

HW: Dungeon Crawler

Overview

Objectives

- Create and store values in a 2D dynamic array.
- Deallocate a 2D dynamic array.
- Resize a 2D dynamic array.
- Avoid memory leaks.
- Use pass by reference.

Introduction

You have 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 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.

Getting Started

- [Get the starter code](#) (same as what you submit).
 - `Dungeoncrawler.cpp`
 - Contains the high level game control logic.
 - `logic.h`
 - Contains function prototypes for the game.
 - `logic.cpp`
 - **Where you will implement the function definitions.**
 - `helper.h`
 - Contains function prototypes for helper functions.
 - `helper.cpp`
 - Contains function definitions for helper functions.
- Allowed includes:
 - `<iostream>`
 - `<fstream>`
 - `<string>`
 - `"logic.h"`
 - `"helper.h"`
- Read over the starter code.
 - There is a struct in `logic.h` for the player with their position in the grid and a count of their treasure.

- There are tile status constants in `logic.h` that represent the tile type on the map.
- There are constants for the player's status and keyboard inputs in `logic.h`.
- Read over the [input file format](#) for the program.
- Testing locally is very important for this assignment.
 - `g++ -std=c++17 -Wall -Wextra -pedantic-errors -Weffc++ -fsanitize=undefined, address *.cpp`
 - `./a.out`
- General procedure:
 - Pick a function to implement (start with functions that won't need other functions to work).
 - Read over the function specifications in `logic.h`.
 - Read over the function's [requirements](#) in this document.
 - Write some test cases you expect your function to work with.
 - Write the function in `logic.cpp`.
 - Test your code.
 - If there are any failures, debug and re-evaluate your function.
- Refer to the [appendix](#) as needed.

Requirements

loadLevel()

1. Refer to the [input file format](#).
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.
 - a. **Correct behavior:**
 - i. if the input file is invalid in any way, do not create the map but return `nullptr` instead.
 - ii. For how to check validity, refer to the [appendix](#) and think back to checks done on previous homeworks (this is intentionally left up to you to discern).
 1. One check is to ensure the total number of spaces does not exceed `INT32_MAX`.
4. Consider what values are created or updated when the dungeon map file is valid and write code that sets those values correctly.
 - a. **Correct behavior:**
 - i. `maxRow` is set to the number of rows in the dungeon level map
 - ii. `maxCol` is set to the number of columns in the dungeon level map
 - iii. player's row and column attributes are set to the players starting location in the dungeon level map
 - iv. 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).

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.
 - a. **Correct behavior:**
 - i. `nextRow` and `nextCol` are set to the appropriate values depending on the input direction
 - ii. Input other than `MOVE_UP`, `MOVE_DOWN`, `MOVE_LEFT`, or `MOVE_RIGHT` is ignored (don't move)

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.
 - a. **Correct behavior:**
 - i. 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?)
 - ii. Update `map` to be `nullptr`
 - iii. Update `maxRow` to be 0

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
 - a. **Correct behavior:**
 - i. An enlarged version of the dungeon map is created
 1. A new 2D array, with $(2 * \text{maxRow})$ rows and $(2 * \text{maxCol})$ 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`.
 - a. Visually, we can divide this 2D array into four $\text{maxRow} \times \text{maxCol}$ subarrays:
 - ii. The original map is deallocated
 - iii. `maxRow` is doubled
 - iv. `maxCol` is doubled.
 - v. A pointer to the enlarged map is returned
 - vi. A `nullptr` if the map is null or one of the dimensions is invalid.

A	B
C	D

doPlayerMove()

1. [Refer to Appendix C: Gameplay](#)
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.
 - a. **Correct behavior:**
 - i. 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.
- ii. 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`.
- iii. Return the appropriate status flag.

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.
 - a. **Correct behavior:**
 - i. 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.
 - ii. 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...).

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.
 - a. **Correct behavior:**
 - i. If the input values are invalid, do not create a new map but instead return the null pointer.
 - ii. Otherwise, dynamically allocate a 2D char array with `maxRow` rows and `maxCol` columns. Initialize each array element to `TILE_OPEN`.
 - iii. Return a pointer to the new map in memory.

Appendix

A: Testing Your Code

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.](#) 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]

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

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](#). 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](#) 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!

B: 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 5x3 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.
 - It is possible for a "final level" to have both an exit and a door (for example, a "secret" level).

C: 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] 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] 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] 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](#)

[2] 2 It is possible to escape it in 65 steps with 1 treasure, or 88 steps with 4 treasures. Can you do better?

[3] [WASD keys](#)

[4] [Rook](#)

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