THE UNIVERSITY OF SYDNEY

COMP2129 Assignment 2

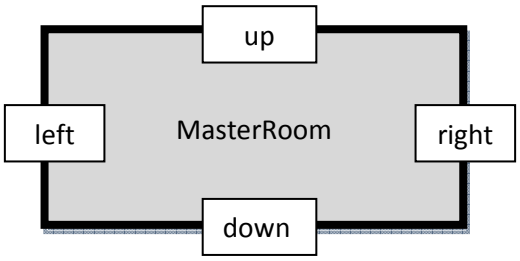
Due: 11:59pm Friday 8th April

Task

Design and implement the computer game *Doughnut World* in C, using dynamic data structures. Full marks can only be achieved if your program has no dangling references nor memory leaks.

Doughnut World

The computer game *Doughnut World* is a game in which the player assumes the role of a lost person trying to escape from *Doughnut World*. *Doughnut World* consists of rooms which are connected by doors[1]. The doors of a room are addressed by their location, i.e., left, right, up, down. For example, the room below has the name *MasterRoom* and all four doors are depicted:



The player can either move left, right, up, and down to enter another room, or can consume doughnuts and milkshakes. Moves might not be successful, i.e., if a door does not connect to another room and the player attempts to move through that door, the player will stay in the current room without losing the game.

At the beginning of the game, the person is placed in the start room. From the start room the person can move from one room to the next room until the person finds the exit room. If the player finds the exit room, the player will win the game and the game terminates.

For each move the player burns energy and loses fluids. The player may consume *doughnuts* and *milkshakes* to overcome the loss of energy and fluids, respectively. Every move the player makes, successful or otherwise, burns *one* doughnut and *one* milkshake.

[1]The rooms are not necessarily in a geometric space also known as a planar layout; we assume a topology only.

Doughnuts and milkshakes are initially deposited in the rooms of the game. These deposits will not be refilled once consumed by the player.

A player has a limit on how many doughnuts and milkshakes they can fit in their stomach. The players stomach can hold only 3 doughnuts and only 2 milkshakes. If a player consumes more doughnuts and milkshakes than their stomach can hold, the player will burst and the game will terminate. Conversely, if a player moves without having at least one doughnut *and* one milkshake in the stomach, the player will lose and the game will terminate.

When a player starts playing the game, the stomach is empty and the player needs to consume at least one doughnut and one milkshake to move on.

Implementation Details

The computer game is text base, i.e., the game is defined first and then the moves of the player will follow. The user-interface is done via standard input/output using the C functions: **`putchar()`**, **`printf()`**, and **`scanf()`**.

The header file **`dw.h`** defines the dynamic data structures you need for this game. Here is the contents of **`dw.h`**:

```
1  #ifndef DW_H_
2  #define DW_H_
3
4  // for directions for door placement in a room.
5  enum direction {UP=0, RIGHT=1, DOWN=2, LEFT=3};
6
7  // data structure to store information about a room.
8  struct room {
9    char name[21];           // name of the room
10   int num_doughnuts;       // number of doughnuts in the room
11   int num_milkshakes;      // number of milkshakes in the room
12   struct room *doors[4];   // doors of the room for each direction
13                            //  doors[LEFT] -> left door
14                            //  doors[RIGHT] -> right door
15                            //  doors[UP] -> door up
16                            //  doors[DOWN] -> door down
17 };
18
19 // data structure is a singly-linked list to store all rooms.
20 struct list {
21   struct room *room;    // pointer to room
22   struct list *next;    // next node in the list
23 };
24
25 #endif
```

All rooms are stored in a singly-linked list. Use a variable that points to the first element in the linked list, e.g., **`struct room_list *base;`**. The **`room`** data structure contains the name of the room, the number of available doughnuts and milkshakes, and a pointer array that contains 4 pointers pointing to 4 possible adjacent rooms. Use the **`enum`** data type to access the four directions of the pointer array.

You will need to build and maintain the data structure for defining the game and for playing the game.

In order to pass the test cases, your program *must* free all of the dynamic memory it allocates. As part of the testing process, this will be automatically checked via **`valgrind`**. Even if your program produces the correct output for a particular test case, if it does not free all of the memory it allocates, your program will fail the test case.

Input Data Format

The input data format is strict and you will be given test cases to make sure that you follow the input data format:

```
<number of rooms>
<room-name> <number of doughnuts> <number of milkshakes>
...
<number of doors>
<room-name> {L|R|U|D} <room-name>
...
<start-room-name> <exit-room-name>
<move 1>
<move 2>
...
```

The input file starts with the number of rooms followed by the room definitions. Each room has a unique name, number of doughnuts and milkshakes. The number of doughnuts and milkshakes needs to be non-negative. Additionally, the maximum room name length is 20 characters long.

The next section defines the doors of the game. First, the number of doors is defined. Note that each door is only defined once; that is, you need to connect the two rooms in both directions. For example, a door declaration `secret-room L bathroom` implies that `secret-room` has a door to the left which connects to the `bathroom`, and the `bathroom` has a door to the right which connects to `secret-room` in opposite direction.

Note that a subsequent door declaration cannot overwrite a previous door declaration and the program has to report an error.

After defining the rooms, the name of the start room and exit room is defined. The start and exit room need to be defined as rooms before.

Following this, the moves the player makes are then stated until the end of the file. Between each movie, the current room name along with the available numbers of available doughnuts and milkshakes are printed to the screen. In addition, the numbers of doughnuts and milkshakes in the stomach of the player are also printed. The moves are as follows:

- `L` move to the left if there is a door; otherwise stay in the current room. Consumes one doughnut and one milkshake even if the move was not successful.
- `R` move to the right if there is a door; otherwise stay in the current room. Consumes one doughnut and one milkshake even if the move was not successful.

- `U` move a room up if there is a door; otherwise stay in the current room. Consumes one doughnut and one milkshake even if the move was not successful.
- `D` move a room down if there is a door; otherwise stay in the current room. Consumes one doughnut and one milkshake even if the move was not successful.
- `G` eat a doughnut. There needs to be at least one doughnut in the room and the stomach needs to have the capacity to hold the doughnut, otherwise player will lose the game.
- `M` drink a milkshake. There needs to be at least one milkshake in the room and the stomach needs to have the capacity to hold the milkshake, otherwise player will lose the game.

Below you find an example input:

```
2
bathroom 2 2
secret-room 1 1
1
secret-room L bathroom
bathroom secret-room
G
M
L
G
M
R
```

Output Data Format

If any part of the input format for defining the game is incorrect, print **`error`**, and terminate the program. This includes incorrect input format, doubly-defined doors, references to rooms (whilst constructing doors) that have not previously been defined, etc.

Your program should print **`lost`** and terminate if any of the following situations occur:

- The player attempts to move when they have run out of fluid or energ.
- The player attempts to consume a resource which does not exist
- The player consumed too many resources such that they burst
- All of the players moves are sucessfully read in but the player did not end up in the end room

If the player reaches the exit room, print **`won`** and terminate the program.

In between each move the player makes, your program needs to print the status information. This status line contains five pieces of information, each one separated by a single space, and the entire line terminated by a newline character. The five pieces of information are:

- The room name
- The number of doughnuts in the current room
- The number of milkshakes in the current room
- The number of doughnuts in the players stomach
- The number of milkshakes in the players stomach

The expected output for the previous example input file is as follows:

```
bathroom 2 2 0 0
bathroom 1 2 1 0
bathroom 1 1 1 1
bathroom 1 1 0 0
bathroom 0 1 1 0
bathroom 0 0 1 1
secret-room 1 1 0 0
won
```

Writing your own test cases

Since your assignment will be automatically marked, it is crucial that you follow our instructions carefully. Your output will need to be in exactly the right format. To assist with this, we have made available some sample test cases. If you are logged in to the **`ucpu[01]`** machines, you can run these tests using the Makefile we provided by typing **`make test`**. If your output is incorrect, you will see both the expected output and your output. If you passed the test, then you will only see the name of the test.

The sample test cases are by no means exhaustive. You will need to test your code more thoroughly by thinking carefully about the specifications and writing your own tests.

As with the previous assignment, you can view the sample test cases manually. They are located in the folder **`~comp2129/assignment2/sample/question1`**.