Tutorial 11: Project — Jungle Chess

Introduction

TAs

- **Zhihang Hu (class A)**
- **zhhu21@cse.cuhk.edu.hk**
- **SHB 1026**
- **Consultation hours: Wed 3:30-5:30pm**

- **Sadaf Khan (class B)**
- **skhan@cse.cuhk.edu.hk**
- **SHB 1026**
- **Consultation hours: Tue 3:30-5:30pm**

Project: Jungle Chess (A classic strategy board game)

Object-Oriented Programming

Inheritance && Polymorphism

… …

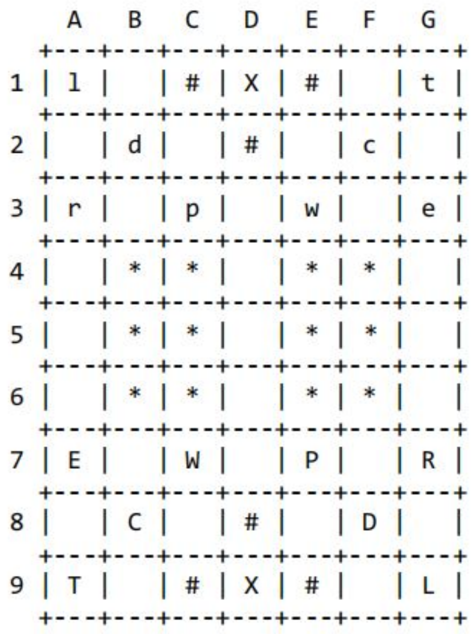
Project: Jungle Chess (A classic strategy board game)

Player 1

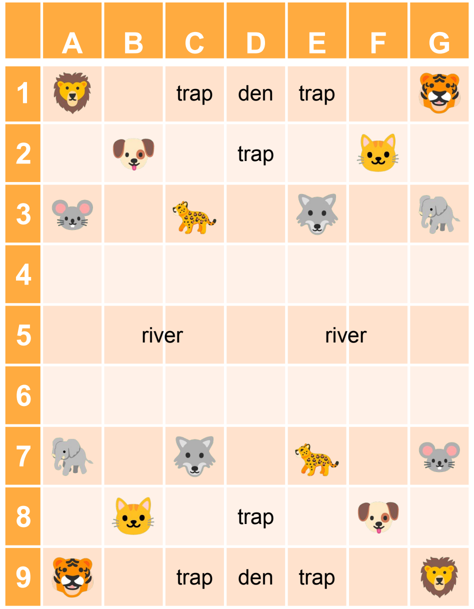
Lower Case

Player 2

Upper Case



```
    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	C	D	E	F	G
1	🦁		trap	den	trap		🐯
2		🐶		trap		🐱	
3	🐭		🐆		🐺		🐘
4							
5		river			river		
6							
7	🐘		🐺		🐆		🐭
8		🐱		trap		🐶	
9	🐯		trap	den	trap		🦁

Initial game setup: 7 × 9 board

Project: Jungle Chess (A classic strategy board game)

- Each Piece is associated with different animals, each with a different rank, as listed in the right table.

Rank	Piece	Face	Chinese	Red Label	Blue Label
8	Elephant	🐘	象	E	e
7	Lion	🦁	獅	L	l
6	Tiger	🐯	虎	T	t
5	Leopard	🐆	豹	P	p
4	Wolf	🐺	狼	W	w
3	Dog	🐶	狗	D	d
2	Cat	🐱	貓	C	c
1	Rat	🐭	鼠	R	r

General Rules

- Two players, namely "Blue" and "Red", make moves in turns.
- Must make a movement in it’s turn.
- Piece can capture and replace another player’s piece by some specific rules.
- Different animals’ movement differs, and rules in different terrain differs.
- 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.

	A	B	C	D	E	F	G
1	🦁		trap	den	trap		🐯
2		🐶		trap		🐱	
3	🐭		🐆		🐺		🐘
4							
5		river			river		
6							
7	🐘		🐺		🐆		🐭
8		🐱		trap		🐶	
9	🐯		trap	den	trap		🦁

Movement Rules

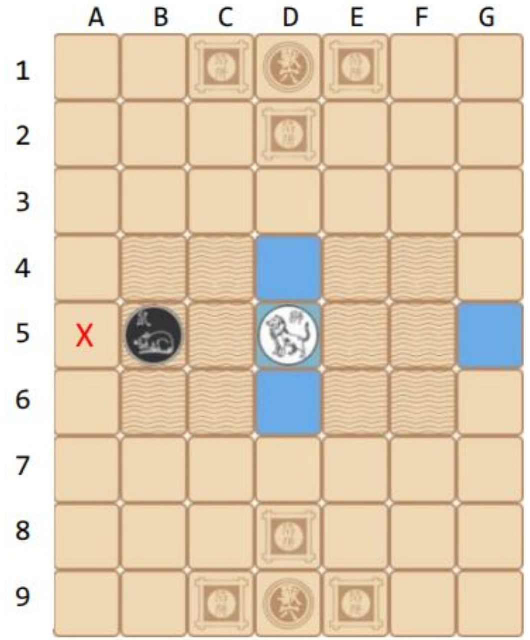
Rules 4 to 7 are special, rules related to the river (or water) squares.

1. All pieces can move <u>one square</u> horizontally or vertically (not diagonally).
2. A piece may not move into <u>its own den</u>.
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.

	A	B	C	D	E	F	G
1	🦁		trap	den	trap		🐯
2		🐶		trap		🐱	
3	🐭		🐆		🐺		🐘
4							
5		river			river		
6							
7	🐘		🐺		🐆		🐭
8		🐱		trap		🐶	
9	🐯		trap	den	trap		🦁

Movement Rules

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.



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.

Capturing Rules

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).

	A	B	C	D	E	F	G
1	🦁		trap	den	trap		🐯
2		🐶		trap		🐱	
3	🐭		🐆		🐺		🐘
4							
5		river			river		
6							
7	🐘		🐺		🐆		🐭
8		🐱		trap		🐶	
9	🐯		trap	den	trap		🦁

Capturing Rules

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.

	A	B	C	D	E	F	G
1	🦁		trap	den	trap		🐯
2		🐶		trap		🐱	
3	🐭		🐆		🐺		🐘
4							
5		river			river		
6							
7	🐘		🐺		🐆		🐭
8		🐱		trap		🐶	
9	🐯		trap	den	trap		🦁

C++ Implementation

Problem Specification

- Develop a C++ program that implements this Jungle game using OOP concepts.
- We have provided you with a starter code base for this project.
- You need not design but implement the given design of the game. (TODO parts)
- The whole project consists of up to 33 files (16 pairs of header files and source files for implementing 15 classes, one file I/O utility for loading customized initial boards, plus the main or client source file).

Header Files

	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	

(1) "Provided": all these source files need no changes and should be kept unmodified.

(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.

Cpp Files

	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

(1) "Provided": all these source files need no changes and should be kept unmodified.

(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.

Main

- The `main()` function in jungle.cpp creates a Game object and calls its `run()` method to start a Jungle chess game.

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.)

game.h

```
enum Mode { H2H = 1, H2M, M2M };

enum State { RUNNING, GAME_OVER };
```

jungle.cpp

```
int main(int argc, char* argv[]) {
    char* filename = nullptr;
    if (argc > 1)
        filename = argv[1];

    cout << "Choose game mode (1, 2, 3): ";
    int mode;
    cin >> mode;
    Game game((Mode)mode, filename);
    game.run();
    return 0;
}
```

game.h

```
enum Mode { H2H = 1, H2M, M2M };
enum State { RUNNING, GAME_OVER };
```

Initialize

- A game is created according to the user settings and the related players and game board are initialized.

game.c

```
Game::Game(Mode mode, char* filename) : state(RUNNING) {
    Player* blue = (Player*) (mode == M2M) ?
        (Player*) new Machine("Blue", BLUE) : (Player*) new Human("Blue", BLUE);

    Player* red  = (Player*) (mode == H2H) ?
        (Player*) new Human("Red", RED) : (Player*) new Machine("Red", RED);
    /* TODO (Optional): change this line from Machine to GreedyMachine if you have,
       then RED player is supposed to be "cleverer" in terms of move strategy
     */

    players[0] = blue;
    players[1] = red;
    turn = BLUE;
    board = new Board(this, filename);
}
```

Board

- Board is initialized with each cell a pointer to the corresponding piece.

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.

```
Board::Board(Game* game, char* filename) :
    game(game), cells() {  // initalize cells' elements to nullptr
    // Set up the initial game board
    if (filename != nullptr)
        loadFromFile(filename, this);
    else
        init();
}


// initial gameboard configuration
void Board::init() {
    // TODO:
    // set up the Jungle chess standard initial gameboard properly;
    // add Elephants, Lions, ..., Rats to the cells array at correct
positions

    // TODO: also add the created pieces to each player's vector of pieces
}
```

Board

- Board is initialized with each cell a pointer to the corresponding piece.

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.

You need to use the **new** operator which allocates memory space to store data of the object to be created and returns a pointer to the created object:

```
cells[y][x] = new Lion(color, y, x);
```

Human::MakeMove()

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.

- 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.

```
void Game::run() {
    for (int round = 1; state == RUNNING ; round++)  {
        cout << "Round " << round << ":" << endl;
        board->print();                         // print game board
        getPlayer(turn)->makeMove(board); // move a chess piece
        if (state == RUNNING)
            turn = Color(1 - int(turn));  // flip turns
    }
    // Game over
    cout << "Game over:" << endl;
    board->print();

    // TODO: print who wins
    // (assumed the winner is the player who made the last turn)
}
```

Machine ::MakeMove()

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.

For your self-learning's sake, you are highly encouraged to try some greedy algorithms for picking "good moves".

Machine ::MakeMove()

```
// a sample machine that makes random valid moves
void Machine::makeMove(Board* board) {
    // TODO: make a random but valid move of a randomly picked piece on a board
    // Hint: there exist many ways to do so, one way is as follows:
    // - generate a random integer for picking a piece r from the player's pieces vector
    // - set y1, x1 to r->getY(), r->getX()
    // - generate random integers y2 and x2 in range of [0, H) and [0, W) respectively
    //   [or better in the y, x range of the 4 neighboring cells around (y1, x1), note to
    //    handle jumpable cells, 2 or 3 cells away, as well for Tiger and Lion]
    // - call board's move(y1, x1, y2, x2)
    // - once a valid move is returned, print the from and to cell addresses
    //   and exit this function
    // Note: it can happen that no valid move can be found despite repeated picks.
    //       For example, only a Rat remains alive at a corner of the board while
    //       the two cells it may go have been occupied by a Cat and a Dog.
    //       In this case, the player must surrender (set the opponent as winner).
}
```

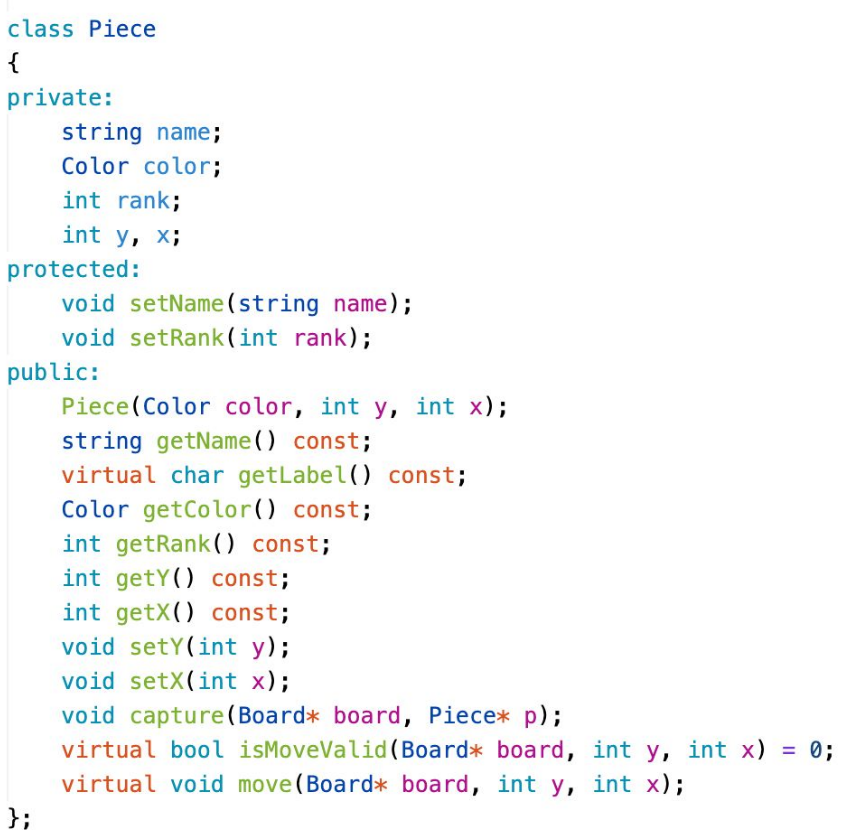
Piece(**Base classes)**

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

- **Base classes**

Class **Piece** (piece.h and piece.cpp) It serves as the base or parent class of all game pieces like **Rat**, **Cat**, etc.

1. 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.
2. 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 nee 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.
3. It has a *pure* virtual function of signature isMoveValid(Board *board, int y, int x).



```
class Piece
{
private:
    string name;
    Color color;
    int rank;
    int y, x;
protected:
    void setName(string name);
    void setRank(int rank);
public:
    Piece(Color color, int y, int x);
    string getName() const;
    virtual char getLabel() const;
    Color getColor() const;
    int getRank() const;
    int getY() const;
    int getX() const;
    void setY(int y);
    void setX(int x);
    void capture(Board* board, Piece* p);
    virtual bool isMoveValid(Board* board, int y, int x) = 0;
    virtual void move(Board* board, int y, int x);
};
```

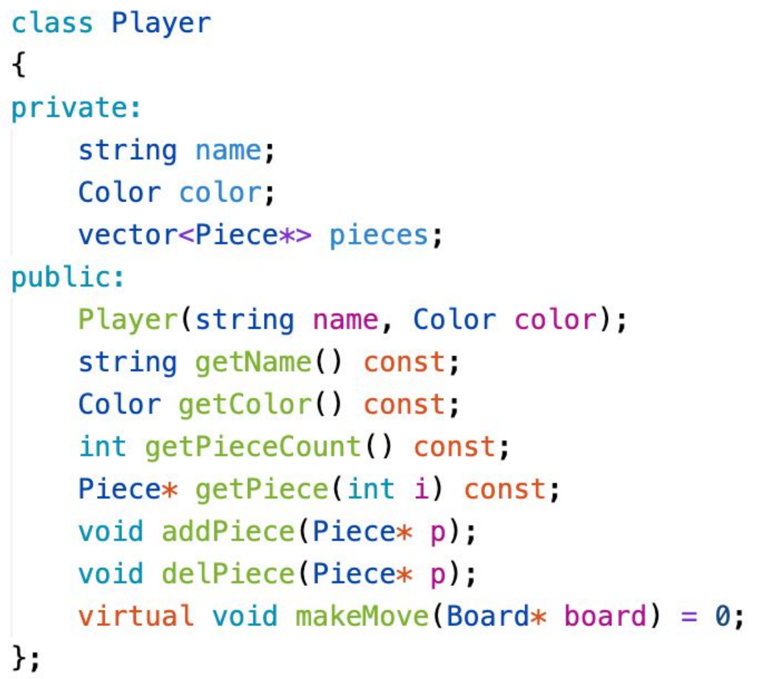
Player(**Base classes)**

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

- **Base classes**

It serves as the base or parent class of **Human** and **Machine** classes.

1. 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.)
2. 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).
3. 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.)
4. 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.



```
class Player
{
private:
    string name;
    Color color;
    vector<Piece*> pieces;
public:
    Player(string name, Color color);
    string getName() const;
    Color getColor() const;
    int getPieceCount() const;
    Piece* getPiece(int i) const;
    void addPiece(Piece* p);
    void delPiece(Piece* p);
    virtual void makeMove(Board* board) = 0;
};
```

Subclasses of **Piece**

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

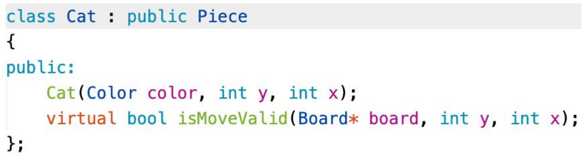
- Class **Cat** (cat.h and cat.cpp)
- Class **Dog** (dog.h and dog.cpp)

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.

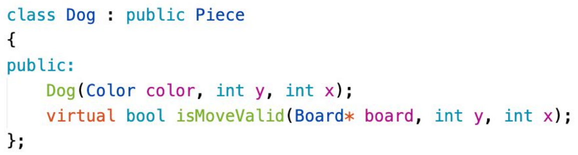
- Class **Tiger** (Tiger.h and Tiger.cpp)
- Class **Lion** (lion.h and lion.cpp)

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

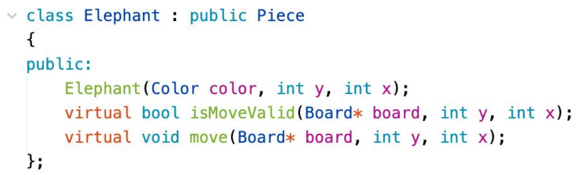
The design or the existence of the Jumper class is for enhancing code reuse. Since both Lion and Tiger can jump over a river, and their jump behavior is the same, we choose to put the code for checking if a jump can be made inside the Jumper class (i.e., write once only). Then both Lion and Tiger can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



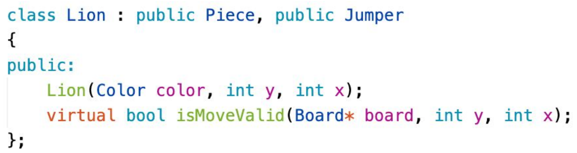
```
class Cat : public Piece
{
public:
    Cat(Color color, int y, int x);
    virtual bool isMoveValid(Board* board, int y, int x);
};
```



```
class Dog : public Piece
{
public:
    Dog(Color color, int y, int x);
    virtual bool isMoveValid(Board* board, int y, int x);
};
```



```
class Elephant : public Piece
{
public:
    Elephant(Color color, int y, int x);
    virtual bool isMoveValid(Board* board, int y, int x);
    virtual void move(Board* board, int y, int x);
};
```



```
class Lion : public Piece, public Jumper
{
public:
    Lion(Color color, int y, int x);
    virtual bool isMoveValid(Board* board, int y, int x);
};
```

Subclasses of **Player**

Class **Human** (human.h and machine.cpp)

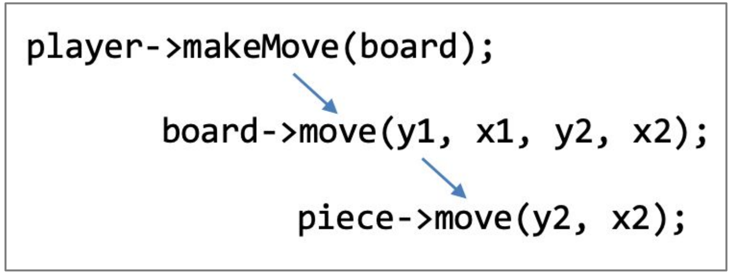
It implements `makeMove(Board* board)` to repeatedly prompt the current human player to enter two cell addresses of a move until a valid move is obtained.

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. 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)

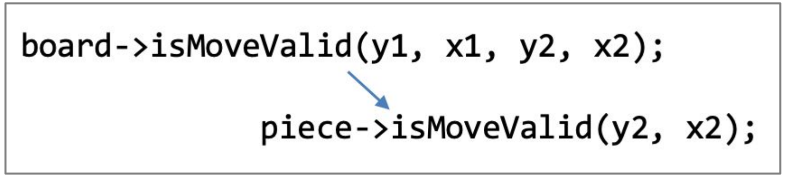
It implements `makeMove(Board* board)` to make a random valid move on the board by a nested loop. First, pick a piece randomly from the player's `pieces` vector and get its position (y1, x1). Then generate a random position (y2, x2) and try moving the piece from (y1, x1) to y2, x2).

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



```
player->makeMove(board);
              board->move(y1, x1, y2, x2);
                      piece->move(y2, x2);
```

Move validity checks also have two levels:



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

Move Validity Checks

The Board-level `move()` calls `Board::isMoveValid()` before executing the move. This level of checking 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.

```
// Carry out the move from (y1, x1) to (y2, x2)
bool Board::move(int y1, int x1, int y2, int x2) {
    if (isMoveValid(y1, x1, y2, x2)) {
        get(y1, x1)->move(this, y2, x2);
        return true;
    }
    return false;
}
```

Move Validity Checks

```cpp
// Check if the move from (y1, x1) to (y2, x2) is valid
bool Board::isMoveValid(int y1, int x1, int y2, int x2) {
    // TODO:
    // check against invalid cases, for example,
    // - the source is an empty cell
    // - the source and destination are the same position
    // - the destination is out of bound of the board (hint: use OUT_BOUND)
    // - the source piece is not of same color of current turn of the game
    // (... and anymore ?)
    // [Note: you can modify the following code if it doesn't fit your design]

    // Piece-specific validation
    if (p->isMoveValid(this, y2, x2) == false)
        return false;

    return true;
}
```

Move Validity Checks and Move

The Piece-level `piece->isMoveValid(board, y2, x2)` has validation against the moving rules of the specific piece type, e.g. only Rat can move into a water square. The Piece-level `move()` includes capturing the opponent piece, adjusting the animal's rank when entering and leaving a trap of the opponent side, and checking if winning condition is met.

```
// Carry out the move of this piece to (y, x)
void Piece::move(Board* board, int y, int x) {
    // TODO:
    // Hint: by calling suitable existing methods

    // capture opponent piece
    // ...


    // handle rank changes when entering and leaving traps
    // ...


    // check winning conditions
    // (moved into opponent's den or captured all opponent pieces)
    // ...


    // carry out the move
    // ...
}
```

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 and Rat); for file names, use lower case for all letters (e.g., elephant.h, rat.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.

- e.g. add winner field to Game class; create a new subclass GreedyMachine for implementing some greedy moves based on score (rank of captured pieces) of candidate valid moves.

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.