**Project 3 Report Justin Chang**

**UID: 504732893**

a) For this project, presenting the board was actually quite difficult because the orientation of the r and c passed in as a point had to be mapped a certain way. In order to actually display the board as well, since there was a dedicated 2 space buffer before the start of a row, and a dedicated row for numbering from 0->n-1 as well as 0->n-1 for columns, for a number passed in 1<=n<=10 (since MAXROWS and MAXCOLS=10), I ended up using a 2D array of ints to “mark” the values of a specific board orientation and a 2D array of chars to map the ints into some type of symbol based on either shipIDs (which were just indexes in a vector of ints). The ships were declared as a class that really could have been a struct, which basically holds a char symbol, an int length, and a std::string shipName. All of these can be accessed based on the index of the vector of ship types (which I first tried to dynamically allocate as objects but found out that was unnecessary and excessive). These integer ids were transferred onto the board when needed to be placed on the board, and in the char display, sent to the char symbol accessor method of the game to display the symbol of the char of the ship.

Additionally, after setting up those things, actually getting the display of the numbered row of ints and cols also took a while because passing in a row number actually corresponded to the “column” of the board and a column number actually corresponded to the “row” of the board. This was a huge problem in my debugging process because I had to constantly change some of the way I mapped the values into the board. At first, originally, I thought for example, the point (4,2) would go nicely into the char display board (4,2) but instead, would go into the point (2,4) on the board and update that and give me a lot of undefined behavior with things because it would be say, expecting a shipID for example but get water and just map to something totally different. Another downside of this was if I had allocated ships and needed to unplace ships or if I had tried to map the value into something weird and displayed it, it would fail and display extremely undefined behavior, also making it hard to figure out where exactly my problems were. For instance, the int boardDisplay that needed to be updated had -2 for blocking values or # in charBoardDisplay, -1 for water value or . for charBoardDisplay, 0-> nShips as the shipId and corresponding charSymbol for charBoardDisplay, 1000 for a hit water, or o for charBoardDisplay and 1001+nShips for a hitShip in general or X for charBoardDisplay.   
  
Even coming up with the above values took me a lot of time planning out and experimenting as well discarding some of the values because initially, I almost thought about using enumerations but that would take too long and be too specific. Additionally, I forgot that a shipId could be 0 and also tried to make the water 0, meaning it wouldn’t actually get the shipId 0 and fail to place that ship on the board. Additionally, there was also a problem with trying to set a number to low to correspond to a hit because the values I used assumed that there would never be more than 1000 ships which is realistic assuming my MAXROWS and MAXCOLS are limited to 10x10, and also meaning that my ships can only expect to have 100 ships. If I try to push more than 100 ships, there should be no way for that to pass into the function of placing ships realistically and I wouldn’t need to worry about the mapping of that into the charDisplay.

Some other small things to notice about charDisplay is that it required a minimium size of MAXCOLS+3 MAXROWS+1 because of the inverted nature of the display, and the fact that there should be more cols than rows anyways. Additionally, each value to be represented from the int displayBoard to charDisplay would require for instance, if I had

for(int r=0;r<n1;r++)

{

for(int c=0;c<n2;c++)

{

if(displayBoard[r][c]==somethingInt)

charDisplay[r+2][c+1]=somethingChar;

}

}

I would need that r+2, c+1 mapping, as well as keeping track of what r and c really correspond to which would take me some more inversion steps to keep track of and properly map into its rightful locations.

Additionally, after placing a specific ship, I would need to take the ship placed and also insert that particular shipID to an existingShipID private vector in the Board class to make sure when I need to unplace and replace some ships that I don’t accidentally place the same ship more than once or have any problems with that. This allows me to also have a way to track unique shipIDs without getting too deep into trying to implement something or add something that really shouldn’t be there. I also have a destroyedShips vector to keep track of when a ship will be destroyed and the final function of the board class will be called in the game play interface when needing to check for a win condition, which basically returns if existingShipIDS is == destroyedShips in terms of size. This is helpful in many ways and consistent because existingShipsID should not change after successfully placing ships, and destroyedShips accurately determines how many ships would be left on the playing field since it can be updated by the attack function (which I will describe in the next paragraph).

The attack function was also pretty intense and detailed because it had so many things going on in general. It had to interact with not only the board in some ways but also player because of how the board needed to be updated, how a player needed to record its attacks on enemy players, and how the good player also needed to record attacks enemy players made on it. The shotHit, shipDestroyed, and shipID paramters were passed be reference and intended to be used to check when a shot would either hit, which would update displayBoard to either 1000 or 1001+shipID (not updating char displayBoard because that is unique to display function, which I should also add, takes in a Boolean that either shows everything or shotsOnly based on if you are a human player or not). The shotHit, only turns true when a value not -1 is Hit, and would update the play function that a player has shot a board that contained a ship. If its not returning true, we know it hit water, or an already shot Boat/water. This means that the person “wasted” a shot, which came up quite often a lot even in debugging mediocre player, good player, human player (which actually also took a while due to string stream problems and needing more than 1 int and the double cin.ignore enter to continue problem).

The shipDestroyed and shipID parameters also come into play when a shotHit clears a ship, which is checked if the shipID no longer exists on the displayBoard, we know that we cleared a ship and we can update a destroyedShips private vector to sweep up and take the ship destroyed (spent a while here to be honest before realizing I didn’t actually need a way to deallocate a ship because deallocating the ship from here also was quite difficult and might need the power of the friend key word to properly do so) and continue normal play. We would update the shipID with the destroyed ship and tell the play function in game so that the players can know which ship was sunk as a result of the destroyedShip parameter being true.

Another small debugging problem I ran into in this function actually lied in the fact that there was still room for throwing bad points. All of the game functions protect against bad points but not board, so I needed to implement some form of bounds checking in the board class. Returning false on a shot hit and true on a shot not hit also almost gave me inspiration to use these functions in the testing of my player.cpp’s good player class to check for even more unique ways of separating/ partitioning good spots for my ship. However, I realized that was a little bit unnecessary and I could have just used a check on if unplaced ship works on specified coordinates (Which ill get into in part b).

Lastly, getting the play function to do everything required by the sample game and spec took a lot of time as well since there was a lot of error bounds checking and cout displays that needed to be correct. This was not an easy task as there was a lot of room for mistakes in spaces, mistakes in calling the wrong display version or marking the wrong boards to record attacks and also testing the wrong kinds of points to be marked or even just blocking and unblocking, though it shouldn’t even have been too bad of a problem. Just getting all of these things lined up and ready to go though was extremely difficult but definitely doable and fun in the end. At some point I realized I was really messing with an extremely high amount of for loops, if statements, and maybe even having extremely terrible efficiency in terms of run time, but I knew it would work and be safe in the end. Though the board overall interface implementations was definitely challenging and had a lot of problems to debug, the main problems I think I encountered and the more technically challenging in terms of design and implementation would still go to desigining the mediocre and good player classes, which I will describe now in part b.

b) My good player’s strategy for placing ships would be extremely similar to the mediocre player’s strategy for placing ships (which will be described in both pseudocode and description in part c). However, I added another functional check to make sure that theres an extra 3 by 3 central grid blocked off for adding ships, making it even harder to place a good player’s ships and making the ships more spread out in general. This however, doesn’t always guarantee that the ships would always not be centralized, it just reduces the chances of being centralized and since it doesn’t have a consistent way of always telling that it should be able to fit the ships in a bunch of moves, I also gave it a “worst case” mediocre style of ship placements which is still decent because blocking random squares and spreading out ship placements already improves the situation for a player in terms of hiding and placing ships in good locations. The main idea was to just make sure since, most players have a higher chance of randomly shooting the middle areas, I wanted to make sure that there was a higher variance of staying away from the middle as well as still spreading out ships in a non trivial way (which will be described in part c). Since the spec didn’t really specifically state that there was a worst case should still be able to place all the ships in a bad board scenario if the given ships should fit on the board in a specific orientation, I assumed that condition didn’t need to be covered. Well since it needs to be covered now, I need to be able to palce all the ships so if it keeps failing the mediocre player’s attempt (say it fails like 100 times) then I’ll try to use the same idea of ship placement by rearranging recursive call until it fits the board, thus guaranteeing that the good player must be able to place everything on the board. The biggest part about ensuring all ships could be placed though would basically lie in the fact that I needed to clear board and make sure all existing ships or destroyed ships were also removed, for safe access to use an algorithm to literally try to fit on the board. This also forced me to catch another subtle error that lied in the fact that clear board needed to not only just set coordinates on the display board to water, but also to remove all existing traces of ships to ensure that unless I had a board where there was too many ships, I can still ensure a way to place all ships on the board.

The goal for recommending moves for the good player was to not have a random valid point shooting method used by mediocre player. A good player, I think (based on an online analysis of battleship), can actually divide up all possible orientations and try to shoot based on the highest probable ship placements until a ship is hit and enter stage 2. However, that would probably be too hard to implement and surpass the 5 second time limit imposed by the spec, so I went for a different approach. I used the concept that there should be a way to partition, based on the lengths of the ships in nShips (you can get that using game().nShips loop to get shipId and passing that into the length function) and partition based on that to search for a particular ship as stage 1. In other words, for example, if there is a chance to shoot for a boat of length 2, which will basically be in every game, the partition method will cut the board in half and literally only shoot every other tile alternating in diagonal down each column, chosen at random. When a hit is made, which has a little more strategy than for mediocre player, then I know how to aim when I go to stage 2 and 3. When a ship is hit, we activate stage 2 which will have a stronger algorithm than mediocre player to shoot either 1 up or 1 down or 1 left or 1 right, by random. If it hits 1 up or 1 down, then it disregards 1 left or 1 right and either works up or down continuously, by activating stage 3 , and blocks off one side (if it misses on say, down), and activates stage 4 and just goes in 1 direction (in this case up), or continues shooting both directions. If it sinks a ship in any of the stages above not=1, it will go back to stage 1, and continue partitioning, assuming it has destroyed the biggest ship if it destroys a ship. If it fails to destroy the right ship, and there are still ships on the board, it will reloop the partition to keep shooting from that size down. (It will also keep track of already shot locations just like mediocre player so it will end up not reshooting already shot locations in any of the stages).

c) Pesudocode for Mediocre Player placing ships is as follows : (sorry, it might be very very long and difficult to understand still since my actual code was already really bad and probably not optimal but It seems to work and follow the spec guidelines )

**MedicorePlayer Place Ships:**

(after calling block, unblock, and having the for loop that checks if the recursive implementation would work on 50 tries, here is the function for describing if I can recursively place my ships)

if size of the game board dimensions doesn’t allow the ships to fit based on length,

return false

if n is last ship (base case),

for every possible point on the board,

check if the ship can be placed at a given orientation (an orientation is a struct that holds a direction (vertical or horizontal), point (row and col), and has a Boolean condition for whether or not it can fit based on this specific configuration)

check if board can place ship calling board’s place ship function

if can, since n is last ship, we are good and can return true

else just keep testing if possible

if at the end we can’t we will have to unwind previous ships, if any exist and recall this function until all possible orientations on the board has been tested. This is checked with another vector of provisional temp orientations to make sure all possible orientations possible have been properly pushed.

else if n is not last ship

Must run a check to make sure before attempting placement (which must check orientation as well on the specified row col point) that this position has not been attempted before (hence why we have a unwind algorithm that also must push these unwinded points)

for every possible point on the board,

check if ship can be placed at given orientation

check if board can place ship calling board’s place ship function

if can, place it and increment a counter for placing ships, which will help us unwind if we have a problem later

same as above procedure, if can place ship after calling board place ship, and if not, we will need to unwind if all possible points considered for n!=game’s number of ships.

Must run a check to make sure before attempting placement (which must check orientation as well on the specified row col point) that this position has not been attempted before (hence why we have a unwind algorithm that also must push these unwinded points)

if unable to place ships earlier

unwind algorithm that pushes all the provisional test points to a bad vector check to make sure we don’t keep trying those points for specified shipId

call unplace function on the same provisional orientation points vector for each ship placed. If no ships placed, we can just return.

The case for if unplaceable configuration due to overrun ship size has also been configurated in the top base case

function will call itself after unable to place ships with reset parameters, but with marked down orientation vectors and bad vectors to ensure unique different points are tested.

Else if able to place ships earlier

Function call itself with an increased position to insert next shipId (how we reduce to base case essentially, and try to reach a point of placing all the ships given a configuration.

(somewhere outside the recursive function) remember to call unblock! And reset the vectors by calling clear function.

**Mediocre Player Attack:**

Using either enumrations or a set of bools to track states,

if in state1,

attack randomly.

If already attacked, don’t select that point.

If in state2,

Attack in a limited row col space, using a vector to mark specific attack values

Also make sure that in this limited row col space that if already attacked, don’t select that ppoint.

Can select points at random in this limited row col space. Keep choosing until is unique.

How to determine states (in a different helper function that is called recordAttacks)

record attack point

check if shotHit, shotDestroyedShip, and make sure that if in state1 (random attacks) and killed a ship, stay in state1

if shotHit but no ShipDestroyed, switch to state2 and do below

if size of limited squares vector is not 0, (means we had diff limited squares from a previous attack that switched into state2)

clear the vector.

record from 4 left to 4 right and 4 down to 4 up of point of hitting ship but not destroy,

if unique non attacked points,

push into a vector of limited squares

else

ignore the values

if just switched into state 2,

run the above algorithm again to make sure that nothing leaks out

else

check if a shot in state2 killed something.

If shipDestroyed

Switch back to state1.

Else

Stay in state2.

If run out of points in state2

Go to state1. (for ship sizes > 5)

**Good Player Ship Placement:**

(same as mediocre player, except it does an extra check on if unplaceable with the block,)

place all the ships with same algorithm without calling block (which ensures if given a certain orientation, should be possible if recursive method works, which should be the case).

Good Player doesn’t record any attacks, since its not always clear if an attack will be consistent to a player’s, especially mediocre player’s, attack strategy for changing ship placements.

**Good Player Attack:**

Good player is slightly similar but has a few more strategies in its game, such as a better random shot selection based on a partitioning of the board, and a more consistent, centralized recursive attack algorithm to not go too far out after reaching state2 by entering a state3.

Basically, the same as the above mediocre player’s attack strategy pesudocode +

If in state1,

Limit squares to values from row and column separated by 1 length, and spread out diagonally, making it seem like a “checkerboard” to reduce range of random shots fired.

If in state2,

Push only the 4 adjacent (max) cells to attack and attack all of them

If attacked 2 horizontal misses, we go to state 3 and focus on vertical line

If attacked 2 vertical misses, we go to state 3 and focus on horizontal line

If attacked with neither of those, conditions, we assign new “points” to recursively call in state2.

If in state3,

If vertical line

Continue to only push vertical coordinates

If horizontal line

Continue to only push horizontal coordinates

And of course like earlier, the same conditions apply to state 2 and state 3 and state 1 when to switch states.

State 2->state3 is dictated by

If attacked 2 horizontal misses, we go to state 3 and focus on vertical line

If attacked 2 vertical misses, we go to state 3 and focus on horizontal line

All of this was done using a few vectors as well as recursion. I also used structs to organize most of the tasks to simplify readability, but still ended up doing a lot of work to manage and work through them. Additionally, going through the vectors that contained custom data types took a lot longer when comparisons were needed due to non overloaded assignment operators and non overloaded comparison operators.