The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2018
CSSE2310 / CSSE7231 - Assignment 1
Due: 11:10pm 17 August, 2018
Marks: 50
Weighting: 25% of your overall assignment mark (CSSE2310)
Revision: 1.2

Introduction

Your task is to write a C99 program (called `fitz`) which allows the user to play a game(described later). This will require I/O from both the user and from files. Your assignment submission must comply with the C style guide (v2.0.4) available on the course website.

This assignment is to be your own individual work. Using code which you did not write[1] is against course rules and may lead to a misconduct charge.

The Game

The game begins with an empty board, displayed like this:

```
.....
.....
.....
.....
```

The two players in the game will take turns getting a tile from a file and placing it onto the board. Tiles can be rotated clockwise (in multiples of 90°) before being placed. For example, a tile shaped like this:

```
****
```

could be placed in any of the following ways (as well as others):

```
*....      .....      ....*      .....
*....      .....      ....*      .....
*....      ****.      ....*      .....
*....      .....      ....*      .****
```

The first player will indicate their tiles with `*` and the second player will use `#`. A tile can not be placed where it will overlap with existing tiles on the board. The game ends when a player can not place their next tile. That player loses.

For example, the `*` player needing to place

```
**
**
```

on a board that looks like:

[1] See the first lecture for exceptions

```
#####
.**##
**..#
.****
```

would lose the game.

Note: the tiles must be used in the order they appear in the file. If you run out of tiles, start again at the beginning of the file.

Interaction

At the beginning of the game and after each player's turn, the board will be displayed. There are three types of players in the game,

`h` where moves will be read from `stdin`.

1 or 2 where moves will be generated by your program.

Types 1 and 2 will be referred to collectively as "automatic players". Their moves will be printed like this (substitute # for * as needed):
`Player * => 1 0 rotated 90`
then go to the next player's turn. For players reading from stdin, the tile the user must place is displayed, followed by a prompt:
`Player *]`
(there is a space following the])
and wait for the user to enter *row column rotate* (single space separated with no leading nor trailing spaces). For example:

```
,!,,,
,!!,,
,,!,,
,,!,,
,,,,,
Player *]
```

Note that, row and column describe the middle of the tile (described as a 5×5 grid). For illustration purposes only, the middle point is indicated below with an @.

```
!!!,,
!!,,,
,,@,,
,,,,,
,,,,,
```

To place this tile like this:

```
........
..@.....
...**...
..***...
```

would require an input of `1 2 180` (@ for illustration purposes only).

If the user's input is empty or invalid, reprompt immediately (without redisplaying the board). Some reasons the input could be invalid are:

- it is not precisely three space separated integers
- placing the tile at that position would result in a non-empty part of the tile being off the board. (In the example above, `3 1 0` would be legal, but `1 1 0` would not.)
- Placing the tile at that position would cause it to overlap with a tile already on the board.
- rotate is not one of $\{0, 90, 180, 270\}$

Note: row and column do not necessarily need to be on the board for a move to be legal. For example: playing the following shape as -1 -1 0

```
,,,,,
,,,,,
,,@,,
,,,!!
,,,!!
```

on this board:

```
.....
.....
!!!!.
```

results in

```
##...
##...
!!!!.
```

After each valid move is chosen, whether automatically or manually, the updated board will be displayed.

When a winner is determined, the following is displayed:

```
Player ? wins
```

(substitute */# for ? as appropriate).

Invocation

When run with an incorrect number of arguments, `fitz` should print usage instructions to stderr:

```
Usage: fitz tilefile [p1type p2type [height width | filename]]
```

and exit (see the error table).

`tilefile` is the filename of a file containing the tiles to be used in this game. `p1type` and `p2type` must be either `h`, `1` or `2`. If five command line arguments are given, the last two should be interpretted as integers between 1 and 999 inclusive. If only four command line arguments are given, the last one should be interpretted as the name of a file containing a saved game to load. If only one argument is given, the contents of the tile file should be output to standard out as described below.

Note: End of game checks should be performed before each player needs to choose a move (either manually or automatically).

Tile file format

The tile file should contain at least one tile, tiles after the first have a blank line separating them from the previous tile. Each tile is described by a 5×5 grid of characters (`'\n'` terminated). A `'!'` indicates part of the tile which is present while, `','` indicates part of the tile which is empty. See below for an example file.

Note: read the whole tile file into memory rather than trying to go back to the file each time.

Shape display

For each tile in the file (in order), print the tile with its three rotations next to it (space separated). Put a blank line between each row of tiles.

For example, a file which contains:

```
,,,,,
,!!,,
,!!!,
,,,,,
,,,,,

,,,,!
,,,,!
,,,,!
,,,,!
,,,,!
```

Would display as:

```
,,,,, ,,,,, ,,,,, ,,,,,
,!!,, ,,!!, ,,,,, ,,!,,
,!!!, ,,!!, ,!!!, ,!!,,
,,,,, ,,!,, ,,!!, ,!!,,
,,,,, ,,,,, ,,,,, ,,,,,

,,,,! ,,,,, !,,,, !!!!!
,,,,! ,,,,, !,,,, ,,,,,
,,,,! ,,,,, !,,,, ,,,,,
,,,,! ,,,,, !,,,, ,,,,,
,,,,! !!!!! !,,,, ,,,,,
```

Saved game file format

The first line of a saved game file will contain parameters separated by spaces. The remainder of the file will store the board. Every line of the file is to be terminated with '\n'.

The parameters are (in order):

- number next tile to play (starting from zero).
- the next player to have their turn — 0 or 1
- the number of rows in the board
- the number of columns in the board

For example:

```
3 1 4 3
*..
..#
..#
**#
```

The board above describes a game with a 4×3 board where the # player is next to play and they will try to place the fourth[2] tile in the file.

Things to watch out for:

- Files with fewer lines than the grid size requires.
- Files with short lines (less content than expected).
- Files with lines which are longer than expected.
- Files with characters in the wrong places.
- Next player being something other than 0 or 1.
- Next tile expecting more tiles than are actually in the tile file.

Automatic player moves

The automatic players will generate their moves using the following algorithms. Automatic players should continue generating moves until a legal placement is found or until all possible placements have been tried.

Type 1:

1. Let r_{start}, c_{start} be the most recent legal play by either player (or $-2, -2$ if no moves have been made yet).
2. Let $r = r_{start}, c = c_{start}$
3. Let θ =zero
4. Repeat
5. Repeat
6. If r, c, θ is a valid placement, stop.
7. Add one to c
8. If c is larger than maximum column(+2), set c to -2 and add one to r.
9. If r is larger than maximum row(+2), set r to -2.
10. Until $r = r_{start}$ and $c = c_{start}$
11. Add 90 to θ
12. Until $\theta > 270$

[2]Tile numbers start at 0

Type 2: For this type, the starting location and search direction changes depending on which player this is. The first player will start in the top left corner $(-2, -2)$ and search left→right, top→bottom. The second player will start at the bottom right corner $(\text{rows} + 2, \text{cols} + 2)$ and search right→left, bottom→top.

1. Let r_{start}, c_{start} be the most recent legal play by this player.
2. Let $r = r_{start}, c = c_{start}$
3. Repeat
4. Let θ =zero
5. Repeat
6. If r, c, θ is a valid placement, stop.
7. Add 90 to θ
8. Until $\theta > 270$
9. Move to next position (wrapping as needed)
10. Until $r = r_{start}$ and $c = c_{start}$

Saving games

To save a game, instead of entering a row and column, enter `save` followed immediately by the path of the file to save to. That is, no space between `save` and the start of the path.

Errors and messages

When one of the conditions in the following table happens, the program should print the error message and exit with the specified status. All error messages in this program should be sent to standard error and followed by a newline. Error conditions should be tested in the order given in the table.

Condition	**Exit Status**	**Message**
Program started with incorrect number of arguments	1	Usage: fitz tilefile [p1type p2type [height width | filename]]
Tile file can't be read	2	Can't access tile file
Tile file is incorrect	3	Invalid tile file contents
Invalid player type	4	Invalid player type
Invalid height or width (< 1)	5	Invalid dimensions
Save file can't be read	6	Can't access save file
Invalid save file contents	7	Invalid save file contents
End of file while waiting for user input	10	End of input

If end of input is encountered mid way through processing a line, process that line as normal. For a normal exit, the status will be 0 and no special message is printed.

There are a number of conditions which should cause messages to be displayed but which should not immediately terminate the program. These messages should also go to standard error.

Condition	**Action**	**Message**
Error opening file for saving save game	Prompt again	Unable to save game

Compilation

Your code must compile with command:
`make`
When you compile, you must use at least the following flags: `-Wall -pedantic -std=c99`.

You must not use flags or pragmas to try to disable or hide warnings.

Specification Updates

It is possible that this specification contains errors or inconsistencies or missing information. Clarifications may be issued via the the course discussion forum. Any such clarifications posted 5 days (120 hours) or more before the due date will form part of the assignment specification. If you find any inconsistencies or omissions, please notify the teaching staff.

Test Data

Testing that your assignment complies with this specification is your responsibility. Some test data and scripts for this assignment will be made available.
`testa1.sh` will test the code in the current directory. `reptesta1.sh` will test the code which you have in the repository. The idea is to help clarify some areas of the specification and to provide a basic sanity check of code which you have committed. They are not guaranteed to check all possible problems nor are they guaranteed to resemble the tests which will be used to mark your assignments.

Submission

Submission must be made electronically by committing using subversion. In order to mark your assignment the markers will check out `https://source.eait.uq.edu.au/svn/csse2310-s???????/trunk/ass1`. Code checked in to any other part of your repository will not be marked.

Note: No late submissions will be accepted for this assignment (see ECP). The markers will evaluate the last commit in your repository before the deadline. Late by 1 minute (or less) is still late.

Marks

Marks will be awarded for both functionality and style.

Functionality (42 marks)

Provided that your code compiles (see above), you will earn functionality marks based on the number of features that your program correctly implements (as determined by automated testing), as outlined below. Partial marks may be awarded for partially meeting the functionality requirements[3]. Not all features are of equal difficulty. If your program does not allow a feature to be tested then you will receive 0 marks for that feature, even if you claim to have implemented it. For example, if your program can never open a file, we can not determine if your program would have loaded input from it. The markers will make no alterations to your code (other than to remove code without academic merit). Your programs should not crash or lock up/loop indefinitely. Your programs should not take a long time to complete.

[3] ie passing some but not all of the tests of a feature

Please note that some features referred to in the following scheme may be tested in other parts of the scheme. For example, if you can not display the initial board, you will fail a lot of tests, not just the one which refers to "initial board". Students are advised to pay close attention to their handing of end of input situations.

- Command args — correct response to
  - incorrect number of args (1 mark)
  - problem with tile file (2 marks)
  - problems with save file (2 marks)
  - other invalid args (2 marks)
- Correctly display tile file (5 marks)
- Correctly process first move by human player (3 marks)
- Correctly make first move by automated players
  - Type 1 (3 marks)
  - Type 2 (3 marks)
- Reject invalid moves (2 marks)
- Save game (3 marks)
- Load saved games (4 marks)
- Play complete games (12 marks)

Style (8 marks)

Style marks will be calculated as follows:
Let A be the number of style violations detected by simpatico plus the number of build warnings. Let H be the number of style violations detected by human markers. Let F be the functionality mark for your assignment.

- If $A > 10$, then your style mark will be zero and M will not be calculated.
- Otherwise, let $M_A = 4 \times 0.8^A$ and $M_H = M_A - 0.5 \times H$ your style mark S will be $M_A + \max\{0, M_H\}$.

Your total mark for the assignment will be $F + \min\{F, S\}$.

A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name) up to the maximum of five. You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final — it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style mark can never be more than your functionality mark — this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

Sample games

```
 ./fitz d1 1 1 6 5
.....
.....
.....
.....
.....
.....
Player * => 1 1 rotated 0
**...
***..
.....
.....
.....
.....
Player # => 2 1 rotated 0
```

```
**.#.
***#.
...#.
...#.
...#.
.....
Player * => 3 1 rotated 0
**.#.
***#.
**.#.
***#.
...#.
.....
Player # => 3 2 rotated 0
**.#.
```

```
***##
**.##
***##
...##
....#
Player * => 5 1 rotated 0
**.#.
***##
**.##
***##
**.##
***.#
Player * wins
```

```
 ./fitz d1 h 1 5 6
......
......
......
......
......
,,,,,
,!!,,
,!!!,
,,,,,
,,,,,
Player *] 1 1 90
.**...
.**...
.*....
......
......
Player # => 2 -2 rotated 0
```

```
#**...
#**...
#*....
#.....
#.....
,,,,,
,!!,,
,!!!,
,,,,,
,,,,,
Player *] aarg
Player *] 3 2 270
#**...
#**...
#**...
#**...
#**...
Player # => 2 1 rotated 0
```

```
#**#..
#**#..
#**#..
#**#..
#**#..
,,,,,
,!!,,
,!!!,
,,,,,
,,,,,
Player *] 3 5 270
#**#..
#**#..
#**#.*
#**#**
#**#**
Player * wins
```

Notes and tips

1. A well written program should gracefully handle any input it is given. We should not be able to cause your program to `crash`.
2. You can assume that no `valid` line of input contains more than 70 characters.
3. Remember that the functionality of your program will be marked by comparing its output for certain inputs against the expected output. Your program's output must match exactly.
4. Be sure to handle unexpected end of file properly. Many marking tests will rely on this working.
5. Debug using small boards if possible.

6. Do not hardcode board dimensions.
7. You are expected to check for inability to open files, but do not need to consider other system call failures such as malloc fails.

Changes

Changes $1.1\zeta \rightarrow 1.2$

- Corrected human prompt in example

Changes $1.1 \rightarrow 1.1\zeta$

- Fixed commandline argument counts in the invocation section. Counts do not include argv[0].

Changes $1.0 \rightarrow 1.1$

- Fixed incorrect year in due date
- Fixed incorrect possibilities for player types in invocation.
- Fixed minor spelling error.
- Added sample games.
- Game over check changed to be done before each player is asked to choose a new move. This wording is simpler than it was before and should be equivalent.
- Valid dimensions must be ≥ 1.
- Corrected wrapping instructions in Type 1.