CS 202 - S22 - Assignment 3

Mr. Piotrowski

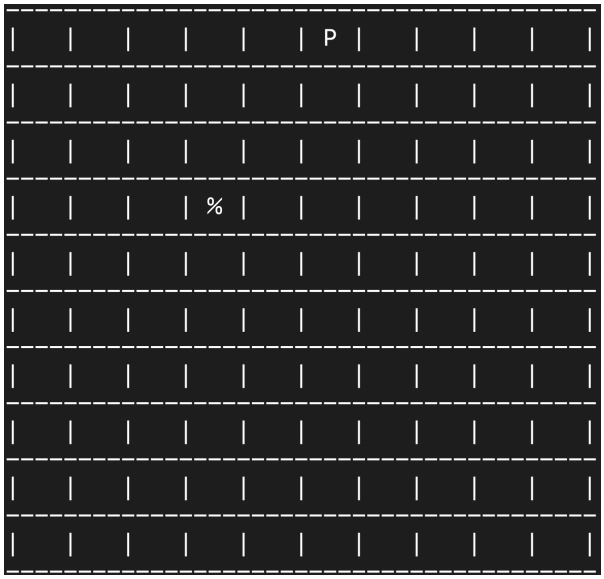
Inheritance

1 Assignment Introduction

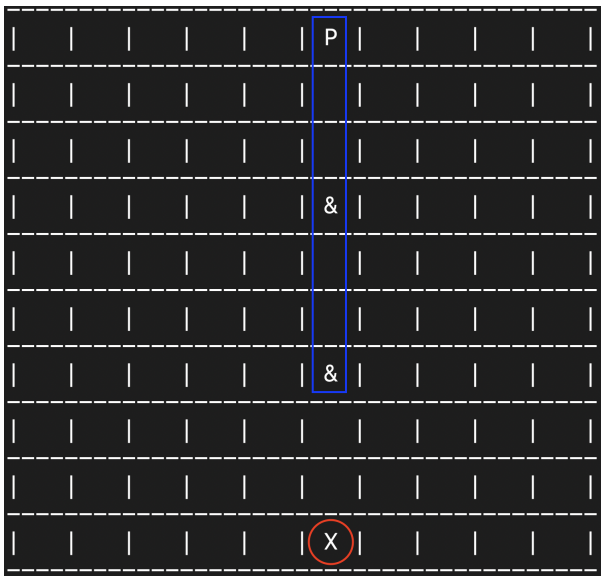
Welcome to assignment 3! You are a programmer for an up-and-coming game titled "Battle Chess!" and you are responsible for setting up the classes for the

program.

This is how Battle Chess works. There is a 10x10 grid on the terminal screen - this is the chess board. See below:



The grid will be filled with 2 to 4 chess pieces. The player controls one of the pieces and the remaining pieces are computer controlled. The player is always placed at the bottom of the chess board when the game starts (*in red*) and the remaining pieces are above the player (*in blue*).



The object of the game is for the player to knock out all other pieces. This is a tricky task as each piece gets their turn to move. Each piece has their own moving pattern as well. These patterns are described later in the assignment.

Note: It is possible that the pieces chosen for the game will never connect on the same tile as their movement pattern may prevent it. For this reason, at least one **pawn** is always guaranteed as an opponent because pawns can traverse every tile.

2 Notes

1. Unfortunately, you will not be coding the whole game. The focus of this assignment is to understand inheritance, so you will be only coding within the classes.

2. This program makes decisions without user input. Instead of using AI, this program makes decisions based on random numbers. This means the computer controlled pieces will not make intelligent decisions, rather, they just move randomly.

   In order to get a random number, a function in <iostream> called **rand()** is used. This function returns a random integer. Purely random integers generated by a computer are not possible. An unsigned integer called a **seed** is required. This seed is the same thing as a Minecraft seed which you may be familiar with. The video game Minecraft randomly generates worlds from a seed. Seeds can be saved and then re-used to generate the exact same world.

   In order to match output on codegrade, this program will prompt the player for a seed. This seed ensures consistent output for grading. When you are testing your program, feel free to enter any seed. Consider using the same seed used in the sample outputs. As long as you test code on Sally/Bobby, the output should match 100%.

3 Program Flow

For a comparison, please see the sample output.

1. The program prompts the player for a seed. This player can really enter any integer they want.

```
b346-15:S22 - A3 piotrj1$ ./a.out
Welcome to Battle Chess (TM)!
Please enter the seed for your game :
3
```

2. Next, the program asks the player if they would like to play tutorial mode. If "Y" is entered, all of the opponents will be pawns. If "N" is entered, the opponents are selected randomly (*aside from the one guaranteed pawn*).

```
Tutorial Mode? (Y/N) :
N
```

3. After that, the program prompts the player for the difficulty level. The number entered controls the number of opponents.

```
Please enter the difficulty level (1-3) :
2
```

4. Now the program asks the user to pick their chess piece.

```
Pick your chess piece!

||      The Barracks       ||

|| (0) pawn           [P] ||
|| (1) forwardKnight  [X] ||
|| (2) leftRook       [#] ||
|| (3) rightRook      [%] ||
|| (4) leftBishop     [&] ||
|| (5) rightBishop    [Q] ||

4
```

5. Now the game can begin. All pieces are placed on the board like so:

```
Piece 101 added to the board!
Player is piece 102
**************************************
||            GAME START              ||
**************************************


 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   | P |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   | # |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   | & |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
```

After the board is set, each piece starts taking turns. See below for what the player will see before they make their first move. In this case, the 2 computer pieces all make their moves.

```
100 is moving.
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   | P |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   | # |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
|   |   |   |   |   |   |   |   |   |   |
 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
```

```
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   | & |   |   |   |   |
 ---------------------------------------------
101 is moving.
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   | P |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   | # |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   | & |   |   |   |   |
 ---------------------------------------------
```

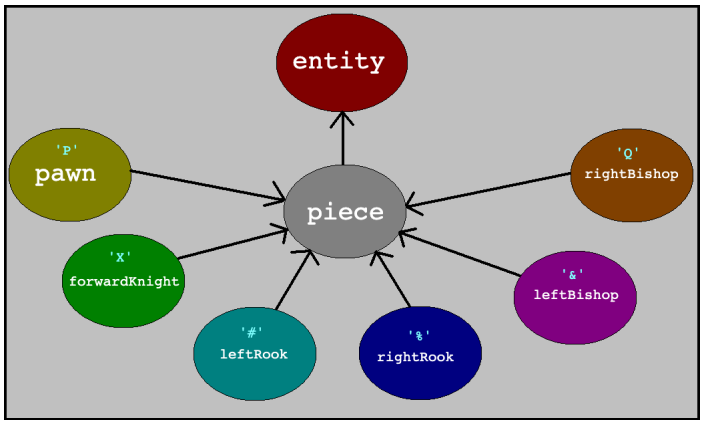
Now the player may make their move by selecting (U)p, (L)eft, (R)ight, or (D)own. The player will only be allowed to make valid moves.

```
Player is moving.
What direction would you like to move?
  -> (L)
  -> (R)
  -> (U)
  -> (D)
 : D
Invalid entry.
 : D
Invalid entry.
 : U
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   | P |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   | # |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   | & |   |   |   |   |   |   |   |
 ---------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
 ---------------------------------------------
```

6. The game continues until the player gets knocked out or all the computers get knocked out.

4 The Inheritance

This program works well for inheritance. All chess pieces share similar attributes. These shared attributes are stored in a base class. The pieces differ in their movement patterns, thus, the derived classes will control these attributes. See the below inheritance diagram.



4.1 entity

The board is comprised of entities. An empty 10x10 board will have 100 "empty" entities, which are just objects of the base class. Because all pieces will inherit from the entity class, we can easily place a piece on the board in this way:

```
// The board
entity grid[10][10];
// Place a pawn on the grid at y = 3, x = 2
pawn p;
grid[3][2] = p;
```

This works because all pawns are also entities. Entity objects have coordinates in the grid, a shape, and their own personal ID.

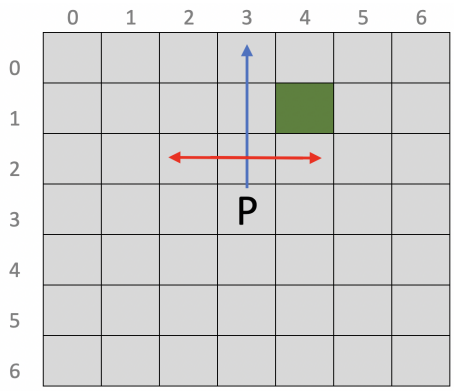
4.2 piece

The piece class is a derived class of the entity class. It is also the base class for all the distinct pieces. All pieces follow a moving pattern. This pattern can be described with 2 variables:

1. Direct Distance
2. Perpendicular Distance

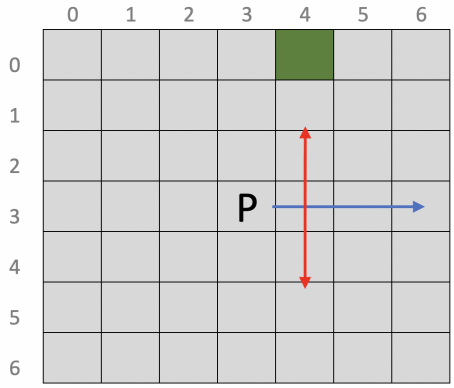
The Direct Distance is the number of spaces moved in the direction specified. The Perpendicular Distance is the number of spaces moved in the direction that is perpendicular to the direct distance.

For example, let's suppose that a piece (*the* **P** *pictured below*) moves up and lands on the green square. The direct distance is pictured as the blue arrow (*the direction specified)* and the perpendicular distance is pictured as the red arrow.



The Direct Distance is **2** here as the piece moves 2 spaces up. The Perpendicular Distance is **+1** as the piece moves **+1** in the perpendicular direction.

Let's assume a different piece moves right.



The Direct Distance is **1** here as the piece moves 1 space right. The Perpendicular Distance is **-3** as the piece moves negatively in the perpendicular direction.

The Direct Distance is always positive so it is unsigned. The perpendicular distance could be negative or positive, so it is signed. Please reference the table below to move your pieces properly using these two variables:

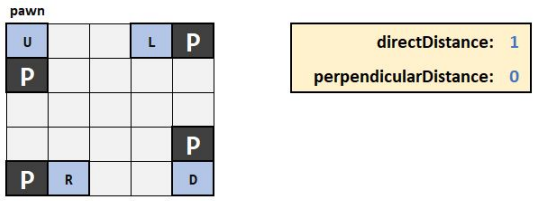
Direction	i (*y coord*)	j (*x coord*)
UP	**subtract** direct_distance	**add** perpendicular_distance
RIGHT	**add** perpendicular_distance	**add** direct_distance
DOWN	**add** direct_distance	**subtract** perpendicular distance
LEFT	**subtract** perpendicular_distance	**subtract** direct_distance

4.3 All Classes Derived from Piece

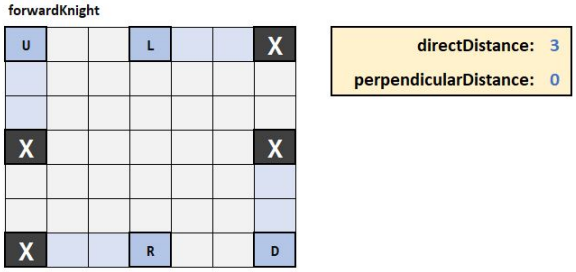
Once you get down to these classes, there is not much more work to be done. All the pieces have their own unique shape and movement pattern. It will be as simple as setting the constructors to initialize each piece with the proper values. Below you will see each piece and its movement pattern.

The dark gray square in the grid shows the specific piece. The darker blue square shows where that piece will end up if they move in that direction.

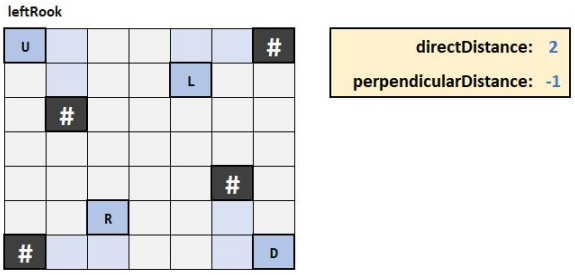
4.3.1 pawn



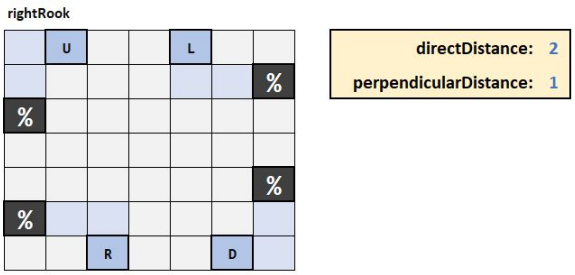
4.3.2 forwardKnight



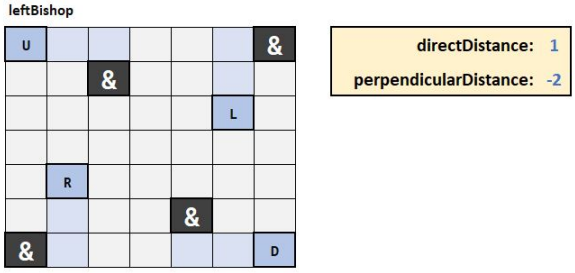
4.3.3 leftRook



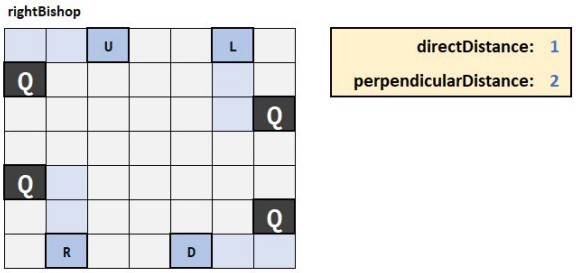
4.3.4 rightRook



4.3.5 leftBishop



4.3.6 rightBishop



5 Coding The Assignment

There are 3 files:

1. main.cpp
2. battleChess.h (**your code here**)
3. battleChess.cpp (**your code here**)

Before writing any code for function definitions, try just adding all the class declarations to the header files by referencing the UML diagrams. Then add

the skeletons of each function in the cpp file. This will get you to a point where the code compiles. After this, start coding the function definitions.

5.1 Entity Class

I provided the class to you with function definition skeletons in the cpp file.

base class **entity**

MAS	Member	const?	static?	Type	Name	Params
private	variable	N	N	char	shape	
protected	variable	N	N	unsigned int	id	
protected	variable	N	Y	unsigned int	id_counter	
protected	variable	N	N	unsigned int	i_coord	
protected	variable	N	N	unsigned int	j_coord	
protected	variable	N	Y	unsigned int	number_of_entities	
protected	variable	Y	Y	unsigned int	grid_height	
protected	variable	Y	Y	unsigned int	grid_width	
protected	variable	N	Y	entity[10][10]	grid	
protected	constructor	N/A	N/A	N/A	entity	
protected	constructor	N/A	N/A	N/A	entity	unsigned int, unsigned int, char
protected	constructor	N/A	N/A	N/A	entity	const entity&
protected	function	N	N	void	move	unsigned int, unsigned int
public	destructor	N/A	N/A	N/A	~entity	
public	function	N	Y	bool	inBounds	unsigned int, unsigned int
public	function	N	Y	unsigned int	getWidth	
public	function	N	Y	unsigned int	getHeight	
public	function	Y	N	char	getShape	
public	function	Y	N	unsigned int	getId	
public	function	N	Y	unsigned int	getId	unsigned int, unsigned int
public	function	Y	N	bool	isEmptyEntity	
public	function	N	Y	bool	isEmptyEntity	unsigned int, unsigned int
public	function	N	Y	void	printGrid	
public	function	N	Y	void	placeEmptyEntity	unsigned int, unsigned int
public	function	N	Y	unsigned int	getNumEntities	

- **char shape -** This char is what the entity looks like on the chess board. An empty entity has the shape of a space ' '. Something that is not empty, like a pawn, will have a 'P'.
- **unsigned int id -** The unique identifier for an entity. Empty entities will not have a unique id here because these objects are considered to be nothing.
- **static unsigned int id_counter -** This static variable is used to initialize an entity's object id. For example, suppose this static variable started out at 100. The first piece to be created would have id 100. This id_counter would then be incremented so that the next object to be declared would have id 101. Your job is to **initialize this variable to 100**.
- **unsigned int i_coord -** This variable represents the y-coordinate of the entity in the chess board. i coordinates are always used as the first index. (*grid[**i**][j]*)
- **unsigned int j_coord -** This variable represents the x-coordinate of the entity in the chess board. j coordinates are always used as the second index. (*grid[i][**j**]*)

- **static unsigned int number_of_entities -** This static variable counts the number of entities existing in the program. A 10x10 grid will always have 100 entities (*including the empty entities*), thus, this variable would be 100. Your job is to **initialize this variable to 0.**
- **static const unsigned int grid_height -** The height of the chess board.
- **static const unsigned int grid_width -** The width of the chess board.
- **static entity[10][10] grid -** This is the chess board. Notice the chess board is within the entity class and it is protected. This means that main will not be able to directly manipulate the chess board. **Initialize this grid.** All you have to do is declare it in the cpp file. The default constructor for an entity will be called in each grid space automatically.
- **static bool inBounds(unsigned int, unsigned int) -** This function is used in the program to determine if an i,j coordinate is within bounds of the grid. This function should return true if both i and j parameters are within bounds, false otherwise.
- **entity() -** The default constructor for an entity. Whenever this constructor is called, the entity is guaranteed to be an empty entity. Initialize the shape to be ' ' (*space*). Next, compute the i_coord and j_coord. At the beginning of the program the grid will be initialized with 100 default entities. These entities should place themselves in the grid. A default entity can compute it's intended i_coord by using this equation:

```
i_coord = number_of_entities % grid_width;
```

  The intended j_coord is this equation:

```
j_coord = number_of_entities / grid_width;
```

  Once that is done, place the entity in the grid. Recall that the *this* keyword is the address to the current object, so we can place it in the grid by dereferencing it:

```
grid[i_coord][j_coord] = (*this);
```

  Lastly, increment the number of entities by 1 and initialize the entity id to 99,999,999. This large number is used because empty entities are not intended to have ids.
- **entity(unsigned int, unsigned int, char) -** This parameter constructor will be called by derived class objects (*piece*). The first parameter is used to intialize the i_coord, the second is for the j_coord, and the char is to initialize the shape. Next, place the entity in the grid as it was done in the default constructor. Next, increase the number of entities. The id does not need to be initialized here because it will be done in the derived class constructor.

- **entity(const entity &) -** The copy constructor for the entity class. Copy all the instance members from the parameter (*including the id*) and then place the entity in the grid. Increase the number of entities by 1.
- **~entity() -** The destructor. The only thing it needs to do is decrease the number of entities by 1.
- **void move(unsigned int, unsigned int) -** This function will move an entity to a new location. The coordinates are passed as parameters. You can move the object by placing it in the grid as we did in the constructor. You will also want to update the objects i_coord and j_coord to reflect the object's new position.
- **static unsigned int getWidth() -** Returns the width of the grid.
- **static unsigned int getHeight() -** Returns the height of the grid.
- **char getShape() const -** Returns the shape of the entity.
- **unsigned int getId() const -** Returns the id of the entity.
- **static unsigned int getId(unsigned int, unsigned int) -** Returns the id of the entity in the grid that is located at the coordinates passed as parameters.
- **bool isEmptyEntity() const -** Checks to see if the entity is empty (*has a space as it's shape*). If it is an empty entity, return true. False otherwise.
- **static bool isEmptyEntity(unsigned int, unsigned int) -** Checks to see if the entity located at the coordinates passed in as parameters is an empty entity.
- **static void printGrid() -** Prints the chess board in a pretty format. I coded this one for you.
- **static void placeEmptyEntity(unsigned int, unsigned int) -** This function will place an empty entity in the grid located at the coordinates specified as parameters. The new entity's members should also be updated to reflect the location in the grid.
- **static unsigned int getNumEntities() -** Returns the number of entities.

5.2 Piece Class

You will have to code 100% of this class.

derived class **piece** public inheritance of base class **entity**

MAS	Member	const?	static?	Type	Name	Params
private	variable	N	N	unsigned int	direct_distance	
private	variable	N	N	int	perpendicular_distance	
private	variable	N	Y	unsigned int	number_of_pieces	
private	function	Y	N	char	getRandomDirection	
protected	constructor	N/A	N/A	N/A	piece	unsigned int, int, unsigned int, unsigned int, char
protected	function	Y	N	void	computeNextCoordinate	char, unsigned int&, unsigned int&
protected	function	Y	N	char	promptDirection	
public	destructor	N/A	N/A	N/A	~piece	
public	function	N	N	void	move	
public	function	N	N	void	randomMove	
public	function	N	Y	unsigned int	getNumberOfPieces	

- **unsigned int direct_distance -** The direct distance traveled by a chess piece. (*See section 4.2*)
- **int perpendicular_distance -** The perpendicular distance traveled by a chess piece. (*See section 4.2*)
- **static unsigned int number_of_pieces -** This variable keeps track of the total number of pieces in the game. **Initialize this to begin at 0.**
- **piece(unsigned int, int, unsigned int, unsigned int, char) -** The only constructor allowed to be used for piece objects. The first parameter is used to initialize the direct_distance, the second is for the perpendicular_distance, the third is for the i_coord, the fourth is for the j_coord, and the char is for the shape. Here you will want to inline a call to the entity constructor that takes 3 parameters. You will also want to initialize the direct_distance, initialize the perpendicular_distance, increase the number of pieces by 1, initialize the id of the object to be equal to the id counter, and then increment the id counter by 1.
- **~piece() -** The destructor for the piece class. It should decrease the number of pieces by 1.
- **char getRandomDirection() const -** This function will return a random direction that a piece can move in. I coded this for you. Reading this function might help you with the **promptDirection** function.
- **void computeNextCoordinate(char, unsigned int&, unsigned int&) const -** This function needs to compute where the piece will move in a specified direction. The char parameter is the direction the piece will go 'L', 'R', 'U', 'D'. The second parameter is where you should store the new i_coord for the object. The third parameter is where you should store the new j_coord for the object. You can compute a piece's next location by using the direct distance and the perpendicular distance. To code this function, you will want to reference the table from section 4.2.

  Please note that this function should only save the new coordinates if the move is valid. That is, this function should return the original coordinates

of the object if the requested direction results in a move that is not possible. This only happens when the requested move will place the piece out of bounds.

For example:

```
unsigned int i_new, j_new;
computeNextCoordinate('L',i_new,j_new);
cout << "New: " << i_new << " , " << j_new << endl;
cout << "Old: " << i_coord << " , " << j_coord << endl;
```

If the coordinate pairs are different, this means that the piece **could** move left and the i_new and j_new variables contain the coordinates of where it would move. If the coordinate pairs are the same, then left is an invalid move.

- **char promptDirection() const -** This function should prompt the player for the direction they would like to move their piece:

```
What direction would you like to move?
  -> (L)
  -> (R)
  -> (U)
  -> (D)
 :
```

The player should only be allowed to enter **valid moves**. You will have to first make sure that the input is one of the 4 letters 'L', 'U', 'R', 'D'. If it is not, re-prompt and check again. If the user does enter a proper direction, use the function computeNextCoordinate() to see if the piece could move in the requested direction. When the function returns, you should see coordinates that are different than the piece's current position. If this is the case, you can return the char that the player entered. If the coordinates are the same as they were, reprompt the user to enter a different direction.

```
 : bug
Invalid entry.
 : g
Invalid entry.
 : D
Invalid entry.
 : U
```

- **void move() -** This function will move the piece. First, get the direction the user wants to move by calling promptDirection. Next, computeNextCoordinate() will get you the new location for the piece. After that, call entity::move() to actually move the piece to the new location. Finally, call placeEmptyEntity to overwrite the old location of the piece. If you don't do this part, you will duplicate the players piece.

void randomMove() - This function will move a piece randomly. It works exactly the same way the function above does, except it calls getRandomDirection() instead of promptDirection(). I coded this one for you.

static unsigned int getNumberOfPieces() - Returns the number of pieces.

5.3 Derived from Piece Classes

5.3.1 pawn class

derived class **pawn** public inheritance of base class **piece**

MAS	Member	const?	static?	Type	Name	Params
public	constructor	N/A	N/A	N/A	pawn	
public	constructor	N/A	N/A	N/A	pawn	unsigned int, unsigned int

- **pawn() -** The default constructor. This should inline the piece constructor. All pawns have a direct distance of 1 and a perpendicular distance of 0. The default constructor should also place the pawn at i = 0, j = 0. Lastly, the shape of the pawn is 'P'.
- **pawn(unsigned int, unsigned int) -** The parameter constructor. This should inline the piece constructor. All pawns have a direct distance of 1 and a perpendicular distance of 0. This constructor should also place the pawn at the location specified by the parameters. Lastly, the shape of the pawn is 'P'.

5.3.2 fowardKnight class

derived class **forwardKnight** public inheritance of base class **piece**

MAS	Member	const?	static?	Type	Name	Params
public	constructor	N/A	N/A	N/A	forwardKnight	
public	constructor	N/A	N/A	N/A	forwardKnight	unsigned int, unsigned int

- **fowardKnight() -** The default constructor. This should inline the piece constructor. All fowardKnights have a direct distance of 3 and a perpendicular distance of 0. The default constructor should also place the fowardKnight at i = 0, j = 0. Lastly, the shape of the fowardKnight is 'X'.
- **fowardKnight(unsigned int, unsigned int) -** The parameter constructor. This should inline the piece constructor. All fowardKnights have a direct distance of 3 and a perpendicular distance of 0. This constructor should also place the fowardKnight at the location specified by the parameters. Lastly, the shape of the fowardKnight is 'X'.

5.3.3 leftRook class

derived class **leftRook** public inheritance of base class **piece**

MAS	Member	const?	static?	Type	Name	Params
public	constructor	N/A	N/A	N/A	leftRook	
public	constructor	N/A	N/A	N/A	leftRook	unsigned int, unsigned int

- **leftRook() -** The default constructor. This should inline the piece constructor. All leftRooks have a direct distance of 2 and a perpendicular distance of -1. The default constructor should also place the leftRook at i = 0, j = 0. Lastly, the shape of the leftRook is '#'.
- **leftRook(unsigned int, unsigned int) -** The parameter constructor. This should inline the piece constructor. All leftRooks have a direct distance of 2 and a perpendicular distance of -1. This constructor should also place the leftRook at the location specified by the parameters. Lastly, the shape of the leftRook is '#'.

5.3.4 rightRook class

derived class **rightRook** public inheritance of base class **piece**

MAS	Member	const?	static?	Type	Name	Params
public	constructor	N/A	N/A	N/A	rightRook	
public	constructor	N/A	N/A	N/A	rightRook	unsigned int, unsigned int

- **rightRook() -** The default constructor. This should inline the piece constructor. All rightRooks have a direct distance of 2 and a perpendicular distance of 1. The default constructor should also place the rightRook at i = 0, j = 0. Lastly, the shape of the rightRook is '%'.
- **rightRook(unsigned int, unsigned int) -** The parameter constructor. This should inline the piece constructor. All rightRooks have a direct distance of 2 and a perpendicular distance of 1. This constructor should also place the rightRook at the location specified by the parameters. Lastly, the shape of the rightRook is '%'.

5.3.5 leftBishop class

derived class **leftBishop** public inheritance of base class **piece**

MAS	Member	const?	static?	Type	Name	Params
public	constructor	N/A	N/A	N/A	leftBishop	
public	constructor	N/A	N/A	N/A	leftBishop	unsigned int, unsigned int

- **leftBishop() -** The default constructor. This should inline the piece constructor. All leftBishops have a direct distance of 1 and a perpendicular distance of -2. The default constructor should also place the leftBishop at i = 0, j = 0. Lastly, the shape of the leftBishop is '&'.

- **leftBishop(unsigned int, unsigned int) -** The parameter constructor. This should inline the piece constructor. All leftBishops have a direct distance of 1 and a perpendicular distance of -2. This constructor should also place the leftBishop at the location specified by the parameters. Lastly, the shape of the leftBishop is '&'.

5.3.6 rightBishop class

derived class **rightBishop** public inheritance of base class **piece**

MAS	Member	const?	static?	Type	Name	Params
public	constructor	N/A	N/A	N/A	rightBishop	
public	constructor	N/A	N/A	N/A	rightBishop	unsigned int, unsigned int

- **rightBishop() -** The default constructor. This should inline the piece constructor. All rightBishops have a direct distance of 1 and a perpendicular distance of 2. The default constructor should also place the rightBishop at i = 0, j = 0. Lastly, the shape of the rightBishop is 'Q'.
- **rightBishop(unsigned int, unsigned int) -** The parameter constructor. This should inline the piece constructor. All rightBishops have a direct distance of 1 and a perpendicular distance of 2. This constructor should also place the rightBishop at the location specified by the parameters. Lastly, the shape of the rightBishop is 'Q'.

6 Compiling and Submitting

Compile your code as:

```
g++ -std=c++11 main.cpp battleChess.cpp
```

Run your code using:

```
./a.out
```

Submit your battleChess.cpp file and battleChess.h file to codegrade.

7 Comprehension Questions

1. Referencing the code you wrote for this assignment, suppose this code exists in main:

```
1    int main(){
2        pawn p1;
3        return 0;
4    }
```

   (a) For the pawn declaration on line 2, list every constructor that is called in order.

(b) When this pawn goes out of scope, list every destructor that is called in order.

2. Referencing the code you wrote for this assignment, explain why code snippet 1 would compile and why code snippet 2 would not. Assume the code written below exists somewhere within the forwardKnight class.

Code Snippet 1

```
1    forwardKnight f;
2    entity e = f;
```

Code Snippet 2

```
1    entity e;
2    forwardKnight f = e;
```

3. Imagine creating this program **without** using inheritance. List at least 1 thing that would be more difficult to code and why.