Maze solver

Introduction

Mazes and labyrinths have fascinated people for centuries. The first reference to a maze can be found in Greek mythology where Daedalus is said to have build a maze to trap the mythical Minotaur. Throughout the middle ages mazes have been popular and even now you can find them in amusement parks or public places.

Assignment

For this programming assignment you have to implement your own maze solver. The assignment can be broken down into three main parts:

- The data structures used to represent the walker and the maze.
- The support functions that deal with the walker and the maze.
- The maze solvers functions.

Data structures

First you will have to define the data structures used by the maze solver. The two main data types are `maze_t` and `walker_t`. For this assignment we provide you with some incomplete skeleton code that you can use as a starting point for your program. You can download the code as a tar file from the course website.

For the maze you have to complete the `maze_t` typedef in `maze.h`. The structure should contain the number of rows and the number of columns of the maze. The structure should also contain the maze itself as a 2D-array of characters (see tips in the last section). The file that contains the actual maze lists the number of rows and columns in the first line. This makes it easy to dynamically allocate the memory needed for the 2D-array when you read the maze from file. The `maze.h` file also contains some useful predefined constants that you can use.

An actual input maze will be in ASCII format. A small example maze file (`map1.txt`) is:

```
6,10
##########
#    #   #
# ##   #S#
# #######
#       E#
##########
```

The first line has the format: `rows,columns` Walls are presented as #'s, the start position of the walker as `S` and the exit as `E`. A maze should have walls all around it, this makes it easy to check moves later on.

In the header file `walker.h` you need to complete the typedef `walker_t` that represents the walker. It should contain the position of the walker in the maze.

Support functions

The support functions are used by the main function and the solvers to deal with data structures. The support functions for the maze are:

init_maze()

This function allocates and initialises a maze of the desired size. The arguments are the number of rows and columns and it should return a pointer to a `maze_t` structure.

read_maze()

The `read_maze()` function takes the filename of the maze file and returns a pointer to a filled `maze_t` structure. This function will use `init_maze()`.

print_maze()

Takes a `maze_t` pointer and a `walker_t` pointer. Print the maze together with the step count to the terminal. Mark the current position of the walker with an `x`. This way you can see your walker move through the maze. This is fun and also useful for debugging your code (less fun).

cleanup_maze()

Takes a pointer to a maze and frees the memory.

The support functions for the walker are:

init_walker()

The initialisation function of the walker takes a pointer to a filled maze structure and returns a pointer to an initialised walker structure. The maze is used to find the initial position of the walker.

check_move()

Takes as arguments: a maze pointer, a walker pointer and a direction. Returns an integer indicating if the move is valid.

move_walker()

Takes as arguments: a maze pointer, a walker pointer and a direction. Performs the move if it's valid, otherwise returns a value signaling failure.

at_exit()

Returns true if the walker is at the exit of the maze. It needs a maze pointer and walker pointer as arguments.

cleanup_walker()

Takes a pointer to a walker and frees the memory.

Maze solvers

The tar file with the skeleton code also contains `main.c` with an incomplete `main()` function. You can use this function as a starting point. The main function should solve the input maze with each of your solver algorithms. You should implement at least the following two maze solvers:

Random walker

This is the simplest maze solver, and a good one to start with. The walker just chooses a random direction for every move and hopes to reach the exit by chance. You could think of slightly smarter versions of the random walker, maybe one that does not go back to the previous location unless there are no other options.

Wall follower

This walker uses the well known trick of following the wall. Image the walker always touching the wall with his left hand. If the maze is connected (eg. no disconnected islands) this method will always lead you to the exit.

You are of course encouraged to implement more advanced maze walkers. Some solvers can find the exit on disconnected mazes where the wall follower might fail. They might also be able to find the exit quicker.

You are free to implement any maze walker that you want, but there are some rules. The maze walker is not allowed to use any global knowledge about the maze. But you are allowed to leave 'breadcrumbs' or other markings on the map if you want to. Basically it should be the same as if you yourself would be dropped into a maze without a map and had to find the exit.

You should print the maze after every move. If you resize your terminal to the height of the maze it allows you to animate the walker moving through the maze. Just redirect the output of your solver to a file and

use a pager such as `less`, hold the `page-down` key and you have a basic animation of the progress of your walker.

Tips

Here are some tips to help you with this assignment:

- The easiest way to dynamically allocate a 2D array with dimensions N by M in C is to use "Iliffe" vectors. This is actually a one dimensional array of pointers to other arrays. This link has an illustration of such an Iliffe vector: http://en.wikipedia.org/wiki/Array_data_structure. It is always a good idea to make a quick sketch of the data structures you are implementing yourself.

- You have already used pseudo random numbers in the first assignment, so you know how to use them for the random walker. Normally you would initialise the seed with the output of `time()` to get a different sequence for every run. But it can be useful to use the same sequence of random numbers during debugging. You do this by using a fixed seed.

- Check your maze walker with our test mazes, but feel free to generate your own.

- Try to write a maze solver that finds the exit in the smallest number of steps. It's unlikely that this will be the random walker.

Report

For this assignment you will also need to write a report. The report should not be longer than two pages and should describe:

The problem

Describe the problem that you had to solve for this assignment in your own words. This is an important part of the report, "copy-and-paste" work will not be accepted here.

The solution

Present your solution to this problem. What design decisions did you make and why did you make them. This should also serve as high level documentation for your source code.