Due Date: May 1, 2020 @ 11:59:59

Free for all – Battleship game

This assignment will focus on the concepts you have learned so far in this course and implement a game. The entire purpose of this project is to have a little fun and, account for flexibility in the game implementation. In this assignment, a baseline version of the Battleship game is defined for specific constraints. In this assignment, we create 3 classes to represent the game. The functions related to different game pieces are modularized and made easy for code development.

You will submit your work to Gradescope and it will be both auto graded and human graded. The autograder will compile your code with g++-7.4.0. Be sure to watch Piazza for any clarifications or updates to the assignment.

Don't forget the Assignment 7 Canvas Quiz! Your assignment submission is not complete without the quiz.

Your submission must consist of the following files: **Board.h**, **Ship.h, Point.h, BattleShip.cpp, BattleShip.h, README.md** and a **Makefile**. There are 3 classes to be implemented in total. **BattleShip.cpp** implements the **main()** function. **BattleShip.cpp** and **BattleShip.h** files are provided and you are expected to appropriately comment the files.

For header files, in general *(*.h):*

- Be sure to have include guards.
- Don't import anything that is unnecessary.
- Include file header comments that have your name, student id, a brief description of the functionality, and be sure to cite any resource (site or person) that you used in your implementation.
- Each function should have appropriate function header comments.
  - You can use javadoc style (see doxygen) or another style but be consistent.
  - Be complete: good description of the function, parameters, and return values.
  - In your class declarations (and header files in general), only inline appropriate functions.

For source files, in general *(*.cpp):*

- Don't clutter your code with useless inline comments (it is a code smell [Inline Comments in the Code is a Smell, but Document the Why], unless it is the why).
- Follow normal programming practices for spacing, naming, etc. Be reasonable and consistent.
- Be sure to avoid redundant code.

For classes, in general:

- Remember the rule of 3: Explicitly define the destructor, copy operator=, and copy constructor if needed.
- Do not make a member variable public or protected, unless it makes sense to do so.

Battleship game: Brief introduction

Typically, a two-player game, where the two players each place 5 ships on the board. The ships have different sizes and can be placed vertically or horizontally. Refer to 1990 version of link for the ship rules. Ships can touch each other but can't be overlapped. After the ships are placed, each player guesses the position where the ship could have been placed by the opponent. The player who sinks all the opponent's ships first is the winner of the game.

For *Point class: Point <unsigned, unsigned>*

- The two unsigned parameters represent the maximum value that the x and y co-ordinates of Point can hold. They are basically the board dimensions that are set to (10, 10) by default.

- "Status" enumeration with the fields: EMPTY, SHIP, MISS, HIT – to represent the status of each point.
  EMPTY – initialize the points on the board to empty.
  SHIP – a ship is placed at this point on the board.
  MISS – a ship is not placed at this point on the board and the player has incorrectly guessed it.
  HIT – a ship is placed at this point on the board and the player has guessed it right.

Data member variables:

- x and y unsigned fields to represent the co-ordinates of the board.

- Status 'status' data member to store the status of the point

Member functions:

- Default constructor

- Parameterized constructor: Point (char, unsigned) – to set the x and y fields of Point object. Here, the x field (alphabet) is converted to unsigned. Throw a "std::out_of_range" exception if any of the arguments passed is not within the bounds of the board size.

- Parameterized constructor: Point (unsigned, unsigned) – to set the x and y fields of Point object. Throw a "std::out_of_range" exception if any of the arguments passed is not within the bounds of the board size.

- Overload the == operator that returns true, if the two co-ordinates, representing the x and y data members of the Point objects are equal.

- Overload the << operator that displays the respective Point object's co-ordinates in the form (x,y), where x denotes the row number and y denotes the column alphabet. For example, if the point is (0, 0), the displayed co-ordinate is (1, A), to represent the board. (Refer to output_format.txt for more details)

- static char xToChar (unsigned) – To convert an integer (x field of Point object) to character, relative to the character 'A'.

- static unsigned xToInt (char) - To convert a character (x field of Point object) to an integer, relative to the character 'A'.

- Status getStatus () const - To return the enum status member of the Point object.

- Define Ship and Board as friend classes.

For *Ship class: Ship <unsigned, unsigned>*

- The two unsigned parameters represent the board dimensions, like in Point class.

- "Direction" enumeration with the fields – VERTICAL and HORIZONTAL to represent the direction in which the ship is placed on the board.

Data member variables:

- string name – the name of the ship

- vector <Point<*unsigned, unsigned*>*> ship – the co-ordinates occupied by the ship. Note that this class does not manage the memory of the pointer.

Member functions:

- Parametrized constructor: Ship(string, const vector <Point<*unsigned, unsigned*>*>) – to set the name and the co-ordinates occupied by the ship. The contents of the vector can be shallow copied.

- Parametrized constructor: Ship(string, const Point <*unsigned, unsigned*> &, unsigned len, Direction direction, Board <*unsigned, unsigned*> &) – to set the name and the co-ordinates occupied by the ship, depending on the direction (the direction in which the ship

is placed) and len (length of the ship). Place the ship on the board, by calling the getShipPoint() function of Board class with the relevant arguments. In case of invalid ship placement, catch and throw the "std::invalid_argument" exception that might be thrown by the getShipPoint() function, and reset the status of the previously placed ship points to EMPTY.

- void setName (string) and string getName() - setter and getter methods for the string "name" member variable.

- bool isHit (const Point *<unsigned, unsigned>* &) const – returns true if the point co-ordinate represents any ship on the board and sets the status to HIT.

- bool isSunk () const – returns true if the ship has been sunk by the opponent player.

***For** Board class: Board<unsigned, unsigned>*

- The two unsigned parameters represent the board dimensions, like in Point and Ship class.

Data member variables:

- board[unsigned][unsigned] - A 2D grid of type Point*<unsigned, unsigned>* to represent the internal board, which is updated based on the player's guesses.

- bool hideShips – set to true by default. When the board is displayed, the player's ships will not be hidden, if hideShips is set to false.

Member functions:

- Overload the << operator that displays the status of the respective co-ordinates of the Board object with the rows as numbers and columns as alphabets. (Refer to output_format.txt for more details). Note that the row numbers are right justified with the width set to '3'.
    1. '~' -> default character
    2. '*' –> for the hit positions
    3. 'X' –> for the miss positions
    4. '=' -> for the positions where the ship is placed on the board (hide this, if the hideShips member is true)

- Explicit parameterized constructor: explicit Board (bool hide) – to set the hideShips data member variable and is set to true, by default.

- void setStatus (const Point *<unsigned, unsigned>*, Status) – to set the status of the co-ordinate in the 2D board, based on the arguments passed.

- Point *<unsigned, unsigned>** getShipPoint (const Point*<unsigned, unsigned>* &p, unsigned X, unsigned Y) – returns the Point co-ordinate of the board, where the ship is to be placed and sets the status to SHIP after placement. If the point is already occupied by another ship, throw a "std::invalid_argument" exception. The X and Y parameters represent the offset with respect to the Point p. X and Y should be given as input, according to the direction in which the ship is to be placed on the board.

Baseline version of Battleship game:

- Number of rows and columns = 10.
- Number of ships = 5.
- Names and length of ships = {destroyer, 2} {cruiser, 3} {submarine, 3} {battleship, 4} {carrier, 5}.
- Two players: human and cpu.
- A basic version of the game is provided in BattleShip.cpp and BattleShip.h files.

***For** Battleship source and header files:*

- It is already implemented for you. You are expected to appropriately comment the files and add them back. This would be manually graded.

For extra credits (Up to 25%):

- Any features added should be described briefly and the game should be run in new files: BattleShipV2.cpp and BattleShipV2.h (will be used for manual grading).
- Extend the number of rows and columns, ship, length of the ships, with respect to difficulty level set by the user. If the number of columns is more than 26, you could extend to AA, AB, AC etc.
- Implement a multi-player game with human players, dumb AI, intelligent AI etc.
- Implement an intelligent AI to win the game faster, by changing the ship placement and the guessed co-ordinates.
- Most importantly, feel free to add features that would improve the experience of the game.
- Any extensions should not change the basic functionality of Point, Ship, or Board classes.

Makefile

Your submission must include a *Makefile*. The minimal requirements are:

- Compile using the BattleShip executable with the *make* command.
- Use appropriate variables for the compiler and compiler options.

- Have a rule to build the object file, with appropriate dependencies, for each module (header and source pairs).
- It must contain a clean rule.
- The executable must be named something meaningful (not *a.out*).

README.md

Your submission must include a README.md file (see https://guides.github.com/features/mastering-markdown/). The file should give a brief description of the program, the organization of the code, how to compile the code, and how to run the program. Using markdown, make a section for each item required in your README.md.

Submission

You will upload all the required files to Gradescope. Reminder to complete the Assignment 7 Canvas quiz. There are no limits on the number of submissions. See the syllabus for the details about late submissions.

Grading

For this assignment, half of the marks will come from the auto grader. For this assignment, none of the test details have been hidden.

The other half of the marks will come from human grading of the submission. Your files will be checked for style and to ensure that they meet the requirements described above. In addition, all submitted files should contain a header (comments or otherwise) that, at a minimum, give a brief description of the file, your name and wisc id.

HAPPY CODING! HAVE FUN!