



The University of New Mexico  
CS 241—Data Organization: Project 5

### **Maze Generator**

By Torin Adamson (a.k.a. Wolf Coder)

edited by Joel Castellanos

**Due: Wednesday, April 8 at midnight in WebCT**

## Problem Specification

Author a C file named `YourFirstName_YourLastName_mazegen.c` which creates random mazes. You must implement the functions defined in the header file:

`http://cs.unm.edu/~joel/cs241/mazegen.h`

Your implementation will be tested by linking it with a test file and running that test file which will that makes various calls to your functions. The test file is given on the web at:

`http://cs.unm.edu/~joel/cs241/mazetest.c`

Your implementation will be tested for memory leaks using valgrind-3.3.0-Debian. Valgrind is a dynamic analysis tool that can detect many memory management and threading bugs.

Your maze generator must create two-dimensional mazes with the following properties:

1. Your generator must create mazes of any specified width and height (within the computer's memory limitations) where each of these dimensions are odd integers greater than or equal to 3.
2. Each cell in a  $(2n+1) \times (2m+1)$  maze is either *open space* or a *wall*.
3. The mazes must be random: A pair of mazes of the same, sufficiently large size, must have a low probability of being identical.
4. Each maze must have exactly two openings along its outer edge: one in the leftmost location of the top edge, the other in the rightmost location of the bottom edge.
5. Each maze must have exactly one path that connects the two outer edge openings.
6. Every open cell in each maze must be reachable from either opening.
7. Every open cell must be adjacent to at least one wall cell.
8. Every wall cell must be adjacent to at least one open cell.
9. Each maze must contain dead ends which, on average, increase in number, get longer, have more turns, and have more branches as the maze size increases.

You may draw your maze using whatever characters you want; however, if you choose to use extended ASCII characters, then you must specify the character set. Display your mazes with a monospace font. In Windows, displaying in Courier looks better than New Courier because the former has less space between lines.

The easiest way to compile this project is to create a new directory and place within it the your .c file, mazeset.c and mazegen.h. Then, compile with the command: gcc \*.c.

## Example Mazes

The smallest allowed maze is a 3x3. This trivial maze has only one possible configuration:

```
# #
# #
# #
```

Possible 7x5 mazes are:

```
# #####
#   # #
### # #
#     #
##### #
```

or

```
# #####
#   # #
# # # #
# #   #
##### #
```

Figure 3a shows a possible 19x11 maze using the # symbol as a wall. Figure 3b shows the same maze using character 219 of character set CP866 (this character set can be used in PuTTY by setting: Window → Translation → Received data assumed to be in which character set → CP866).

```
# #####
#           # # #
##### # ##### # #
#           # # # # #
# ##### ### # ### #
#   #   # # # # #
### # ### ##### # #
#   #           # #
# #####           #
#           # #
#####           #
```

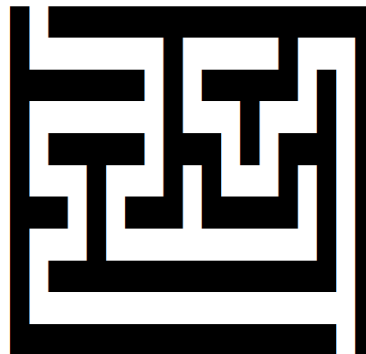


Figure 3a

Figure 3b

Figure 4 contains two versions of the same maze as figure 3. In Figure 4a, walls are drawn with the space character, and empty spaces are drawn with various pipe characters of CP866: | (186), = (205), 7 (187), 4 (188), L (200), F (201), 4 (185), 2 (202), 7 (203), 4 (204), 6 (206), 1 (181), 2 (208), 7 (210), and 1 (198). The cool thing about using the pipe characters is that every other row and column can be removed without changing the maze's logic. This is shown in Figure 4b. This method of display allows a much more complex maze to be drawn in the given space.

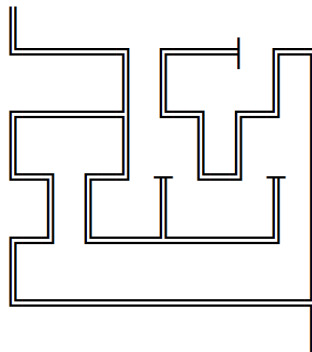


Figure 4a

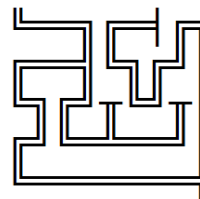


Figure 4b

## Grading

-10x	Where $x$ is the number of compiler warning messages.
+10	Comments are sufficient, accurate, and informative
+10	The program is well organized (no spaghetti code)
+10	Adheres to the UNM CS-241 coding standards
+10	Passes the error cases (test #5) in mazeset.c.
+5	Draws the 5 mazes in test #1 of mazeset.c with no more than 3 minor errors.
+5	Draws the 5 mazes in test #1 of mazeset.c with no more than 2 minor errors.
+10	Draws the 5 mazes in test #1 of mazeset.c with no errors.
+10	Draws the 5 mazes in test #1 of mazeset.c with no errors and uses dynamic memory allocation. This test does not include a check for memory leaks.
+10	Passes test #2 of mazeset.c. This test uses Valgrind to check for memory leaks.
+5	Passes test #3 of mazeset.c. This test creates and prints two a large mazes which must be different.
+15	Passes test #4 of mazeset.c. This test creates many large mazes and checks both for memory leaks and efficient runtime.

## Challenge Options

For extra credit, you may implement any of the extra challenge options. You receive points for every option you successfully complete.

### Challenge Option 1: Compact Pipe Characters (10 points)

Implement the extended ASCII, CP866 pipe characters shown in figure 4b.

### Challenge Option 2: Maze Solver (15 points)

Can't solve your own maze? Write a function that does it for you using the prototype:

```
extern void maze_solve();
```

The answer should be displayed, and should not contain any mistakes (i.e. running into a dead end) even though your algorithm might have to make them in order to solve the maze. If you wrote the maze generator cleverly enough, this should be easy to write a program that finds out how to solve the maze. There's an old Windows 9x screen saver that can give you ideas...

### Challenge Option 3: Image Writer (25 points)

Look online for the Windows .BMP format. You can actually use any common image format, but the .BMP format is the easiest to write. Research is required, but after that, simply add a function to your generator that can save the mazes into an image file so they can be printed out. You might want to scale the maze so that each tile is a square of 10×10 pixels or whatever you think works best.

## Challenge Option 4: Random Dungeon Generator (20 points)

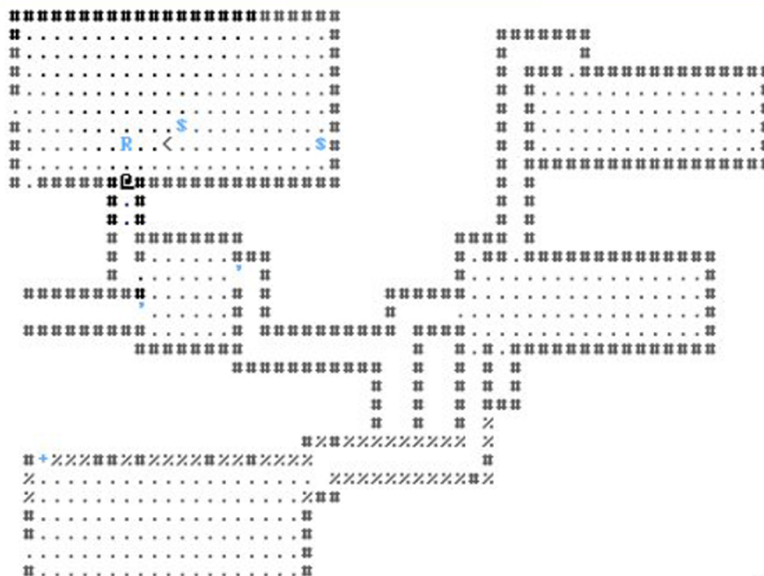
Some games have featured a random dungeon generator, such as Chocobo's Dungeon 2 for the PS1 and the classic Moria and Angband (<http://www.thangorodrim.net/>) for Unix, DOS, Windows, Linux, Apple 2e, and Mac. Although random dungeons get boring easily, they make for a great "side mission/quest" type of feature. This idea can be adapted to many different kinds of games, and could also be great for generating random game content to create a prototype that stress-tests the game-play of your game.

Write a function with the prototype:

```
extern void dungeonGen();
```

This function should create a more interesting maze filled with rooms, tunnels, and other things. You could use various characters to represent different types of tiles such as the screen capture below where # for a wall, % for a dirt wall that can be tunneled through, @ is the player, R is a giant reptile, \$ is gold, . is empty space that has been explored, < is the exit (stairs going up), + is a closed door, and a comma is an open door.

These will be graded with an arbitrary test program that can request any size of map (perhaps wider than 80 tiles) and look for creativity by examining both the maps and the code. Whatever map you create, it must be possible to move from entrance to the exit, although the entrance and exit can be generalized to be something different than a break in the outer wall (such as a > character for entrance and a < character for exit). Each map you generate must look sufficiently different from the last to be considered random.



**\*\*\*Warning\*\*\*** Angband and many other dungeon programs are open source. You are welcome to read through and learn from open source; however, if you copy it, there is a good chance you will be caught. That is a sure path to an **F** in this class, and a department wide red mark on your name.