Programming Project 4: Breadth First Finding Nemo

IMPORTANT: Setup the project and get it to compile right away!

Overview: Read everything carefully

This project can be viewed as 4 components.

- Implementing the stack and queue using a linked list.
- Being able to understand and modify a significant existing code base.
- Implementing breadth first search as outlined in the lecture slides week 7(no backtracking)
- Incorporating backtracking.

I suggest approaching this project in this order and ensure that each step is validated prior to moving onto the next. This is critically important since if there are any bugs with any of the earlier portions of your development, they become many times more difficult to debug later on because they are hidden and compounded by other possible bugs.

In total, 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 17 source files, 5 supplementary files, and you will use the **list.h** implementation and **studentinfo.h** files from the last project. You will only modify and submit 3 files in addition to the two files you modified for the last project, 5 total files submitted.

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. Your submission must not change the class definition, only the member function implementations. **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. Your submission must not change the class definition, only the member function implementations. **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/etc: 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, 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).
12. settings.ini: Used to change various parameters of the program without the need to recompile.
    - `mazeFile:` maze file to open, for windows, if your file is in the same folder as the executable you can just use the file name. For OSX, you might want to use the full path to the file.
    - `frameTimeDelay:` The time between frames; shorter duration, faster the game.
    - `numSharks:` The number of sharks: Sharks have no effect on anything that you do but having them float around might make it difficult to compare your implementation, so set this to zero.

- `havePlayerBackTack:` Enable/Disable backtracking, set to true or false.

There are some checks to catch some input errors, but not for everything, so be a bit careful with what you change, e.g. true is not the same thing as True.

Example Executables:

I will be providing example executable binaries demonstrating a fully implemented project. You can use these to get a sense how your program should behave. That is to say that, for all possible mazes, for all possible starting/ending positions, if your program generates the same console output as the example executable frame-by-frame, then it is highly likely that every function called is implemented correctly. This also includes the dialog Nemo has with himself while navigating the maze.

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. They are often used to synchronize different parts of your program based in various events. For us, this means not only do we need to make sure that each classes' member variables are consistent with the objects' intended state, we also need to set these enums to be consistent with the intended program state in order to effectively synchronize. 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 empty () const: returns whether or not the stack or queue is empty.
5. Type top/front() 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 print() const: Prints the items in the stack or queue for debugging purposes. If either Stack or Queue is empty, this prints nothing.

player.cpp:

There is only one function to be 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_btStack/Player:: m_btQueue: Structures to support backtracking algorithm. You may or may not need one or both. It depends on how you plan your backtracking algorithm. You are free to design and algorithm you wish, the only constraint to backtracking is that nemo must navigate towards the next cell to LOOK, but can only move one **adjacent** cell per frame in doing so.

  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. While Nemo is in the process of `LOOKING` if he notices that his current position is not adjacent to the next position he wishes to look around, he must `BACKTRACK` his previous steps until he is adjacent to the cell he wishes to look around so he can continue with the process of `LOOKING.`

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.

- Player::update():
  This function is called each iteration where Nemo will move at most **one** cell, per iteration, and will either be LOOKING or BACKTRACKING, never both; there are also the states of FREEDOM and STUCK with should initiate the termination of the program. To begin with you should just translate the algorithm in the slides (without backtracking). This also means that you should use the same order of cells to look at, west east north south, otherwise your implementation will have different behavior. Once your implementation performs identically to the example binaries without backtracking you can tackle the backing tracking problem. Note that the algorithm we discussed in the slides corresponds to the state of LOOKING.

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, queue, 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 **only** submit 4 files:

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. If I take these 4 files, I must be able to compile them using VS2017/19 without any errors.