COM S 327, Spring 2019
Programming Project 1.04
Player Character and Monsters

We've had dungeons for a few weeks, and now we also have path finding. We've reached a point where we can add an adventurer and monsters. User interface comes next week, so for this week, add some routines of your own devising to move the player character (PC) around. The PC movement doesn't have to be clever; you can move it randomly, along a chosen vector, whatever you like; even allow it to stand still (i.e., don't bother adding movement routines for the PC at all). 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. Rooms are easy to find since you have a room array, so it's easier to put the PC in one of those than in a corridor. Add a switch, `--nummon`, that takes an integer count of the number of monsters to scatter around, and hard code a default number of monsters to place in the dungeon when the switch is not present. Something close to 10 is reasonable for the size of our dungeons[1].

Monsters are usually represented by letters, but occasionally numbers and punctuation are used. 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`, humanoid non-humans by `h`, giants (big people) by `P`, and dragons by `D`, but Smaug, Ruth, Falcor, Saphira, and Norbert are all `D`s (and Rand would still be a `p`). If you've got color, you might make those `D`s gold, white, white, blue, and black, respectively. In the future, we'll design our monsters and assign letters to them with some care, but for now, we're going to use letters *and numbers* based on monster characteristics (see below) so that we can know their abilities at a glance.

Our monsters will have a 50% probability of each of the following characteristics:

- **Intelligence**: Intelligent monsters understand the dungeon layout and move on the shortest path (as per path finding) toward the PC. Unintelligent monsters move in a straight line toward the PC (which may trap them in a corner or force them to tunnel through the hardest rocks). Intelligent monsters also can remember the last known position of a previously seen PC.
- **Telepathy**: Telepathic monsters always know where the PC is. Non-telepathic monsters only know the position of the PC if they have line of sight (smart ones can remember the position where they last saw the PC when line of sight is broken). Telepathic monsters will always move toward the PC. Non-telepathic monsters will only move toward a visible PC or a remembered last position.
- **Tunneling Ability**: Tunneling monsters can tunnel through rock. Non-tunneling monsters cannot. These two classes will use different distance maps. Tunneling monsters when attempting to move through rock will subtract 85 from its hardness, down to a minimum of zero. When the hardness reaches zero, the rock becomes corridor and the monster immediately moves. Reducing the hardness of rock forces a tunneling distance map recalculation. Converting rock to corridor forces tunneling and non-tunneling distance map recalculations.
- **Erratic Behavior**: Erratic monsters have a 50% chance of moving as per their other characteristics. Otherwise they move to a random neighboring cell (tunnelers may tunnel, but non-tunnelers must move to an open cell).

[1]You don't have to hard code this. You're welcome to implement some logic that decides how many monsters to place, but there should *always* be at least one.

These characteristics are somewhat loosely defined, and you may interpret them as you like so long as you keep with the spirit of the assignment; don't try to take the easy way out. I get many, many questions on the order of, "What does a smart, non-telepathic, non-eratic monster do?". My interpretation is that that monster will move toward the PC if it can see it, move toward the last known position has previously seen but cannot presently see the PC, and rest (stay still) otherwise. If your interpretation is different, that's okay (again, so long as your "interpretation" isn't engineered to minimize your work). It's your game.

Since there are 4 binary characteristics, we can map the 16 possible monsters to the 16 hexadecimal digits by assigning each characteristic to a bit. We'll assign intelligence to the least significant bit (on the right), telepathy next, tunneling third, and erratic behavior last (on the left), thus we have:

Binary	0000	0001	0010	0011	0100	0101	0110	0111
Decimal	0	1	2	3	4	5	6	7
Represented by	`0`	`1`	`2`	`3`	`4`	`5`	`6`	`7`
Binary (cont.)	1000	1001	1010	1011	1100	1101	1110	1111
Decimal (cont.)	8	9	10	11	12	13	14	15
Represented by (cont.)	`8`	`9`	`a`	`b`	`c`	`d`	`e`	`f`

So you can observe the behavior of, say, a `b` monster, and know that it has the intelligence, telepathy, and erratic characteristics, or of a `3` and know that it is smart and telepathic.

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 (PC and NPCs) generates an event when it is created. That event has a time, when it will occur (based on the current game turn and the character's speed), and goes in a priority queue, prioritized on the event time. Each character's move event occurs every floor(1000/speed) turns (integer math, so this is just a normal division; you don't have to call floor()). Each time a character's move event is removed from the queue, that character gets to move, and a new move event is placed in the queue for its next turn. The game turn starts at zero, as do all characters' 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 event simulator*. It's not actually necessary to keep track of the game turn. Since the priority queue is sorted by game turn, the game turn is, by definition, the next turn of the character at the front of the queue.

You already have at least one priority queue. You can use it again, or write or find another. Using it again is the best choice.

So the PC moves around like a drunken, roving idiot (or doesn't) according to an algorithm that you devise (or don't). All monsters who know where the PC is, move toward it, and all smart 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 (`thug_life.gif`). 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. It's also possible, depending on what types of monsters are loaded, to end up in an infinite loop, and that's okay.

Redraw the dungeon after each PC move, pause so that an observer can see the updates (use `usleep(3)`, which sleeps for *argument* number of microseconds; something like 250000 is reasonable), and when the game ends, print the win/lose status to the terminal before exiting.

All code is to be written in C.