Lab 7

4/5 October 2021

In this lab, you'll add to your `Maze` class a method `doesPathExist` that uses a `stack` to determine whether there's a path from start to finish. But first...

Stacks

I mentioned in class that the "call stack" was a special case of something more general called a *stack*. The concept of a stack is that, like a stack of (say) cafeteria trays, you can only add more on top and only remove from the top and you can't look at or access anything except on the top of the stack. As an abstract data type, that simplifies its interface considerably; an implementation of a stack only needs to support the following five (maybe just four) operations:

- push(item) adds a given item to this stack
- pop removes the appropriate item from this stack (and returns it, in some implementations)
- top [aka peek] returns the appropriate item from this stack (without changing the stack)
- isEmpty determines whether this stack is empty
- size computes the number of elements in this stack (omitted in some implementations)

Knowing that all stack operations only interact with the top of the stack, trace by hand the following series of stack operations, starting with an empty stack, by drawing a representation of a stack and adding elements as they're pushed and crossing them out as they're popped. Make sure you understand what each call will return (if anything) before you continue with this lab.

```
push (4)
push (7)
peek ()
```

```
push (10)
pop ()
push (13)
pop ()
pop ()
peek ()
push (15)
push (19)
pop ()
peek ()
```

Today we'll use the C++ builtin implementation `stack`, which implements these five operations as:

ADT name	C++ name
push(item)	`push(item)`
pop	`pop()`
top/peek	`top()`
isEmpty	`empty()`
size	`size()`

Note that in the standard C++ implementation, `push` and `pop` are `void` methods (they don't return anything).

Back to the maze

I know some of you didn't fully complete Labs 2 and 3, and I don't want that to hold you back on this lab, so I put my solution in `/home/shared/162/lab7/` that you can use as your starting point.

However, many of you had a complete or very-nearly-complete version of those labs, which I encourage you to build on in this lab. If so, copy your Lab 3 work into this week's directory and add methods `west`, `north`, and `south` to `Location` (following the pattern of `east`); and look through my solution just to see if there was anything else you missed. (Do look at `test_Maze.u` to see how `print` was tested; if you had a mostly working `Maze` but didn't test it, feel free to copy over just the test file from my version.)

Other than that, though, the end of Lab 3 is effectively your starting point for this lab. Our chief task this week is to write `doesPathExist`, which

determines whether this `Maze` has any legal path from its start to its finish; so, start off by declaring that method (with appropriate header) in the `.h` file, and define a stub for it in the `.cpp` file.

Test cases

Before we get into the maze solving algorithm, let's think about examples and test cases. Before, we were mostly concerned with making examples that would test reading and printing—and less concerned with the mazeness of the examples. Now, though, we want examples that will be able to effectively test a method that asks, "is there a path from the start to the finish?".

(A few of you were already thinking along these lines in your Lab 3 test cases. Well done! But ok if you didn't, too.)

So, let's add another few test cases. Draw out a few very short and simple mazes—some of which have solutions and some of which don't. (Hopefully at least: two that are super-simple, one where the start is surrounded on four sides by walls, and one where the start and finish are adjacent and surrounded by walls; and more that are still relatively small but with added complexity: some with and some without a solution, some with actual branching, and at least one where the border of the grid is not all walls, with the correct path to the finish skirting the edge of the grid.)

Add them to the fixture in the `.u` file.

Finally, write some test blocks for `doesPathExist` that call that method on each of the example `Maze`s defined in the fixture and records the expected result (true or false) for each one. (You could put them in a single test block, but it's sometimes nice to break them up: the tests that verify the simplest cases, the tests that check edge conditions, the tests that check branching, etc.) Compile and run the tests to confirm that they run, and that at least some of them fail.

Using `stack` in our maze

In the recursive backtracking technique we saw last week, functions on the call stack remembered where they left off—both the board itself (via the `current` variable) and which possible moves they had yet to try (via the `posn` variable). A different way to use the same basic technique is to use an actual stack variable (instead of the call stack) to more explicitly make those notes and keep track of places that we know we have yet to explore. These places constitute a sort of to-do list, or agenda, of work we have yet to do.

In the definition for `doesPathExist`, start by declaring a local variable to store the agenda of locations we have yet to explore:

```
    stack<Location> agenda;
```

To start the solution process, the first thing you'll need is to put an initial value in our `agenda`—namely, the start location. So, call the `push` method of the `agenda` object to add the start location to it.

You'll need to `#include <stack>` at the top of the file as well.

We're about to set this code aside for a moment, but before we do, check again that it compiles and runs (though the test will, of course, fail).

Doing the solution

On the next page, I've sketched out a framework for a maze-solving process. You'll have to think about it a little; in particular, although I've stated four different questions to ask at different points in the algorithm, I have omitted what to do in each case if the answer is yes vs. no. Should we keep going? Continue to the next iteration? Break the loop? Return something? Think about the algorithm, try using it on your examples above, and figure out how to fill in those blanks. Actually make notes (written down somewhere, not just in your head) that indicate what to do ("If so, ... If not, ...") and make sure you understand them. (If you're stuck, talk about it with the other students!)

```
Loop over the following:
    Is the agenda empty?


    Peek at the next agenda item, and pop it from the agenda.
    Is the element at that location the finish?


    Is the element at that location a wall?


    Make a local vector with all four adjacent locations.
    For each of them:
        Is it within the outline of the maze?


        Add it to the agenda.
```

After you feel moderately comfortable with the algorithm, implement it as the body of `doesPathExist`. You will want to make use of some of those other methods of `Maze` and `Location` that we defined earlier.

NOTE 1: look at the rubric on the back page before you start implementing; I've written it in a "suggested implementation order" that should make it both easier to write and easier to test.

NOTE 2: as written, this algorithm will still hang (infinite loop) on a number of inputs. Once you have an implementation that you think fully covers the above algorithm, but it's not working, consider the statement "Add the location to the already-explored list" and the question "Is that location in the already-explored list?"—and ask yourself where they fit in the algorithm and how to usefully integrate them into your program.

Handing in

Bundle up all your files, and don't forget a good readme, and hand in the work as `lab7`. I will want you to hand in the work-in-progress on Monday the 11th at 4pm; the final version will be due on Monday the 18th at 4pm.

Rubric

RUBRIC

1	Present and engaged in lab, good readme
	Tests and basics
1	Creates additional examples, using `stringstream`, with good coverage
1	Compiles, including stub (or more) for `doesPathExist` w/ correct header ♣
1/2	Test cases for `doesPathExist` (true and false, various cases, uses examples) ♣
	Implementation
1	Declares `stack` variable, uses at least one `stack` method
1	Accesses correctly all four `Locations` adjacent to current `Location`
1/2	...and adds them to the agenda under at least some conditions
1	Loops, returns false when the stack is empty ♣
1	...and true when the finish is found ♣
1	Declares and maintains explored list, ignoring previously-explored `Locations` ♣
1	Correctly implements algorithm on p. 4 of lab ♣

♣ indicates point is only available if the code compiles, with at least a stub for the relevant method(s).