Assignment 5: Dual-Knight's Tour

Due: 20:00, Thu 22 Nov 2018 Full marks: 100

Introduction

In the game of chess, a knight (馬) is a piece moving like the letter L (「日」字) on a chessboard. That is, it moves either two squares horizontally and one square vertically, or two squares vertically and one square horizontally. A *dual-knight's tour* is to put two knights on a chessboard, each making moves, such that they never visit a square more than once. For simplicity in this assignment, we use a smaller 6×6 chessboard. Note that the two knights can eventually end up in squares where both can have no more possible moves but other squares remain unvisited. Figure 1 shows one dual-knight's tour, in which both knights ♘ at (rol,col) = (5,4) and ♞ at (0,1) have no more moves eventually. You are going to write a program, using object-oriented programming, to let users put two knights somewhere on a chessboard and moves them until no more moves can be made.

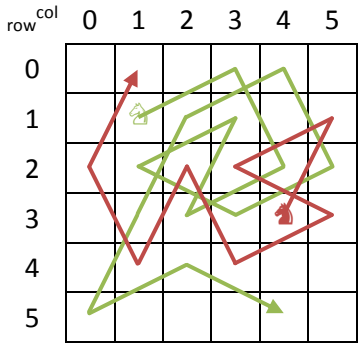

Figure 1: An example 6×6 dual-knight's tour. The symbols ♘ and ♞ denotes the starting positions of the two knights.

Program Specification

You have to write your program in two source files `DualKnightsTour.cpp` and `tour.cpp`. The former is the implementation of the class `DualKnightsTour`, while the latter is a client program of class `DualKnightsTour` which performs the program flow. You are recommended to *finish the `DualKnightsTour` class first* before writing the client program. When you write the `DualKnightsTour` class, *implement the member functions and test them individually one by one*. *Your two files will be graded separately*, so you should not mix the functionalities of the two files.

Class `DualKnightsTour` (`DualKnightsTour.cpp`)

You are given the interface of the `DualKnightsTour` class in the header file `DualKnightsTour.h`. *Do not* modify this header file. Descriptions of the class are given below.

```
class DualKnightsTour {
public:
    DualKnightsTour(int r1, int c1, int r2, int c2);
    void print() const;
    bool isValid(char knight, int r, int c) const;
    bool hasMoreMoves() const;
```



```
    bool move(char knight, int r, int c);
private:
    char board[6][6];
    int posR1, posC1, posR2, posC2;
    int steps1, steps2;
    int consec1, consec2;
};
```

Private Data Members

`char board[6][6];`

The configuration of a dual-knight's tour is represented by a 6×6 two-dimensional `char`-array. The elements `board[0][0]`, `board[0][5]`, `board[5][0]`, and `board[5][5]` denote the top-left, top-right, bottom-left, and bottom-right corners of the board respectively. Each array element stores either a dot '`.`' or a letter 'A'–'Z'/'a'–'z'. A dot '`.`' means that the square was not visited by any knight before. Uppercase letters 'A'–'Z' denote the move sequence of the one knight ♘: A → B → C → …, where 'A' is the starting position of ♘. Lowercase letters 'a'–'z' denote similarly for the other knight ♞. For example, the tour in Figure 1 would be stored in the array `board` as shown in Figure 2.

row \ col	0	1	2	3	4	5
0	.	i	.	B	I	.
1	.	A	J	E	.	b
2	h	F	f	c	C	H
3	.	K	D	G	a	d
4	.	g	M	e	.	.
5	L	.	.	.	N	.

Figure 2: The contents of data member `board` for the dual-knight's tour in Figure 1.

`int posR1, posC1, posR2, posC2;`

Denote the current positions of the two knights on the chessboard. (`posR1`,`posC1`) is the current position of ♘, while (`posR2`,`posC2`) is the current position of ♞.

`int steps1, steps2;`

The data members `steps1` and `steps2` store respectively the number of moves that ♘ and ♞ have already done since the beginning.

`int consec1, consec2;`

To balance the two knights, we restrict that one knight cannot move three consecutive steps. That is, if a knight has moved twice consecutively, then the next move must be by the other knight. The data members `consec1` and `consec2` store the number of consecutive moves by ♘ and ♞ respectively.

Public Constructor and Member Functions

`DualKnightsTour(int r1, int c1, int r2, int c2);`

This constructor creates a dual-knight's tour where the knights ♘ and ♞ are initially positioned at (`r1`,`c1`) and (`r2`,`c2`) respectively. You have to set all array elements of the data member `board` as '`.`'

(unvisited) except the initial position, which should be set as 'A' for ♘ and 'a' for ♞. You also have to set the data members (a) `posR1`, `posC1`, `posR2`, and `posC2` using the parameters `r1`, `c1`, `r2`, and `c2` respectively, and (b) all the other data members as 0. You can assume that parameters `r1`, `c1`, `r2`, and `c2` are always in [0 … 5], and the two positions are always different.

`void print() const;`

Prints the configuration of the dual-knight's tour in the format shown in Figure 3. Note that the current positions of ♘ and ♞ are specially printed as `@` and `#` respectively. (The actual `board` contents would still be letters; they are just printed specially to easily identify the current positions.)

```
  0 1 2 3 4 5
0 . # . B I .
1 . A J E . b
2 h F f c C H
3 . K D G a d
4 . g M e . .
5 L . . . @ .
```

Figure 3: Printing format of a dual-knight's tour.

`bool isValid(char knight, int r, int c) const;`

Checks whether the parameter `knight` in the tour can be moved from its current position to the destination at (`r`,`c`). Note that the knight is *not* actually moved in this member function; it just checks if the move is valid or not. It should return `true` if *all* the following conditions are satisfied:

- `knight` is either '@' or '#' (meaning ♘ and ♞ respectively);
- `r` and `c` form a proper position ($0 \le r,c < 6$);
- the proposed destination is an unvisited square;
- the proposed move is not a third consecutive one for the same knight;
- the destination is an L-shape move.

If any of the above conditions is `false`, the member function should return `false`.

`bool hasMoreMoves() const;`

Checks whether either knight has more possible moves to make. This member function should return `true` if at least one knight can make at least one valid move; and should return `false` otherwise. This member function can be implemented by *calling* `isValid(…)` several times.

`bool move(char knight, int r, int c);`

Tries to move the parameter `knight` from its current position to the destination at (`r`,`c`). This member function *should call* `isValid(…)` in its implementation to verify whether the move can actually be made. If the destination forms a valid move, this member function should update all related data members to reflect the corresponding knight's change of position, and returns `true`. Otherwise (that is, the move is invalid), this member function updates nothing and returns `false`. (For example, a valid move by ♘ should: (a) update its move sequence and its current position, (b) increment `steps1` and `consec1`, (c) reset `consec2` to 0, and (d) return `true`.)

Client Program (`tour.cpp`)

Your main program is a client of the `DualKnightsTour` class. You create a `DualKnightsTour` object here and call its member functions to implement the following program flow.

1. The program starts with prompting the user to enter the two knights' starting positions. You can assume that *this user input is always four integers*, where the first two inputs are the row and column of ♘ (@), and the next two inputs are the row and column of ♞ (#).
2. If either input position is invalid (row or column outside [0 … 5]) or the two positions coincide, display a warning and go back to step 1.
3. Create a `DualKnightsTour` object using the input positions.
4. Prompt the user to make a knight's move. You can assume that *the input is always a character followed by two integers*, in which the character is the knight and the integers are the destination of the move. You need to check if a user input is valid or not. (See definition in the description of the `isValid(…)` member function of `DualKnightsTour` class.) You should call the `isValid(…)` or `move(…)` member functions to do the checking here.
5. If the input is valid, you should move the corresponding knight to the destination. Otherwise, you should print a warning message and go back to step 4.
6. After moving a knight, check if there are still more possible moves. (At least one knight can make at least one move.) If so, you should go back to step 4 to obtain the next user input destination.
7. When there are no more possible moves, display the message "*No more moves!*".

Some Points to Note

- You *cannot declare any global variables* in *all* your source files (except `const` ones).
- You can define extra functions in any source files if necessary. However, extra *member* functions (instance methods), no matter `private` or `public`, are *not* allowed.

Program Output

The following shows some sample output of the program. The blue text is user input and the other text is the program output. You can try the provided sample program for other input. *Your program output should be exactly the same as the sample program* (i.e., same text, same symbols, same letter case, same number of spaces, etc.). Otherwise, it will be considered as *wrong*, even if you have computed the correct result. Note that in the program output, *only the chessboard is printed from the* `DualKnightsTour` *class* (`DualKnightsTour.cpp`). *All other program output (the prompts, warnings, ending message) are printed from the client program* (`tour.cpp`).

```
Knights' starting positions (row1 col1 row2 col2): 1 6 3 4↵
Invalid position(s)!
Knights' starting positions (row1 col1 row2 col2): 1 1 -1 3↵
Invalid position(s)!
Knights' starting positions (row1 col1 row2 col2): 2 3 2 3↵
Invalid position(s)!
Knights' starting positions (row1 col1 row2 col2): 1 1 3 4↵
  0 1 2 3 4 5
0 . . . . . .
1 . @ . . . .
2 . . . . . .
3 . . . . # .
4 . . . . . .
5 . . . . . .
```

```
Move (knight row col): @ -1 2↵
Invalid move!
Move (knight row col): @ 0 4↵
Invalid move!
Move (knight row col): @ 0 3↵
  0 1 2 3 4 5
0 . . . @ . .
1 . A . . . .
2 . . . . . .
3 . . . . # .
4 . . . . . .
5 . . . . . .
Move (knight row col): @ 2 4↵
  0 1 2 3 4 5
0 . . . B . .
1 . A . . . .
2 . . . . @ .
3 . . . . # .
4 . . . . . .
5 . . . . . .
Move (knight row col): @ 3 2↵
Invalid move!
Move (knight row col): # 1 5↵
  0 1 2 3 4 5
0 . . . B . .
1 . A . . . #
2 . . . . @ .
3 . . . . a .
4 . . . . . .
5 . . . . . .
Move (knight row col): # 0 3↵
Invalid move!
Move (knight row col): @ 3 2↵
  0 1 2 3 4 5
0 . . . B . .
1 . A . . . #
2 . . . . C .
3 . . @ . a .
4 . . . . . .
5 . . . . . .

    ⋮      (Some moves are skipped to save space. See Blackboard for full version.)

  0 1 2 3 4 5
0 . . . B I .
1 . A J E . b
2 . F f c C H
3 . K D G a d
4 . # . e . .
5 @ . . . . .
Move (knight row col): # 2 0↵
```

```
  0 1 2 3 4 5
0 . . . B I .
1 . A J E . b
2 # F f c C H
3 . K D G a d
4 . g . e . .
5 @ . . . . .
Move (knight row col): @ 4 2↵
  0 1 2 3 4 5
0 . . . B I .
1 . A J E . b
2 # F f c C H
3 . K D G a d
4 . g @ e . .
5 L . . . . .
Move (knight row col): # 0 1↵
  0 1 2 3 4 5
0 . # . B I .
1 . A J E . b
2 h F f c C H
3 . K D G a d
4 . g @ e . .
5 L . . . . .
Move (knight row col): @ 5 4↵
  0 1 2 3 4 5
0 . # . B I .
1 . A J E . b
2 h F f c C H
3 . K D G a d
4 . g M e . .
5 L . . . @ .
No more moves!
```

Submission and Marking

- Your program file names should be `DualKnightsTour.cpp` and `tour.cpp`. Submit the *two files* in Blackboard (https://blackboard.cuhk.edu.hk/). You do *not* have to submit `DualKnightsTour.h`.
- Insert your name, student ID, and e-mail address as comments at the beginning of all your source files.
- Besides the above information, your program should further *include suitable comments as documentation*.
- You can submit your assignment multiple times. Only the latest submission counts.
- Your program should be *free of compilation errors and warnings*.
- *Plagiarism is strictly monitored and heavily punished if proven.* Lending your work to others is subjected to the same penalty as the copier.