Department of Computer Science
COS110 - Program Design: Introduction
Assignment 1

1 Introduction

Deadline: 8th October, 23:30

1.1 Objectives and Outcomes

The objective of this assignment is to provide a comprehensive practical exercise that tests the use of classes, objects, constructors, destructors, dynamic memory and the use of operator overloading.

1.2 Submission

All submissions are to be made to the **ff.cs.up.ac.za** page under the COS 110 page, and for the correct assignment slot. Submit your code to Fitchfork before the closing time. Students are **strongly advised** to submit well before the deadline as **no late submissions will be accepted**.

1.3 Plagiarism

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying textual material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to **http://www.ais.up.ac.za/plagiarism/index.htm** (from the main page of the University of Pretoria site, follow the *Library* quick link, and then click the *Plagiarism* link). If you have questions regarding this, please ask one of the lecturers, to avoid any misunderstanding.

1.4 Implementation Guidelines

Follow the specifications of the assignment precisely. For the assignment, you will be required to create your own makefile so pay attention to the names of the files you will be asked to create. If the assignment requires you to submit additional files of your own, follow the file structure and format exactly. Incorrect submissions will use up your uploads and no extensions will be given. In terms of C++, unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the practical, will not be allowed. Some of the appropriate files that you submit will be overwritten during marking to ensure compliance to these requirements. If the specification makes use of text files, for providing input information, be sure to include blank text files with the specified names.

1.5 Classes

The class is one of the foundations of the Object Oriented (OO) paradigm. The purpose of classes is to organise methods and data into logically coherent units that provide can reusable functionality to building complex programs. Strongly associated with classes, are their two most recognisable features of a class that allow for creating and destroying instances of that class.

1.6 Operator Overloading

Like function overloading, operator overloading is the practice of extending the usage of the set of operators defined in C++ in ways that go beyond their normal usage. For example, while it is possible to overload a function by defining multiple versions of the same function with different argument lists, overloading an operator, like the +, would then allow for the usage of the plus in contexts outside of simple mathematical addition. It would be possible to define and use concepts like the addition of two classes in programs, extending the flexibility and ease of use possible within a given domain.

1.7 Scenario

For this assignment you are going to implement a variation of a popular chess puzzle, namely the Checkmate-in-One puzzle. The puzzle takes place on a standard 8 by 8 chess board with a pre-existing setup for both sides and all of the pieces. The puzzle works with a single player getting to make one move in the game. The goal is to checkmate the opponent with this single move and thus end the game.

This variation has a number of important deviations from the standard formulation of the puzzle which are noted here:

1. The opposing player, the player who does not move first, can only move their king.
2. The opposing king, the one being subject to the potential checkmate, cannot take any enemy pieces.
3. No opposing piece can be used to block an attempt at a checkmate.
4. No pieces will be captured during play.
5. The use of castling, en passant capture or promotion of pawns is not allowed.

Given the two sides, white and black, and white being the one trying to checkmate black, the opposing player in this example would be black. If black were the side to attempt to checkmate white, then white would be the opposing player. You are going to implement a program that enables the solving of this kind of puzzle.

1.8 Mark Distribution

Activity	**Mark**
(Task 1) Piece Class	15
(Task 2) Board Class	60
(Task 3) Solver Class	30
Total	**105**

2 Assignment

The assignment is broken up into 3 separate tasks that follow on from each other. In each task, you will be required to implement code, and design components that will be used by later tasks to build more complex systems. It is recommended to work sequentially from Task 1 to Task 3.

2.1 Game Rules

Presented here, is an outline of the rules of the game around which this assignment is centred. This is not meant to be a completely exhaustive explanation, as you should defer to the specifics provided in the Task descriptions for more detail, but rather this is a primer for the general details of chess, the movement mechanics and the checkmate conditions that end the game.

2.1.1 Game Board

Chess is played on a standardised 8 by 8 board. In real life, chess boards are labelled with a combination of letters and numbers to make the moves easier to interpret by the players and spectators. However for this assignment the chess boards will only referred to by numbers.

For example, consider,

```
 01234567
0--------
1--------
2--------
3--------
4--------
5--------
6--------
7--------
```

This is a board made of 8x8 tiles. Each tile represents the open sea and the board is currently completely empty. The positions of the each tile, in terms of coordinates it reflects is given by the numbers. The top leftmost corner is (0,0) and the bottom rightmost corner is (7,7). If compared to a standard chess board such as in Figure 1

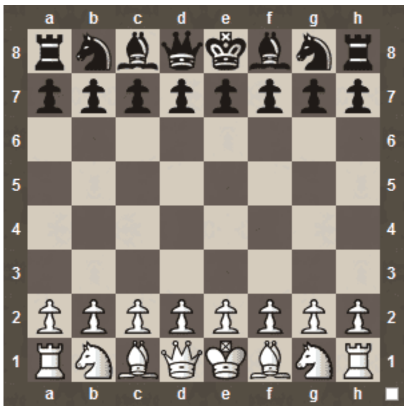

Figure 1: Standard Chess Board

Position (0,0) would correspond to A8 and position (7,7) would correspond to H1. The x coordinate refers to the row and the y coordinate refers to the column.

The use of the − string is used to indicate an empty board space. All spaces not taken up by a piece on the board should be set to this string.

2.1.2 Pieces

The standard game of chess is played with equal pieces between the players. For the purposes of this assignment the pieces are referred to on the board by a combination of their colour and their type. For example the string, "bb" would refer to a black bishop whereas "wp" would refer to a white pawn. The full list of types for the pieces is produced below:

- p: pawn
- b: bishop
- k: knight
- K: king
- q: queen

For example, consider a board with two black pawns, a black king, a white king and two white rooks,

```
wr-------
----wK----
--------
--------
-bpbp----wr
-bK------
--------
--------
```

2.1.3 Move

For this puzzle, only one side has to submit a move. The move has the following format:

```
X1,Y1,X2,Y2
```

The list is delimited by commas and the first two values refers to the x and y coordinates of the piece to be moved and the second set of coordinates refer to its new position. Note that X refers to the row and Y refers to the column in the matrix. This could be interpreted as chessboard[X1][Y1] for instance.

2.1.4 List File

During Task 3 you will be asked to evaluate multiple files during a single execution of your program. The file will consist of lines which consist of a file name followed by an integer priority value. These two values are comma delimited.

2.2 Task 1: Piece Class

The basis of the assignment is the **piece** class. It is comprised of two files: **piece.h** and **piece.cpp**. You will be required to produce both of these files for yourself. A UML diagram for the class is provided to you below:

```
piece
-pieceType:string
-side:char
-xPos:int
-yPos:int
------------------------
+piece()
+piece(newPiece: piece *)
+piece(pType:string, side:char,x:int,y:int)
+∼ piece()
+getSide():char
+getPieceType():string
+getX():int
+getY():int
+setX(x:int):void
+setY(y:int):void
+friend operator<<(output:ostream &, t:const piece &):ostream&
+operator[](pos:int):void
+operator+(move:string):piece&
```

The class variables are as follows:

- pieceType: The string variable indicating what kind of piece it is. The values are as follows:

  - king
  - pawn
  - queen
  - bishop
  - knight
  - rook

- xPos: The X coordinate of the piece on the board.
- yPos: The Y coordinate of the piece on the board.

The class methods have the following behaviour:

- piece: The default and empty constructor for the piece class. It is blank.
- piece(newPiece:piece*): This is a copy constructor for the piece class. It receives an instantiated piece object and copies the values of that object into a newly constructed instance.
- piece(pType:string,side:char,x:int,y:int): This is a value-based constructor for the class. It takes 4 separate arguments which all correspond to values inside the class.
- ∼piece: The class destructor. This deallocates any allocated memory of the class. It also prints out (with a new line at the end), the following message: "(X,Y) side type deleted" where side refers to the side of the ship of the piece, type the piece type and the X,Y coordinates where it was found on the board. An example of this output is:

  ```
  (0,0) b rook deleted
  ```

  The coordinates are printed first, followed by the side and then the piece type.
- getSide: This returns the side char for the piece.
- getPieceType: This returns the type of the piece.
- getX: This returns the X coordinate variable.
- getY: This returns the Y coordinate variable.
- setX: This sets the x coordinate to the passed in value.
- setY: This sets the y coordinate to the passed in value.
- operator[](pos:int): This is an overload of the [] operator. When called, this will receive an int that takes either the values 0 or 1. If any other value is provided print the message: "Invalid Index" with a new line at the end. If the index is valid, it should print out one of two messages with a new line at the end. If the value is 0, it should print out the x coordinate and if 1, the y coordinate. The format of the messages are as follows

```
x coord: 1
y coord: 2
```

The above examples show the output for two different calls of the operator. Only one line of output should be given per call.

- operator+(move:string): This operator will receive a string with the format

  ```
  x,y
  ```

  where x represents an x coordinate and y represents a y coordinate. If the coordinates are valid (that is within the valid range of the chess board) then the appropriate variables should be set with these variables. If the coordinates are invalid, then nothing should happen.

- operator<<: It also prints out (with a new line at the end), a message in the following format:

  ```
  b rook at [0,0]
  ```

  The side is printed first, then the piece type followed by the coordinates at the end. This message must be sent to the output variable.

2.3 Task 2: Board Class

The next most important component is the **board** class. It is comprised of two files: **board.h** and **board.cpp**. You will be required to produce both of these files for yourself. A definition for the class is provided below:

```
board
-numWhitePieces:int
-numBlackPieces:int
-whitePieces: piece **
-blackPieces: piece **
-chessboard: string **
-move: string
-sideToMove:char
-operator++():board&
------------------------
+board(pieceList:string)
+~ board()
+operator--():board&
+checkIfPieceHasCheck(pieceType:string ,xPos:int,yPos:int ,kingX:int,kingY:int):bool
```

The variables are defined as follows:

- numWhitePieces:The number of white pieces left on the board.
- numBlackPieces:The number of black pieces left on the board.

- whitePieces: A dynamic array of piece objects representing all of the white piece objects.
- blackPieces: A dynamic array of piece objects representing all of the black piece objects.
- chessboard: A 2D string array representing the chessboard.
- move: The move being submitted as a solution to the given board puzzle.
- sideToMove: The side which is to make the move to solve the puzzle.

The methods are defined as follows:

- board(pieceList:string): The constructor for the board class. It will receive the game board setup information in the form of a name to the file containing that information. You must read this file to extract the required information to instantiate the class variables with appropriate values as defined by the text file format.

  When the constructor has finished executing, the chessboard 2D array should be populated as defined above with the corresponding pieces in their appropriate locations. Additionally, all of the pieces should have their appropriate objects instantiated in their respective arrays.

- ∼board(): The destructor for the board object. When called all of the dynamic memory that has been assigned should be deleted. The order (from first ot be deleted to last) of the deletion order is as follows: blackPieces, whitePieces, chessboard. Once deleted, you should print a final message with a new line at the end. The message has the following format:

  ```
  Num Pieces Removed: X
  ```

  X in this example refers to the total number of pieces (considering both sides) that were originally placed on the board at the start of the program.

- operator++(): This operator moves the piece located at the first set of coordinates to the board position denoted by the second set of coordinates. You can assume that the coordinates will always be correct for the piece and its movement types and that no move will expose a king to being checked. The change in position for said piece should also be reflected in the respective pieces object in the appropriate pieces array for the side that is moving. The coordinates for this move come from the move variable.

- checkIfPieceHasCheck(pieceType:string ,xPos:int,yPos:int ,kingX:int,kingY:int): This function is used to check if a given piece has a check status on the opposing king. It receives the type of piece being checked for as well as its position (xPos and yPos) and the position of the king to compare against. If the piece has a check then a true is returned otherwise false is returned.

  Note that pawns, queens, knights, bishops and rooks are the only pieces capable of checking a king.

- operator−−: This is the most important operator. It is used to determine if the solution, the move provided, actually solves the Checkmate-in-One puzzle. This operator should, based on the conditions described about the puzzle, determine if the side that made the move induced a checkmate scenario on the opposing player. Remember that the opposing player is only going to be able to move their king, and cannot block checks, or capture pieces that might cause checks.

  If a checkmate is reached the operator should output the following message with new line at the end:

  ```
  Success: Checkmate of w King at [X,Y]
  ```

  where X,Y refers to the original coordinates of the opposing king and where they were first put into check. The side of the king that got checkmated should be before the word King. For reference, w refers to the white side and b refers to the black side.

  If the move failed to produce a checkmate, either because it did not check the enemy king, or the enemy king was able to escape, then the message (with a new line at the end) should be:

  ```
  Failed: No Checkmate of w King
  ```

2.4 Task 3: Solver Class

The final part of the assignment is the **Solver** class. It is comprised of two files: **solver.h** and **solver.cpp**. This class is where the bulk of the processing and complexity of the assignment will be found as implementing it, implements the game logic. You will be required to produce both of these files for yourself. A UML diagram for the class is provided to you below:

```
solver
-numGames:int
-boards: board **
------------------------
+solver(games:string)
+~ solver()
```

The variables are defined as follows:

- numGames: This is the number of boards that are going to be created.
- boards: This is a dynamic array that will store a number of board objects. Each represents a board and potential solution that will have to be evaluated.

The methods have the following behaviour:

- solver(games:string): The class constructor. It will receive the name of the list file which contains the list of all of the boards to be loaded into the boards array, as well as their solving priority. This function should instantiate the boards array based on the appropriate information and then determine the checkmate status of every board. The output for this has the following format:

  ```
  [X] Y
  ```

  X refers to the solving priority associated with a given board and Y refers to the output produced by the determineIfCheckMate function. You should make use of the appropriate board function to output this information. Finally, the boards must be processed in ascending order of solving priority. The order in which they were read into the array should remain unchanged. For example, if there are 3 boards to solve, the output would look something like this:

  ```
  [0] Success: Checkmate of w King at [1,0]
  [4] Failure: No Checkmate of b King
  [8] Failure: No Checkmate of w King
  ```

- ∼solver(): The class destructor. This deallocates any allocated memory of the class. It will delete the boards from index 0 to index X where X refers to the last board index that is stored in the array. After this, print out the following message with a new line at the end:

  ```
  Num Boards Deleted: B
  ```

  where B refers to the number of boards that were deleted.

2.5 File Structure

Provided here are file structures that the program will need. These are structural examples, that show how the files should be composed. You are required to produce your own files for testing purposes.

2.5.1 Input File

```
w
2,4,1,2
b,king,0,0
b,pawn,1,0
w,king,4,5
w,rook,6,1
w,rook,2,7
w,bishop,7,5
w,knight,2,4
```

The first line of the file indicates the side that is going to move. This is either w or b for white and black respectively. The next line is the move that side is going to do. The remaining lines (unspecified size) of the file indicate the the pieces for each side and their positions and types.

2.5.2 List File

```
game1.txt,4
game2.txt,2
game3.txt,8
```

This file is used for task 3. It will contain a list of file names and priorities, separated by a comma. The list will be of unspecified size. Each line associates that board and therefore puzzle with a solution priority represented as an integer value.

2.6 Submission

You will have to submit your code to 3 separate upload slots. You will have a maximum of 5 uploads per slot for this assignment. The slots are very limited so do not rely on Fitchfork to debug your code.

2.6.1 Task 1 Checklist

The following files need to be submitted for Task 1 of the assignment:

- piece.h
- piece.cpp
- main.cpp
- makefile

2.6.2 Task 2 Checklist

The following files need to be submitted for Task 2 of the assignment:

- piece.h
- piece.cpp
- board.h
- board.cpp
- main.cpp
- makefile
- game1.txt

2.6.3 Task 3 Checklist

The following files need to be submitted for Task 3 of the assignment:

- piece.h
- piece.cpp
- board.h
- solver.cpp
- solver.h
- board.cpp
- main.cpp
- makefile
- list.txt
- game1.txt
- game2.txt
- game3.txt
- game4.txt

The text files should be blank. These will be replaced during testing.

The includes for each file are as follows

2.6.4 Task 1 Checklist

The following includes are needed for the piece class:

- string
- iostream
- sstream

2.6.5 Task 2 Checklist

The following includes are needed for the board class:

- fstream
- piece.h

2.6.6 Task 3 Checklist

The following includes are needed for the solver class:

- board.h

Be sure to work incrementally and test as you go to ensure that your code is robust under a number of possible conditions. The very limited number of uploads means that the margins for error are relatively low and therefore caution is advised. Remember that while your code might appear to be working, there could be a number of non-obvious problems under the surface.