CS246—Assignment 5, Group Project (Spring 2015)

Due Date 1: Friday, July 17, 04:55pm
Due Date 2: Tuesday, July 28, 04:55pm

This project is intended to be doable by two people in two weeks. Because the breadth of students' ablilities in this course is quite wide, exactly what constitutes two weeks' worth of work for two students is difficult to nail down. Some groups will finish quickly; others won't finish at all. We will attempt to grade this assignment in a way that addresses both ends of the spectrum. You should be able to pass the assignment with only a modest portion of your program working. Then, if you want a higher mark, it will take more than a proportionally higher effort to achieve, the higher you go. A perfect score will require a complete implementation. If you finish the entire program early, you can add extra features for a few extra marks.

Above all, MAKE SURE YOUR SUBMITTED PROGRAM RUNS. The markers do not have time to examine source code and give partial correctness marks for non-working programs. So, no matter what, even if your program doesn't work properly, make sure it at least does something.

The Game of PawnPusher9000

In this project, you and a partner will work together to produce PawnPusher9000 more commonly known as chess.

The rules of chess are readily available online, so we outline them only briefly here.

Chess is played on an 8x8 checkerboard, arranged so that there is a white square at the bottom right. Players take turns making one move at a time. The player controlling the white pieces makes the first move.

There are six types of pieces:

- **King (K)** Moves one square in any direction.
- **Queen (Q)** Moves in any of the eight possible directions, any distance, but cannot move past any piece that blocks its path.
- **Bishop (B)** Moves in any of the four diagonal directions, any distance, but cannot move past any piece that blocks its path.
- **Rook (R)** Moves in any of the four vertical/horizontal directions, any distance, but cannot move past any piece that blocks its path.
- **Knight (N)** If it sits on square (x, y), it can move to square $(x \pm 2, y \pm 1)$ or $(x \pm 1, y \pm 2)$. Can "jump over" any piece that blocks its path.

- **Pawn (P)** Moves one square forward.

A piece captures another piece by moving onto the square occupied by that piece. The captured piece is then permanently removed from the board. A piece that could capture another piece is said to attack that piece. A piece may only capture a piece of the opposite colour.

The object of the game is to place your opponent's king under attack, such that your opponent's king cannot escape in one move. This is known as *checkmate*. An attack on the king, whether it can escape or not, is known as *check*.

The following additional rules govern the movement of pieces:

- The pawn is the only piece whose standard move is different from its capturing move. A pawn moves only forward, but it captures on the forward diagonals (one square). Thus, on capturing, a pawn must move diagonally forward, one square, to take over a square occupied by another piece.
- A pawn, on its first move, may move either one square forward or two squares forward.
- If a pawn, by moving two squares forward, avoids capture by another pawn (i.e, if moving one square forward would have put it under attack by another pawn), the would-be attacking pawn may still capture it by moving one square diagonally forward to the square the other pawn skipped over. This is known as pawn capture *en passant*. This option is only available immediately following the two-square move by the opposing pawn. If you wait, you can't do it.
- A pawn, upon reaching the other end of the board is replaced by either a rook, knight, bishop, or queen (your choice).
- A move known as *castling* helps to move the king to a safer square while simultaneously mobilizing a rook. To execute it, the king moves two squares towards one of the rooks, and that rook then occupies the square "skipped over" by the king. This happens in one move. For castling to be legal, the king and rook used must not previously have moved in the game; there must be no pieces between the king and rook used; and the king must not be in check on either its starting position, its final position, or the position in between (to be occupied by the rook).
- It is not legal to make any move that puts your king in check.
- If any player ever has no legal moves available, but is not in check, it is known as *stalemate*, and the game is a draw.

Display

You need to provide both a text-based display and a graphical display of your game board. A sample text display follows:

```
8 rnbqkbnr
7 pppppppp
6  _ _ _ _
5 _ _ _ _
4  _ _ _ _
```

```
3 _ _ _ _
2 PPPPPPPP
1 RNBQKBNR

  abcdefgh
```

In this display, capital letters denote white pieces, and lower case letters denote black pieces. Unoccupied squares are denoted by a blank space for white squares, and an underscore character for dark squares. The above board also represents the initial configuration of the game.

After every move, the board must be redisplayed. If one player or the other is in check, additionally display `White is in check.` or `Black is in check.`, as appropriate. If one player has won the game, display `Checkmate! White wins!` or `Checkmate! Black wins!` If the game is stalemated, output `Stalemate!` If the game is resigned, output `White wins!` or `Black wins!`, as appropriate.

Your graphical display should be set up in a similar way. Do your best to make it visually pleasing. It is permitted to represent the pieces as letters in your graphical display.

Players

Your system should accommodate both human and computer players. In particular, human vs. human, human vs. computer, and computer vs. computer should all be possible. When both players are human, the role of your program is to provide the game board, to detect checkmate and stalemate, and to prohibit illegal moves. Computer players should operate at one of several difficulty levels:

- Level 1: random legal moves.
- Level 2: prefers capturing moves and checks over other moves.
- Level 3: prefers avoiding capture, capturing moves, and checks.
- Levels 4+: something more sophisticated.

For levels 4 and above, try to come up with more sophisticated strategies for your player to use. Attempt level 4 (or any higher levels) only when the rest of the project is done.

Question: Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

Question: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

Question: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

Command Interpreter

You interact with the system by issuing text-based commands. The following commands are to be supported:

- `game` *`white-player black-player`* starts a new game. The parameters *`white-player`* and *`black-player`* can be either `human` or `computer[1-4]`.
- `resign` concedes the game to your opponent. This is the only way, outside of winning or drawing the game, to end a game.
- A move consists of the command `move`, followed by the starting and ending coordinates of the piece to be moved. For example: `move e2 e4`. Castling would be specified by the two-square move for the king: `move e1 g1` or `move e1 c1` for white. Pawn promotion would additionally specify the piece type to which the pawn is promoted: `move e7 e8 Q`. In the case of a computer player, the command `move` (without arguments) makes the computer player make a move.
- `setup` enters setup mode, within which you can set up your own initial board configurations. This can only be done when a game is not currently running. Within setup mode, the following language is used:
  - `+ K e1` places the piece `K` (i.e., white king in this case) on the square `e1`. If a piece is already on that square, it is replaced. The board should be redisplayed.
  - `- e1` removes the piece from the square `e1` and then redisplays the board. If there is no piece at that square, take no action.
  - `=` *`colour`* makes it *colour*'s turn to go next.
  - `done` leaves setup mode.

  Upon completion of setup mode, you must verify that the board contains exactly one white king and exactly one black king; that no pawns are on the first or last row of the board; and that neither king is in check. The user cannot leave setup mode until these conditions are satisfied. We recommend doing setup mode early, as it may facilitate testing.

It would be in your best interest (and will help during your demo) to make sure that your program does not break down if a command is misspelled.

The board should be redrawn, both in text and graphically, each time a move command is issued. For the graphic display, redraw as little of the screen as is necessary to make the needed changes.

Command Line Options

Your program should have the ability to process an optional single command line argument, a file that specifies the layout of a pre-saved game. The file will consist of exactly 9 lines with the first 8 lines containing exactly 8 characters representing the 8 positions in a row of the chess board. Each character is either the representation of a chess piece as discussed earlier or an underscore to represent an empty board position. The ninth line will contain exactly one character: W if it is White's turn next or B if it is Black's turn next.

For example, the following file represents a game that was saved as soon as it started i.e. before any player made a move.

```
rnbqkbnr
pppppppp
--------
--------
--------
--------
PPPPPPPP
RNBQKBNR
W
```

Note: The above representation of a saved game is a simplification. In particular, it does not keep track of whether a king or rook had previously moved in the game before it was saved (one of the conditions for castling). You may assume, that if a king or rook happens to be in their correct starting position, then they had not moved in the game before it was saved.

Question: Assuming you have already implemented all the commands specified in the **Command Interpreter**. Is there a way to implement the "Load a pre-saved game" feature, with maximal code reuse. Explain.

Scoring

A win awards one point to the winner and zero points to the loser. A draw awards half a point to each team. When the program ends (ctrl-D is pressed), it should print the final score to the screen. For example:

```
Final Score:
White: 2
Black: 1
```

Grading

Your project will be graded as follows:

Correctness and Completeness	60%	Does it work? Does it implement all of the requirements?
Documentation	20%	Plan of attack; final design document.
Design	20%	UML; good use of separate compilation, good object-oriented practice; is it well-stuctured, or is it one giant function?

Even if your program doesn't work at all, you can still earn a lot of marks through good documentation and design, (in the latter case, there needs to be enough code present to make a reasonable assessment).

If Things Go Wrong

If you run into trouble and find yourself unable to complete the entire assignment, please do your best to submit something that works, even if it doesn't solve the entire assignment. For example:

- can't handle castling, en passant, or pawn promotion
- program only produces text output; no graphics
- only one level of difficulty implemented
- can't detect check or checkmate

You will get a higher mark for fully implementing some of the requirements than for a program that attempts all of the requirements, but doesn't run.

Make sure that your program is at least able to load a pre-saved game and display the game.

A well-documented, well-designed program that has all of the deficiencies listed above, but still runs, can still potentially earn a passing grade.

Plan for the Worst

Even the best programmers have bad days, and the simplest pointer error can take hours to track down. So be sure to have a plan of attack in place that maximizes your chances of always at least having a working program to submit. Prioritize your goals to maximize what you can demonstrate at any given time. We suggest: save the graphics for last, and first do the game in pure text. One of the first things you should probably do is write a routine to draw the game board (probably a `Board` class with an overloaded friend `operator<<`). It can start out blank, and become more sophisticated as you add features. You should also develop the ability to read in pre-saved games earlier on. This will ensure that we are able to test features faster by loading our pre-saved games. You should also do the command interpreter early, so that you can interact with your program. You can then add commands one-by-one, and separately work on supporting the full command syntax. Work on your pieces one at a time. Once you figure out the right interface and implemenation for one piece type, the remaining piece types will be easier to build. Take the time to work on a test suite at the same time as you are writing your project. Although we are not asking you to submit a test suite, having one on hand will speed up the process of verifying your implementation.

You will be asked to submit a plan, with projected completion dates and divided responsibilities, as part of your documentation for Due Date 1.

If Things Go Well

If you complete the entire project, you can implement extra features for a few extra credits. These should be outlined in your design document, and markers will judge the value of your extra features.

Submission Instructions

See **Guidelines.pdf** for instructions about what should be included in your plan of attack and final design document.

Due Dates

Due Date 1 (July 17, 2015): Due on due date 1 is your plan of attack, deadlines.txt and initial UML diagram for your implementation of PawnPusher9000.
Due Date 2 (July 28, 2015): Due on due date 2 is your actual implementation of PawnPusher9000. All .h, .cc and any text files needed for your project to compile and run should be included in pp9k.zip. The zip file must contain a suitable Makefile such that typing make will compile your code and produce an executable named `pp9k`. In addition, upload a5-uml-final and a5-design to the appropriate place on marmoset.

A Note on Random Generation

If you require the random (or rather, pseudo-random) generation of numbers, you have two options available to you. In `<cstdlib>`, there are commands **rand** and **srand**, which generate a random number and seed the random generator respectively (typically, seeded with **time()** from `<ctime>`). Alternatively, you can use the **prng class** from `prng.h`, which provides an encapsulated random number generating algorithm. Either is fine for the project; it is not required to use one over the other.