CS 32 Spring 2013
Project 3
Battleship
Due: 11 PM, Tuesday, May 30

Introduction

You have been hired to write a computer version of the classic Battleship game. Battleship is a simple game where each player attempts to sink their opponent's fleet of ships before the opponent sinks theirs.

Here's how you play it:

1. Each player has a number of different ships. In Standard Battleship, each player has five ships:
    a. An aircraft carrier which is 5 segments long: AAAAA
    b. A battleship which is 4 segments long: BBBB
    c. A destroyer which is 3 segments long: DDD
    d. A submarine which is 3 segments long: SSS
    e. A patrol boat which is 2 segments long: PP
2. Each player has their own up-to-10x10 board (with rows and columns numbered 0 through 9), in which they place their ships. Each ship may be placed horizontally or vertically, but not diagonally. No ship may overlap another on the board. Each player sees only their own board; they cannot see where their opponent has placed their ships. A player's board in a 10x10 game might look like this:

```
  0123456789
0 ....AAAAA.
1 .......P..
2 .......P..
3 ..........
4 ...D......
5 ...D......
6 ..SD......
7 ..S.......
8 ..S...BBBB
9 ..........
```

3. After both players have placed their ships, game play begins. The first player picks a row and column coordinate to attack on the opponent's board and announces it to the opponent. The opponent then tells the player whether or not the player hit a ship, and if so, whether or not the entire ship has been sunk. A ship is sunk when all of its segments have been hit. If a player has sunk the entire ship, the opponent must indicate which ship was sunk, e.g. "You sank my battleship!"
4. After the first player attacks, the second player is given an opportunity to attack the first player. Attacks alternate between players until one player has sunk all of the other player's ships.
5. The first player to sink all of their opponent's ships wins the game.

The Assignment

For this assignment, you will have to complete the Battleship game we started building, including the following classes:

1. **A Board class:** This class maintains the board data structure. During a game, there will be an instance of the Board class for each player, to record the locations of and damage to that player's ships. You will need to implement most of this class's member functions.
2. **A Player abstract base class:** This base class specifies the interface for all players. You will be given this class. A Player has two main tasks:
    a. It must place the ships on the board using a specified algorithm.
    b. It must make attacks against its opponent in a game of Battleship.
3. **An AwfulPlayer class,** derived from the Player class: We provide this class as an example of a concrete class derived from Player. It plays the game legally, but very stupidly.
4. **A HumanPlayer class,** derived from the Player class: You will implement a Human Player class. The functions in this class will prompt the user to decide where to place ships, and where to attack. (This way, the user can play against the computer.)
5. **A MediocrePlayer class,** derived from the Player class: We will provide a set of requirements so you can design and build a Mediocre Battleship Player. You will

need to implement most of this class's member functions. Your Mediocre Player must work exactly as specified later in this specification.

6. **A GoodPlayer class**, derived from the Player class: You will be asked to implement your own Battleship player algorithms. Your goal is to build an artificially intelligent player class that can beat the Mediocre Player as well as your fellow students' GoodPlayers.
7. **A Point class:** This simple class is used to specify a (row,column) coordinate. You don't have to write any code for this class! ☺
8. **A Game class:** This class is used to run a complete game between two players.
9. A **main function** allowing the user to create and play a game. We will not look at your main function, so you can use it for testing purposes.

The Direction and Point Types

The file **globals.h** that we supply defines two types used in many places in the program. A Direction is an orientation for a ship, and a Point is a (row,column) coordinate for a position on the board.

```
enum Direction {
      HORIZONTAL, VERTICAL
};

class Point
{
   public:
      Point() : r(0), c(0) {}
      Point(int rr, int cc) : r(rr), c(cc) {}
      int r;
      int c;
};
```

The Game Class

The Game class is used to run a complete game between two players. It also answers queries about game parameters:

```
class Game
{
  public:
    Game(int nRows, int nCols);
    ~Game();
    int rows() const;
    int cols() const;
    bool isValid(Point p) const;
    Point randomPoint() const;
    bool addShip(int length, char symbol, string name);
```

```
    int nShips() const;
    int shipLength(int shipId) const;
    char shipSymbol(int shipId) const;
    string shipName(int shipId) const;
    Player* play(Player* p1, Player* p2, bool shouldPause = true);
};
```

Game(int nRows, int nCols)

Construct a game with the indicated number of rows and columns, which must not exceed MAXROWS and MAXCOLS, respectively. (Those constants are defined in **globals.h**.)

int rows() const

Return the number of rows in the game board.

int cols() const

Return the number of columns in the game board.

bool isValid(Point p) const

Return true if and only if the point denotes a position on the game board.

Point randomPoint() const

Return a random point on the game board.

bool addShip(int length, char symbol, string name)

Add a new type of ship to the game. (Later, when play() is called, each player will place one of each type of ship on their board.) The ship has the indicated length, which must be positive and must allow the ship to fit on the board. When the board is displayed, the indicated symbol will be displayed in the positions occupied by that ship; that symbol must be a printable character other than X, o, and . (which are used for other purposes in the display), and must not be the same as for any other ship. The indicated name is what will be used to denote the ship when writing messages. If g is a Game, the function might be called like this:

```
g.addShip(5, 'A', "aircraft carrier")
```

If the arguments in a call to addShip satisfy the constraints, the call results in a new ship type being added to the game, and the call returns true. Otherwise, a new ship type is not added, and the call returns false.

int nShips() const

Return the number of ship types in the game, which will be the number of times addShip was called returning true. The integers from 0 to nships()−1 will be the ship IDs of the ships in the game, each one corresponding to the ship type added by one of the successful calls to addShip. Although each ship type's ship ID will be a distinct integer in the range 0 to nShips()-1, it is not required that the first ship added have ship ID 0, the next 1, the next 2, etc. (although that may what naturally happens with your implementation).

int shipLength(int shipId) const

Return the number of positions occupied by the ship whose ID is shipId.

char shipSymbol(int shipID) const

Return the character that, when displaying a board, will be used to represent the ship whose ID is shipId.

string shipName(int shipID) const

Return the string that, when writing messages, will be used to denote the ship whose ID is shipId.

Player* play(Player* p1, Player* p2, bool shouldPause)

This function runs a complete game between the two indicated players, returning a pointer to the player who won the game, or nullptr if some situation occurred that makes it impossible to complete the game (e.g., a player being unable to place the ships on their board). The parameter p1 is a pointer to the player who goes first; p2 is a pointer to the other player.

If the third parameter is true, then after displaying the result of each shot, the function displays a "Press enter to continue: " prompt and waits for the user to press enter before continuing. This parameter defaults to true if the play function is called with only two arguments. If the parameter is false, no such pause occurs; this is useful when testing the function merely to determine the winner, such as might be done from a script that discards the actual output.

The play function must do the following:

1. It creates two Board objects, one for each of the players.
2. It calls the placeShips function of each player to place the ships on their respective board. It is possible that the placeShips function may fail to place the ships in some instances (e.g., if the game parameters are such that there is no configuration of ships that will fit, or if a MediocrePlayer is unable to place all of

the ships because of blocked squares). If a player's placeShips function returns false, your play function must return nullptr.

3. Once both players have successfully placed their ships, game play starts.
4. Until one of the players wins the game:
    a. Display the second player's board (since the first player will be attacking that board). If the first player is human, do not show undamaged segments on the opponent's board, since that would be cheating. If the first player is not human, then show the entire opponent's board, since we're just an onlooker to what the computer player will decide for itself.
    b. Make the first player's attack.
    c. Display the result of the attack.
    d. Repeat these steps with the roles of the first and second player reversed.
5. If the losing player is human, display the winner's board, showing everything.

You can't go wrong by having your play function conduct the game the way it does in our sample program.

To ensure that you do not change the interface to the Game class in any way, we will implement that class for you. But don't get your hopes up that we're doing any significant work for you here: Our implementation is to simply give Game just one private data member, a pointer to a GameImpl object (which you can define however you want in Game.cpp). The member functions of Game simply delegate their work to functions in GameImpl.[1] You still have to do the work of implementing those functions, although we have done some of that work for you. (Examine the last half of **Game.cpp** to see what we did.)

Other than Game.cpp, no source file that you turn in may contain the name GameImpl. Thus, your other classes must not directly instantiate or even mention GameImpl in their code. They may use the Game class that we provide (which indirectly uses your GameImpl class).

The Board Class

The Board class maintains the board data structure. Here is the required interface that you must implement:

```
class Board
{
  public:
    Board(const Game& g);
    ~Board();
    void clear();
    void block();
    void unblock();
```

[1] This is an example of what is called the pimpl idiom (from "**p**ointer-to-**impl**ementation").

```
    bool placeShip(Point topOrLeft, int shipId, Direction dir);
    bool unplaceShip(Point topOrLeft, int shipId, Direction dir);
    void display(bool shotsOnly) const;
    bool attack(Point p, bool& shotHit, bool& shipDestroyed, int& shipId);
    bool allShipsDestroyed() const;
};
```

Board(const Game& g)

Initialize a board. The parameter g refers to the game the board is being used in. Board member functions that need to know the number of rows or columns or the characteristics of ships will presumably find out from the game object the board was constructed with.

void clear()

This function clears the board so it is empty, ready to be populated with ships.

void block()

This function blocks about half the positions on the board. This function will be used only by the MediocrePlayer when placing ships on the board at the start of play. If a position is blocked, then that player may not place a ship that occupies that location.

void unblock()

This function unblocks all the blocked positions on the board. This function will be used only by the MediocrePlayer, *after* placing ships on the board, but before the start of play.

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

This function attempts to place the specified ship at the specified coordinate, in the specified direction. The parameter topOrLeft specifies the topmost segment of the ship if dir is VERTICAL, or the leftmost segment if dir is HORIZONTAL. The parameter shipId is the ship ID number. This function returns false if the ship cannot be placed because:

1. The shipId is invalid
2. The ship would be partly or fully outside the board.
3. The ship would overlap an already-placed ship.
4. The ship would overlap one or more positions that were blocked by a previous call to the block function.
5. The ship with that ship ID has previously been placed on this Board and not yet been unplaced since its most recent placement.

If this function returns false, then the board must remain unchanged (i.e. no part of the ship may remain on the board upon failure). If this function successfully places the ship, the board is updated to reflect that, and the function returns true.

bool unplaceShip(Point topOrLeft, int shipId, Direction dir)

This function attempts to remove the specified ship from the board, so the positions it occupied may be used to place other ships. The parameters are the same as for placeShip. This function returns false if the ship cannot be removed because:

1. The shipId is invalid
2. The board does not contain the entire ship at the indicated locations.

If this function returns false, then the board must remain unchanged (i.e. no part of the ship may be removed upon failure). If this function successfully removes the ship, the board is updated to reflect that, and the function returns true.

void display(bool shotsOnly) const

This function displays the board, using the following format:

1. First line: The function must print two spaces followed by the digits for each column, starting at 0, followed by a newline. You may assume there will be no more than 10 columns.
2. Remaining lines: The function must print a digit specifying the row number, starting at 0, followed by a space, followed by the contents of the current row, followed by a newline. You may assume there will be no more than 10 rows. In each of the positions of the row, use the following characters to represent the playing field:
    a. If the shotOnly parameter is false, use the ship's symbol to display an undamaged ship segment; if the shotsOnly parameter is true, show a period to display an undamaged ship segment.
    b. Use an X character to display any damaged ship segment.
    c. Use a period to display water where no attack has been made.
    d. Use a lower case letter o character to display water where an attack has been made that missed a ship.

Here’s an example with shotsOnly being false:

```
  0123456789
0 ....AXXAX.
1 .......P..
2 .o.....P..
3 ..........
4 ...D..o...
5 ..oD......
6 ..XD..o...
7 ..X.......
8 ..X...BBBB
9 ..o.......
```

Here's that same board with shotsOnly being true:

```
  0123456789
0 .....XX.X.
1 ..........
2 .o........
3 ..........
4 ......o...
5 ..o.......
6 ..X...o...
7 ..X.......
8 ..X.......
9 ..o.......
```

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

This function is used to submit an attack against the board. The function must return false if the attack is invalid (the attack point is outside of the board area, or an attack is made on a previously attacked location). The function returns true if the attack is valid, regardless of whether or not any ship is damaged.

If any undamaged segment of a ship is at position p on the board, then the shotHit parameter must be set to true, and the segment must henceforth be considered a damaged segment. Otherwise the shotHit parameter must be set to false.

If this specific attack destroyed the last undamaged segment of a ship, then the shipDestroyed parameter must be set to true and the shipId parameter must be set to the ship ID of the ship that was destroyed; otherwise the shipDestroyed parameter must be set to false and shipId must be left unchanged.

It's up to you whether to set to some value or leave unchanged:

1. shotHit, shipDestroyed, and shipId if the attack is invalid
2. shipDestroyed and shipId if the attack missed

bool allShipsDestroyed() const

This function returns true if all ships have been completely destroyed on the current board and false otherwise. (If this is true, it means that the player who was attacking that board has won the game.)

As with the Game class, the real work will be implementing the auxiliary class BoardImpl in Board.cpp. **Other than Board.cpp, no source file that you turn in may contain the name BoardImpl.** Thus, your other classes must not directly instantiate or even mention BoardImpl in their code. They may use the Board class that we provide (which indirectly uses your BoardImpl class).

The Player Abstract Base Class

The Player abstract base class specifies the interface for all players. We have implemented this class for you.

```
class Player
{
  public:
    Player(string nm, const Game& g);
    virtual ~Player();
    string name() const;
    const Game& game() const;
    virtual bool isHuman() const;
    virtual bool placeShips(Board& b) = 0;
    virtual Point recommendAttack() = 0;
    virtual void recordAttackResult(Point p, bool validShot, bool shotHit,
                                                    bool shipDestroyed, int shipId) = 0;
    virtual void recordAttackByOpponent(Point p) = 0;
};
```

```
Player* createPlayer(string type, string nm, const Game& g);
```

Player(string nm, const Game& g)

Initialize a player whose name is nm. The parameter g refers to the game the player is being used in.

string name() const

Return the name of the player.

const Game& game() const

Return a constant reference to the game the player is being used in. (Derived classes can use this to find out the number of ships in the game, for example.)

bool isHuman() const

Return true if the player is a human. (This affects how an opponent's board will be displayed.)

bool placeShips(Board& b)

This function must be implemented in concrete derived player classes. It must implement the logic required to place the ships on the board before game play begins. It returns true if all ships could be placed, and false otherwise (e.g., because the game parameters are such that there is no configuration of ships that will fit, or because a MediocrePlayer is unable to place all of the ships after 50 tries).

Point recommendAttack()

This function must be implemented in concrete derived player classes. It must return a point that indicates the position on the opponent's board that the player would like to attack next.

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

This function must be implemented in concrete derived player classes. Game::play() must call this function after an attack is made on an opponent's board to report the result of the attack to the attacking player. The position p is where the attack was made. The parameter validShot will be true if the attack was valid, or false if it was invalid (i.e., p is outside of the board area, or is a position that had already been attacked previously). If validShot is true, then shotHit is true if the attack hit a ship segment, and false otherwise. If shotHit is true, then shipDestroyed is true if that was the last undamaged segment of the ship that was hit, and false otherwise. If shipDestroyed is true, then shipId is the ship ID of the ship that was destroyed. You can not assume anything about the value of

1. shotHit, shipDestroyed, and shipId if validShot is false
2. shipDestroyed and shipId if shotHit is false
3. shipId if shipDestroyed is false

This function is intended to let an intelligent non-human player update its data members in a way that will let the next call of recommendAttack make a good choice.

void recordAttackByOpponent(Point p)

This function must be implemented in concrete derived player classes. Game::play() must call this function when an attack is made on this player's board. The position p is where the attack was made. The player may very well have this function do nothing. However, an intelligent non-human player might want to keep track of an opponent's attacks and adjust its own attack strategy accordingly. For example, the player might notice the opponent is attacking in a certain pattern, use that to predict how far from winning the opponent is, and play differently depending on whether or not an opponent is far from or close to winning.

Player* createPlayer(string type, string nm, const Game& g)

This is *not* a member function of any class. It is a so-called *factory function* that lets you create a player without the compiler's needing to have seen a class declaration for that player at the point where you create that player. The parameter nm is the name of the player and g is the game the player is being used in. The first parameter determines the kind of concrete player to be created:

"awful" for an AwfulPlayer
"human" for a HumanPlayer
"mediocre" for a MediocrePlayer
"good" for a GoodPlayer

The function returns nullptr if the type string is not one of those four. Otherwise, it returns a pointer to a dynamically allocated object of the indicated type. It is the caller's responsibility to ultimately delete the object.

All concrete classes derived from Player must have a constructor that takes a string for the player's name and a constant reference to a Game, since that what createPlayer assumes.

Other than Player.cpp, no source file that you turn in may contain the names AwfulPlayer, HumanPlayer, MediocrePlayer, or GoodPlayer.

The AwfulPlayer Class

This is a concrete class derived from Player, for which we have provided the code. It places ships in the same dumb way every time, and recommends attacks in a systematic pattern, paying no attention to the effect of previous shots.

The HumanPlayer Class

This is a concrete class derived from Player. It places ships and recommends attacks based on user input. The isHuman function must return true, so that Game's play function can ensure that a human player doesn't get to see where the opponent's ships are during play. The placeShips and recommendAttack functions must prompt the user for

where to place the ships and where to attack, respectively, in the manner of the posted sample program. The recordAttackResult and recordAttackByOpponent probably need not do anything.

The MediocrePlayer Class

This is a concrete class derived from Player class that implements an artificially intelligent mediocre Battleship player. You are required to use particular algorithms for placeShips and recommendAttack/recordAttackResult/recordAttackByOpponent.

bool placeShips(Board& b)

For a MediocrePlayer, this function must place the ships on the board using the following algorithm:

1. First, it must call the block function on the specified board. This will randomly block off about half the positions on the board. No segment of any ship will occupy one of these squares.
2. Second, you must use a **recursive algorithm** to place each of the five ships on the board, ensuring that no ship overlaps any other ship and that no ship overlaps a blocked position. You must use the board class's placeShip function to attempt to place a ship on the board, and unplaceShip to remove it from the board if the position isn't appropriate. To fit each ship, you may need to place some of them vertically, others horizontally, etc. You may find that after placing some of the ships, you have misallocated the space and need to **backtrack**, removing the ships you've already placed to try a different configuration.
3. After attempting to place the ships, your function must call the unblock function to remove the blocked squares. This will leave just the ships on your board.
4. If all ships could be placed, this function returns true. If it is *impossible* to fit all of the ships on the board, given the set of blocked positions from the first step, then your function must go back to step 1 and try again if it hasn't yet done so 50 times. If the function has not returned true after 50 tries, then it must return false.

(Hint: placeShips itself will probably not be recursive. For step 2 above, though, it may well call an auxiliary function that *will* be recursive.)

Here's an example of the backtracking algorithm. Suppose we wish to place three ships into the board below, a 4-segment battleship, a 3-segment submarine, and a 2-segment patrol boat. (Note that # represents a blocked position; the actual block() function will not block this high a percentage of the squares):

```
  0123456789
0 ##########
1 ###.######
2 ###.#.####
3 ###.#.####
4 ##....####
5 ##########
6 ##########
7 ##########
8 ##########
9 ##########
```

If our algorithm attempted to place the battleship first vertically at (1,3), our board will look like this:

```
  0123456789
0 ##########
1 ###B######
2 ###B#.####
3 ###B#.####
4 ##.B..####
5 ##########
6 ##########
7 ##########
8 ##########
9 ##########
```

Next our algorithm would have to place the submarine on the board. It fits vertically at (2,5):

```
  0123456789
0 ##########
1 ###B######
2 ###B#S####
3 ###B#S####
4 ##.B.S####
5 ##########
6 ##########
7 ##########
8 ##########
9 ##########
```

However, as you can see, there is no room left to place the patrol boat, so the placement algorithm must backtrack, removing the submarine and then attempting to place it elsewhere. But it won't fit elsewhere, so the algorithm backtracks to the battleship placement; it must remove the battleship and place it somewhere else. It will fit horizontally at (4,2):

```
  0123456789
0 ##########
1 ###.######
2 ###.#.####
3 ###.#.####
4 ##BBBB####
5 ##########
6 ##########
7 ##########
8 ##########
9 ##########
```

Trying the submarine again, we find it fits vertically at (1,3). Finally, the algorithm would place the patrol boat vertically at (2,5):

```
  0123456789
0 ##########
1 ###S######
2 ###S#P####
3 ###S#P####
4 ##BBBB####
5 ##########
6 ##########
7 ##########
8 ##########
9 ##########
```

void recordAttackByOpponent(Point p)

For a MediocrePlayer, this function does nothing. What attacks an opponent has made do not affect a MediocrePlayer's attack decisions.

Point recommendAttack()

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

For a MediocrePlayer, these functions work together to implement the following algorithm:

0. The Mediocre Player is in one of two states; initially it is in state 1.
1. In state 1, recommendAttack returns a randomly chosen position on the board that has not been chosen before. There are then three possible outcomes when this position is attacked:

a. The attack missed. In this case, the player stays in state 1 (so it will again make a random choice for the next recommendation).
b. The attack hits a ship, destroying its final undestroyed segment, so the ship is destroyed. In this case, the random attack got lucky and completely destroyed a ship that was almost destroyed earlier. In this case, the player stays in state 1.
c. The attack hits a ship, but does not destroy it. In this case, the player switches to state 2 (so will use the algorithm below for the next recommendation)

2. In state 2, let (r,c) be the position that was hit that caused the transition from state 1 to state 2. In state 2, recommendAttack returns a randomly chosen position drawn from a limited set: the set of valid positions no more than 4 steps away from (r,c) either horizontally or vertically, that have not been chosen before. This set defines a cross around (r,c). For example, assume that in state 1, a shot was fired on position (5,3) that hit a ship, but did not destroy it, resulting in:

```
  0123456789
0 ....AAAAA.
1 ..o.......
2 ....BBBB..
3 .o........
4 ...D.....o
5 ...X......
6 ...D......
7 ..S...P...
8 ..S...P...
9 ..S.o.....
```

Subsequent shots in state 2 must be to the positions denoted with a + below:

```
  0123456789
0 ....AAAAA.
1 ..o+......
2 ...+BBBB..
3 .o.+......
4 ...+.....o
5 +++X++++..
6 ...+......
7 ..S+..P...
8 ..S+..P...
9 ..S+o.....
```

By attacking this set of coordinates, the mediocre player is guaranteed to eventually sink the ship that was hit in state 1 if that ship's length is 5 or less. As it turns out, it is possible that the mediocre player may also sink other ships inadvertently with this algorithm, but that's OK.

If the game has ships of length 6 or more, it's possible that every position no more than 4 steps from (r,c) has been previously chosen. In that case, switch to state 1 and use its selection algorithm.

There are three possible outcomes from the recommended attack:

a. The attack missed. In this case, the player stays in state 2.
b. The attack hits a ship, but does not destroy it. In this case, the player stays in state 2.
c. The attack hits a ship, destroying its final undestroyed segment, so the ship is destroyed. (The ship that was destroyed is likely to be the ship that was hit to cause the transition from state 1 to state 2, but it might have been a different ship that was near that one.) In any event, the player switches to state 1.

The GoodPlayer Class

Here's your chance to shine. This is a concrete class derived from Player class that implements an artificially intelligent Battleship player. You may use any algorithms you like to place your ships and to attack the other player, but a call to recommendAttack, then Board::attack, then recordAttackResult must not take more than 5 seconds. (The FAQ addresses the issue of how to limit the time.) See if a GoodPlayer object can beat a MediocrePlayer object most of the time.

Your main() Function

Your main routine must be in a file that you will not turn in. It can do whatever you like.

What We Provide You

We provide you with the following files:

main.cpp	a sample main routine
globals.h	pervasive constants and types and a function for generating random integers
Board.h	the declaration of the Board class
Player.h	the declaration of the Player abstract base class and the implementations of some of its member functions
Game.h	the declaration of the Game class.
Board.cpp	a skeleton of the implementation of the board functionality
Player.cpp	the declaration and implementation of AwfulPlayer, and placeholders for the declarations and implementations of HumanPlayer, MediocrePlayer, and GoodPlayer. Also, the implementation of createPlayer
Game.cpp	a skeleton of the implementation of the game functionality

Other Requirements

You must not make any changes whatsoever to the .h files that we provide you. You will not be turning in those files; we will test your code using our versions of those files, to ensure you didn't sneak in any changes to the required interfaces.

If you write any helper functions, put then in one of the files you will be turning in: Board.cpp, Player.cpp, or Game.cpp.

Functions that *you* write must not write to cout except for the following member functions and helper functions you write that they might call: GameImpl::play, BoardImpl::display, and any member functions of HumanPlayer. Functions you write may freely write to cerr. **Functions that *you* write must not read from cin** except for the following member functions and helper functions you write that they might call: GameImpl::play and any member functions of HumanPlayer.

Subject to the constraints we imposed (e.g., no changes to the public interface of Game, no mention of GameImpl in any file other than Game.cpp), you're otherwise pretty much free to do whatever you want in Game.cpp as long as it's related to the support of only the Game implementation; for example, you may add members (even public ones) to the GameImpl class (but not the Game class, of course) and you may add non-member support functions. Similar freedom applies to Board, BoardImpl, and Board.cpp, and also to Player, HumanPlayer, MediocrePlayer, GoodPlayer, and Player.cpp.

A program consisting of the main.cpp and the .h files that we provided you, along with the Board.cpp, Player.cpp, and Game.cpp files that you turn in, must build successfully under both g32 and either Visual C++ or clang++.

What You Must Turn In

You can still get a good amount of partial credit if you implement most of the project. We will test your Board.cpp in a framework containing our correctly implemented Game.cpp, so that even if your Game.cpp is incorrect, you can earn points for a correctly implemented Board.cpp. Similarly, we will test your Player.cpp in a framework containing our correctly implemented Board.cpp and Game.cpp, and we will test your Game.cpp in a framework containing our correctly implemented Board.cpp and Player.cpp.

You will turn in a zip file consisting of four files:

1. Board.cpp, containing the implementation of board functionality.
2. Player.cpp, containing the declarations and implementation of AwfulPlayer, HumanPlayer, MediocrePlayer, and GoodPlayer, along with the implementation of createPlayer.
3. Game.cpp, containing the implementation of the Game functionality.
4. report.doc, report.docx, or report.txt, your report. This must contain:

a. A description of the design of your data structures. For example, how do you represent the board? How do you record shots made?
b. A prose description of your GoodPlayer's strategies for placing ships and recommending moves. If they're easy to describe, this may well be only a paragraph or two. This is a high-level summary; details go in the pseudocode.
c. Pseudocode for non-trivial algorithms. For example, how does a MediocrePlayer place ships? How does your GoodPlayer implement its strategies?