Assignment 0: Image Processing

- I construct an output file name from the input file name. This construction is slightly buggy, but you can fix it if you want (no bug bounty, though). See the comment in the source code for details.

- I don't use a loop for the convolution. Instead, I implement the whole convolution in a single expression. The reason is entirely one of performance. That the expanded, hard-coded expression is much, much, much faster than the little 3x3 loop that you probably wrote. Go ahead and time them to confirm. You can time an application running in the shell in Linux by running it using the "time" utility, e.g., "time sobel motorcycle.pgm". This timer is not good enough for real scientific analysis, but it's plenty good enough for something like this.

- I cast all of my index operations into the input image to int16_t (a 16-bit, signed integer type) to avoid the sign and overflow issues that lead to "grainy" output.

Assignment 1.01: Dungeon Generation

- My solution is attached. You may post your solutions to this thread as well. You are welcome to use any code posted to this thread in future assignments unless the poster explicitly states otherwise. Please attribute any code you use to its respective author(s).

- My solution uses some features of C that we haven't discussed yet, most notably preprocessor macros and enumerated constants. We'll get to them soon. It also uses a version of Dijkstra's algorithm to run twisty corridors, for which it needs a priority queue. heap.[ch] contain the implementation of this.

- The queue is a Fibonacci queue, which has very nice asymptotic performance, but big constants. For a relatively small graph like the cells in our dungeon (less than 20000 nodes), it's probably a wash whether the Fibonacci heap or a simpler type of heap like a binary heap is the better choice. I haven't run a performance comparison. On very large graphs, the Fibonacci heap will win hands down.

- The next assignment requires no limit on the maximum number of rooms, so you'll need to modify the room array to be dynamically allocated. You can keep the internal max rooms for dungeon generation purposes, but you should be able to read and write dungeons with any number of rooms. I have posted some test dungeons so that you can test your implementation with compliant input with many rooms.

- If you are picking up this code drop, please don't be daunted by its length or complexity. You don't have to understand it! You only have to understand the dungeon data structure! Beyond the small issue of dynamically allocating the rooms array, you'll never have to understand or to make any changes to the dungeon generation code.

Assignment 1.02: Dungeon Load/Save

- My solution to 1.02 is attached. Same rules as always: You may use my solution, your own solution, a (willing) classmate's solution, or even code that you find on the web as a basis for the next assignment so long as you attribute it appropriately to the author(s).

- Some stuff that my code does that yours (probably) doesn't:
  - I add an optional parameter to the --load and --save switches to take a file in the non-default position.
  - I accept short forms of all switches, using a single dash and the first letter, so --save == -s.
  - In order to generate the "special" test dungeons, I wrote some code to read portable gray map (PGM) images (like we used in assignment 0). This is a very simple image format. I could then use an image editing program (Gimp) and my mouse to "draw" the dungeons. The image reading code is far from robust. It will break if you pass it PGMs that aren't exactly like mine. If you'd like to make your own special dungeons, add black (0) and white (255) pixels on top of an otherwise gray 80x21 image. You could then post your special dungeons in a thread here on Piazza and we can all pathfind them.
  - I included code to automatically create $HOME/.rlg327 if it doesn't already exist.

- I don't check the return values of fwrite(), which, really, I should. Setting bad examples, here! I don't check return values of fread(), either, but in this case there's not a good reason to, because I instead check that the internal size of the file matches the actual size of the file (on disk) using the stat(2) system call. Technically, it's still possible that there would be a read error after making this confirmation; however, the kind of things that would cause these are, for instance, disk failures, in which case you've got much bigger problems!

- The code (aside from the heap) is still in one file. I think this makes sense organizationally, because it all deals with the dungeon. With the new assignment, pathfinding, I think it starts to make sense to break the code into multiple files. My solution to 1.03 has separate files for the heap, the main program, path finding, dungeon, and utility code (making a directory, for instance)

Assignment 1.03: Path Finding

- My solution is attached. Same deal as always: You're welcome to use this code. You're also welcome to post your own solutions to this thread, and in so doing, you grant permission for your classmates to use your code.

- I've now broken my game into several compilation units (files). Dungeon stuff (the bulk of the code in earlier drops) is in dungeon.c. main() and functions that are only useful to main() are in rlg327.c. utils.c gets the directory creation code from last time. This is nice to pull out for two reasons; the first being that it really isn't part of the game proper, and the second that it's the only system dependent code that we have (if you want to compile this on a non-POSIX OS, you'll need to port this file). The new code for pathfinding is in path.c.

- Appropriate header files have been created to export data structures and functions as needed. The dungeon structure has two new maps of the dungeon, one for each of the two different distance metrics. It also has a PC structure, currently containing only a position. render_dungeon() checks for existence of the PC before rendering terrain. The PC is not represented as a terrain type. Many students do initially represent it that way, and even go on to represent all of the monsters as terrain in 1.04. If you give it some thought, I think you'll see why that is a poor solution.

Assignment 1.04: Player Character and Monsters

- My solution to 1.04 is attached. You are welcome to attach your own solution to this thread. As always, you may use my solution (or any other attached to this thread) as your basis for future assignments. Something which I haven't emphasized in the past, but which discussions with students have brought to the fore, is that you may also take parts of solutions and use it in your own base. If, for instance, you want to continue to use your own code, but you have a bug which you can't figure out, you may extract from my code, adapt it as necessary, and directly use it in your code. Please be sure to attribute the original author(s) when using others' code.

- My solution now has a few more files, including files for characters, PC, and NPCs, events, and movement routines.

- The main loop is in main() in rlg327.c. This continually renders the dungeon and processes the event queue (by calling do_moves()) until the PC dies. do_moves() inserts the PC, then runs until the PC is removed.

- The event queue is in the dungeon structure. Also added to the dungeon structure is a dungeon-sized matrix of character_t pointers. This allows constant time detection of characters in movement targets. A character dies whenever another character moves into it's position. Since the character is in the event queue and we have no function that allows us to remove an arbitrary node from the queue, we instead mark it as dead, then it is cleaned up when the node is removed by do_moves().

- We have a pseudo object hierarchy with character as a base and pc and npc derived from it. Observe the use of pointers to achieve this, with common fields in character and unique fields in the respective "sub-objects". Since there are only two pointers, we only waste 4 bytes (on a 64-bit system) in doing it this way (compare to using an enumerated value to indicate which pointer is valid in a union). If we had many pointers, or if we used instances instead of pointers, I would create a union within character with each necessary type within and an enumeration to indicate the actual type.

- Rendering a dungeon cell is still constant time since I can use the character map to find monsters in constant time.

- We have 16 different movement patterns, but observe that erratic movement is a coin toss: when "heads" we make a random move, and "tails" we move as if the monster is not erratic, so that's trivial. The other's also have aspects in common. Imagine any monster that can see the PC; it moves toward the PC according to line of sight. Now consider a telepathic but unintelligent monster. It can't see the PC, but it still moves in a straight line toward the PC, same as line of sight! So these two cases can use the same routine to calculate the next position! Movement is reduced to a handful of helper functions, calling the appropriate one when needed.

- Also, rather than complicated, nested "if"s or a slightly less complicated switch statement, I've built a function jump table (an array of function pointers) indexed by the 4 ability bits. See npc_next_pos() in npc.c.

Assignment 1.05: User Interface with NCurses

- My solution is attached. Same rules as always.
- I've moved all code related to the user interface into io.c, with exported function prototypes in io.h.
- The act of displaying the dungeon with curses is pretty straightforward; I expect we've all done it nearly precisely the same, so I'm not going to discuss that here. User input is handled primarily in io_handle_input(), which is a loop around a big switch statement. The loop ensures that erroneous input and commands that shouldn't actually use a turn (like listing monsters) actually don't use a turn.
- I've added a handful of fun or useful commands that weren't part of the assignment specification. Look through this function for them. In particular, I recommend that you see the 'q' command which gives you a usage example for a message queue. The message queue will allow you to do some printf()-style debugging despite the curses interface, which (if you haven't already noticed) makes that kind of difficult. Take a look at the message functions and data structures at the top of io.c to get a more complete understanding of the system.
- 'm' executes the monster list routine, which lists all of the monsters in the dungeon. It sorts the monster list by distance from the PC. Take a look at that for another example usage of qsort(). Just for fun, I also add an adjective to each monster in the monster list. You can add your own simply by extending the "adjectives" array.
- Stairs requires deleting the dungeon and creating a new one. new_dungeon(), in dungeon.c, handles this.

Assignment 1.06: "Fog of War" and porting to C++

- My solution is attached. The standard rules apply.

- My changes are mostly the minimal possible to comply with the assignment specification. The only real counterexample to that is the addition of the bold in the visibility map to show what is illuminated. When I say "minimal", I mean that, while I've changed dungeon, character, pc, and npc to classes, I'm still using them like structs. The fields are public. I've accepted the C++ compiler's (no-op) default constructor, and I persist in initializing the objects with C-style functions. This misses a lot of the power that comes with an object model, and I will make changes to some of these things week to week as the semester winds down.

- "Converting" these three structures to classes was a matter of changing the data structure itself (in the respective headers), and renaming the respective source files to C++, then fixing compiler errors until it builds. I have some mutator and accessor functions, e.g., in character, that aren't needed. These are leftovers from code that was required in earlier semesters.

- To do the lighting, I modified can_see() in character.cpp, so that if the viewer is the PC, it short circuits at the light radius and it updates the PC's known terrain and visibility maps (calling pc_learn_terrain()). pc_observe_terrain() (in pc.cpp) calls can_see() every player turn for every cell in the PC's light radius. There are definitely more efficient solutions.

- The PC has changed to a pointer in the dungeon structure. This means it needs to be deleted. If you simply delete it unconditionally at the end of main(), you'll introduce a double free bug. This is because the turn queue holds an instance of the PC, and it will delete it when the heap is destroyed. The simplest solution is to test if the PC is alive before calling delete_pc().

Assignment 1.07: Parsing Monster Definitions

- Two attachments this time. rlg327-s2019-1.07.tar.gz is the standard solution drop. object_descriptions.tar.gz contains a standalone object parser for students who are writing their own code and don't want the extra work of writing an object parser. To be clear, it is not 100% standalone. First, it includes dice.h (also included, should you want it) and uses a dice class set method expecting three ints, base, number, and sides, in that order. Second, it includes dungeon.h, using my dungeon structure to get to a vector of object_description. And third, it includes utils.h because I forgot to delete that line; it's not needed. Modifying these things to work with your code (assuming you already have a dice implementation) is literally only a few lines of code. The interface should be clear enough from object_descriptions.h.

- As for the full solution:

- The normal rules apply.

- I've made calls to the parsers at the front of main(), so you'll want to remove those to continue with the next assignment.

- In the parsers, I use macros to build lookup tables of many of the keywords to convert token strings to integers. I do build the tables in sorted order, so bsearch() could be used, but they're all small enough that a linear search is much faster, so I search them linearly. I use sentinel values to mark the ends of the tables.

- The parser pulls out individual tokens from the input stream and keeps a lookahead token to choose where to go next. Code is reused as much as possible, for instance, dice parsing is always the same, so all dice are parsed by the same method. When parsing a monster or object, each field must appear exactly once. I keep a boolean which is false before a field has been parsed and true after. I check and set it before parsing the field. I also track the number of fields parsed. When the loop terminates, it is not necessary to check the state of all the booleans, because if I've looped the correct number of times and never repeated a field, then I must have covered all fields.

- I use a mixture of input functions, including getline(), get(), peek(), and operator>>(),

Assignment 1.08: Loading Monsters and Objects

- My solution is attached. The regular rules apply.

- I added a static generate_monster() method to the monster_description class. This gets called from gen_monsters() to do all of the work that the old gen_monsters() did, but using the new definitions. Objects are generated in essentially the same way, but since objects are new (not converting old C code), I've got a constructor that takes an object description, and I call it rather than a factory method. Monster generation and object generation could be essentially the same. This gives you two different examples showing how they could be done.

- Objects are stored in the object map (analogous to our character map), but unlike NPCs, this is the only place they are stored, so when leaving a level or quitting the game, it's necessary to iterate through the entire map to locate and clean up objects.

- Displaying objects correctly with fog of war is a bit tricky. Since objects don't move, the semantics are slightly different. I added a "seen" flag to the object class and can_see() has been updated such that whenever the PC sees a cell, any objects in that position are marked as having been seen, then only previously seen objects are displayed. Another solution would be to display objects only in cells that have been seen. If we were to implement monsters that can pick up items, neither of these would be sufficient, but they're good for now.

- Multi-colored monsters use the select() system call to monitor standard input and as a high-precision timer. The games loops on select(), redrawing the dungeon when the timer goes off (colors are chosen randomly at print-time), and continuing into io processing when input is available.

- Pass wall monsters are implemented, but PICKUP and DESTROY are not.

- Unique monsters are implemented by the monster description class. It checks if it is allowable for a monster to be created before actually creating it, and a reference to the description is passed to the npc. The NPC's destructor then takes care of calling back into the description to update the records there.

- If you see a river crab, you're probably already dead. Not to mention Rory and SpongeBob. I have to admit that Rory was an accident. She was supposed to have speed 10. I'm not quite sure how she ended up with speed 50, but it was not intentional. As it stands, she now moves as fast as she talks.

- I found a minor GCC bug while doing this update. Only the second one I've ever submitted. The first one (I don't remember what it was) was in 2007. This one was a known bug, so my report only confirms that it still exists in 8.2.0. Look for "pragma" in io.cpp.

Assignment 1.09: PC Equipment and Updated Combat

- The normal rules apply.

- I think that this assignment shouldn't have been "hard", but there are a lot of details. For completion, this update gets our game into a reasonably playable state, so take it home and show it to your parents. If you want, you can print it out and hang it on their refrigerator.

- For inventory and equipment, the PC gets a pair of arrays of object pointers. This means that the PC needs a non-default destructor (which can clean those up). Equipment slots could be indexed by the enums that were initially used by the parser, but there's an off-by-one issue there. I made another set of enums, but in retrospect, I think that dealing with the off-by-one and keeping one set would have been the better design.

- Combat is straightforward enough. Displacement is more challenging, and is implemented exactly as I discussed in class, with an array of possible displacement positions; I start at a random position in that array and traverse it circularly to find a position to displace to. If none is available, the monsters swap positions. This does introduce a bug that can place non-pass wall monsters inside walls. I will not give a bug bounty for this bug. It's known. It doesn't lead to crash. And it's not hard to fix. The difficulty is more a matter of deciding how to fix it.