Homework 1 — "Maze" Game

Tuesday, September 14th, 2021 at 11:59 pm (**part 1 - 5 points**)

Tuesday, September 21st, 2021 at 11:59 pm (**part 2 - 95 points**)

(100 points total)

Objectives:
- Work with a multi-file c++ program
- Create a program that involves structs, classes, and enums
- Become (more) familiar with pointers and references
- Understand and review how objects are instantiated and how to use and manipulate them via methods and fields
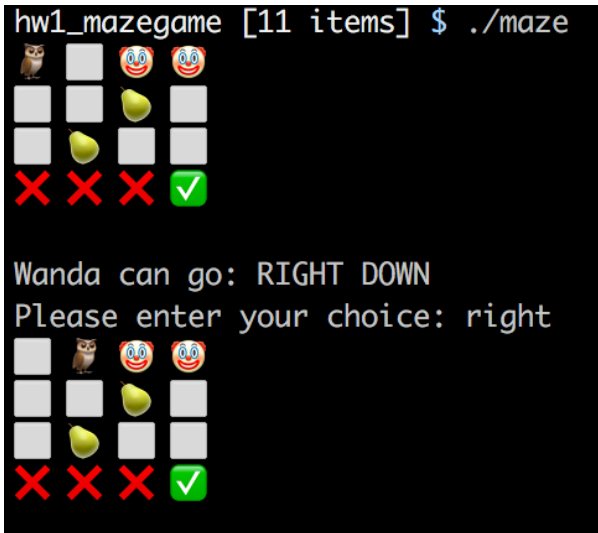
Turn In:
- the individual files listed here: `main.cpp`, `Maze.h`, `Maze.cpp`, `Player.h`, `Player.cpp`, `Makefile`
- You *may* split Board into its own `Board.h` and `Board.cpp` files if you choose.
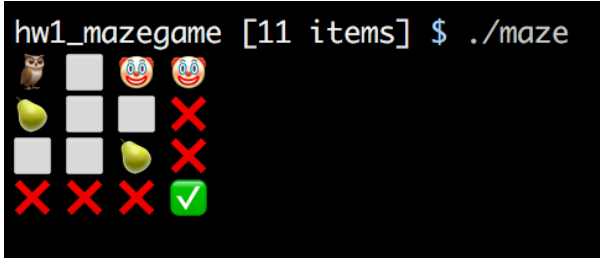
Instructions:

Your job is to implement a "maze" game. The word "maze" is in quotes here because "grid with random walls in it" is more accurate. As you'll see from this description, in its most basic incarnation, this game **can** produce "mazes" that aren't solvable—the player can't move and the game will never end.

We'll implement a basic, but forgiving text UI to play our game. You can think of this game as the first thing that you would implement if you were trying to implement pacman. There is a single human player who will ask which direction they want to go each turn, and the player will walk around collecting treasure until they reach the exit (or die).

Here are some screenshots of what our version looks like (see the last page for a complete run-through on a small board):

Here, the player is the owl, walls are red crosses, treasures are pears, enemies are clown faces, and the exit is the green check mark.

Each turn, the current player should be told which directions are legal moves. Every move that does not go off the board or run into a wall is a legal move. Diagonal moves are not legal moves.

The current player will earn 100 points each time they collect a treasure.

If an enemy moves onto a square with the human on it, the human is destroyed. If the human moves onto a square with an enemy on it, the human is destroyed.

	If an enemy moves to a square with another enemy, neither player moves and the turn is forfeited. (Hint: think of this as the enemy "doing nothing"). The game ends when the player successfully reaches the exit or the enemies destroy the human.
	Here is an example of an impossible board that was generated since the exit has no path to it from the player (you can't move diagonally). The rules for board layout are as follows: - The player always starts in the upper left corner - The exit is always in the lower right corner - Walls appear with a 20% chance in the spaces that are not the beginning space or the exit - Treasures appear with a 10% chance in spaces that are not walls, the beginning space, or the exit - You can decide where the enemies start, so long as they do not start on a wall or the exit.

Part 1 - Diagram your Program (due Tuesday, September 14th at 11:59pm)

We've provided you with the Makefile and headers for this homework. Your first goal is to plan out how they will interact. We've given you an example for the Player object here, so your job is to complete this for Board and Maze.

For each method of the given object, you should *briefly* describe what it does (feel free to re-word our comments into your own words) and which other methods it will need to call. The main goal here is for you to think through the structure of your program before writing it. If you find when you are implementing it, that you need to change some decisions that you made here, that is fine.

Example: Player

Player(const std::string name, const bool is_human)	initialize a Player with the given name and whether or not it is human
void ChangePoints(const int x)	update the points_ value according to the int passed in
void SetPosition(Position pos)	update the Player's pos_ to the new position
std::string ToRelativePosition(Position other)	translate the other position into a direction relative to the Player by comparing other with pos_

- Note: none of the methods here rely on calling other methods—this is not the case for Board and Maze!

Part 2 - Implement your Program (due Tuesday, September 21st at 11:59pm)
Step 1 — the "basic" game (80 points)
We are providing header files and a Makefile. Your job is to implement the guts of the objects and re-create this game. Your display doesn't need to exactly match ours (use your design and text UI skills as you wish), but the functionality should match ours.

The header files that we provide currently have most of the code commented out. This is so that you can uncomment and implement as you go. You **should not attempt to implement everything before compiling**.

Here are two strategies that you might use when implementing this game:
1) build your methods based on the workflow you're working on. For example: if you're working on building a board, you could work on writing the constructor and then overriding the << operator so that you can print out the Board.
2) start with `Player`, test out the `Player` object, then move on to `Board`, then `Maze`. Again, <u>test as you go</u>. Most of the functionality in `Maze` will depend on you having a working `Board`.

In either case, you should not implement more than 1 function at a time before testing it out.

The `TakeTurn` function in `Maze` will likely be the last one that you implement, as it will depend on having all the other functions in place and working.

There are screenshots of an entire run-through of the game on the last page of this write-up.

Once you have finished implementing your game so that it works with a single human player, move on to adding enemies to your game.

Number of enemies: you may pick any number from 2 - 4. You do not need to let the user choose how many enemies are in the game. Your enemies should be implemented in such a way that adding another does not require you to add more code other than the initial set up of the additional enemy.

<u>Note</u>: we have used emoji to display our game, but this is not a requirement. Some systems and linux configurations do not nicely display emoji on the terminal. It is 100% acceptable to use regular ASCII characters to display your game!

Step 2 — improving the game (5 points)
The game we have described is not *super* fun, nor is it *entirely* functional—it's possible (fairly likely even) that the player can't even reach the exit. There is only one kind of treasure. There aren't any traps. The enemies move via user input.

Once you have implemented the basic game, choose one of the following features to implement:
1) Ensure that there is always an open path from the player to the exit. For example, you could just generate new boards until one has a path. You should not just generate a board with the edges free of walls or with no walls at all.

2) Add different kinds of treasure. You should make the new kinds of treasure **both** have a different appearance from the basic treasure, and have a different point value. (add at least 2 new kinds of treasure)
3) Add some danger to the board in the form of stationary traps. Make it possible for the player to lose lives and perish before exiting the board. These traps could be large, they could be invisible, they could be made of fire—it is up to you.
4) Give the player a fixed amount of time or turns to complete the puzzle.

Make sure that you have included a comment in your main.cpp describing which of these 4 options you implemented!

General guidelines
The program that you turn in **must compile.** Even if your program is incomplete, **make sure it compiles**. (You'll get partial credit for a partially complete and compiling submission).

Similarly, the code that you turn in **must run**. You will get credit for programs that are partially complete, but you will get a hefty deduction for programs that do not run. **Make sure that it runs.** There will be autograder tests on Gradescope to test compilation and some functionality of your work. In addition, we will be testing your code on the coding.csel.io and the vm.

10 points will go to style and comments. Your files and functions should have comments. We should know how to run your program and how to use your functions from your comments. Refer to the style guide on the course github to help you out here.

Your variables should have meaningful names. Your code should be easily readable. If you have complex sections of code, you should use inline comments to clarify. You should follow the style guidelines posted on our course github.

You should not have redundant code—you should use loops, conditionals, and helper functions to avoid copy+pasting code. If you find yourself copy+pasting lots of code—pause and think about your design. Refer to the example code and guidelines associated with lecture 5 to help you here.

As a reference, our Maze.cpp is about 350 lines, our Player.cpp is about 40 lines, and our main.cpp is 25 lines (all counts including whitespace and comments).

Happy coding and remember to post questions on Ed! If you'd like to see more examples of what happens in different situations, just let us know and we will provide more examples!

Step 3 — extra credit (up to 15 bonus points)
You cannot get points for extra credit unless you have completed both step 1 and step 2. We **highly recommend** choosing #1 as the feature that you implement for step 2 if you are doing the extra credit.

Give the enemies strategies so they are computer controlled. If you have multiple enemies, they can all follow the same strategy or different strategies. Extra credit will be scaled on how sophisticated your enemy strategies are.
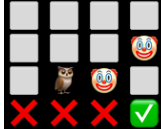
The example to the left shows a complete run through of the game.

The user's choice is case insensitive. If they enter an invalid option, they are prompted again.

Once the user steps onto a space with a treasure, that treasure should be collected — the user should have 100 more points and the treasure should disappear.

```
Romeo0 can go: LEFT DOWN
Please enter your choice: down
[grid]
X X X ✅

Romeo1 can go: UP LEFT RIGHT
Please enter your choice: up
[grid]
X X X ✅

Wanda can go: UP LEFT RIGHT
Please enter your choice: right
[grid]
X X X ✅

Romeo0 can go: UP LEFT DOWN
Please enter your choice: up
[grid]
X X X ✅

Romeo1 can go: UP LEFT RIGHT DOWN
Please enter your choice: up
[grid]
X X X ✅

Wanda can go: UP LEFT RIGHT
Please enter your choice: right
[grid]
X X X ✅

Romeo0 can go: LEFT DOWN
Please enter your choice: down
[grid]
X X X ✅
```
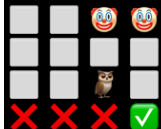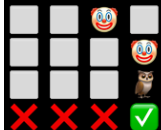
```
Romeo1 can go: LEFT RIGHT DOWN
Please enter your choice: down
[grid]
X X X ✅

Wanda can go: UP LEFT DOWN
Please enter your choice: down
[grid]
X X X ✅

Wanda has 101 points.Romeo0 has 0 points.Romeo1 has 100 points.
```

The human player earns 1 point for successfully getting to the exit.

When the game ends, it should report how many points each player has.