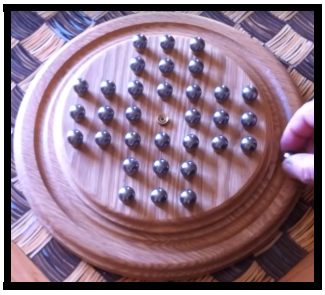
MSc (Computing Science) 2022-2023
C/C++ Laboratory Examination

Imperial College London

Tuesday 10 January 2023, 10h00 – 12h00

"You are never really playing an opponent. You are playing yourself"

Arthur Ashe



- ☞ You are advised to use the first 10 minutes for reading time.
- ☞ Log into the Lexis exam system using your DoC login as both your login and as your password (**do not use your usual password**).
- ☞ You must add to the pre-supplied header file **solitaire.h**, pre-supplied implementation file **solitaire.cpp** and must create a **makefile** according to the specifications overleaf.
- ☞ You will find source files **solitaire.cpp**, **solitaire.h** and **main.cpp**, and data files **board.txt**, **solution.txt**, **board1.txt**, **board2.txt**, **board3.txt**, **target1.txt**, **target2.txt**, **target3.txt**, **target4.txt** and **target5.txt** in your Lexis home directory (**/exam**). If one of these files is missing alert the invigilators.
- ☞ **Save your work regularly.**
- ☞ Please log out once the exam has finished. No further action needs to be taken to submit your files.
- ☞ No communication with any other student or with any other computer is permitted.
- ☞ You are not allowed to leave the lab during the first 15 minutes or the last 10 minutes.
- ☞ **This question paper consists of 6 pages.**

Image Credit: Michael Collinson `https://www.youtube.com/watch?v=kj4kvV7CPVk`

Problem Description

Figure 1: Marble Solitaire starting layout (left) and goal state (right)

As shown in Figure 1, Marble Solitaire is a one-player game that is played on a cross-shaped board with 33 holes. Initially a marble is placed in each hole, with the exception of the centre one, and the goal is to make moves, each of which removes a marble, until there is only one marble left in the centre of the board.

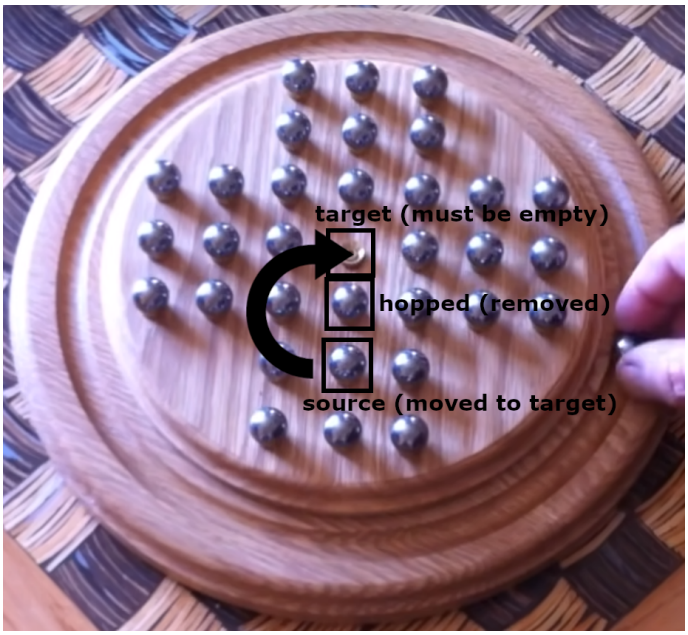


Figure 2: How a Solitaire move is made

As shown in Figure 2, moves are made in a vertical or horizontal direction[1] by hopping some source marble over an adjacent one into an empty target hole that is two positions away. The hopped marble is then removed.

If the board is not in the goal state, and no moves are possible, then the game is lost and the player must restart the game from the starting layout.

[1] Note diagonal moves not are allowed.

Pre-supplied functions and files

You are supplied with a main program in **main.cpp**, and several data files **board.txt**, **solution.txt**, **board1.txt**, **board2.txt**, **board3.txt**, **target1.txt**, **target2.txt**, **target3.txt**, **target4.txt** and **target5.txt** representing board layouts.

You can use the UNIX command **cat** to inspect the data files. For example, the contents of the data file **board.txt** (representing the starting layout), **target4.txt** (representing an intermediate state on path to the goal state) and **solution.txt** (representing the goal state) are, respectively:

```
..OOO..          ..___..          ..___..
..OOO..          ..__O..          ..___..
OOOOOOO          __OOO__          _______
OOO_OOO          _OOOOO_          ___O___
OOOOOOO          __O_O__          _______
..OOO..          ..___..          ..___..
..OOO..          ..___..          ..___..
```

Note how marbles are denoted by '`O`' (i.e. capital letter O, *not* the number zero) characters, unused parts of the board by '`.`' characters and empty holes by '_' (underscore) characters.

You are also supplied with the beginnings of the header file **solitaire.h** (for your function prototypes) and the beginnings of the implementation file **solitaire.cpp** (for your function definitions).

The file **solitaire.cpp** includes the definition of several pre-supplied functions:

- `char **allocate_2D_array(int rows, int columns)` is a helper function that allocates a two-dimensional (`rows` × `columns`) array of characters, returning the 2D array.
- `deallocate_2D_array(char **m, int rows)` is a helper function that frees up the memory allocated for the 2D array `m`.
- `char **load_board(const char *filename, int &height, int &width)` is a function which reads in a board from the file with name `filename`, sets the output parameters `height` and `width` according to the dimensions of the board, and returns a 2D (`height` × `width`) array of characters representing the board.
- `void print_board(char **board, int height, int width)` is a function which prints out the board stored in the 2D (`height` × `width`) array of characters `board`. Row and column labels are also shown (as numbers and letters respectively).

For example the code:

```
char **puzzle = load_board("board.txt", height, width);
print_board(puzzle, height, width);
```

should result in the output:

```
Loading puzzle board from 'board.txt'... done (height = 7, width = 7).
     ABCDEFG
   0 ..OOO..
   1 ..OOO..
   2 OOOOOOO
   3 OOO_OOO
   4 OOOOOOO
   5 ..OOO..
   6 ..OOO..
```

Specific Tasks

1. Write a Boolean function `are_identical(first, second, height, width)` which takes two `height` × `width` 2D arrays (`first` and `second`) and returns true if and only if all elements in the arrays are identical (equal).

   For example, given three files **board1.txt**, **board2.txt** and **board3.txt** with contents:

```
..O..        ..O..        ..O..
.O.O.        .O.O.        .O.O.
OOOOO        OOOOO        OO...
O...O        O...O        O....
O...O        O...O        O....
.....        .....        .....
.....        .....        .....
```

   Then the code:

```
int height1, width1, height2, width2, height3, width3;
char **board1 = load_board("board1.txt", height1, width1);
char **board2 = load_board("board2.txt", height2, width2);
char **board3 = load_board("board3.txt", height3, width3);
assert(height2 == height1 && width2 == width1);
assert(height3 == height1 && width3 == width1);
bool same = are_identical(board1, board2, height1, width1);
cout << "Boards 1 and 2 are " << (same ? "" : "not ") << "identical." << endl;
same = are_identical(board2, board3, height1, width1);
cout << "Boards 2 and 3 are " << (same ? "" : "not ") << "identical." << endl;;
```

   should display the output

```
Loading puzzle board from 'board1.txt'... done (height = 7, width = 7).
Loading puzzle board from 'board2.txt'... done (height = 7, width = 7).
Loading puzzle board from 'board3.txt'... done (height = 7, width = 7).
Boards 1 and 2 are identical.
Boards 2 and 3 are not identical.
```

2. Write a Boolean function `make_move(board, move, height, width)` which attempts to make a move described by a 3-character string `move` on the `height` × `width` puzzle `board`.

   The first character of the move is a letter describing the column of the source marble ('`A`'–'`G`'). The second character is a digit describing the row of the source marble ('`0`'–'`6`'). The third character is a letter describing the direction in which the move should be attempted: '`N`', '`S`', '`E`' or '`W`' for north, south, east and west, respectively.

   For example, the code:

```
char **puzzle = load_board("board.txt", height, width);
cout << "Starting from:" << endl;
print_board(puzzle, height, width);
cout << endl;

cout << "Move D1S is";
if (make_move(puzzle, "D1S", height, width)) {
  cout << " valid" << endl;
  print_board(puzzle, height, width);
} else
   cout << " invalid" << endl;
cout << endl;
```

should display the output

```
Loading puzzle board from 'board.txt'... done (height = 7, width = 7).
Starting from:
     ABCDEFG
   0 ..OOO..
   1 ..OOO..
   2 OOOOOOO
   3 OOO_OOO
   4 OOOOOOO
   5 ..OOO..
   6 ..OOO..

Move D1S is valid
     ABCDEFG
   0 ..OOO..
   1 ..O_O..
   2 OOO_OOO
   3 OOOOOOO
   4 OOOOOOO
   5 ..OOO..
   6 ..OOO..
```

3. Write a Boolean function `find_solution(begin, end, height, width, solution)` which attempts to generate a sequence of **no more than six** valid moves which will transform board `begin` into board `end` (both boards having dimensions `height` × `width`). If a sequence of valid moves can be found, the function should return `true` and output parameter `solution` should contain a comma-separated list of moves. Otherwise the function should return `false`.

   For example, given files **target5.txt**, and **solution.txt** with contents:

```
..___..                    ..___..
..._O_..                   ..___..
___O___                    _______
__OOO__                    ___O___
_______                    _______
..___..                    ..___..
..___..                    ..___..
```

   Then the code:

```
int h,w;
char **first = load_board("target5.txt", h, w);
char **second = load_board("solution.txt", h, w);
char sequence[512];
bool result = find_solution(first, second, h, w, sequence);
if (result)
  cout << "Success! Move sequence: " << sequence << endl;
else
  cout << "No solution found" << endl;
```

   should display the output:

```
Loading puzzle board from 'target5.txt'... done (height = 7, width = 7).
Loading puzzle board from 'solution.txt'... done (height = 7, width = 7).
Success! Move sequence: D3E,D1S,C3E,F3W
```

   For full credit for this part, your function – or helper function if you choose to use one – should be recursive.

(The three parts carry, resp., 20%, 40% and 40% of the marks)

What to hand in

Place your function implementations in the file **solitaire.cpp** and corresponding function declarations in the file **solitaire.h**. Use the file **main.cpp** to test your functions. Create a **makefile** which will compile your submission into an executable file entitled **solitaire**.

Hints

1. You will save time if you begin by studying the `main()` function in **main.cpp**, the pre-supplied functions in **solitaire.cpp** and the given data files.
2. Feel free to define any of your own helper functions which would help to make your code more elegant. This will be particularly useful when answering Questions 2 and 3.
3. Question 3 will be **much** easier and more elegant if you exploit the answer to Questions 1 and 2 in your solution.
4. You are explicitly required to use recursion in your answer to Question 3. You are welcome to use a recursive helper function although using a default parameter can avoid this (e.g. you might want to use a default parameter to control the depth of the recursion). You are not obliged to use recursion in answering any other question; however, you may use it if you feel it would make your solution more elegant.
5. The test harnesses for Questions 2 and 3 in the `main()` function may appear relatively complex at first. You may wish to begin by commenting these out and uncommenting the simple test cases that appear immediately above in the code.
6. It should be possible to produce a full solution with no memory leaks. You can check for memory leaks using the command:

   ```
   valgrind --tool=memcheck --leak-check=full ./solitaire
   ```

7. The answer to each question will be assessed individually and independently, under the assumption that the answers to the other questions are correct. Thus it is a good idea to try to attempt all questions. If you cannot get one of the questions to work, try the next one.

GOOD LUCK!
WITH BEST WISHES FOR
SUCCESS, HEALTH AND HAPPINESS
IN 2023