Programming Project 3, Part 2: Finding Queued Nemo

Overview: Read everything carefully

In this project you will be implementing basic artificial intelligence (AI) for Nemo. Nemo is lost in a large aquarium with a maze of coral. There are hungry sharks roaming the aquarium who will take a bite out of the poor fish if they encounter him. You will be provided with 19 files, you will only modify and submit 3 of them in addition to the two files you modified for the part 1.

You will be using your templated list implementation for this project. You should include both your list.h and studentinfo.h files with these files.

1. actor.h/.cpp: Base class for the Actors in the Aquarium. Notable members include the actors position in the Aquarium, the actor's state and interaction (discussed later), and also a pointer to the Aquarium itself. The actor does not create the Aquarium, it just points to it so it can obtain information about its surroundings. Similar to how a student may point to his or her classroom to use the classroom's computers, but the student was never a part of the classroom's creation. Do not modify these files, do not include this file in your submission.

2. player.h/.cpp: Derived from Actor, will be instantiated as the player Nemo. You will implement the update function for the Player, discussed later. Do not modify the header file, **the cpp file will be part of your submission.**

3. shark.h/.cpp: Derived from Actor, non-player characters that randomly move around in the maze. They will take a bite at Nemo if they are in the same cell as Nemo at any step. They will greet each other if there are multiple sharks in the same cell and Nemo is not. Do not modify these files, do not include this file in your submission.

4. aquarium.h/.cpp: Creates the maze and Actors, manages the Actors, and draws everything on the console. Do not modify these files, do not include this file in your submission.

5. game.h/.cpp: Creates the Aquarium and manages the game loop. It is important to understand that the game is already being driven with a loop in Game::play(). Each iteration of this loop is a single step of the game. That means when you implement the logic for the player to update his position, it is to update a **single** step; you should have no loops related to moving the player, just decide where to move and move once. Do not modify these files, do not include this file in your submission.

6. point.h/.cpp: A point class representing the (x,y) coordinates in the maze and Actors' position. Do not modify these files, do not include this file in your submission.

7. queue.h: Templated Queue interface using your linked list implementation. You will implement the functions for this class, discussed later. **This file will be part of your submission.**

8. stack.h: Templated Stack interface using your linked list implementation. You will implement the functions for this class, discussed later. **This file will be part of your submission.**

9. utils.h/.cpp: Some utility functions not specific to any of the above classes. Do not modify this file, do not include this file in your submission.

10. main.cpp: Entry point of the program, you will use this to test the code you develop. You are free to make any changes you want to this file, it will not be part of your submission. Note that the file I provide makes use of preprocessor macros separating different versions of a main function each testing various aspects of implemented code. This is to provide you some additional methods with which you can test your code (instead of commenting in/out code).
11. maze.txt/maze_lecture.txt: Input files for the maze. The aquarium will build its maze based on what is in whichever maze file passed to it. This is to allow ease of modifying/creating mazes. The mazes in these text files must follow certain rules since there are no checks in the program itself to ensure format:
    a. Walls are marked by 'X' and open cells are spaces.
    b. Mazes must be rectangle and completely enclosed.
    c. There can be no whitespace after the rightmost wall/column.
    d. There must be one newline after the last row.
    e. There must be a cell marked 'S' to indicate the starting position for Nemo. There must be a cell marked with 'E' to indicated the ending point for the maze.

    Note that when developing, in Visual Studio 2017, these input files can be placed within the same directory as your source files. In XCode, you must determine the location of your Working Directory, or set it through the project preferences/schemes. After that, you can put these files into your Working Directory.

    I provide some example executables, the windows version can have these files in the same directory as the executable. The OSX executables must have the files in your users/<USER> directory (the same directory that has your Downloads and Desktop directories).

Example Executables:

As mentioned above, I have, or will be providing example executables demonstrating a fully implemented project. There will be two versions, one that incorporates back tracking and one that does not. The details of backtracking will be discussed below, but you should refer to the example that does not have backtracking initially and make sure you code functions without backtracking first.

A few new things, maybe:

This project makes use of a C++ feature that may be new to you. This is found in actor.h with the declarations of:

```
enum class State { LOOKING, STUCK, BACKTRACK, FREEDOM };

enum class Interact { GREET, ATTACK, ALONE };
```

`enum class` are custom data types (like structs and classes) which group together constants that have some common theme among them. These are often used to clearly indicated various states of objects within the program and/or facilitate control flow depending on these constants. For example:

```
State myState = State::FREEDOM; // Declare a state, initialize to be FREE, MURICA!

// ... Some code that changes the state ...

switch (myState) {
case State::LOOKING:
	// ... code relating to looking ...
	break;
case State:::STUCK:
	// ... code relating to being stuck ...
	break;
default:
	// ... All other cases ...
}
```

The `enum class` is different from the `enum` you may have seen prior to this course. There are many differences, but suffice it to say that for most modern applications (C++11 and after) you should be using `enum class` rather than `enum`.

Meet the files you'll need to Modify:

stack.h/queue.h:

These are a stack and queue interface for your linked list. Be sure you understand the behavior of the stack and queue and how they must operate.

1. Stack()/Queue(): Default Constructor. Already fully implemented, creates and empty list.
2. void push(Type item): Adds the item into the stack or queue being consistent with how stacks or queues behave.
3. void pop(): Removes the item from the stack or queue consistent with how stacks or queues behave.. Note that some pop implementations also return the item that was popped, this implementation does not. Users must call the peek function if they wish to see the top of the stack or the front of the queue prior to popping.
4. bool isEmpty() const: returns whether or not the stack or queue is empty.
5. Type peek() const: Gets the item on the "Top" of the stack without changing the stack, or the "Front" of the queue without changing the queue.
6. void printStack() const/ void printQueue() const: Prints the items in the stack or queue for debugging purposes.

player.cpp:

There is only one function to implement in the player, Player::update(), all other functions are already implemented . You'll be completing the logic for the player , Nemo, to find the exit to the aquarium maze. The **queue based algorithm** is largely the same as what is illustrated in the lecture slides, so it is a good idea to familiarize yourself with that particular example prior to tackling this function.

The member variables for the Player include those of the Actor, since Player inherits from Actor. Of particular import are the following along with their setter/getter functions

- Actor::m_curr: The Actor's, thus Player's, current position in the Aquarium, you will be directly manipulating and checking this to make decisions on where to move next, and then move there.
- Actor::m_state: The Actor's, thus Player's, state will influence what actions will occur based on their state.
- Actor::m_aquarium: The Actor will need to obtain information from the aquarium such as if they are at the end point or if the cell they are examining is open. For this you will want to familiarize yourself with the Aquarium class. In particular, you could examine how Sharks (another type of Actor) behaves to get some hints.

Within the Player class:

- Player::m_look: This is exactly the queue that is used in the algorithm in the lecture slides.
- Player::m_discovered: A List that keeps track of all the cells that Nemo has discovered, this is different from what is used in the slides, in that the slides actually marks the cells in the world and checks that (think using chalk to mark your progress). What we're doing here instead is remembering the points we've discovered and keeping the aquarium free of graffiti.
- m_toggleBackTracking: If you decide to tackle the challenge of backtracking, all back tracking behavior should be toggled using m_toggleBackTracking, if false the player teleports the way seen in the algorithm from the lecture slides. This is largely to allow for testing your code without backtracking. I will first test with disabling backtracking to make sure you implementation behaves in the same way as indicted by the algorithm in the lecture slides.
- Player::m_backTrack: A stack to keep track of steps the player has taken. One major limitation to the algorithm in the slides is that it does not take into consideration of actual movement through the maze. It just tells you whether or not it is possible to reach the exit given the starting position. You should notice that the player will jump directly to the points to look around. This teleportation makes for unnatural movement.

  We want to smooth out that process so instead of jumping directly to the point we want Nemo to move one cell at a time. For example, in the slides after the player is at (2,1) and looks around. The next point in the queue is (1,2). However, if we have the player move directly to (1,2) it will appear as though the player teleported between iterations. We want the player to back track to the previous cell (1,1) first.

  Once at (1,1) the player should notice that his current position is **adjacent** to the cell he wants to visit next (1,2), that is in his m_look, note that adjacency does not include diagonal cells here. So, he no longer needs to be BACKTRAKING and should continue on LOOKING for the exit as normal. Note that when the player is in the process of BACKTRAKING he is doing nothing else, i.e. LOOKING for the exit. It isn't until he finishes BACKTRACKING that he continues LOOKING. In other words, the player, Nemo, can only do one thing at a time each iteration, either BACKTRACING or LOOKING, not both.

Back tracking is tricky, only attempt this after everything else. That is to say your Nemo should behave the same way as the example in the lecture slides; teleport around.

- Player:: m_btBufferStack/Player:: m_btBufferQueue: Helper structures to support backtracking algorithm. You may or may not need one or both. It depends on how you plan your backtracking algorithm. Both are used in the naive implementation of backtracking seen in the example executable. Again, that does not mean both are required. If you are able to figure out a way to facilitate backtracking with just the single stack:m_backTrack, then you are free to do so.

The first two steps of the algorithm are already done for you in Player's constructor, which should make sense since those steps set the initial state for the solving process and thus the initial state of Nemo, who will go through the process of solving the maze. Note that step 4 in popping the m_look queue, this implies that Nemo has actually moved to that location and so your implementation should manage that.

Submission:

What you submit should be code that has had no changes made to any of the other files and no changes to the class definitions to the classes you are to modify(list, stack, player). What this means is that, in the process of development you can tinker with everything as you please, to help you understand the functionality of all the classes. But, whatever you submit, must be able to compile with code, in the files/definitions mentioned, that have had no changes made to them. You submit 5 files:

list.h studentinfo.h stack.h queue.h player.cpp

You will have your own main.cpp for testing, but do not include it with your submission. Combine everything into a zip file name <lastname>_<id>.zip, for example nguyen_123456.zip. Note that when you resubmit on canvas it will postpend your file name with a number indicating its submission order, this is ok. If I take these 5 files, I must be able to compile them using VS2017 without any errors or warnings. Be sure to you do not introduce any compilation or link errors.