CS32: Project 3 Report

1. For this project, I mainly used vectors to store data. However, the game board itself, defined in Board.cpp as gameBoard, is represented using char \*\* gameBoard. I found this implementation more resilient, as I previously had a 2-dimensional array, which required a fixed size, which would have been potentially problematic. Also, to store the length, ID, and symbol of a ship, I made a struct of type ship to store this data. In this way, I could access these data members by means of a dot operator, which made my program much more organized and coherent. And as for vectors, they are my favorite data structure to use by far, especially considering the limitations of its counterpart, the array. Using the STL I was able to minimize coding time and maximize efficiently, such that I effectively stored data and could later quickly access it. For example, each time I added a ship, I pushed it into a vector, so that the program would remember its ID, and know not to place it again. This also applies for shots hit before, as if the user entered the same coordinates, the program could let them know of the wasted shot. This also applies for generating a shot for AI, as to remember that it had already tried that previous shot. Thus, using this implementation, the program is more effective.
2. A good player strategizes a game in terms of placing its ships and attacking its opponent’s ships logically. I coded the GoodPlayer class to place its ships randomly. The possible downfall is that ships could be placed together, but I believe this is made up for by the implemented attack. By choosing every other point, the likelihood of hitting a ship is cut in half. Also, if the attack returns true for hitting a ship, it will then attack the points around that initial point. If one of those points returns as a hit, the process repeats until the ship is destroyed.
3. Pseudocode:

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

*Clear board objects.*

*Place ships. If can’t be placed, return nullptr.*

*While there are non-destroyed ships on both boards*

*If shouldPause, wait for enter.*

*If human, display boards with shots only presented.*

*If not human, display boards with all objects.*

*Call recommend attack.*

*Display boards.*

*If player 2’s board returns true for all ships destroyed*

*Return player 1.*

*If player 1’s board returns true for all ships destroyed*

*Return player 2.*

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

*If constraints not satisfied, return false.*

*Store ship data into vector that holds added ships.*

*Add specified symbol to board.*

*Increment added ships total.*

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

*Check if passed data, shipId, matches with Point p coordinate. shotHit is true is Point p belongs to a placed ship.*

*If hit ship amount of times of ship length, shipDestroyed is true.*

bool MediocrePlayer::placeShips(Board& b)

*For 50 iterations*

*Block the board object.*

*Attempt to place ships using recursive call*

*If can be placed horizontally, return true. Pass the next ship’s Id to attempt to*

*place.*

*Otherwise, call the function with the temporary point.*

*If can be placed vertically, return true. Pass the next ship’s Id to attempt to*

*place.*

*Otherwise, call the function with the temporary point.*

*Unblock the board.*

*If can’t be placed*

*Unblock the board.*

*Continue the loop.*

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

*If in state 2*

*Check if hit ship is destroyed.*

*If so, reset to state 1.*

*If in state 1*

*Check if hit ship is destroyed.*

*If so, reset to state 1.*

*Push the points surrounding Point p, such that it forms a cross.*

bool GoodPlayer::placeShips(Board& b)

*Place ships in random direction*

*Horizontal or vertical*

*Generate random point*

*If can place ship at random point at this direction*

*Break*

*Otherwise, continue the loop until able to.*

Point GoodPlayer::recommendAttack()

*If not in state 1*

*Generate random point*

*Search made shots, random point not found, push it.*

*Set to state 1.*

*If in state 1*

*Generate random point*

*Return this point.*