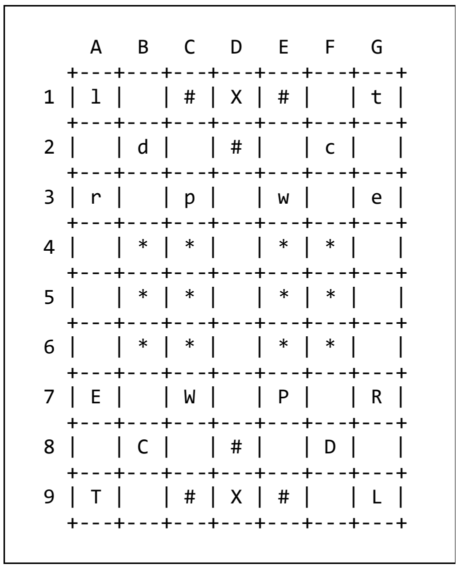
Project: Jungle

Due: 20:00, Sat 11 Dec 2021 Full marks: 100

Introduction

The objective of this project is to let you apply the OOP concepts learned, including inheritance and polymorphism, to programming through developing a board game application – Jungle – in C++.

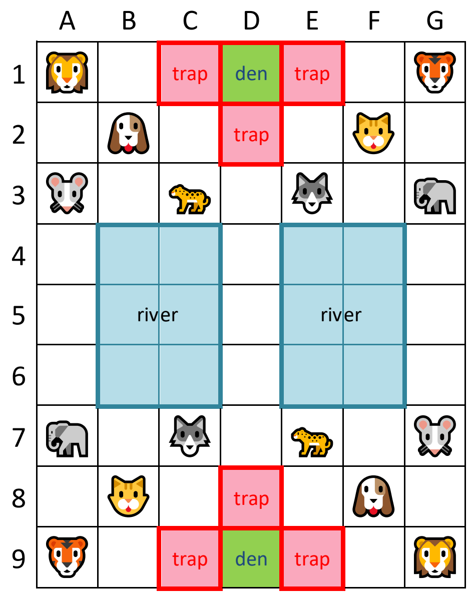
Jungle Chess or Animal Chess or Dou Shou Qi (Chinese: 鬥獸棋) is a traditional Chinese board game. It is a two-player strategy board game played on a board that consists of 7 columns and 9 rows of squares. The initial board configuration is shown in Figure 1. In a text console environment, we use the single-letter labels listed in Table 1 to denote the eight types of chess pieces on the board. To denote a position on the board, we use a coordinate system similar to a spreadsheet program, e.g. "A2" refers to the cell in the first column and the second row. Suppose that a C++ array `cells` is used to represent the board, cell address "A2" will be mapped to `cells[1][0]`.



```
     A   B   C   D   E   F   G
   +---+---+---+---+---+---+---+
 1 | l |   | # | X | # |   | t |
   +---+---+---+---+---+---+---+
 2 |   | d |   | # |   | c |   |
   +---+---+---+---+---+---+---+
 3 | r |   | p |   | w |   | e |
   +---+---+---+---+---+---+---+
 4 |   | * | * |   | * | * |   |
   +---+---+---+---+---+---+---+
 5 |   | * | * |   | * | * |   |
   +---+---+---+---+---+---+---+
 6 |   | * | * |   | * | * |   |
   +---+---+---+---+---+---+---+
 7 | E |   | W |   | P |   | R |
   +---+---+---+---+---+---+---+
 8 |   | C |   | # |   | D |   |
   +---+---+---+---+---+---+---+
 9 | T |   | # | X | # |   | L |
   +---+---+---+---+---+---+---+
```

(a)

⇔



(b)

Figure 1: Initial configuration of the game board

The gameboard represents a jungle terrain with dens, traps, and rivers, highlighted in green, pink, and blue respectively in Figure 1(b) and marked with symbols X, #, and * respectively in Figure 1(a) as in console mode.

Each player has eight game pieces representing different animals, each with a different rank, and in their own color (blue versus red). In console mode, we use lowercase and uppercase letters to denote the blue and red pieces respectively. Stronger-ranked animals can capture ("eat") animals of weaker or equal rank, e.g., Elephant captures Lion, Lion captures Tiger, … Dog captures Cat, Cat captures Rat. But Rat, with the lowest rank, can capture Elephant (many published versions of the game say the rat kills the elephant by "running into its ear and gnawing into its brain").

The animal ranking, from strongest to weakest, is given in Table 1:

Table 1: Chess pieces

Rank	Piece	Face	Chinese	Red Label	Blue Label
8	**E**lephant	🐘	象	E	e
7	**L**ion	🦁	獅	L	l
6	**T**iger	🐯	虎	T	t
5	Leo**p**ard	🐆	豹	P	p
4	**W**olf	🐺	狼	W	w
3	**D**og	🐶	狗	D	d
2	**C**at	🐱	貓	C	c
1	**R**at	🐭	鼠	R	r

Movement Rules

The two players, namely "Blue" and "Red", make moves in turns, with Blue moving first. During their turn, a player must move. Below is a summary of the rules about movement. Rules 4 to 7 are special rules related to the river (or water) squares.

1. All pieces can move one square horizontally or vertically (not diagonally).
2. A piece may not move into its own den.
3. Animals of either side can move into and out of any trap square.
4. Rat is the only piece that may go onto a water (river) square.
5. Lion and Tiger can jump over a river vertically and horizontally.
6. (Rule 5 related) If the target square contains an enemy piece of equal or lower rank, the Lion or Tiger captures it as part of their jump.
7. (Rule 5 related) A jumping move is blocked (not allowed) if a rat of either color currently occupies any of the intervening water squares.

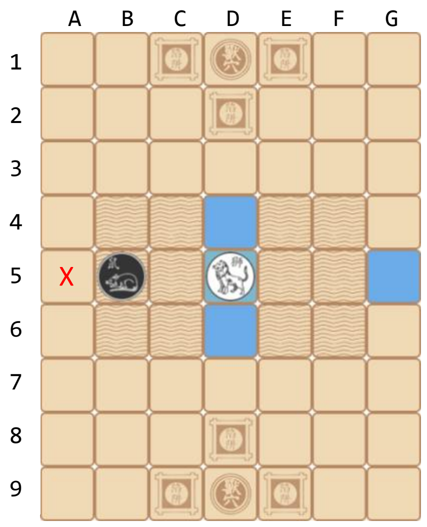
See Figure 2 to visualize rules 5 – 7.

Capturing Rules

A moving piece can capture (eat) an opponent – the attacking piece replaces the captured piece on its square; the captured piece is removed from the gameboard. A piece can capture any enemy piece that has the same or lower rank, with the following exceptions:

8. A Rat can "kill" (capture) an Elephant, but only from a land square, not from a water square.
9. Similarly, a Rat in a water square cannot kill an opponent Rat on an adjacent land square.
10. A Rat in the water is invulnerable to capture by any piece on land, i.e., a Rat in the water can only be killed by another Rat in the water (not by a Rat on a land square).
11. A piece that enters one of the opponent's trap squares is (temporarily) reduced to 0 in rank. So, the trapped piece can be captured by any piece of the defending side, regardless of rank. Also, a trapped piece (e.g., Lion) cannot kill any adjacent enemy (e.g., Cat) while it is still in the trap.
12. A trapped piece has its normal rank restored when it exits an opponent's trap square, i.e., when it has moved to an empty square.

13. An animal can enter its own trap squares with no effect on its rank. An opponent piece of a higher rank can capture it as usual. But after the capturing, the opponent piece's rank becomes 0 because it is now in the enemy's trap.



The blue cells are the valid positions that this Lion piece can move into. It can jump to G5 but not to A5 because a Rat piece has blocked its way of the jump.

```
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | # | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   |   | # |   | c |   |
  +---+---+---+---+---+---+---+
3 |   | l | p |   |   |   | e |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | R | * |   |
  +---+---+---+---+---+---+---+
5 |   | r | * | L | * | * |   |
  +---+---+---+---+---+---+---+
6 | w | * | * | T | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   | t |   |   |
  +---+---+---+---+---+---+---+
8 | E |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | # |   |   |
  +---+---+---+---+---+---+---+
```

In the console mode, Lion L at D5 cannot jump to A5 as blocked by Rat r at B5. Tiger T at D6 can jump to G6 or to A6 (capturing the wolf w by the way). Lion l at B3 can jump to B7. Tiger t at E7 cannot jump to E3 as blocked by Rat R at E4. So, using a Rat to block jumps of Lion and Tiger pieces is a good defending strategy.

Figure 2: Lion and Tiger can jump over a river from one edge to another.

Game Over Conditions

The player who is first to move any one of their pieces into the opponent's den wins the game. Another way to win is to capture all the opponent's pieces. There is no draw game.

Program Specification

Your work is to develop a C++ program that implements this Jungle game using OOP concepts. For consistent grading and to ease your OO design workload, we have provided you with a starter code base for this project. The whole project consists of up to 33 files (16 pairs of header files and source files for implementing 15 classes and one file I/O utility, plus the main or client source file). The files are listed in Table 2 and Table 3. We have provided a zip file of code base with some essential files provided to assist your design of this big program. Basically, you need not design but implement the given design of the game.

Note: you can actually have more than 33 files in your submission if you would take up the **optional** challenge of making a subclass `GreedyMachine` of the `Machine` class for implementing some greedy algorithm for picking moves that may allow the computer player to get a higher chance of winning the game.

Table 2: Header files for class definitions

	File	Description	Remarks
1	`game.h`	The Game class interface	Provided
2	`board.h`	The Board class interface	
3	`piece.h` *	The Piece class interface (an abstract base class)	
4	`player.h` *	The Player class interface (an abstract base class)	
5	`human.h`	The Human class interface (inheriting the Player class)	
6	`machine.h`	The Machine class interface (inheriting the Player class)	
7	`jumper.h`	The Jumper class interface	
8	`elephant.h`	The Elephant class interface (inheriting the Piece class)	To be created
9	`lion.h`	The Lion class interface (inheriting Piece and Jumper classes)	
10	`tiger.h`	The Tiger class interface (inheriting Piece and Jumper classes)	Provided
11	`leopard.h`	The Leopard class interface (inheriting the Piece class)	To be created
12	`wolf.h`	The Wolf class interface (inheriting the Piece class)	
13	`dog.h`	The Dog class interface (inheriting the Piece class)	Provided
14	`cat.h`	The Cat class interface (inheriting the Piece class)	
15	`rat.h`	The Rat class interface (inheriting the Piece class)	
16	`fileman.h`	A utility function for reading input initial board config. files	

Table 3: Source files for client code and class implementations

	File	Description	Remarks
17	`jungle.cpp`	The client program (containing the main function)	Provided
18	`game.cpp`	The Game class implementation	To fill in TODO parts
19	`board.cpp`	The Board class implementation	
20	`piece.cpp` *	The Piece class implementation	
21	`player.cpp` *	The Player class implementation	
22	`human.cpp`	The Human class implementation	
23	`machine.cpp`	The Machine class implementation	
24	`jumper.cpp`	The Jumper class implementation	
25	`elephant.cpp`	The Elephant class implementation	To be created
26	`lion.cpp`	The Lion class implementation	
27	`tiger.cpp`	The Tiger class implementation	To fill in TODO parts
28	`leopard.cpp`	The Leopard class implementation	To be created
29	`wolf.cpp`	The Wolf class implementation	
30	`dog.cpp`	The Dog class implementation	Provided
31	`cat.cpp`	The Cat class implementation	
32	`rat.cpp`	The Rat class implementation	To fill in TODO parts
33	`fileman.cpp`	A utility function for reading input initial board config. files	Provided

(* abstract classes)

Note the last column of the above tables:

(1) "Provided": all these source files have been provided with full code; basically, you don't need to change them unless you think of a strong need to adjust them to your redesign of the code.

(2) "To fill in TODO parts": in these files, some code segments are missing. **Please look for all TODO comments in them** for instructions about what to do and fill in the missing code to finish the implementation of each class.

(3) "To be created": such files are not provided and are to be created by you from scratch.

Some classes cannot compile until the classes they depend on have first compiled successfully. So, we suggest the following sequence of development:

Piece, Player, Game, … , Board, Human, Machine

Before Board can compile, you need to create all animal classes first. You may give them rather simple or almost empty or skeleton implementations of the Lion, Tiger, …, Rat classes first, so that the Board can be developed and compiled before each animal classes gets implemented in detail.

Game Flow

The `main()` function in jungle.cpp creates a Game object and calls its `run()` method to start a Jungle chess game. The `Game()` constructor accepts a game ***mode*** argument taking one of the values below:

- H2H (value 1) means "**Human vs. Human**", i.e., two human players enter moves via console.
- H2M (value 2) means "**Human vs. Machine**", i.e., one human player enters moves via console and the opponent player is the computer, which makes moves on its own.
- M3M (value 3) means "**Machine vs. Machine**", two computer players make moves in turns on their own without any human attention.

(The `Mode` enum in game.h has defined these three constant values.)

When the `Game` object is being created, the constructor will create two Player objects of type (Human / Machine) according to the above mode setting, set game state as *RUNNING (0)* and create the Board object. The Board constructor will call `Board::init()` method to set up the initial game board by filling the `cells` array with pointers to `Elephant`, `Lion`, `Tiger`, … objects that are being created in the method. The created pieces will be added to each player's list of pieces as well. Note that for easing your program testing, the Game and Board constructors also have a `filename` parameter; its value is either null or set to a command-line argument when the program is called from a command prompt or terminal. The command-line argument specifies a file path of a text file which contains user-customized initial board configuration (see more details later).

The Game object's `run()` is the main loop of the program, which repeats calling the current player's `makeMove(board)` function and printing the board. The current player is flipped in each iteration. For a Human player, its `makeMove()` will prompt the user for two cell addresses, e.g. A7 B8, with the former denoting the source cell and the latter denoting the destination cell on the board. These text-based cell addresses will be translated to array indexes to access the Board object's `cells` array.

Since a game may run long, we design a special way to end the game program prematurely. During a game run, either Human player can enter a *special imaginary move*, namely "Z0 Z0" (from cell "Z0" to cell "Z0" which are both invalid positions). The program will detect this input to halt the game. This is a graceful mean to end the program instead of pressing Ctrl + C (keyboard interrupt).

For a Machine player, its `makeMove()` makes a random (or intelligent) move as long as it is valid. The focus of this project is on OOP instead of AI. So, the machine player's strategy of moves is unimportant. Implementing a cleverer machine player won't be rewarded with higher marks in this project. However, for your self-learning's sake, whenever time permits, you are highly encouraged to try some greedy algorithms for picking "good moves". For example, you may create a subclass called `GreedyMachine`, inheriting the `Machine` superclass. When it comes to the machine's turn, its `makeMove()` method may scan over the player's `pieces` vector to choose a piece that can capture the opponent piece of the highest rank found so far (reaching highest score). However, it may also need to avoid entering some cells which are surrounding by animals stronger than the chosen piece (deducting score for that target position). So, the greedy machine can evaluate moves and find the best move that results in highest score currently.

If you feel this is too hard for you, you may choose to skip it and just implement a purely random move strategy in the Machine class. One naïve way is to first pick a piece randomly from the player's `pieces` vector and get its position (y1, x1). Then generate a random target position (y2, x2) and test if moving the piece from (y1, x1) to (y2, x2) is valid. If the move is invalid, repeat these steps until a valid move is resulted. Finally, carry out the valid random move.

Class Hierarchy

The basic structure of this complicated program can be broken down as follows.

Abstract base classes

There are two abstract base classes in this program:

Class **Piece** (piece.h and piece.cpp)

- It serves as the base or parent class of all game pieces like **Rat**, **Cat**, etc.
- Each Piece object has a name (from which its single-letter label is extracted for showing the piece on the gameboard, e.g., 'E' for Elephant, 'P' for leoPard, etc.), color (BLUE or RED), rank, and position (y, x) on the board.
- It has a virtual function `move(Board *board, int y, int x)` function for making a move. A basic implementation is available at the superclass level. It is up to you to decide whether certain subclasses of Piece need to override this function to customize the move behavior. In this case, you need to modify the subclass header to define the virtual function prototype in your subclass. Two other functions canCapture() and capture(), relating to capturing an opponent piece, are also marked as virtual. The subclasses may override them if you see it necessary.
- It has a *pure* virtual function of signature `isMoveValid(Board *board, int y, int x)`. For this one, every subclass of Piece must override it in order to be a concrete instantiable class.

Class **Player** (player.h and player.cpp)

- It serves as the base or parent class of **Human** and **Machine** classes.
- Each Player object has a name (in string, "Blue" or "Red") and a color (in Color enum, BLUE or RED). (note: Color enum is defined in piece.h.)
- It has a *pure* virtual function of signature `makeMove(Board* board)`. This method must be implemented by the concrete classes Human and Machine (otherwise, they will stay as abstract classes and can't be instantiated as objects).
- It keeps a list of pieces that are still alive and on the game board. (The list is implemented using a vector storing pointers to the Piece objects.)

- It provides methods to retrieve the count of pieces in the list, get a piece from the list, add a piece to or delete a piece from the list.

Concrete classes

Subclasses of **Piece**

We have provided the following classes that inherit from Piece as examples of how to complete the implementation of a working game piece:

- Class **Cat** (cat.h and cat.cpp)
- Class **Dog** (dog.h and dog.cpp)

They have implemented their own `isMoveValid(Board *board, int y, int x)`.

One of your main tasks is to add the following subclasses that inherit from Piece, each of which must implement the pure virtual function `isMoveValid(int y, int x)` according to the rules governing the moves of each piece type. Follow our provided examples (Cat and Dog classes) and remember to add “include guard” compiler directives in all header files.

- Class **Elephant** (elephant.h and elephant.cpp)
- Class **Lion** (lion.h and lion.cpp)
- …

Lion and Tiger are two most special subclasses. Besides Piece, they inherit from a second base class which is Jumper via multiple inheritance syntax:

```
class Tiger : public Piece, public Jumper
{ ... };
```

The design or the existence of the Jumper class is for enhancing code reuse. Since both Lion and Tiger can jump over a river, and their jump behavior is the same, we choose to put the code for checking if a jump can be made inside the Jumper class, so we can write it once only. Then both Lion and Tiger can simply reuse it.

Subclasses of **Player**

- Class **Human** (human.h and machine.cpp)
  - It implements `makeMove(Board* board)` to repeatedly prompt the current human player to enter two cell addresses of a move until a valid move is obtained. The ad hoc move "Z0 Z0" is detected here and if received, the game state is set to GAME_OVER and the method returns at once. (The ad hoc move is a surrender mechanism for ending the game earlier or when there is really no valid move on the board while the last animal is still alive)
  - In the loop body, it calls the `move()` method of the board object, which will validate the move and update the board if the move is confirmed valid.
- Class **Machine** (machine.h and machine.cpp)
  - It implements `makeMove(Board* board)` to make a random valid move on the board by a nested loop. First, pick a piece randomly from the player's `pieces` vector and get its position (y1, x1). Then generate a random position (y2, x2) and try moving the piece from (y1, x1) to

(y2, x2). If the trial move is found invalid, repeat the second step. If it happens that a valid move cannot be found within some trials bound, pick a piece from the player's pieces list randomly again to redo the random search process. The maximum times of the repeated piece picking can be, for example, 10 times the piece count in the vector. In case that the program cannot find any valid move within the trials bound for all piece pickings, the player must surrender – set game state to GAME_OVER and the turn finishes with no actual move (no update of the board), and the turn is flipped to the opponent player. This is possible, say, when a player has only one piece left, and the piece is fully surrounded by the opponent's pieces all of which have higher ranks. In this case, the single piece has no way to move.

Note that each move will go through a three-level call hierarchy:

```
player->makeMove(board);            // player can be a Human or Machine object

         board->move(y1, x1, y2, x2);

                  piece->move(y2, x2);   // piece is an object of any Piece subclasses
```

Note: piece = the piece at (y1, x1) on the board, which is being moved to the cell (y2, x2).

For move validity checks, there are also two levels of checking:

```
board->isMoveValid(y1, x1, y2, x2);

                 piece->isMoveValid(y2, x2);
```

(1) **Board-level**: done by `Board::isMoveValid(…)`, which includes validation against illegal cases like accessing out-of-bound positions or moving a piece that belongs to the opponent. Read the TODO comments in board.cpp for more details.
(2) **Piece-specific**: done by `Piece::isMoveValid(Board* board, int y, int x)`, which is a virtual method and can also be overridden by subclass's version (polymorphism) to include validation against the moving rules of the specific piece type, e.g. only Rat can move into a water square; e.g. an Elephant can move to a square occupied by a *trapped* opponent Rat.

The board-level `move(…)` will call board-level `isMoveValid(…)` to confirm validity before it carries out an actual move while the piece-level `move()` won't call piece-level `isMoveValid()` anymore since it has been called by `Board::isMoveValid(…)` already. Piece-level `move(…)` carries out the move updating the board and piece's data, taking care of some required actions associated with a move, e.g. capturing the opponent piece, handling the animal's rank when entering and leaving a trap of the opponent side, and checking if any winning condition has been satisfied.

Assumptions

- The board size is always 9x7. Still, we use two global constants H and W to represent the board's height (number of rows) and width (number of columns).
- We assume that cell address inputs always follow the format of one letter (A-Z or a-z) plus one integer (1-26). Lowercase inputs like "a1", "h10", etc. will be accepted as normal. Except for "Z0" (case-sensitive) which is treated as a *sentinel* for game stop, you need NOT handle weird inputs

like "AA1", "A01", "abc", "A3.14", "#a2", … for cell addresses and inputs like "-1", "6.28", "#$@", … for the game mode ~~number of pits~~. The behavior upon receiving these is unspecified, probably running into forever loop or program crash.

Restrictions and Freedom

- You cannot declare any global variables (i.e. variables declared outside any functions). But global constants or arrays of constants are allowed.
- Your final program should contain all the necessary files (the C++ header and source files listed above). Note your spelling - do not modify any file names.
    - For class names, use camel case (e.g., Elephant, Rat, GreedyMachine); for file names, use lowercase for all letters (e.g., elephant.h, rat.cpp, greedymachine.cpp).
- For the provided code, basically, you need not (or even should not) modify them. Having that said, you are still allowed to modify the code if you have some better ideas or needs of working something around. For example, for the Game class, our provided design does not have a field for specifying the game winner. We exploited the `turn` field (keeping track of who is playing the current turn) to report the winner and assumed the player playing the last turn before game is over is the winner. If you feel necessary, you can add an explicit field, called `winner` say, for the purpose of reporting the game winner. You may also add new methods to any class (preferably only subclasses) if necessary.

Test Cases

There are many possible test cases in this project. Think carefully for a complete set, e.g., around those special rules on moves related to rivers, traps and dens, Elephant vs. Rat, and the jumping of Lion and Tiger over a river. However, playing the game from beginning by moving one square only every time when we want to test specific cases is very tedious and time-consuming. To cut the time, we have provided code (fileman.cpp/.h) for the program to read a text file which is used to list only a subset of animal pieces and their specific positions of user choice.

If you call the program in a terminal and provide the file name as the command-line argument, e.g.,

```
C:>sample-jungle.exe board1.txt
```

Then the program will read the file content and the board will be initialized with a subset of pieces defined in the file instead of presetting all 16 animals at their standard initial positions as in Figure 1. For example, the sample text file board1.txt contains the following lines:

```
F2 c
B3 l
C3 p
G3 e
E4 R
B5 r
D5 L
A6 w
D6 T
E7 t
```

```
A8  E
```

The first column marks the cell addresses of the animals denoted by the case-sensitive labels listed in the second column. Lowercase letters refer to Blue's pieces, and uppercase to Red's pieces.

Loading this file will initialize the game board to be the one as shown in Figure 2 (right). The game can continue from this state onwards with Blue playing the first move.

Note that we didn't implement full validation logic for this part. So, it is possible that you can create multiple pieces of Lion for one player, and the game behavior could be unknown. It is the user's responsibility to ensure a valid initial game setting is in place while shortcutting the moving process from the game beginning using this facility.

Sample Runs

In the following sample runs, the blue text is user input and the other text is the program printout. You can try the provided sample program for other input. *Your program output should be exactly the same as the sample program* (same text, symbols, letter case, spacings, etc.). Note that *there is a space after the ':' in the program printout*.

There are too many possible test cases. Due to space limitation, we can show only a few sample runs below. Please check your program correctness against the results produced by our sample program executable posted on Blackboard. A few sample text files for defining customized initial game boards are also available on Blackboard.

Sample run in Human vs. Human mode:

```
C:\Desktop\Jungle\Debug>Jungle.exe board4.txt
```

```
Choose game mode (1, 2, 3): 1↵
Round 1:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | # | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   | d | L |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   | p |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   | t |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | # | r |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | # | E |   |
  +---+---+---+---+---+---+---+
Blue's turn: d7 d8↵
Round 2:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | # | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   | d | L |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   | p |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t | r |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | # | E |   |
  +---+---+---+---+---+---+---+
Red's turn: d2 c2↵
Invalid. Try again!
Red's turn: d2 d3↵
Round 3:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | # | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   | d | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   | p | L |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t | r |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | # | E |   |
  +---+---+---+---+---+---+---+
Blue's turn: e8 e9↵
Round 4:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | # | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   | d | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   | p | L |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | r | E |   |
  +---+---+---+---+---+---+---+
Red's turn: d3 c3↵
Leopard of Blue captured!
Round 5:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | # | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   | d | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   | L |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | r | E |   |
  +---+---+---+---+---+---+---+
Blue's turn: c2 c1↵
Round 6:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | d | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   | L |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | r | E |   |
  +---+---+---+---+---+---+---+
Red's turn: c3 c2↵
Round 7:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | d | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   | L | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | r | E |   |
  +---+---+---+---+---+---+---+
Blue's turn: d8 c8↵
Invalid. Try again!
Blue's turn: c1 d1↵
Invalid. Try again!
Blue's turn: e9 e8↵
Round 8:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | d | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   | L | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t | r |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | # | E |   |
  +---+---+---+---+---+---+---+
Red's turn: c2 c1↵
Dog of Blue captured!
Round 9:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | L | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t | r |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | # | E |   |
  +---+---+---+---+---+---+---+
Blue's turn: e8 e9↵
Round 10:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | L | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | r | E |   |
  +---+---+---+---+---+---+---+
Red's turn: f9 e9↵
Rat of Blue captured!
Round 11:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | L | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | t |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | E |   |   |
  +---+---+---+---+---+---+---+
Blue's turn: d8 d9↵
Game over:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 |   |   | L | X | # |   |   |
  +---+---+---+---+---+---+---+
2 |   |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   | C | # |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | t | E |   |   |
  +---+---+---+---+---+---+---+
Player Blue wins!
```

Sample run in Machine vs. Machine mode (a special case that Blue has no valid move at all):

```
C:\Desktop\Jungle\Debug>Jungle.exe board2.txt
```

```
Choose game mode (1, 2, 3): 3↵
Round 1:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 | r | C | # | X | # |   |   |
  +---+---+---+---+---+---+---+
2 | T |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | # |   |   |
  +---+---+---+---+---+---+---+
Game over:
    A   B   C   D   E   F   G
  +---+---+---+---+---+---+---+
1 | r | C | # | X | # |   |   |
  +---+---+---+---+---+---+---+
2 | T |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
4 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
5 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
6 |   | * | * |   | * | * |   |
  +---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+
8 |   |   |   | # |   |   |   |
  +---+---+---+---+---+---+---+
9 |   |   | # | X | # |   |   |
  +---+---+---+---+---+---+---+
Player Red wins!
```

Submission and Marking

- Submit a zip of your source files to Blackboard (https://blackboard.cuhk.edu.hk/).
- Insert *your name*, *student ID*, and *e-mail* as comments at the beginning of your main.cpp file.

- You can submit your assignment multiple times. Only the latest submission counts.
- Your program should be *free of compilation errors and warnings*.
- Your program should *include suitable comments as documentation*.
- ***Do NOT plagiarize.*** Sending your work to others is subject to the same penalty for copying work.