# COM S 327, Fall 2018
## Programming Project 1.02
### Dungeon Load/Save

It's time to save our dungeons to disk. If we're going to save them, we also want to load them, and of course, if there's nothing to load, we'll still want to generate new ones. This functionality will provide us with the ability to test our implementations of a number of the future assignments against "gold" output to confirm correctness.

In Linux and UNIX, we hide files by beginning their names with a dot. We call them *dot files*. If you list a directory with `ls`, you won't usually see dot files by default (it's possible your shell is configured to give different behavior, however). Passing the `-a` switch to `ls` will force it to list *all*, which includes hidden files. Try it.

Be very careful of the rookie sysadmin mistake of trying to clean up all dot files with '`rm -rf .*`'. This probably doesn't do what you expect. To understand why, consider that '.' is another name (an alias) for the current directory and '..' is the parent of the current directory. To get some idea of just how much damage the command above can do, try issuing the command '`ls -aR`'.

We'll store all of our game data for our Roguelike game in a hidden directory one level below our home directories. We'll call it, creatively enough, `.rlg327`. You can create this directly manually in the shell with the `mkdir` command, or alternatively, you may use the `mkdir(2)` system call to do it in program control. Since this is a system call, it's more something for an operating systems class than for this one, so we won't require you to use it (but it's not complicated). Since the game data directory is under your home, you need to know how to find that. Use `getenv(3)` with the argument `"HOME"`. Concatenate `.rlg327/` on to that. Then our dungeon will be saved there in a file named `dungeon`.

For now, our default will always be to generate a new dungeon, display it, and exit. We'll add two switches, `--save` and `--load`. The save switch will cause the game to save the dungeon to disk before terminating. The load switch will load the dungeon from disk, rather than generate a new one, then display it and exit. The game can take both switches at the same time, in which case it reads the dungeon from disk, displays it, rewrites it, and exits. You should be able to load, display, and save the same dungeon over and over. If things change from run to run, you have a bug. This is a very good test of both your save and load routines!

Planning ahead a bit, we're going to include a position for our PC (player character). The only rule is that the PC must be someplace on the floor (i.e., not embedded in the rock). For maximum simplicity you may use, say, the upper left corner of room zero. You're welcome to display the PC if you want (not required, yet). If you do, use an '`@`'.

Our dungeon file format follows. All data is written in network order (big-endian). You'll need to ensure that you are doing the endianness conversions on both ends (reading and writing). All code in C.

| Bytes | Values |
|---|---|
| 0–11 | A semantic file-type marker with the value `RLG327-F2018` |
| 12–15 | An unsigned 32-bit integer file version marker with the value 0 |
| 16–19 | An unsigned 32-bit integer size of the file |
| 20–21 | A pair of unsigned 8-bit integers giving the $x$ and $y$ position of the PC, respectively |
| 22–1701 | The row-major dungeon matrix from top to bottom, with one byte, containing cell hardness, per cell. The hardness ranges from zero to 255, with zero representing open space (room or corridor) and 255 representing immutable rock (probably only the border). |
| 1702–end | The positions of all of the rooms in the dungeon, given with 4 unsigned 8-bit integers each. The first byte is the $x$ position of the upper left corner of the room; the second byte is the $y$ position of the upper left corner of the room; the third byte is the $x$ size (width) of the room; and the fourth byte is the $y$ size (height) of the room. |

Everybody's program, if correct, should load anybody else's file! If your dungeon doesn't have a "hardness" concept, add it now. It will be needed for a later stage of development. If you've implemented non-rectangular rooms, you have two choices:

1. Change your code to make them rectangular.

2. Decompose them into contiguous, rectangular subrooms. The decomposition could even be to contiguous $1 \times 1$ rooms, which would be an incredibly inelegant implementation, but which would get the job done.

Even if your dungeon generation routines limit your dungeon to a fixed maximum number of rooms, your load routines should have no limit, otherwise, some of us will write dungeons that others cannot read.