Project Maze

Due: September 26, 2012

1 Introduction

While on her way to the official Disney Princess tryouts at the Watson Castle of Information Technology, Princess Doeppna has lost her way and become ensnared in a seemingly-endless labyrinth! She'll have no choice but to wander throughout her prison for hours, manually performing a randomized search for the exit so she can find her way to stardom.

Fortunately, you, who have surely recoiled with horror at the thought of her attempting to perform such a task without a stack or queue, have offered to provide your assistance.

2 Assignment

This C programming assignment contains two parts: first you will write a program *generator*, which generates mazes; then you will write a program *solver*, which solves those mazes. Your generator program should take a single argument:

```
./generator <output maze file>
```

and your solver program should take six arguments:

```
./solver <input maze file> <output path file> <starting x-position>
    <starting y-position> <ending x-position> <ending y-position>
```

Both programs will output their solutions to the given file as specified below.

3 The Layout of a Maze

A maze can be represented as a two-dimensional array of rooms. Your code should be able to handle variable size mazes, although for this assignment mazes will always have the same dimensions: 25×10. Room indices are counted starting from zero at the upper-left corner; thus, the lower-right corner of a 25×10 maze will have coordinates $(24, 9)$.

Each room in the maze has four connections, one for each neighboring room. These connections can either be walls or openings. Rooms which are on the edges of the maze should always have a wall on that border - for example, a room on the south border of the maze should always have a wall on its southern end.

3.1 Encoding a Maze in Memory

You will need to represent the maze in two different ways: first, as a data structure within your program; second, as text in a file on disk. You'll need to be able to read your mazes from and write them to files for this assignment.

3.1.1 Within the Program

Representation of a maze within your program is up to you. However, for each room you will need to keep track of the following:

- the `x` and `y` coordinates of the room.
- whether or not the room has been visited.
- for each connection of the room, whether that connection is a wall, opening, or is uninitialized.

You should use a `struct` to represent each room.

3.1.2 Within a File

Mazes will be written to files on disk by your generator program, and read from files on disk by your solver program. In addition, the files you create may be read by other solver programs or support programs (for example, the visualizer). It is therefore imperative that your maze files conform to our file specification.

Each line of the output file will correspond to a row of rooms.

For each room, you need a concise way to write that room's set of connections. Since there are four possible directions, with two options per direction(wall/opening), there are a total of 16 unique combinations of room connections.

Ideally, we would only need to write one character per room. Since there are 16 options, we can store the output as a one-byte hexadecimal number.

So how do we convert our connections (currently stored as 0 or 1) into hexadecimal values? We already have 1 field per connection, and conveniently, hexadecimal numbers are made up of four bits.

- the highest-order bit represents the east connection;
- the next-highest bit represents the west connection;
- the next-lowest bit represents the south connection; and
- the lowest-order bit represents the north connection.

A bit with value `1` corresponds to a wall, and a bit with value `0` corresponds to an opening.

For example, a room with openings to the north and south and walls to the east and west would be stored as 1100. The binary number 1100 is equal to 12 in decimal and `c` in hexadecimal.

As an example, the following is a sample maze representation in a file:

```
5973333313953973133333313b
c6339595adccd639633b59639
cd53286a70ac619c5333a639c
c4a59e5396969cc6ad51b53ac
ce5a632bc5a5ac61b4ac5a738
432339ddcc5a5adc5ad6a5958
cd5396accccdc58cc5239c6ac
cccd633accc4aec6a639cc59c
68c43339cccc5949719cc6acc
7a6a7332a6a6a6a63a6a633ae
```

3.2 Translating Between Representations

Both parts of this project must translate between the above representations of mazes. Your `generator` must translate from the program represenation to the file representation, and your `solver` must translate from the file representation to the program representation.

3.2.1 From Program to File

You need a way to convert from four integers to one hexadecimal character. Although you could do so with bit-shifting, it is quite a bit simpler to convert by hand to decimal, and then let `fprintf()` deal with the conversion to hex.

Each bit corresponds to a power of two.

E	W	S	N
8	4	2	1

For each bit, the value is incremented by the corresponding power of two.

For example, if I have a room with connections 1100, the sum would be $1(8) + 1(4) + 0(2) + 0(1)$.

Once you have a decimal representation of your room, there is a lovely trick you can pull to print out a hex value. Instead of giving `fprintf() "%d"`, you can give it `"%x"` and it will print your decimal value as a hex character.

3.2.2 From File to Program

Before you can perform any operations on the hexadecimal character from a file, you must properly convert it back into an integer. Fortunately, you can combine both of these steps into one by using the `fscanf()` function with the format string `"%1x"`.

Once you have the hexadecimal number back in integer form, use the following formula to extract each bit:

```
east = hex / 8
hex = hex % 8
west = hex / 4
hex = hex % 4
south = hex / 2
hex = hex % 2
north = hex
```

An alternative way to do this is to use bit shifting operations.

4 Input and Output

The C `<stdio.h>` library contains several definitions that enable you to easily write to or read from files. Included in these definitions is a `FILE` struct, which represents a file within a C program.

4.1 Opening a File

The `fopen()` function opens a file, returning a pointer to a `FILE` struct that corresponds to the desired file.

```
FILE *fopen(char *filename, char *mode)
```

The desired file is indicated by `filename`. The `mode` argument refers to how the file will be used; if you intend to write to the file, this value should be `"w"`, and if you intend to read from the file, it should be `"r"`.

If the desired file does not exist or an error occurs, `NULL` is returned.

4.2 Writing to a File

You can write data to a file using the `fprintf()` function. This function works in very much the same way as `printf()`.

```
int fprintf(FILE *stream, char *format, ...)
```

The only difference is that `fprintf()` takes an additional argument: the `FILE *` that you obtained with `fopen()`.

Remember: To print a hexadecimal number, pass that number as an argument to `fprintf()`, using `"%x"` as your format string.

4.3 Reading From a File

`<stdio.h>` defines many functions that you can use to read from files. The function that you will likely find most useful for this project is `fscanf()`.

```
int fscanf(FILE *stream, char *format, ...)
```

`fscanf()` is to `scanf()` as `fprintf()` is to `printf()`: it does the same thing as its counterpart function, but with a `FILE *` as the source of the input or output operation. To receive a hexadecimal number from a hexadecimal character, use the format string `"%1x`, which will scan a single character into an `unsigned int` variable.

4.4 Closing a File

After your program has finished writing to or reading from a file, it should close that file. Do this with the function `fclose()`.

```
int fclose(FILE *fp)
```

This function returns 0 if no error occurred and 1 otherwise.

5 Generator

Your Generator program should generate a maze with the above representation and save it to a file.

There are a few ways to generate a maze, but the simplest uses the *drunken-walk algorithm.* This algorithm recursively constructs a maze by visiting each room, and then randomly choosing connections for that room by visiting the adjacent rooms.

Starting at room (0,0) drunken-walk randomizes an order to visit the adjacent rooms in, and then visits each of those rooms.

- If the neighboring room has not yet been visited, then the algorithm recurs on that room.
- If it has already been visited and already has a connection defined in the given direction, then the current room should copy that connection to be consistent.
- Otherwise, a connection should be randomly chosen in that direction and stored in the current room. Note that increasing the probability of choosing a wall increases the difficulty of the maze. To mimic the demo exactly, always choose to make a wall.

If implemented correctly, there is guaranteeably a path to from any room to any other room in the maze.

Psuedocode:

```
drunken_walk(x, y):
    r = rooms[x][y]
    set r.visited to true
    for each direction dir in random order:
        neighbor = rooms[x + x-offset of dir][y + y-offset of dir]
        if neighbor is out of bounds
            store a wall in r at direction dir
        else if neighbor has not yet been visited
            store an opening in r at direction dir
            drunken_walk(neighbor.x, neighbor.y)
        else
            opposite_dir = the opposite direction of dir
            if neighbor has a connection c (door or wall) in direction opposite_dir:
                store c in r at direction dir
            else
                store a wall in r at direction dir
```

5.1 Random Number Generation

To ensure that the maze generated by your program is not always the same, you'll need to use random number generation. One way to generate a random integer in C is to use the `rand()` function, which takes no arguments and returns an integer between 0 and `RAND_MAX`[1] To get a random number between 0 and `n`, you can take `rand() % (n + 1)`.[2]

The `rand()` function is actually a *pseudorandom number generator*, meaning that it outputs a consistent sequence of values when given a particular *seed* value. By default, `rand()` has a seed value of 1, so unless you change this your program will generate the same sequence of random numbers each time it is run.

To change the seed value, include the line `srand(time(NULL))`[3] at the beginning of your `main()` function.

To randomize the order of directions through which you will search, declare the directions in some fixed order in an array. You can then shuffle that array in-place with the following algorithm[4]:

```
shuffle_array(A[n]):
    for i from 0 to n-1:
        choose a random number r between i and n, inclusive
        switch A[i] and A[r]
    end for
```

This procedure produces all possible orderings with equal probability.

[1]`RAND_MAX` is defined in *stdlib.h*. Its value is library-dependent, but is guaranteed to be at least 32767.
[2]This is slightly biased towards lower numbers, but it's good enough for this sort of application.
[3]`time()` returns the number of seconds that have occurred since January 1, 1970.
[4]This algorithm is known as the Fisher-Yates shuffle, described in 1938 by Ronald A. Fisher and Frank Yates.

6 Solver

Your Solver program should solve a maze defined by the above representation, printing either the path to the exit or the order of its search (see below) *in reverse order* to a file.

There are also several ways to solve a maze. Your program should employ a *depth-first search.* Such a search begins at the maze's start room and explores adjacent, accessible rooms recursively.

```
dfs(x, y):
    if x, y are the coordinates of the goal
        return true
    mark the room at [x][y] as visited
    for each direction dir:
        neighbor = rooms[x + x-offset of dir][y + y-offest of dir]
        if the connection in direction dir is open and neighbor is unvisited:
            if dfs(neighbor.x, neighbor.y) returns true
                return true

    /* if the program reaches this point than each neighbor's branch
       has been explored, and none have found the goal. */

    return false
```

Beginning from the indicated room, this algorithm traverses all rooms reachable from that path before returning to that room to choose a new path.

We expect your solver to produce two different modes of output:

- Your program outputs the coordinates of only the final route from beginning to end. In order to print only the room coordinates which you know to be on the way to the goal, print the coordinates for each time the recursive call returns true.
- Your program outputs the entire path traversed up until the goal is reached. In order to print every room traversed, you should print the coordinates both when the recursive call returns true *or* false.

The choice should be made when your program is compiled. This is done using preprocessor macros.

```
#ifdef FULL
printf(<something>);
#else
printf(<something else>);
#endif
```

The above code fragment executes the `printf(<something>)` statement only if the macro `FULL` is defined, and executes the `printf(<something else>)` statement otherwise. You can toggle this definition by using the `gcc` compiler flag, `-D <macro>`, which defines `<macro>` for the preprocessor. You can also use the macro `#ifndef <macro>` to execute code only if `<macro>` is not defined.

Your program should print the entire search if a macro `FULL` is defined and print only the path to the exit otherwise. Rooms should be printed with format `x, y` with the upper left corner of the maze corresponding to coordinate `0, 0`. The first line of output should be the starting room, and the last line should be the ending room. If those rooms happen to be the same, your output should only contain one line.

7 Compiling and Running

You have been provided with a *Makefile*, a short text file that contains scripts for compiling, running, or cleaning up projects (for example). You'll learn more about Makefiles in Lab02.

This Makefile contains the following *targets*, or scripts you can run:

- `all`, which compiles both the `generator` and `solver` programs into executable files *generator* and *solver*.
- `generator`, which compiles only the `generator` program into an executable file *generator*.
- `solver`, which compiles only the `solver` program into an executable file *solver*.
- `solver_full`, which compiles the `solver` program with the macro `FULL` defined into an executable file *solver*.

To make a particular target, run

```
make <target>
```

from the command line, or just `make`, which defaults to the first target in the Makefile (in this case, `all`).

If you want to add extra files to either part of your program, add them to the file list defined by `GEN_OBJS` and `SOL_OBJS` for each part respectively.

Once you have compiled your programs, you can run them with the commands

```
./generator <output maze file>
./solver <input maze file> <output solution file> <starting x-position>
    <starting y-position> <ending x-position> <ending y-position>
./solver_full <input maze file> <output solution file> <starting x-position>
    <starting y-position> <ending x-position> <ending y-position>
```

8 Support

To make your maze experience simpler, the course staff has provided two programs which will assist you with this project: a maze viewer and a maze runner. The viewer displays a maze in a graphical form. The runner also displays a maze in graphical form, additionally showing Princess Doeppna's attempts to reach its exit.

To run the viewer you can run the script

```
cs033_maze_view <maze file>
```

To run the maze runner:

```
cs033_maze_vis <maze file> <solution file> <speed>
```

Both of these scripts will check your input files for correctness, and will print out a list of errors if problems are found. If there are problems in the maze file, the viewer will still be launched, giving you the opportunity to find the errors in your maze graphically. However, if problems are found in a solution file, the `cs033_maze_vis` script will terminate. The `<speed>` argument to `cs033_maze_vis` must be an integer between 1 and 3, inclusive; higher values make the character move faster.

8.1 Demo

Lastly, the course staff has provided you with a demo implementation of both parts of this project. These demos are located in */course/cs033/bin/maze-solver-demo* and */course/cs033/bin/maze-generator-demo*; you should be able to run them with `maze-solver-demo` and `maze-generator-demo`.

9 Grading

Your grade for this project will be composed as follows:

Generator functionality	40%
Solver functionality	40%
Style	20%
Total	100%

Your programs should perform error checking on their input, with one exception: your `solver` need not check that its input maze is valid. Also, your programs should not crash for any reason; before you hand in your project, make sure that your programs do not terminate due to a segmentation fault or floating point exception.

Consult the C Style document (which is on the website and provided in the project stencil) to improve your C coding style.

10 Handing In

To hand in your project, run the command

```
cs033_handin maze
```

from your project working directory. You should hand in your source code, a Makefile and a README; you need not include any other files in your handin. Your README should describe the organization of your program and any unresolved bugs.

If you wish to change your handin, you can do so by re-running the script. Only your most recent handin will be graded.