MSc (Computing Science) 2023-2024
C/C++ Laboratory Examination

Imperial College London

Tuesday 9 January 2024, 10h00 – 12h00

"Every great achievement begins with the decision to try and the commitment to keep pushing until the goal is reached."

Amelia Earhart

- ☞ 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 **sokoban.h**, pre-supplied implementation file **sokoban.cpp** and must create a **makefile** according to the specifications overleaf.
- ☞ You will find source files **sokoban.cpp**, **sokoban.h** and **main.cpp**, and data files **level0.txt**, **level1.txt**, **level2.txt**, **level3.txt** and **level4.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 7 pages.**

Image Credit: Moby Games

Problem Description

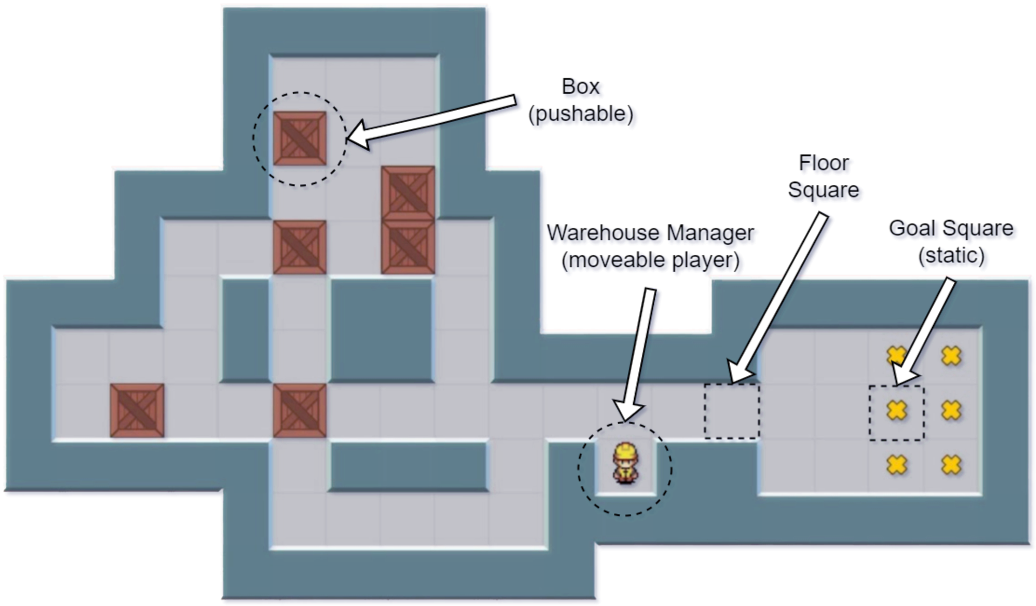

Figure 1: An example Sokoban level

Sokoban is a classic grid-based videogame in which the player takes on the role of a warehouse manager. As illustrated in Figure 1, at each level of the game, the player aims to organise the *boxes* in the warehouse by *pushing* them onto *goal squares*[1].

The player can move one square at a time in four directions: left, right, up and down. The player cannot pass through walls, but can move onto unoccupied floor squares and unoccupied goal squares. The player may also *push* a box in a direction, provided the space behind the box being pushed is an unoccupied floor square or an unoccupied goal square. In this case, both the box and the player move one square in the direction of the push.

The player wins a level when all boxes have been pushed onto goal squares. Conversely, the player loses a level if it becomes impossible to push a box onto a goal square.

wall	`#`
player	`@`
player on goal square	`+`
box	`$`
box on goal square	`*`
unoccupied goal square	`.`
unoccupied floor square	(space)

```
    #####
    #   #
    #$  #
  ###  $##
  #  $ $ #
### # ## #   ######
#   # ## #####  ..#
# $  $          ..#
##### ### #@##  ..#
    #     #########
    #######
```

```
#######
#@ $ .#
# $  .#
#######
```

Figure 2: Elements of Sokoban level textual representation (left), textual representation of the example Sokoban level (middle), and textual representation of a simple Sokoban level (right)

As shown in Figure 2, Sokoban levels can be represented in text format. Solutions can also be represented as a string of concatenated directions, with non-push moves encoded as lowercase letters `l` (left), `r` (right), `d` (down) and `u` (up) and push moves encoded as the uppercase equivalents (i.e. `L`, `R`, `D` and `U`). Thus a solution of the simple Sokoban level on the right in Figure 2 is:

`rRRllldRRR` .

[1]There are always an equal number of goal squares and boxes on every level

Pre-supplied functions and files

You are supplied with a main program in **main.cpp**, and several data files **level0.txt**, **level1.txt**, **level2.txt**, **level3.txt**, and **level4.txt** representing Sokoban levels.

You can use the UNIX command **cat** to inspect the data files. The contents of the data files **level0.txt**, **level1.txt**, **level2.txt**, **level3.txt** and **level4.txt** are, respectively:

```
#######
#@ $ .#
# $  .#
#######
```

```
##########
#        #
#   $ $  #
# ###   ##
#   # ###
 ## # #
#   # #
# ### ###
#  ##   ##
#+      .#
##########
```

```
################
#  #  #  #  #  #
#  .  #  #  #  #
# $#  *  #  #  #
#  #  # @*  #$ #
#  #  #  #  .  #
#  #  #  #  #  #
################
```

```
    ####
#####  #
# $   $#
#  .#. #
## ### ##
 # .#.  #
 #$ @ $ #
 #  #####
 ####
```

```
    #####
    #   #
    #$  #
  ###  $##
  #  $ $ #
### # ## #   ######
#   # ## #####  ..#
# $  $          ..#
##### ### #@##  ..#
    #     #########
    #######
```

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

The file **sokoban.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_level(const char *filename, int &height, int &width)` reads in a level from the file with name `filename`, sets the output parameters `height` and `width` according to the dimensions of the level, and returns a 2D (`height` × `width`) array of characters representing the level.
- `void print_level(char **level, int height, int width)` is a function which prints out the level stored in the 2D (`height` × `width`) array of characters `level`. Row and column coordinates are also shown.
- `uint64_t level_hash(char **level, int height, int width)` is a function which maps levels onto unsigned 64-bit integers. This will be useful for recognising previously encountered level configurations when answering Question 4.

The file **sokoban.h** contains some useful preprocessor directives:

```
// uncomment to activate simple test cases
//#define SIMPLE_TEST

// uncomment to animate solutions (requires make_move)
//#define ANIMATE_SOLUTION

#define SCREEN_HEIGHT 25

#define MAX_SOLUTION_LENGTH 12800
```

You should uncomment the `#define` for `SIMPLE_TEST` if you wish to use the simpler test cases in `main()`. You can uncomment the `#define` for `ANIMATE_SOLUTION` if you wish to animate your solutions (useful when debugging Question 4); in this case, ensure that `SCREEN_HEIGHT` is correctly defined according to your console window height. Finally, `MAX_SOLUTION_LENGTH` defines the maximum length that level solutions should not exceed (relevant to Question 4).

Specific Tasks

1. Write an integer-valued function `goal_squares_without_boxes(level, height, width)` which counts the number of goal squares on a given `height` × `width` Sokoban `level` which do *not* contain a box. Thus, when a level is complete, this function should return 0.

   For example, the code:

   ```
   char **current = load_level("level1.txt", height, width);
   print_level(current, height, width);
   cout << "Number of goal squares without boxes: "
        << goal_squares_without_boxes (current, height, width) << endl;
   ```

   should produce the output:

   ```
   Loading level from 'level1.txt'... done (height = 11, width = 10).
        0123456789
      0 ##########
      1 #        #
      2 #   $ $  #
      3 # ###   ##
      4 #   # ###
      5  ## # #
      6 #   # #
      7 # ### ###
      8 #  ##   ##
      9 #+      .#
     10 ##########
   Number of goal squares without boxes: 2
   ```

2. Write a Boolean function `find_player(level, height, width, row, column)` which finds the coordinates of the player within a given `height` × `width` Sokoban `level`. When the level contains the player, output parameters `row` and `column` should be set to the row and column coordinates of the player respectively, and the function should return `true`. If the level does not contain the player `row` and `column` should both be set to -1, and the function should return `false`.

   For example, the code:

   ```
   char **current = load_level("level3.txt", height, width);
   print_level(current, height, width);
   int row, col;
   bool success = find_player(current, height, width, row, col);
   if (success)
     cout << "Player found at (" << row << "," << col << ")" << endl;
   else
     cout << "Player not found!" << endl;
   ```

   should result in the following output:

   ```
   Loading level from 'level3.txt'... done (height = 9, width = 9).
        012345678
      0     ####
      1 #####  #
      2 # $   $#
      3 #  .#. #
      4 ## ### ##
      5  # .#.  #
      6  #$ @ $ #
      7  #  #####
      8  ####
   Player found at (6,4)
   ```

3. Write a Boolean function `make_move(level, height, width, dir, is_push)` which attempts to move the player in direction `dir` ('`l`', '`r`', '`d`' or '`u`') within a `height` × `width` Sokoban `level`. The function should return `true` if the player can move (either freely or by pushing a box) in the direction `dir`; otherwise the function should return `false`. If the function returns `true`, `level` should be updated to reflect the move, and the output parameter `is_push` should be set to `true` if the player moves by pushing a box. Otherwise `is_push` should be set to `false`.

   For example, the code:

```
char **current = load_level("level4.txt", height, width);
print_level(current, height, width);
cout << "Trying to move in direction 'u'" << endl;
is_push = false;
valid = make_move(current, height, width, 'u', is_push);
if (valid) {
  cout << "Valid move! Is "
       << (!is_push ? "not": "") << " a push." << endl;
  cout << "New level configuration:" << endl;
  print_level(current, height, width);
} else
    cout << "Invalid move." << endl;
```

   should display the output

```
Loading level from 'level4.txt'... done (height = 11, width = 20).
               1
     01234567890123456789
   0      #####
   1      #   #
   2      #$  #
   3    ###  $##
   4    #  $ $ #
   5  ### # ## #   ######
   6  #   # ## #####  ..#
   7  # $  $          ..#
   8  ##### ### #@##  ..#
   9      #     #########
  10      #######
Trying to move in direction 'u'
Valid move! Is not a push.
New level configuration:
               1
     01234567890123456789
   0      #####
   1      #   #
   2      #$  #
   3    ###  $##
   4    #  $ $ #
   5  ### # ## #   ######
   6  #   # ## #####  ..#
   7  # $  $     @    ..#
   8  ##### ### # ##  ..#
   9      #     #########
  10      #######
```

4. Write a Boolean function `solve_level(level, height, width, solution)` which attempts to find a valid solution sequence for a `height` × `width` Sokoban `level`. The output parameter `solution` should be a string (`char *`) containing the relevant concatenated directions with non-push moves encoded as lowercase letters `l` (left), `r` (right), `d` (down) and `u` (up) and push moves encoded as the uppercase equivalents (i.e. `L`, `R`, `D` and `U`). You should be careful that the solution does not exceed `MAX_SOLUTION_LENGTH` characters.

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

   For example, the code:

```
char solution[MAX_SOLUTION_LENGTH];
cout << "Trying to solve level0.txt..." << endl;
char **current = load_level("level0.txt", height, width);
print_level(current, height, width);
if (solve_level(current, height, width, solution)) {
  cout << "Success! Final level configuration:" << endl;
  print_level(current, height, width);
  cout << "Solution is '" << solution << "' (length "
       << strlen(solution) << ")" << endl;
} else
  cout << "no solution!" << endl;
```

   should produce the output:

```
Trying to solve level0.txt...
Loading level from 'level0.txt'... done (height = 4, width = 7).
     0123456
   0 #######
   1 #@ $ .#
   2 # $  .#
   3 #######
Success! Final level configuration:
     0123456
   0 #######
   1 #    *#
   2 #   @*#
   3 #######
Solution is 'rRRllldRRR' (length 10)
```

   Note that other valid (but potentially longer) solutions (such as `dRRRlllurRR`) are also acceptable.

(The four parts carry, resp., 15%, 15%, 30% and 40% of the marks)

What to hand in

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

(P.T.O. for hints)

Hints

1. You will save time if you begin by studying the `main()` function in **main.cpp**, the pre-supplied functions in **sokoban.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 3 and 4.

3. Feel free to use the answer to previous questions in answering subsequent ones. For example, Question 3 will be more elegant if you exploit the answer to Question 2 in your answer. Likewise, Question 4 should be able to exploit your answers to Questions 1, 2 and 3.

4. You are explicitly required to use recursion in your answer to Question 4. You are encouraged to use a recursive helper function for this since you will likely have to pass extra parameters. 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. You will very quickly realise that a brute force approach to Question 4 is too inefficient to solve all but trivial cases. Here are ways to make things more efficient:

   - To avoid needlessly repeating work, remember which level configurations you have seen before. A combination of the `level_hash(...)` function (which you will recall returns an `uint64_t`) and an STL `set<uint64_t>` container should provide you with an easy way to do this[2].
   - Avoid exploring level configurations that are obviously impossible to solve. For example, if a box has been pushed into a non-goal corner, it is going to be impossible to push it out from there, and so there is no point in exploring the level configuration further.
   - The order in which move direction alternatives are tried influences efficiency. You might implement a simple strategy such as encouraging forward progress by always starting your exploration with the direction of the last move. Or you might order move direction alternatives in increasing order of some distance metric[3] based on (a) the position of the boxes on non-goal squares and the position of the unoccupied goal squares (b) the position of the player and the position of the nearest box on a non-goal square.

6. The test harnesses in the `main()` function may appear relatively complex at first. To get access to simple test cases, you may wish to uncomment the `#define SIMPLE_TEST` line in **sokoban.h**.

7. Using the **-O3** command line option for **g++** turns on many compiler optimisations that will likely speed up the execution of your code. This could be handy when attempting to solve some of the more ambitious Sokoban levels.

8. 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 2024

[2]Useful methods on an STL `set` include `clear()`, `insert(`*item*`)` and `count(`*item*`)`.
[3]For example, the *Manhattan distance* between points (r_1,c_1) and (r_2,c_2) is $|r_1 - r_2| + |c_1 - c_2|$.