THE UNIVERSITY OF SYDNEY

COMP2129 Assignment 3

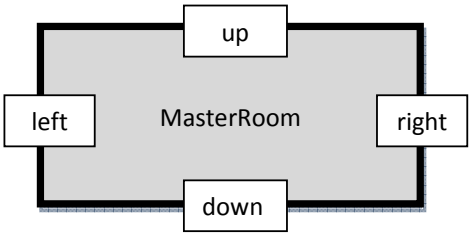
Due: 11:59pm Friday, 10th May 2013

Task description

In this assignment we will implement a game known as "Doughnut World" in the C programming language using dynamic data structures, ensuring that no dangling references or memory leaks occur.

In each game, the player assumes the role of a lost person trying to escape from Doughnut World. The game consists of a defined set of rooms that are connected by portals in a topological layout. The portals of a room are addressed by their location, either left, right, up or down.

For example, a room with four portals is shown below:



The input of your program consists of a series of instructions that are read from standard input. In the first part, the rooms and portals are defined along with the start and end rooms. The second part involves playing the game in which a player can either move left, right, up or down and attempt to enter another room or consume either doughnuts or milkshakes.

At the beginning of the game, the player begins in the start room. From the start room the person can move from one room to another connected room until the player finds the exit room. If the player finds the exit room, the player will win the game.

Each move the player makes, successful or otherwise consumes *one* doughnut and *one* milkshake. The player may consume *doughnuts* and *milkshakes* to overcome this loss of energy and fluids.

A player has a limit on how many doughnuts and milkshakes they can fit in their stomach, holding 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.

When a player begins the game, their stomach is empty and the player will need to consume at least one doughnut and one milkshake to move. Doughuts and milkshakes are initially deposited throughout the rooms, these deposits will not be refilled once they have been consumed by the player.

Implementation details

The header file **dw.h** defines the dynamic data structures needed to implement the game.

The contents of the header file **dw.h** is shown below:

```
1  #ifndef DW_H_
2  #define DW_H_
3
4  /* type definitions */
5  typedef struct room room;
6  typedef struct node node;
7  typedef struct list list;
8
9  /* for directions for portal placement in a room */
10 enum direction {UP=0, RIGHT=1, DOWN=2, LEFT=3};
11
12 /* data structure to store room information */
13 struct room {
14   char name[21];       /* room name */
15   int num_doughnuts;   /* number of doughnuts in the room */
16   int num_milkshakes; /* number of milkshakes in the room */
17   room* portal[4];     /* portals of the room for each direction */
18 };
19
20 /* doubly linked list node to store each room */
21 struct node {
22   node* next; /* pointer to next room in list */
23   node* prev; /* pointer to previous node in list */
24   room* room; /* pointer to room data structure */
25 };
26
27 /* doubly linked list to store all rooms */
28 struct list {
29   int length; /* length of the entire list */
30   node* head; /* pointer to the head of the list */
31   node* tail; /* pointer to the tail of the list */
32 };
33
34 #endif
```

All of the rooms are to be stored in a doubly linked list. You will need to build and maintain these data structures for defining and playing the game, keeping track of all rooms and the head and tail nodes.

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. It is recommended that you use the **enum** data type to access the four directions of the pointer array.

In order to obtain full marks, your program *must* free all of the dynamic memory it allocates. This will be automatically checked using **valgrind**. Even if your program produces the correct output for a given test case, if it does not free all the memory it allocates it will fail that test case.

Program input

The program input consists of two sections . The first section defines the rooms and portals and the second section describes the moves of the player . Each instruction is given on one line of input.

The input format is shown below:

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

The input starts with the number of rooms followed by the room definitions. Each room has a unique name, and a given number of doughnuts and milkshakes. The number of doughnuts and milkshakes cannot be negative. In addition, the maximum room name length is 20 characters.

After defining all the rooms, we then begin to define the portals within the game. First, the number of portals is defined. Each portal is defined only once with rooms being connected in both directions.

For example, a portal declaration `bedroom L bathroom` implies that `bedroom` has a portal to the left which connects to the `bathroom` and the `bathroom` has a portal to the right which connects to `bedroom` in the opposite direction. Note that a subsequent portal declaration cannot overwrite one declared previously, an error must be reported even if the two are the same.

After defining the rooms and portals, we then set two previously defined rooms as the start and exit.

Between each move the player makes the current status of the player is to be outputted.

The possible moves are as follows:

- `L` move to the left room; if it does not exist then stay in the current room
- `R` move to the right room; if it does not exist then stay in the current room
- `U` move to the room above; if it does not exist then stay in the current room
- `D` move to the room below; if it does not exist then stay in the current room
- `G` eat a doughnut; if there is one in the current room and the stomach has enough capacity
- `M` drinks a milkshake; if there is one in the current room and the stomach has enough capacity

The input should be processed one line at a time – after reading a line of input your program should immediately act on the line of input and output accordingly before proceeding to the next line of input. We recommend the use of **`fgets`** and **`sscanf`** to receive input from *stdin*.

Sample input

In the following test case we construct a simple map with 2 rooms. The player successfully maintains their craving for doughnuts and milkshake, navigating through the *maze* reaching the end room.

```
2
bathroom 2 2
bedroom 1 1
1
bedroom L bathroom
bathroom bedroom
G
M
L
G
M
R
```

The corresponding output for the given input is shown below:

```
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
bedroom 1 1 0 0
won
```

Program output

After each move the player attempts to make, your program must output the status of the player. This status must be outputted at the *start* of the game, before the player has moved and also before the *lost* message, if the player loses the game. The player status contains five pieces of information, each one separated by a single space and the entire line being terminated by a newline character.

The five pieces of information within a status 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

If any of the given input is invalid, then you must output **`error`** and terminate the program. This includes any incorrect input not matching the given format, portals being defined multiple times, references to rooms not previously defined when constructing portals, invalid moves, etc.

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

- the player attempts to consume a resource that does not exist
- the player attempts to move when they have run out of fluid or energy
- the player has consumed so many doughnuts or milkshakes that they asplode
- all moves the player makes are read in, but the player does not complete the maze

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

Test cases

Most of the marking is done automatically, and because of this it is important that you follow the specifications of the assignment very carefully. Your program output must be in a very specific format.

To assist you with the testing of your program, a testing tool known as **`haste`** has been developed and allows you to test the correctness of your program, the coverage of your test cases and whether your program has any memory leaks. In addition, we have also included a single test case called **`sample.in`** in the **`tests/`** directory. This test case defines a basic game which is then played.

You can run the correctness, coverage and memory leak testing tools by running:

```
$ make haste
```

Writing your own test cases

We only provide you with one sample test case, but this does not test all the functionality described in assignment spec. Part of this assignment is for you to write your own test cases, based on the sample we provide. This is to ensure that your program will work correctly for any valid or invalid input.

You should place all of your test cases in the **`tests/`** directory. Each testcase should be in the form of *sample.in*. We recommend that the names of your test cases are descriptive so that you know what each is testing, e.g. **`name-too-long.in`** and **`incomplete-game.in`**

We will also be assessing the *coverage* of your test cases, this refers to the percentage of code that the execution of your code passes through. Coverage is an important metric in testing because lines of code that are not covered by a set of test cases are (by definition) never tested. This makes them susceptible to bugs that are difficult to detect. Cases where your coverage may be incomplete include:

- A room with a name of more than **`20`** characters is never tested.
- An *incomplete* game is never tested, where the player does not reach the **`end`** room.

In both of these cases, there could be bugs that may still exist in your code even though all of your other tests pass, as you are not testing all the possible branches of the prototype.

To achieve 100% coverage, you will need to carefuly read the assignment specification and come with test cases that check every single aspect of the task – including valid, invalid and incorrect inputs.

Hint: You can use the included Makefile to check the correctness of your program using your own test cases, and also report on the overall coverage that you have against the staff prototype.
See “The Makefile” section for more details.

The Makefile

We have provided you with a Makefile that can be used to compile and test your code using the test cases you define. The included Makefile will only run correctly on the **`ucpu0`** and **`ucpu1`** machines.

- **`$ make`**
  will compile your **`dw.c`** file into an executable file **`dw`**
- **`$ make test`**
  will compile your program and then check the correctness of your program using the test cases that you define in your **`tests/`** directory and highlight any differences between the **`expected`** output of the staff prototype and the **`observed`** output of your program.
- **`$ make cover`**
  will determine the **`coverage`** obtained using all of the test cases defined in your **`tests/`** directory against the staff prototype, and also output some hints on what is not being covered.
- **`$ make rain`**
  will compile your program and then check whether your program has any memory leaks using the test cases that you define in your **`tests/`** directory. If any test cases require further investigation then run valgrind on each test, e.g. **`$ valgrind ./dw < sample.in`**

Marking

5 **marks** are assigned based on automatic tests for the *correctness* of your program.
This component will use our own (hidden) test cases that cover every aspect of the specification. Note that your test cases will also be assessed and used in this component.

3 **marks** are assigned based on automatic tests for your test case *coverage* of the specification.
The minimum required coverage is 70%, every percent above 70% will attain 0.1 marks.

2 **marks** are assigned based on a manual inspection of the style and quality of your code.

Warning: Any attempts to deceive the automatic marking will result in an immediate 0 for the entire assignment. Negative marks can also be assigned from the manually marked component if your code is unnecessarily and/or deliberately obfuscated.