# COM S 229, Spring 2015
## Programming Project 1.03
### Player Character and Monsters

So far, we've got this lovely dungeon. And we can. . . save and restore it. And, you know. . . look at it.

That's about it. Kind of boring.

So let's add a character to represent us, and some monsters to chase us around. User interface comes next week, so for this week, you add some routines of your own devising to move the player character (PC) around. For extra fun you can make the PC bore through walls; walls are no obstacle and after moving to any location, that location becomes a corridor (where monsters can follow). Remember that even this powerful PC cannot move though immutable cells, however!

The PC is represented by an '`@`'. This is simply the way things are done in Roguelike games, and Roguelike gamers are traditionalists, so, sorry, you can't change this. Place your intrepid `@` on any open space that you like in the dungeon. Add a switch, `--nummon`, that takes an integer count of the number of monsters to scatter around. Monsters are represented by letters. For the most part, there are no conventions here, beyond that each letter represents a certain class of bad guy, and that class usually begins with the letter that is used to represent it. For instance, it's common for humans (people) to be represented by `p`, giants (big people) by `P`, and dragons by `D`, but Smaug, Ruth, Falcor, Saphira, and Norbert and all `D`s. If you've got color, you might make them gold, white, white, blue, and black, respectively. Since we haven't worked on NPCs (non-player characters) yet, you don't have to think about this; just scatter some letters around and keep track of them.

Let's add some characteristics to our monsters. At this point, the monsters are randomly generated, so we don't have to worry about consistency. Each monster will be smart or not and each will be telepathic or not. A telepathic monster always knows where the PC is and will move toward the PC. A non-telepathic monster will only move toward the last position where it saw the PC. There's a question of what *toward* means, here, and that's where monster intelligence comes in. A smart monster will understand the dungeon map and move approximately on the shortest path to the PC (think Dijkstra's algorithm). A dumb monster does not understand the map and will move in the direction that minimizes the Euclidean distance between it and the PC, even if that leaves it stuck in a corner.

One more characteristic we'll add is speed. Each monster gets a speed between 5 and 20. The PC gets a speed of 10. Every character goes in a priority queue, prioritized on the game turn of their next move. Each character moves every floor(100/speed) turns. The game turn starts at zero, as do all character's first moves, and advances to the value at the front of the priority queue every time it is dequeued. A system built with this kind of priority queue drive mechanism is known as a *discrete even simulator*.

So the PC moves around like a drunken, roving idiot according to an algorithm that you devise. All monsters who know where the PC is, move toward it, and all monster's who know where the PC was, move toward that location. If you like, you could make non-telepathic, smart monsters guess where to go next, maybe based on an observed direction vector, and non-telepathic dumb monsters wander the corridors randomly. All characters move at most one cell per move. Eventually, two characters are going to arrive at the same position. In that case, the new arrival kills the original occupant. No qualms. If you haven't figured it out already, you're going to need to use dynamic memory for this; and if you've managed to avoid it so far, you have no choice now. The game ends with a win if the PC is the last remaining character (unlikely, since the NPCs can be up to twice as fast as the PC). It ends with a loss if the PC dies.

Redraw the dungeon after each PC move, pause so that an observer can see the updates (experiment with `sleep(3)` and `usleep(3)` to find a suitable pause), and when the game ends, print the win/lose status to the

terminal before exiting.

All code is to be written in C.