Project Maze: Generator

Due: Wednesday, September 17, 2014

1 Introduction

Baby DJ is a precocious learner, and—although still non-verbal—has already begun experimenting with designing mazes! He and his sister enjoy playing together with their favorite alphabet blocks. They often arrange the cryptic symbols on them into stimulating geometric shapes.

They are still are working on their sharing skills and decided it would be best if they split the alphabet blocks to avoid any unnecessary tension. Baby DJ must now protect his blocks using his maze generating skills.

2 Assignment

In this C programming assignment, you will write a program *generator* which generates mazes.

Your generator program should take a single argument:

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

Your program will output its solutions to a file as specified below.

To get started, run

```
cs033_install maze_generator
```

to copy the stencil for this project into your home directory (∼*/course/cs033/maze_generator*).

3 The Layout of a Maze

In the life lab, you learned how to index into a one-dimensional array to represent a two-dimensional array. In this project you are writing all of your own code. The representation of your maze is entirely up to you - you may use either a one-dimensional or two-dimensional array.

Your code should be able to handle variable size mazes, although for this assignment mazes will always have the same dimensions: 10×25 — that is, 10 rows and 25 columns. 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 $(9, 24)$.

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. In this half of the Maze assignment, you will only write mazes from memory to files.

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 `row` and `column` 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. The files you create may be read by our 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 can you convert your 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:

```
597333331395397313333313b
c6339595adccd639633b59639
cd53286a70ac619c5333a639c
c4a59e5396969cc6ad51b53ac
ce5a632bc5a5ac61b4ac5a738
432339ddcc5a5adc5ad6a5958
cd5396accccdc58cc5239c6ac
cccd633accc4aec6a639cc59c
68c43339cccc5949719cc6acc
7a6a7332a6a6a6a63a6a633ae
```

3.2 Translating Between Representations

Your `generator` must translate from the program represenation of the maze to the file representation.

To do this, you need a way to convert from four integers to one hexadecimal character. There are a few ways to do this, such as bit-shifting, but a simpler way is to convert by hand to decimal and delegate the hexadecimal conversion to `fprintf()`.

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, a room with connections 1100 would be represented with $1(8)+1(4)+0(2)+0(1) = 12$, or `0xC` in hexadecimal.

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.

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 Using Library Calls

When using a library call, it is important to check the value the function returns. If the function returns a value associated with an error, it is important to report the error and handle it accordingly. In this program, we should stop program execution, report the error to the user and exit with the program returning an error (`return 1`). Check if an error has occured by checking the return value. If so, write out an appropriate error message to standard error (`stderr`), and exit cleanly.

To write to `stderr`, use:

```
fprintf(stderr, "[Error message goes here.]\n");
```

4.2 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 it will be created. If an error occurs, `NULL` is returned.

4.3 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.

4.4 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 Generation Algorithm

There are a few ways to generate a maze, but the simplest uses the *drunken-walk algorithm*, or in DJ's case, a *drunken-crawl 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 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:
        if (x + x-offset of dir, y + y-offset of dir) is out of bounds:
            store a wall in r at direction dir
        else:
            neighbor = rooms[x + x-offset of dir][y + y-offset of dir]
            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 (opening 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-1`, you can take `rand() % (n)`.[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.

[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. You can find it in the `<time.h>` header file.

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-1, inclusive
        switch A[i] and A[r]
    end for
```

This procedure produces all possible orderings with equal probability.

6 Compiling and Running

You have been provided a *Makefile*, a text file that contains scripts for compiling, running, or cleaning up projects (for example).

Makefiles contains build *targets*, if we look at the Makefile included in this project we can see that there are 3 targets: `all`, `generator`, and `clean`. Each target has an associated command and optional dependencies. For example, the target `all` depends on the target `generator`. The `generator` target depends on the file `generator.c` and will run `gcc -g -Wall -Wextra -std=99 -o generator.c generator`.

The make command only builds files that have been modifed since the last build and allows you to split up your build process (e.g. splitting up maze generator and maze solver binaries to be built).

To build a particular target, run in the directory containing the Makefile:

`make <target>`

from the command line. If no target is specified, it will build the first target (in this case, the target `all`, which in turn will build `generator`).

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

Once you have compiled your program, you can run it with the command:

`./generator <output maze file>`

You will learn more about Makefiles in Lab02 and Lab07.

7 Support

To make your Generator experience simpler, we've provided three programs which will assist you with this project:

- `cs033_generator_demo <output file>`

  This program demonstrates the expected behavior of your `generator` program. It generates a maze in `output_file`.

[4]This algorithm is known as the Fisher-Yates shuffle, described in 1938 by Ronald A. Fisher and Frank Yates.

- `cs033_maze_validator <maze file>`

  This program will check your maze for errors, such as inconsistent or missing walls and inaccessible areas.

- `cs033_maze_view <maze file>`

  This script will open your maze in a graphical interface. 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.

8 Grading

Your grade for this project will be composed as follows:

Functionality	40%
Code Correctness	40%
Style	20%
Total	100%

- **Functionality**: your code produces correct output, performs error checking on its input, and does not crash for any reason. It does not terminate due to a segmentation fault or floating point exception.
- **Code Correctness**: no part of your code relies on undefined behavior, uninitialized values, or out-of-scope memory; your program compiles without errors or warnings.
- **Style**: your code should look nice! Use appropriate whitespace and indentation and well-named variables and functions. Your code should be reasonably factored, and functions should not be too long.

Your programs should perform error checking on their input. Also, your program should not crash for any reason; before you hand in your project, make sure that your program does not terminate due to a segmentation fault or floating point exception.

Consult the C Style document (which is on the website) to improve your C coding style.

9 Handing In

To hand in your project, run the command

```
cs033_handin maze_generator
```

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.