Wayne Wood
CSCI 3010
Instructor Sreesha Nath

Player.h

```
struct Position {
 int row;
 int col;

 bool operator==(const Position &other) {
 return row == other.row && col == other.col;   }
};


class Player {
public:

 Player(const std::string name, const bool is_human);

 std::string get_name() const {return name_; }
 int get_points() const {return points_; }
 Position get_position() const {return pos_; }
 bool is_human() const {return is_human_; }

 void ChangePoints(const int x);

 void SetPosition(Position pos);

 std::string ToRelativePosition(Position other);

std::string Stringify();

private:
 std::string name_;
 int points_;
 Position pos_;
 bool is_human_;

}; // class Player
```

Maze.h

```
enum class SquareType { Wall, Exit, Empty, Human, Enemy, Treasure };

std::string SquareTypeStringify(SquareType sq);
```

```
class Board {
public:
 Board();

 int get_rows() const {return 4; }
 int get_cols() const {return 4; }

 SquareType get_square_value(Position pos) const;   void

SetSquareValue(Position pos, SquareType value);

std::vector<Position> GetMoves(Player *p);

 bool MovePlayer(Player *p, Position pos);

 SquareType GetExitOccupant();


 friend std::ostream& operator<<(std::ostream& os, const Board  &b);

private:
 SquareType arr_[4][4];

 int rows_;
 int cols_;

}; // class Board

class Maze {
public:
 Maze(); // constructor

 void NewGame(Player *human, const int enemies);


void TakeTurn(Player *p);

 Player * GetNextPlayer();

 bool IsGameOver();

 std::string GenerateReport();
 friend std::ostream& operator<<(std::ostream& os, const Maze  &m);

private:
 Board *board_;
 std::vector<Player *> players_;
 int turn_count_;

}; // class Maze
```

1) Annotating Player.h and Maze.h:

a) Draw a square around the constructors for the Player, Board, and Maze objects.

I highlighted them yellow! (apologies, pdf editing in google docs)

b) Draw a circle around the fields (class attributes) for the Player, Board, and Maze objects.

I highlighted them blue!

c) Underline any methods that you think should not be public. (Briefly) Explain why you think that they should not be public.

I didn't underline any because I don't believe that methods should be private unless they are strictly only called within the class, such as a helper. Not sure if any of these qualify.

2) Critiquing the design of the "maze" game:

a) Methods: should do 1 thing and do it well. They should avoid long parameter lists and lots of boolean flags. Which, if any, methods does your group think are not designed well? Is there a method that you think is a good example of being well-designed? which?

I don't believe there are any bad examples in here, they all use limited parameters and have clearly defined granulated functionalities. For example, GetMoves(Player* p) has one argument, a clearly defined method name, and gives us a good idea of the one functionality it could be used for.

b) Fields: should be part of the inherent internal state of the object. Their values should be meaningful throughout the object's life, and their state should persist longer than any one method. Which, if any, fields does your group think should not be fields? Why not? What is an example of a field that definitely should be a field? why?

I think they are all designed well. For my homework, I didn't use the turn_count I don't think, I just went until the game was over and iterated over the players. However, it can be done this way as well.

c) Fill in the following table. Briefly justify whether or not you think that a class fulfills the given trait.

Trait	Player	Board	Maze
cohesive (one single abstraction)	Yes, it defines a player wholly, i.e. the behavior reflects the object's identity.	Yes	Yes
complete (provides a complete interface)	Yes, has all the methods required to function completely.	Yes	Yes
clear (the interface makes sense)	Yes, the method/attribute names are clear in their intention and there isn't more than one functionality for a single function.	Yes	Yes
convenient (makes things simpler in the long run)	Yes, there is the ability for multi-use functions.	Yes	Yes
consistent (names, parameters, ordering, behavior should be consistent)	Naming conventions are followed, behavior reflects the class object's identity, etc.	Yes	Yes