HW: Dungeon Crawler

You have just been hired by *Foobar*[[1]](#footnote-0) Games as a game logic programmer. The company is working on a retro dungeon crawler, which is a type of game where you control a character who is navigating a fantasy dungeon environment.

You are in charge of **implementing** and testing part of the core game logic. The art and design teams are still working on graphics and sound, so you are working on a text-based prototype for developing and debugging your logic code.

In this assignment, you will **implement** the code that reads an in-game map (or “dungeon”) from a file into a dynamic 2D array. You must also **implement** updating the map as the player moves the character through the dungeon. The player’s goal is to pick up the treasure and go to the level’s exit. Additionally, you will need to **implement** magic amulets that resize the dungeon size as well as **implement** the logic for the monsters that chase the player.

# Extra Credit

In test driven development you develop tests prior to implementing code. In test code, you want to exercise every line of code you have written. So in a set of tests, you will call functions with various parameters to cause all parts of the code to be executed. Moreover, you should test the results of test code for correctness. However, for extra credit, you just need to get coverage of all parts of the code regardless of correctness.

If you do this prior to implementing your code, it might give you better insight into all of the test cases you will encounter when implementing the main problem.

[See Appendix B for gameplay examples.](#_3j2qqm3)

# Objectives

## Assignment Objectives

* Create and store values into a 2D dynamic array using loop iteration, and then delete once it is no longer needed.
* Resize a 2D dynamic array without memory leaks.
* Code and call functions with pass by reference.
* Update variables passed into functions as references and check conditions to determine the game state.

## Extra Credit Objectives

* Write unit tests.
* Write tests for creating and storing values in a dynamically-allocated 2D array.
* Write tests for deleting a dynamically-allocated 2D array.
* Write tests for resizing a dynamically-allocated 2D array.
* Invoke functions that pass arguments by reference.
* Write tests for functions that update their arguments (i.e. pass by reference)
* Write tests for determining the state of the game.

# Submission

You will need to submit files for both Dungeon Crawler and the Extra credit at the same time. Regardless, you only need to include the files you want autograded for a submission.

* dungeoncrawler.cpp.
* logic.cpp
* logic.h.
* helper.cpp
* Helper.h

## Additional files to Submit for Extra Credit

* dctests.cpp
* Any test files used (.txt)

# Roadmap

The implementation of the program and the extra credit are different programs. So they should be set up in different directories to compile and run in your programming environment since each has its own main function.

While not required, it could be beneficial to implement the Extra Credit prior to implementing Dungeon Crawler.

## Dungeon Crawler

### Download, Set Up, and First Submission

1. Download the [Dungeon Crawler Start Code](https://drive.google.com/drive/folders/1024rf3Q9D3wozoIPjX-jloyPzpBSZyrl) into your programming environment. Confirm that the starter code consists of the following files:
   * dungeoncrawler.cpp: You must not edit this file, but you may read it.
   * logic.cpp: You must edit this file.
   * logic.h: You must not edit this file, but you must read it.
   * helper.cpp: You must not edit this file, but you may read it..
   * helper.h: You must not edit this file, but you must read it.
2. Compile and run the initial state of the starter code with the following command:
   * g++ -std=c++17 -Wall -Wextra -pedantic-errors -Weffc++ -fsanitize=undefined,address \*.cpp
3. Confirm that the initial state of the starter code executes without errors and with only warnings related to unused variables/parameters.
4. Submit the starter code to Gradescope so you can take a look at the test cases.
   * You need to submit dungeoncrawler.cpp, helper.cpp, helper.h logic.cpp, logic.h
5. Read the starter code.
   * Familiarize yourself with the code in the logic.h file. These will be used in writing the tests you need to cover the code that will be written later in the logic.cpp file.
     1. Player Struct: holds the values of the adventurer’s position in the grid, as well as a count of treasure acquired across all levels of a dungeon.
     2. Tile Status Constants: these constants hold values for representing the tile type on the dungeon map.
     3. Movement Status Flag Constants: these constants hold values for representing the player's movement status flags.
     4. User Keyboard Input Constants: these constants hold values for representing the user's keyboard inputs.
6. Re-familiarize yourself with the input file format for the program.
   * [See Appendix A for information about the map text files.](#_z337ya)

### Test Locally

1. Compile and run the initial state of the starter code with the following command:
   1. g++ -std=c++17 -Wall -Wextra -pedantic-errors -Weffc++ -fsanitize=undefined,address \*.cpp
2. Submit to Gradescope to get an idea of those tests. Note those might not be sufficient and you might need to consider more test cases.
3. The starter code should fail most, if not all, tests written by you or in gradescope.
   1. Borrow any test code that you can from Gradescope (e.g. from visible tests).

### General Instructions for Writing Code to Pass Unit Tests

1. Pick a function to implement.
   1. Since functional units should be independent of each other, you can implement functions in almost any order. But, you should start with functions that have no reason to invoke any other function. That is, implement from the bottom up.
2. Open logic.h and logic.cpp.
3. Read the function specification in logic.h and think about inputs and what should be true after the function executes. You should have enough sets of inputs to test every aspect of your code.
4. Pick a set of inputs to pass.
   1. For each functional unit, you should have many tests (i.e. sets of inputs and expected results). Pick one to target. Start by picking the simplest tests, such as unhappy paths that can be detected early and easily.
      1. You can always comment out code in the main function and make calls to the function you are targeting with different sets of inputs to help you focus on what you are debugging.
      2. Once everything is done, you can comment out your test code and uncomment that was given to you to see how everything works together.
5. Given the expectations for the function’s behavior written in the header file, your test, and the additional specification given below, write just enough code in logic.cpp to pass the test.
6. Compile your code and tests and run.
7. Review and respond to the results
   1. While the code does not pass the test, figure out why and update the code until it passes the test
8. Follow red-green-refactor for each test
   1. red: test failing
      1. write code to pass test, retest
   2. green: test passing
   3. refactor: clean up test code, remove duplicated code, add abstraction, etc.
9. Go to step 1 unless you are satisfied with your code’s correctness.

### Implement loadLevel(...)

1. Refer to [Appendix A: Input Map Text File](#_z337ya).
2. Open logic.cpp and locate the loadLevel function. Make sure to read the block comment describing the function.
3. Consider all the ways in which reading the dungeon map from the file could go wrong and write code that detects when those things happen and responds correctly.
   1. **Correct behavior:**
      1. if the input file is invalid in any way, do not create the map but return nullptr instead.
4. Consider what values are created or updated when the dungeon map file is valid and write code that sets those values correctly.
   1. **Correct behavior:**
      1. maxRow is set to the number of rows in the dungeon level map
      2. maxCol is set to the number of columns in the dungeon level map
      3. player’s row and column attributes are set to the players starting location in the dungeon level map
      4. The return value is a pointer to a dungeon level map in memory and the contents of that map are the same as those in the input file (except the player’s location which is marked using the player tile symbol).

### Implement getDirection(...)

1. Open logic.cpp and locate the getDirection function. Make sure to read the block comment describing the function.
2. Consider what values are updated and write code that sets those values correctly.
   1. **Correct behavior:**
      1. nextRow and nextCol are set to the appropriate values depending on the input direction
      2. Input other than MOVE\_UP, MOVE\_DOWN, MOVE\_LEFT, or MOVE\_RIGHT is ignored (don’t move)

### Implement deleteMap(...)

1. Open logic.cpp and locate the deleteMap function. Make sure to read the block comment describing the function.
2. Consider what values are updated and write code that sets those values correctly.
   1. **Correct behavior:**
      1. Deallocate the 2D array pointed at by the pointer map, and make sure not to leak memory! (Hint: what should the parameter maxRow be used for?)
      2. Update map to be nullptr
      3. Update maxRow to be 0

### Implement resizeMap(...)

1. Open logic.cpp and locate the resizeMap function. Make sure to read the block comment describing the function.
2. Consider what values are created or updated and write code that sets those values correctly
   1. Correct behavior:
      1. An enlarged version of the dungeon map is created
         1. A new 2D array, with (2 \* maxRow) rows and (2 \* maxCol) columns, is dynamically allocated.
         2. The content of the array map is copied into the subarray **A** exactly (including the adventurer). The content of the array map is copied into each of the subarrays **B**, **C**, and **D**, except for the adventurer, which should be replaced by TILE\_OPEN.
            1. Visually, we can divide this 2D array into four maxRow×maxCol subarrays:
      2. The original map is deallocated
      3. maxRow is doubled
      4. maxCol is doubled.
      5. A pointer to the enlarged map is returned
      6. A nullptr if the map is null or one of the dimensions is invalid.

| **A** | **B** |
| --- | --- |
| **C** | **D** |

### Implement doPlayerMove(...)

1. [Refer to Appendix B: Gameplay](#_3j2qqm3)
2. Open logic.cpp and locate the doPlayerMove function. Make sure to read the block comment describing the function. You will be returning a status from logic.h that gives the result of the attempted move.
3. Consider all possible and meaningfully different map/game states and write code that implements the correct behavior for those states.
   1. **Correct behavior:**
      1. Update player.row and player.col based on the value of nextRow and nextColumn.
         1. If the next position places the adventurer outside the bounds of the array or on an impassable tile (a pillar or a monster), set the status to STATUS\_STAY and update nextRow and nextCol to be the adventurer’s current position (i.e., the adventurer did not move). Remember to check that nextRow and nextCol are within bounds before using them to check a tile’s value (short circuit evaluation might be useful, see zyBook).
         2. If the next position is on a treasure tile, set the appropriate status and increment the adventurer’s treasure by one.
         3. If the next position is on an amulet tile, set the appropriate status.
         4. If the next position is on a door (to the next level), set the appropriate status.
         5. If the next position is on an exit (to the whole dungeon) and the adventurer has at least one piece of treasure, set the appropriate status. If the adventurer has no treasure, treat the door as you would a pillar.
      2. Update the map by updating the adventurer’s position to the next position, setting the new position to TILE\_PLAYER and the adventurer’s old position to TILE\_OPEN.
      3. Return the appropriate status flag.

### Implement doMonsterAttack(...)

1. Open logic.cpp and locate the doMonsterAttack function. Make sure to read the block comment describing the function.
2. Consider all possible and meaningfully different map/game states and write code that implements the correct behavior for those states.
   1. **Correct behavior:**
      1. All monsters with a vertical or horizontal line of sight to the player move 1 tile closer to the player
         1. Starting from the tile above the adventurer’s location and working upward, check each individual tile to see if there is a monster on the tile.
         2. If there is a monster on a tile, move the monster one tile closer to the adventurer.
         3. Continue to check until you have reached the top of the map or reach a pillar (monsters can’t see through pillars)
         4. Repeat the same logic with down, left, and right (in that order). Make sure all monsters that are supposed to move do so before you go to the next step.
      2. Return true if a monster reaches the player, false otherwise.
         1. The adventurer is killed if a monster moves onto their tile (check if the player position now contains a monster), return true (adventurer killed, game over) if so, otherwise return false (the monsters did not attack the adventurer, yet…).

### Implement createMap(...)

1. Open logic.cpp and locate the createMap function. Make sure to read the block comment describing the function.
2. Consider all possible input values and write code to implement the correct behavior for the function.
   1. Correct behavior:
      1. If the input values are invalid, do not create a new map but instead return the null pointer.
      2. Otherwise, dynamically allocate a 2D char array with maxRow rows and maxCol columns. Initialize each array element to TILE\_OPEN.
      3. Return a pointer to the new map in memory.

## Extra Credit

It would benefit you to implement this prior to implementing the assignment, but that is not required to get extra credit.

1. Download the [Extra Credit Start Code](https://drive.google.com/drive/folders/1024rf3Q9D3wozoIPjX-jloyPzpBSZyrl) into your programming environment. Confirm that the starter code consists of the following files:
   1. logic.h: You must not edit this file, but you must read it.
   2. dctests.cpp You must write your unit tests in this file and submit it.  
       You should end up with at least 500 lines of test code.
   3. helper.h: You must not edit this file, but you must read it.
2. Implement dctests.cpp.
3. There are two test cases: “Memory Leaks” and “Test Coverage”.
4. Test Coverage
   1. The target code tells you what cases are covered
   2. You must think of more cases that are not yet covered
   3. Your score will be the coverage percentage times 4, so you can get up to 4 points of extra credit for coverage.
5. There is a concrete example of a test for loadLevel in dctests.cpp.
   1. Only invocation is *required* for coverage points
   2. Testing for correctness is optional but **strongly recommended** for preparing to implement
   3. The starter code demonstrates only one (not particularly elegant) way of testing correctness
   4. Deallocating dynamically-allocated memory is *required* for preventing memory leaks (the rest of the points).
6. Make more tests
   1. For loadLevel, this means making more dungeon maps (both valid and invalid) to try loading
      1. You must familiarize yourself with the input file format for the program.
      2. [See Appendix A for information about the map text files.](#_z337ya)
   2. For all the other functions, this means setting up the pre-conditions (like making a dungeon map directly in memory) and invoking the function
   3. For each test, make sure your test only uses 1 of the target functions (i.e. don't use loadLevel in a test for doPlayerMove)

# Testing Your Code

## Testing Dungeon Crawler Implementation

In other words… Integration/Acceptance Testing Your Code (AKA Playing the Game)

### Sample Dungeons

Included in the starter code from Canvas is a folder of sample dungeon levels. You will find the 4 tutorial levels, 2 levels for the “easy” dungeon, and 3 levels for the “hard” dungeon.

[See Appendix B for more information about playing the game.](#_3j2qqm3) As an example, to play the hard dungeon, you would enter “levels/hard 3” at the beginning prompt. You may want to test with the tutorial or easy dungeon. The hard dungeon is not for the faint of heart! Can you escape it?[[2]](#footnote-1)

Of course, feel free to create your own levels! [Review Appendix A for information about the level file structure.](#_z337ya)

### Debug Printouts

Because we want to only print out the map with each action, you will need to remove the INFO statements or other debugging output from your code before submitting for the full program test cases. The full program cases do not include any debugging output, so Gradescope will not allow them to pass if INFO or “cout” statements are present in the final code.

### Test Cases

In addition to the extensive unit tests that you considered (i.e. inputs and expected results), Gradescope also has unit tests associated with each function. There are also full system tests which check the integration of your code into the existing gameplay loop (in dungeoncrawler.cpp).

However, you should remember that you are responsible for correctly implementing the game logic and testing your code, so some test cases may not have thorough coverage of every possible edge case. Try to think of edge cases and test them out yourself while developing!

Note that video games can be particularly challenging to find all bugs through [test-driven development](https://en.wikipedia.org/wiki/Test-driven_development). Games rely extensively on user interaction, so it is possible for a nearly-infinite combination of edge cases to exist. The gaming industry has helped popularize play-testing jobs and [beta software](https://en.wikipedia.org/wiki/Software_testing#Beta_testing) releases as one means of identifying bugs during development.

### More Ideas

After you have finished and earned a score on Gradescope with which you are satisfied, here are some other game mechanics you can think about implementing. You won’t get extra credit for these, but if you have some free time you might find them both enjoyable and good practice.

* Monsters with different behaviors, for example up/down or left/right every turn, diagonally, two or more squares per turn, etc.
  + Note: Most of what we have done so far in this course has been procedural, but as you learn more about object-oriented approaches, think about how we could do more complicated monster behaviors. You could keep a list of monster objects, each with their own location, actions, etc. that you update each turn.
* Doors that require different amounts of treasure to unlock, or let you go back to a previous level.
* Nonlinear dungeons, where levels have multiple doors to different levels.
* Levels that shrink each turn, like a trash compactor.
* More items with different properties, for example a shield to protect you against monsters, a sword to let you attack monsters, a potion that lets you move faster or freeze monsters, bombs that blow up pillars, etc.
* The ability to restart a level if the adventurer is killed.
* Update the map on the console without reprinting it.

Even though this is a very rudimentary game, we hope you can imagine how this assignment relates to modern computer games. With some graphics, music and sound effects, and a little polish, you’ve built the basics of a turn-based dungeon crawler!

## Testing Extra Credit

### General Instructions for Writing Unit Tests

Pick a function to test.

* 1. Since unit tests should be independent, you can write unit tests in any order.

Open logic.h and dctests.cpp

Read the function specification in logic.h

Think about what the behavior of the function is supposed to be. What are relevant and appropriate pre- and postconditions (i.e. what are the various cases for the input values and what are the correct output(s) that correspond to those inputs)?

* 1. Include both happy and unhappy paths
     1. Happy paths := nothing goes wrong, input is valid
     2. Unhappy paths := something goes wrong, input is invalid

Make a function in dctests.cpp to hold the tests you will write and invoke it from main().

Write the tests in the function body. Be careful to keep the tests independent. Unit tests should only rely on the function (“unit”) being tested. Avoid invoking any of the other functions in your tests as this will make the test less informative.

Don’t leak memory in your tests. If you expect the function to allocate memory, then you should deallocate that memory before the end of the test.

Don’t dereference the null pointer. If you expect the function to return the null pointer, don’t dereference it. If you expect the function to return a non-null pointer, verify that it is non-null before dereferencing it.

Submit dctests.cpp and logic.h to Gradescope and all input files your tests require.

### Write Unit Tests for loadLevel(…)

1. [See Appendix A for information about the map text files.](#_z337ya)
2. Consider all the ways in which reading the dungeon map from the file could go wrong and make tests (dungeon map files + code) that cause those things to happen.
3. Consider what values are created or updated and make tests that verify the values are correct.
   1. A test for loadLevel means creating a dungeon map file and directing the function to load the level from the file.
   2. You should end up with many different dungeon maps that exhibit different kinds of flaws and some that are valid.
4. Deallocate the map (if it was successfully created).

### Write Unit Tests for getDirection(…)

1. Consider all possible inputs and verify that the next row and column are correct.

### Write Unit Tests for deleteMap(…)

1. Be honest with the function: it expects to get a pointer to a 2D array and a number of rows, you should tell it the correct number of rows (otherwise, there will be a memory leak and it will be *your* fault).
2. Don’t double free/delete (don’t re-deallocate memory that has already been deallocated).

### Write Unit Tests for resizeMap(…)

1. Here is an explanation of how the function should behave:

| **A** | **B** |
| --- | --- |
| **C** | **D** |

* 1. Copy the contents of the array map into the subarray **A** exactly (including the adventurer). Copy the contents of the array map into each of the subarrays **B**, **C**, and **D**, except for the adventurer, which should be replaced by TILE\_OPEN.
  2. Deallocate the original map and update the dimensions of the map.
  3. Return a pointer to the new array.

1. Verify that the updates are performed correctly.

### Write Unit Tests for doPlayerMove(…)

1. [Refer to Appendix B: Gameplay](#_3j2qqm3)
2. Here is an explanation of how the function should behave:
   1. The next position is determined by nextRow and nextColumn.
      1. If the next position places the adventurer outside the bounds of the array or on an impassable tile (a pillar or a monster), set the status to STATUS\_STAY and update nextRow and nextCol to be the adventurer’s current position (i.e., the adventurer did not move). Remember to check that nextRow and nextCol are within bounds before using them to check a tile’s value (short circuit evaluation might be useful, see zyBook).
      2. If the next position is on a treasure tile, set the appropriate status and increment the adventurer’s treasure by one.
      3. If the next position is on an amulet tile, set the appropriate status.
      4. If the next position is on a door (to the next level), set the appropriate status.
      5. If the next position is on an exit (to the whole dungeon) and the adventurer has at least one piece of treasure, set the appropriate status. If the adventurer has no treasure, treat the door as you would a pillar.
   2. Update the map by updating the adventurer’s position to the next position, setting the new position to TILE\_PLAYER and the adventurer’s old position to TILE\_OPEN.
   3. Return the appropriate status flag.
3. Consider all possible and meaningfully different map/game states and verify that the behavior of the function is correct.

### Write Unit Tests for doMonsterAttack(…)

1. [Refer to Appendix B: Gameplay](#_3j2qqm3)
2. Here is an explanation of how the function should behave:
   1. The logic for the monster AI is as follows:
      1. Starting from the tile above the adventurer’s location and working upward, check each individual tile to see if there is a monster on the tile.
      2. If there is a monster on a tile, move the monster one tile closer to the adventurer.
      3. Continue to check until you have reached the top of the map or reach a pillar (monsters can’t see through pillars)
   2. Repeat the same logic with down, left, and right (in that order). Make sure all monsters that are supposed to move do so before you go to the next step (Step 4).
   3. The adventurer is killed if a monster moves onto their tile (check if the player position now contains a monster), return true (adventurer killed, game over) if so, otherwise return false (the monsters did not attack the adventurer, yet…).
3. Consider all possible and meaningfully different map/game states and verify that the behavior of the function is correct.

Have fun, adventurer!

# Appendix A: Input Map Text File

This assignment will involve reading a text file that contains information of the dungeon map's internal representation. This text file consists of three parts:

* Line 1: Map Dimensions. This line contains two values for representing the map's number of rows and number of columns, respectively. (Neither can be bigger than 999,999)
* Line 2: Player Starting Location. This line contains two values for representing the player's starting row and column, respectively.
* Lines 3+: These lines contain the individual tile information of the dungeon map as char values.
  + The first number in Line 3 represents the map tile at (0, 0), where the first value is the row position and the second value is the column position.
  + Refer to the tile status constants in logic.h for more details.

The following is an example text file for a 5×3 tile representation of a dungeon map.

| 5 3  3 0  M + -  - + -  - + !  - - -  @ - $ |
| --- |

* Line 1; The map has 5 rows and 3 columns.
* Line 2: The player will start at map location (3, 0), where 3 is the row position and 0 is the column position. Remember that indexing starts from 0, so the top left corner will be map location (0, 0).
* Lines 3-7: The map's internal representation for each map tile.
  + **Note #1:** Whitespace for lines 3 and after are purely for aesthetic purposes, so you must not assume that line breaks represent the actual map dimensions or that there will be spaces in between the char values.
  + **Note #2:** In previous assignments you used column-major order when working with 2D arrays and PPM files, which is the norm in the graphics community. In this assignment you will be using row-major ordering when working with your 2D arrays, as this is how a C++ program will typically order a multidimensional array.
  + **Note #3:** Each level of a dungeon has its own map stored in a different file, each named according to the dungeon name followed by the level number. For example, if the dungeon is named “tutorial” and has four levels, the files will be “tutorial1.txt”, “tutorial2.txt”, “tutorial3.txt”, and “tutorial4.txt”.
  + **Note #4:** Only the final level will have an exit (an ! symbol); all the other levels have doors (a **?** symbol) to the next level.

# Appendix B: Gameplay

These are sketches and descriptions from the design team of the expected gameplay that you must implement and test.

## Loading a Game

Each game starts with a printout of the instructions, followed by a prompt for the dungeon name and the number of levels in the dungeon. Here our dungeon is called tutorial and it has 4 levels (user input in dark red):

| ---------------------------------------------------------  Good day, adventurer!  Your goal is to get the treasure and escape the dungeon!  --- SYMBOLS ---  o : That is you, the adventurer!  $ : These are treasures. Lots of money!  @ : These magical amulets resize the level.  M : These are monsters; avoid them!  +, -, | : These are unpassable obstacles.  ? : A door to another level.  ! : A door to escape the dungeon.  --- CONTROLS ---  w, a, s, d : Keys for moving up, left, down, and right.  e : Key for staying still for a turn.  q : Key for abandoning your quest.  ---------------------------------------------------------  Please enter the dungeon name and number of rooms: tutorial 4↵ |
| --- |

## Navigating the Dungeon

Our dungeon crawler uses the WASD[[3]](#footnote-2) method of controlling the in-game adventurer: **w** and **s** move the adventurer up one row and down one row respectively, while **a** and **d** move the adventurer left one column and right one column respectively. Entering **e** will cause the adventurer to stay still for a turn.

Tiles with unpassable obstacles (a pillar) cannot be moved onto, and are represented by the + symbol. The door to the next level is represented by the **?** symbol, while the door out of the dungeon (and thus the game) is represented by the **!** symbol.

| Level 1  +---------+  | o ? |  | + |  | |  +---------+  Enter command (w,a,s,d: move, e: stay still, q: quit): s↵  +---------+  | o ? |  | + |  | |  +---------+  You didn't move. Are you lost?  Enter command (w,a,s,d: move, e: stay still, q: quit): d↵  +---------+  | o |  | + |  | |  +---------+  You have moved to row 0 and column 1  You go through the doorway into the unknown beyond... |
| --- |

## Treasure

Dungeons are dangerous places, so why would you put up with all that risk for no reward? Tiles with treasure on them are represented by the **$** symbol. Pick up a piece of treasure by moving the adventurer to that tile.

Upon exiting, the game will tell you how much treasure was picked up by the adventurer across all levels. But make sure you don’t leave empty-handed, since the door out of the dungeon (represented by the ! symbol) won’t open if you don’t have at least one piece of treasure!

| Level 2  +---------------+  | ? |  | o $ |  | |  +---------------+  Enter command (w,a,s,d: move, e: stay still, q: quit): d↵  +---------------+  | ? |  | o |  | |  +---------------+  You have moved to row 1 and column 1  Well done, adventurer! You found some treasure.  You now have 1 treasure.  Enter command (w,a,s,d: move, e: stay still, q: quit): w↵  +---------------+  | ? o |  | |  | |  +---------------+  You have moved to row 0 and column 1  Enter command (w,a,s,d: move, e: stay still, q: quit): w↵  +---------------+  | ? o |  | |  | |  +---------------+  You didn't move. Are you lost?  Enter command (w,a,s,d: move, e: stay still, q: quit): a↵  +---------------+  | o |  | |  | |  +---------------+  You have moved to row 0 and column 0  You go through the doorway into the unknown beyond... |
| --- |

## Monsters

So, what makes dungeons so dangerous anyway? The monsters kept in them to guard the treasure of course! Monsters are represented by the symbol **M**, and will chase any adventurer in their line of sight (i.e., if the adventurer is a rook’s[[4]](#footnote-3) move from them). Thankfully, they are slow and move only one tile per turn, and cannot see over unpassable obstacles, allowing the adventurer to hide behind them.

The adventurer is quick enough that he can get through a door before a monster attacks on the next turn, but won’t be able to pick up an item and then withstand an attack. Monster attacks are lethal, and being killed by the monster will cause you to lose the game. The adventurer is not strong enough to attack a monster, so the only strategy is to run away. Monsters are powerful, and will destroy any (passable) obstacle in their path - including treasure, amulets, and even doors![[5]](#footnote-4) Make sure that the monsters don’t destroy the only way out...

| Level 3  +---------+  | M $ |  | $ $ |  | |  | o |  | |  | ? |  +---------+  Enter command (w,a,s,d: move, e: stay still, q: quit): s↵  +---------+  | $ |  | M $ |  | |  | |  | o |  | ? |  +---------+  You have moved to row 4 and column 0  Enter command (w,a,s,d: move, e: stay still, q: quit): s↵  +---------+  | $ |  | $ |  | M |  | |  | |  | o ? |  +---------+  You have moved to row 5 and column 0  Enter command (w,a,s,d: move, e: stay still, q: quit): d↵  +---------+  | $ |  | $ |  | M |  | |  | |  | o |  +---------+  You have moved to row 5 and column 1  You go through the doorway into the unknown beyond... |
| --- |

## Magic Amulets

The dungeon also holds many ancient and mysterious artifacts, such as magic amulets. Magic amulets are represented by the @ symbol. Picking up an amulet will cause the level to double in size, with three additional copies of the level (without additional adventurers, but with additional monsters and items) appearing below, to the right, and diagonally below and right of the level. Unfortunately, the amulet is destroyed in the process and isn’t copied as well.

Be careful, magic can be dangerous to use - make sure you don’t accidentally have a monster appear right next to you! But in wise hands, magic can help you escape from otherwise impossible levels...

| Level 4  +---------+  | o |  | + + |  | @ + ! |  +---------+  Enter command (w,a,s,d: move, e: stay still, q: quit): s↵  +---------+  | |  | o + + |  | @ + ! |  +---------+  You have moved to row 1 and column 0  Enter command (w,a,s,d: move, e: stay still, q: quit): s↵  +------------------+  | |  | + + + + |  | o + ! + ! |  | |  | + + + + |  | + ! + ! |  +------------------+  You have moved to row 2 and column 0  The magic amulet sparkles and crumbles into dust.  The ground begins to rumble. Are the walls moving?  Enter command (w,a,s,d: move, e: stay still, q: quit): s↵  +------------------+  | |  | + + + + |  | + ! + ! |  | o |  | + + + + |  | + ! + ! |  +------------------+  You have moved to row 3 and column 0 |
| --- |

| Enter command (w,a,s,d: move, e: stay still, q: quit): d↵  +------------------+  | |  | + + + + |  | + ! + ! |  | o |  | + + + + |  | + ! + ! |  +------------------+  You have moved to row 3 and column 1  Enter command (w,a,s,d: move, e: stay still, q: quit): d↵  +------------------+  | |  | + + + + |  | + ! + ! |  | o |  | + + + + |  | + ! + ! |  +------------------+  You have moved to row 3 and column 2  Enter command (w,a,s,d: move, e: stay still, q: quit): w↵  +------------------+  | |  | + + + + |  | + o + ! |  | |  | + + + + |  | + ! + ! |  +------------------+  You have moved to row 2 and column 2  Congratulations, adventurer! You have escaped the dungeon!  You escaped with 1 treasure and in 16 total moves. |
| --- |

## Quitting the Game

You can quit the game at any time by entering the symbol q. Just be careful, as there is no way to save your progress.

1. [Foobar](https://en.wikipedia.org/wiki/Foobar) [↑](#footnote-ref-0)
2. 2 It is possible to escape it in 65 steps with 1 treasure, or 88 steps with 4 treasures. Can you do better? [↑](#footnote-ref-1)
3. [WASD keys](https://en.wikipedia.org/wiki/Arrow_keys#WASD_keys) [↑](#footnote-ref-2)
4. [Rook](https://en.wikipedia.org/wiki/Rook_(chess)) [↑](#footnote-ref-3)
5. Note: This is actually a simplification so that testing and development will be easier. [↑](#footnote-ref-4)