Homework 4:
Boarder Song

Due date: March 13 at the start of lecture

Assignment

In this lab you will design and implement a simple C++ class for representing a position on a rectangular game board. You will use this class in your Project 2 implementation next week.

Your Homework 3 and Project 1 both use integers to represent the position of a player's move, reflecting the reality that we use 2D arrays to represent board states... but a *position on a board* is a **type of data** which should really be modeled as a class `BoardPosition`. We can then write methods like `ApplyMove` which take `BoardPosition` objects representing where the user wants to go, rather than simple `int` values (which don't carry the same semantic context as a name like `BoardPosition`). Using primitive data types in the place of proper objects is something we call a "bad code smell" in software engineering – this particular "smell" is called **primitive obsession**.

The `BoardPosition` class will be **immutable**: once we make a `BoardPosition` object, it cannot be changed. You will write several operators and member methods that will make manipulating these objects easier than would be possible in a language like Java.

Design

Write a class `BoardPosition` according to the following specification:

1. Instance variables of type `char` to represent the row and column of the position.
2. A public default constructor, setting row and column to 0.
3. A public constructor taking a row and column as parameters, and initializing the instance variables to those values.
4. Inline, const accessors/getters for the row and column instance variables, **but no** mutators/setters.
5. These operators:
   (a) `operator std::string() const`, which converts the position into a string of the format `(r, c)`. Hint: `ostringstream`, or `std::to_string`.
   (b) Global `std::ostream& operator<<(std::ostream &lhs, BoardPosition rhs)`: prints the string form of `rhs` to the `lhs` stream. (**Pop quiz**: why is this parameter not accepted as a const reference?)
   (c) Friend `std::istream& operator>>(std::istream &lhs, BoardPosition& rhs)`: assume the user has entered a string of the form `(r, c)`; extract the row and column from that string and set the row and column instance variables of `rhs` accordingly. Be sure to "read" the `)` character from the `istream` (don't be rude and leave it behind).4
   (d) `bool operator==(BoardPosition rhs)`: two positions are equal if they have the same row and column.
   (e) `bool operator<(BoardPosition rhs)`: position `a` is less than position `b` if and only if `a`'s row is less than `b`'s row, or if their rows are equal, then `a`'s column is less than `b`'s column.
6. These member methods:
   (a) `bool InBounds(int boardSize)`: returns true if this position's row and column are in the bounds of a board of the given size. Assume the board's valid coordinates go from 0 up to `boardSize-1`.
   (b) `bool InBounds(int rows, int columns)`: same as above, except don't assume the board is square.

(c) `static std::vector<BoardPosition> GetRectangularPositions(int rows, int columns)`: this method generates and returns a vector of `BoardPositions`, one object in the vector for each position on a board of size `rows` x `columns`, in **row-major order**. For example, if `rows` is 3 and `columns` is 2, this method returns a vector containing the positions `(0, 0) (0, 1) (1, 0) (1, 1) (2, 0) (2, 1)` **in that order**. (Note that this is also in increasing order, according to the rules for `operator<` written above.)

BoardDirection Class

Next, create a class to represent a 1-square movement in some direction on a game board, called `BoardDirection`. A direction consists of a "row change" and "column change", indicating the amount to change a row and column position by in order to move 1 square in the given direction. Create:

1. Instance variables of type `char` to represent the row direction and column changes.
2. A public default constructor, setting both changes to 0.
3. A public constructor taking a row change and column change as parameters, and initializing the instance variables to those values.
4. Inline, const accessors/getters for the row change and column change instance variables, **but no** mutators/setters.
5. A public, static variable `CARDINAL_DIRECTIONS`, which is an `std::array<BoardDirection, 8>` containing 8 direction objects for the 8 "cardinal directions". (Up-left, up, up-right, ...)

Now go back to your `BoardPosition` class and add one final operator:

1. `BoardPosition operator+(BoardDirection dir)`: returns the position arrived at after moving a single square in the given direction.

Testing Your Code

You should write a `main` function to test your code, rather than assume it's correct from the moment you type it. Create variables of your two types and call the operators and methods you have written. Use your `<<` operator to check your work.

Deliverables

Hand in:

1. A printed copy of your code, **printed from Visual Studio or your IDE.** Print **all .h and .cpp files**.
2. **Note**: your code must declare exactly the methods described in this spec, and **nothing more**.