COM S 229, Spring 2015
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.

In Linux and UNIX, we hide directories by beginning their names with a dot. We call them *dot files*. If you list a directory with `ls`, you won't 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 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, `.rlg229`. 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 your program. Since this is a system call, it's technically something for an operating systems class, so we won't required you to use it. 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 `.rlg229/` 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. For now, this will result in an identical file, but later, after we've implemented save file versioning, it will give us a quick way to "upgrade" save files.

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
0–5	A semantic file-type marker with the value `RLG229`
6–9	An unsigned 32-bit integer file version marker with the value 0
10–13	An unsigned 32-bit integer size of the rest of the file (total size of the file minus 14)
14–61453	The row-major dungeon matrix from top to bottom, with four bytes for each cell. In each cell, the first byte is non-zero if the cell represents open space, zero otherwise; the second byte is non-zero if the cell is part of a room, zero otherwise; the third byte is non-zero if the cell is part of a corridor, zero otherwise; and the fourth byte is the unsigned, 8-bit integer hardness value of the cell, where open space has a hardness of 0 and immutable material has a hardness of 255.
61454–61455	An unsigned 16-bit integer containing the number of rooms in the dungeon
61456–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 width of the room; and the fourth byte is the height of the room.