HW: Dungeon Crawler Part 1: Test First

You've just been hired by *Foobar*[1] *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're 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 **write tests** for the code (that you will write later). You must **write tests** for reading an in-game map (or "dungeon") from a file into a dynamic 2D array. You must also **write tests** for 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 **write tests** for the magic amulets that resize the dungeon size as well as **write tests** for the monsters that chase the player.

You are not implementing any game logic in this part of the assignment. You are to **write the tests first!**

See Appendix B for gameplay examples.

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.

Testing Your Code

Roll Your Own Dungeons

Review Appendix A for information about the level file structure.

Debug Printouts

To help with testing and debugging, the starter code includes `INFO(X)` and `INFO_STRUCT(X)` macros (in `logic.h`). You can use these to display the values of variables during runtime to aid in tracing your program's execution. The test cases on Mimir ignore standard output (but too much of it will slow the tests down and cause them to time out, so don't spam the console with output).

Test Cases

Mimir has a test for memory leaks caused by your code (tests) and a test for line coverage achieved by your code (tests). There is also a 0-point code quality test that can help you to write better code.

[1] Foobar

Roadmap

1. Download and extract the starter code into your programming environment. Confirm that the starter code consists of the following files:
    a. `logic.h` You must not edit this file, but you must read it.
    b. `tests.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.
2. Submit the starter code to Mimir so you can take a look at the test cases.
    a. You need to submit `tests.cpp`
    b. You do not need to submit `logic.h`.
    c. Note that there are 3 tests and 2 are worth points: "Memory Leaks" and "Test Coverage".
        i. Your score on this part of the assignment is equal to 70 points times the percentage of lines of the target code (an instructor-provided test target pseudo-solution) that your test cases cover plus 30 pts if your code does not have memory leaks. For example, if your test cases cover 81% of the target code without any memory leaks, your score will be 70*81% + 30 = 86.7 points. If your test cases cover 95% of the target code but leak memory, your score will be 70*95% + 0 = 66.5 points.
3. Read the details on "Test Coverage"
    a. The target code tells you what cases are covered
    b. You must think of more cases that are not yet covered
4. Read the starter code
    a. 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.
        i. 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.
        ii. Tile Status Constants: these constants hold values for representing the tile type on the dungeon map.
        iii. Movement Status Flag Constants: these constants hold values for representing the player's movement status flags.
        iv. User Keyboard Input Constants: these constants hold values for representing the user's keyboard inputs.
    b. There is a concrete example of a test for `loadLevel` in `tests.cpp`.
        i. Only invocation is *required* for coverage points on part 1 (test first)
        ii. Testing for correctness is optional but **strongly recommended** for preparing for part 2 (development)
        iii. The starter code demonstrates only one (not particularly elegant) way of testing correctness
        iv. Deallocating dynamically-allocated memory is *required* for preventing memory leaks (the rest of the points).
5. Make more tests
    a. For `loadLevel`, this means making more dungeon maps (both valid and invalid) to try loading
        i. You must familiarize yourself with the input file format for the program.
        ii. See Appendix A for information about the map text files.

    b. For all the other functions, this means setting up the pre-conditions (like making a dungeon map directly in memory) and invoking the function
    c. 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`)
6. Submit your `tests.cpp` file and all of your dungeon map files
7. Read the details on "Test Coverage"
8. Go to step 5 unless you are satisfied with the amount of points you have earned

General Instructions for Writing Unit Tests

1. Pick a function to test.
    a. Since unit tests should be independent, you can write unit tests in any order.
2. Open `logic.h` and `tests.cpp`
3. Read the function specification in `logic.h`
4. 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)?
    a. Include both happy and unhappy paths
        i. Happy paths := nothing goes wrong, input is valid
        ii. Unhappy paths := something goes wrong, input is invalid
5. Make a function in `tests.cpp` to hold the tests you will write and invoke it from `main()`.
6. 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.
7. 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.
8. 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.
9. Submit to Mimir `tests.cpp` and all input files your tests require.

Write Unit Tests for `loadLevel(...)`

1. Refer to Appendix A: Input Map Text File
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.
    a. A test for `loadLevel` means creating a dungeon map file and directing the function to load the level from the file.
    b. 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**

   a. 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`.
   b. Deallocate the original map and update the dimensions of the map.
   c. Return a pointer to the new array.
2. Verify that the updates are performed correctly.

Write Unit Tests for `doPlayerMove(...)`

1. Refer to Appendix B: Gameplay
2. Here is an explanation of how the function should behave:
   a. The next position is determined by `nextRow` and `nextColumn`.
      i. If the next position places the adventurer outside the bounds of the array or on an unpassable 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).
      ii. If the next position is on a treasure tile, set the appropriate status and increment the adventurer's treasure by one.
      iii. If the next position is on an amulet tile, set the appropriate status.
      iv. If the next position is on a door (to the next level), set the appropriate status.
      v. 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.
   b. 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`.
   c. 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
2. Here is an explanation of how the function should behave:
    a. The logic for the monster AI is as follows:
        i. 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.
        ii. If there is a monster on a tile, move the monster one tile closer to the adventurer.
        iii. Continue to check until you have reached the top of the map or reach a pillar (monsters can't see through pillars)
    b. 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).
    c. 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.

Prepare for Part 2 (Development)

1. Refactor your tests
2. Submit and review (make sure your refactor didn't break anything)
3. Add functional correctness tests
4. Submit and review
5. Start sketching the code for implementing the functions
6. Run your tests against *your* code
    1. no need to submit to Mimir for testing, you have your own tests now!
7. 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.

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.
- 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[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        |
|     +    |
|     ?    |
+----------+
```

[2] WASD keys

```
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

Enter command (w,a,s,d: move, e: stay still, q: quit): d↵
+---------+
|         |
|    +    |
|    o    |
+---------+
You have moved to row 2 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 2
Well done, adventurer! You found some treasure.
You now have 1 treasure.

Enter command (w,a,s,d: move, e: stay still, q: quit): d↵
+---------------+
|             ? |
|          o    |
|               |
+---------------+
You have moved to row 1 and column 3

Enter command (w,a,s,d: move, e: stay still, q: quit): w↵
+---------------+
|          o  ? |
|               |
|               |
+---------------+
You have moved to row 0 and column 3

Enter command (w,a,s,d: move, e: stay still, q: quit): d↵
+---------------+
|             o |
|               |
|               |
+---------------+
You have moved to row 0 and column 4
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[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![4] Make sure that the monsters don't destroy the only way out...

[3] Rook
[4] Note: This is actually a simplification so that testing and development will be easier.

```
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.