Programming Project 3

Finding Stacked Nemo

Overview:

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:

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 this file, 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 this file, 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 this file, do not include this file in your submission.
5. game.h/.cpp: Creates the Aquarium and manages the game loop. Do not modify this file, 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 this file, do not include this file in your submission.
7. list.h: A template for linked list. You will implement the functions for this template class, discussed later. Note that there is only one file because it is not an actual class, only a template for one. You must have the "implementation" in the same file as the class definition. There are workarounds that simulate separate files for templates but those can be confusing, so we're just sticking having a single file. Do not modify the class definition. **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 two C++ features that may be new to you. The first 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 (C11 and after) you should be using enum class rather than enum.

You will also encounter the assert() function, from assert.h, in implementing your functions for list.h. Typically, when implementing functions we need to ensure that the user provides proper inputs, otherwise improper inputs may cause undesired behavior, for example divide by zero, or the possibility of dereferencing invalid pointers. Simple ways to deal with these situations are to have if statements catching them and some indication that something has gone awry, cout statement, some return value encoding the error, exiting, etc. The point is to have some elegant form of exception handling. The built in feature in C++ involves using try-throw-catch, but the study can be involved and will not be focused on for this project.

assert isn't quite designed for actual released code, but for testing and catching invalid conditions that would result in malformed code. If any assert detects invalid code it will terminate the program and provide information regarding where the assertion failed. It is very handy for developing test cases and is easy to use. For example:

```
int input;
// ... Code modifying input ...
assert(input != 0); // I ASSERT MY DOMINANCE and that x is not zero, otherwise trouble!

int inverse = 1 / input;
```

If the input is set equal to 0 the assertion will fail and the program will terminate in a controlled manor. We can then go back and examine the conditions that resulted in the assertion failing and fix or manage the issue. In list.h, for the functions in which they appear, they're intended purpose is to act as exception handlers for when the function inputs are bad. Again, asserts are not designed for this task, you will want to explore try-throw-catch for a more elegant solution for production code.

list.h:

You will need to implement the member functions for the linked list class. This list (all linked lists for that matter) will be very similar to what we discussed in lecture. So, clearly understanding the how and the why behind the lecture's linked list implementation will go a long way in completing these functions. I cannot emphasize enough the importance of drawing/planning everything out before attempting to implement any of the functions, otherwise you can easily put down erroneous code. Any program with extensive use of pointers can be extremely difficult to debug after the fact, prevention is generally the best approach to anything, but is even truer in this case.

List's Member variables are provided to you. There are two Node pointers, head and tail, and an integer, size. If there are Nodes in the list the head must always point to the first Node, the tail must always point to the last Node in the list, and size must always be the same as the number of Nodes in the list. Your implementations must keep these invariants consistent with the intended state of the list when your functions complete. This is different from the lecture's linked list where there was only a head pointer.

The Node class is strictly internal to the List and so defined within the list itself, its definition is complete and nothing more is needed in its implementation.

list's functions:

1. List(): Default constructor. This should construct an empty List, the member variables should be initialized to reflect this state.
2. ~List(): Destructor. The list dynamically allocates nodes, that means when we destruct your list we need to ensure we deallocated the nodes appropriately to avoid memory leaks.
3. void printItems() const: Traverses the list and prints the items in the list in a single line, followed by a newline. printItems should indicate the Front and Rear of the list for example suppose our list contains the strings "Cash", "Shell", and "Ruby", printItems will display in the console:

   ```
   Front Cash Shell Ruby Rear
   ```
4. bool isEmpty() const: returns boolean value indicating if the list is empty or not.
5. void addToFront(Type item): Adds item to a new Node at the Front of the list. Updates head, tail, and size accordingly. Must appropriately handle cases in which the list is empty and if there are nodes already in the list.
6. void addToRear(Type item): Adds item to a new Node at the Rear of the list. Updates head, tail, and size accordingly. Must appropriately handle cases in which the list is empty and if there are nodes already in the list.
7. void addItem(int index, Type item): Given an index, this function adds the item to a new Node at the index. Updates head, tail, and size accordingly. If the index is less than or equal to zero add the item to the front. If the index is greater than or equal to the size of the list then add it to the rear. Otherwise add the item at the index indicated.

8. Type getFront() const: returns the item in the Node at the front of the list without modifying the list. The function cannot be called if the list is empty.
9. Type getRear() const: returns the item in the Node at the rear of the list without modifying the list. The function cannot be called if the list is empty.
10. Type getItem(index item) const: returns the item in the Node in the index place of the list.
11. int getSize() const: returns the size of the list.
12. int findItem(Type item): Searches the list to see if the item is currently in the list. If it is, the function returns the index of the item, otherwise it returns -1;
13. bool deleteFront():Removes the first item in the list, returns true if the item was deleted, false otherwise. Updates head, tail, and size accordingly. Must appropriately manage cases where the list is empty or has one or more items.
14. bool deleteRear() : Removes the last item in the list, returns true if the item was deleted, false otherwise. Updates head, tail, and size accordingly. Must appropriately manage cases where the list is empty or has one item, or has two or more items.
15. bool deleteItem(int index): Removes the item at the index of the list, returns true if the item was deleted false otherwise. Updates head, tail, and size accordingly. Must check to see if the index is inbounds.

stack.h:

This is a stack interface for your linked list. Be sure you understand the behavior of the stack and how it must operate.

1. Stack(): Default Constructor. Already fully implemented, creates and empty list.
2. void push(Type item): Adds the item onto the "Top" of the stack.
3. void pop(): Removes the item on the "Top" of the stack. 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 prior to popping.
4. bool isEmpty() const: returns whether or not the stack is empty.
5. Type peek() const: Gets the item on the "Top" of the stack without changing the stack.
6. void printStack() const: Prints the items in the stack 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 maze. The 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_brain: This is exactly the stack 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.
- Player::m_backTrack: A stack to keep track of all steps we've 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 after the third iteration the current position jumps for (x,y) = (1,3) to (2,1). This teleportation makes for unnatural movement.

  So 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 , suppose Nemo reaches (1,3) and searches around and discovers he has reached a dead end. A dead end being defined as being surrounded by walls and/or discovered cells. And so we want Nemo to walk backwards to cells he has already been to (in m_backTrack). Nemo will move to (1,2) in one step, and then (1,1) after that.

  Nemo then should notice that his current position is **adjacent** to the cell he wants to visit next (1,2), that is in his m_brain, 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 Nemo 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.

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.

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_brain stack, this implies that Nemo has actually moved to that location and so your implementation should manage that.

Tips:

Work with one thing at a time. Start with the list and make sure you have each function correct before moving onto the next. Do not include anything else; no game, no player, no aquarium, etc. They are entirely irrelevant to the operation of the list. After list is complete, then work on stack, and so on. If possible, test any change you make.

For ease of testing and development you can make your member variables public. Of course, don't forget to change it back to private prior to submission.

Compile and submit. The code that is provided to you should compile without error given an empty main. You should make sure you can do so before doing anything else. When you make significant headway, make sure what you have compiles and then submit it, that way if for any reason when you hit the deadline and for some reason you can't compile you at least have the previous working code to fall back on. Never submit anything that does not compile. Code that does not compile is worth less than code that does but has half the amount of work; in industry it is worth nothing.

Have test cases before executing any code. For any function, in planning how you want to implement that function, spend time jotting down possible ways to test that function. Make sure you have tests that visit all areas of code, meaning if you have if-else statements, you have a test case that will enter each of the branches. Not only that, make sure you know ahead of time what it is you expect to happen based on what you understand the function should be doing. That way you can compare "What is actually happening" with "What you intend". An unfortunate fact-of-life for programmers is that these two are seldom the same. It takes consider practice to be able to effectively delineate between the two notions; we have the tendency to bias one with the other.

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 3 files:

list.h stack.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 8 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.