Winter 2020 CS 32

Homework 2

Time due: 11:00 PM Tuesday, February 4

1. Write a C++ function named `pathExists` that determines whether or not a there's a path from start to finish in a rectangular maze. Here is the prototype:

```
bool pathExists(string maze[], int nRows, int nCols, int sr, int sc, int er, int ec);
  // Return true if there is a path from (sr,sc) to (er,ec)
  // through the maze; return false otherwise
```

   The parameters are

   - A rectangular maze of Xs and dots that represents the maze. Each string of the array is a row of the maze. Each `'X'` represents a wall, and each `'.'` represents a walkway.
   - The number of rows in the maze.
   - The number of columns in the maze. Each string in the maze parameter must be this length.
   - The starting coordinates in the maze: `sr`, `sc`; the row number is in the range 0 through nRows−1, and the column number is in the range 0 through nCols−1.
   - The ending coordinates in the maze: `er`, `ec`; the row number is in the range 0 through nRows−1, and the column number is in the range 0 through nCols−1.

   Here is an example of a simple maze with 5 rows and 7 columns:

```
"XXXXXXX"
"X...X.X"
"XXX.X.X"
"X.....X"
"XXXXXXX"
```

   The function must return true if in the maze as it was when the function was called, there is a path from `maze[sr][sc]` to `maze[er][ec]` that includes only walkways, no walls; otherwise it must return false. The path may turn north, east, south, and west, but not diagonally. When the function returns, it is allowable for the maze to have been modified by the function.

   Your solution must use the following simple class (without any changes), which represents an (r,c) coordinate pair:

```
class Coord
{
  public:
    Coord(int rr, int cc) : m_r(rr), m_c(cc) {}
    int r() const { return m_r; }
    int c() const { return m_c; }
  private:
    int m_r;
    int m_c;
};
```

   (Our convention is that (0,0) is the northwest (upper left) corner, with south (down) being the increasing r direction and east (right) being the increasing c direction.)

   Your implementation must use a stack data structure, specifically, a *stack of Coord*s. You may either write your own stack class, or use the stack type from the C++ Standard Library. Here's an example of the

relevant functions in the library's stack type:

```
#include <stack>
using namespace std;

int main()
{
    stack<Coord> coordStack;  // declare a stack of Coords

    Coord a(5,6);
    coordStack.push(a);          // push the coordinate (5,6)
    coordStack.push(Coord(3,4)); // push the coordinate (3,4)
    ...
    Coord b = coordStack.top();  // look at top item in the stack
    coordStack.pop();            // remove the top item from stack
    if (coordStack.empty())      // Is the stack empty?
        cout << "empty!" << endl;
    cout << coordStack.size() << endl; // num of elements
}
```

Here is pseudocode for your function:

```
Push the starting coordinate (sr,sc) onto the coordinate stack and
    update maze[sr][sc] to indicate that the algorithm has encountered
    it (i.e., set maze[sr][sc] to have a value other than '.').
While the stack is not empty,
{   Pop the top coordinate off the stack. This gives you the current
        (r,c) location that your algorithm is exploring.
    If the current (r,c) coordinate is equal to the ending coordinate,
        then we've solved the maze so return true!
    Check each place you can move from the current cell as follows:
        If you can move EAST and haven't encountered that cell yet,
            then push the coordinate (r,c+1) onto the stack and update
            maze[r][c+1] to indicate the algorithm has encountered it.
        If you can move SOUTH and haven't encountered that cell yet,
            then push the coordinate (r+1,c) onto the stack and update
            maze[r+1][c] to indicate the algorithm has encountered it.
        If you can move WEST and haven't encountered that cell yet,
            then push the coordinate (r,c-1) onto the stack and update
            maze[r][c-1] to indicate the algorithm has encountered it.
        If you can move NORTH and haven't encountered that cell yet,
            then push the coordinate (r-1,c) onto the stack and update
            maze[r-1][c] to indicate the algorithm has encountered it.
}
There was no solution, so return false
```

Here is how a client might use your function:

```
int main()
{
    string maze[10] = {
        "XXXXXXXXXX",
        "X...X..X.X",
        "X.XXX....X",
        "X.X.XXXX.X",
        "XXX......X",
        "X...X.XX.X",
        "X.X.X..X.X",
        "X.XXXX.X.X",
        "X..X...X.X",
        "XXXXXXXXXX"
    };

    if (pathExists(maze, 10,10, 4,3, 1,8))
```

```
        cout << "Solvable!" << endl;
    else
        cout << "Out of luck!" << endl;
}
```

Because the focus of this homework is on practice with the data structures, we won't demand that your function be as robust as we normally would. In particular, your function may make the following simplifying assumptions (i.e., you do not have to check for violations):

- the maze array contains nRows rows (you couldn't check for this anyway);
- each string in the maze is of length nCols;
- the maze contains only Xs and dots when passed in to the function;
- the top and bottom rows of the maze contain only Xs, as do the left and right columns;
- `sr` and `er` are between 0 and nRows-1, and `sc` and `ec` are between 0 and nCols-1;
- `maze[sr][sc]` and `maze[er][ec]` are `'.'` (i.e., not walls)

(Of course, since your function is not checking for violations of these conditions, make sure you don't pass bad values to the function when you test it.)

For this part of the homework, you will turn in one file named `mazestack.cpp` that contains the Coord class and your stack-based `pathExists` function. (Do not leave out the Coord class and do not put it in a separate file.) If you use the library's stack type, your file should include the appropriate header.

2. Given the algorithm, main function, and maze shown at the end of problem 1, what are the first 12 (r,c) coordinates popped off the stack by the algorithm?

   For this problem, you'll turn in either a Word document named `hw.docx` or `hw.doc`, or a text file named`hw.txt`, that has your answer to this problem (and problem 4).

3. Now convert your `pathExists` function to use a queue instead of a stack, making the fewest changes to achieve this. You may either write your own queue class, or use the queue type from the C++ Standard Library:

```
#include <queue>
using namespace std;

int main()
{
    queue<Coord> coordQueue;   // declare a queue of Coords

    Coord a(5,6);
    coordQueue.push(a);           // enqueue item at back of queue
    coordQueue.push(Coord(3,4));  // enqueue item at back of queue
    ...
    Coord b = coordQueue.front(); // look at front item
    coordQueue.pop();             // remove the front item from queue
    if (coordQueue.empty())       // Is the queue empty?
        cout << "empty!" << endl;
    cout << coordQueue.size() << endl; // num of elements
}
```

   For this part of the homework, you will turn in one file named `mazequeue.cpp` that contains the Coord class and your queue-based `pathExists` function. (Do not leave out the Coord class and do not put it in a separate file.) If you use the library's queue type, your file should include the appropriate header.

4. Given the same main function and maze as are shown at the end of problem 1, what are the first 12 (r,c) coordinates popped from the queue in your queue-based algorithm?

How do the two algorithms differ from each other? (Hint: how and why do they visit cells in the maze in a different order?)

For this problem, you'll turn in either a Word document named `hw.docx` or `hw.doc`, or a text file named`hw.txt`, that has your answer to this problem (and problem 2).

5. Implement this function that evaluates an infix integer arithmetic expression that consists of the binary operators `+`, `-`, `*`, and `/`, parentheses, and operands (with blanks allowed for readability). The `/` operator denotes integer division (with truncation), so that the value of eight divided by five is 1, not 1.6. Operators have their conventional precedence and associativity. Multiplication must be explicitly indicated with the`*` operator.

   The operands in the expression are single lower case letters. Along with the expression string, you will pass the function a Map with key type `char` and value type `int`. Each letter character in the expression represents the integer value in the map that is paired with that letter key. For example, if the map maps `a` to 3, `c` to 5, `l` to 2, and `u` to 11, then the expression `u-c+l*a` would evaluate to 12.

   Here is the function:

```
int evaluate(string infix, const Map& values, string& postfix, int& result);
  // Evaluates an integer arithmetic expression
  //   If infix is a syntactically valid infix integer expression whose
  //   only operands are single lower case letters (whether or not they
  //   appear in the values map), then postfix is set to the postfix
  //   form of the expression; otherwise postfix may or may not be
  //   changed, result is unchanged, and the function returns 1.  If
  //   infix is syntactically valid but contains at least one lower
  //   case letter operand that does not appear in the values map, then
  //   result is unchanged and the function returns 2.  If infix is
  //   syntactically valid and all its lower case operand letters
  //   appear in the values map, then if evaluating the expression
  //   (using for each letter in the expression the value in the map
  //   that corresponds to it) attempts to divide by zero, then result
  //   is unchanged and the function returns 3; otherwise, result is
  //   set to the value of the expression and the function returns 0.
```

   Adapt the algorithms presented on pp. 203-207 of the textbook to convert the infix expression to postfix, then evaluate the postfix form of the expression. The algorithms use stacks. Rather than implementing the stack types yourself, you must use the `stack` class template from the Standard C++ library. You may *not*assume that the infix string passed to the function is syntactically valid; you'll have to detect whether it is or not.

   For this problem, you will turn in a file named `eval.cpp` whose structure is probably of the form

```
#include lines you need, including "Map.h"

declarations of any additional functions you might have written
        to help you evaluate an expression

int evaluate(string infix, const Map& values, string& postfix, int& result)
{
    your expression evaluation code
}

implementations of any additional functions you might have written
        to help you evaluate an expression

a main routine to test your function
```

Use a correct implementation of Map (perhaps from the Homework 1 or Project 2 solution). You will not turn in `Map.h` or `Map.cpp`. (We will use correct versions when we test your code.)

If we take your `eval.cpp` file, rename your main routine (which we will never call) to something harmless, prepend the lines

```
#include "Map.h"
#include <iostream>
#include <string>
#include <stack>
#include <cctype>
#include <cassert>
using namespace std;
```

if necessary, and append the lines

```
int main()
{
    char vars[] = { 'a', 'e', 'i', 'o', 'u', 'y', '#' };
    int  vals[] = {  3,  -9,   6,   2,   4,   1  };
    Map m;
    for (int k = 0; vars[k] != '#'; k++)
        m.insert(vars[k], vals[k]);
    string pf;
    int answer;
    assert(evaluate("a+ e", m, pf, answer) == 0  &&
                            pf == "ae+"  &&  answer == -6);
    answer = 999;
    assert(evaluate("", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("a+", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("a i", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("ai", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("()", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("()o", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("y(o+u)", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("y(*o)", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("a+E", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("(a+(i-o)", m, pf, answer) == 1  &&  answer == 999);
      // unary operators not allowed:
    assert(evaluate("-a", m, pf, answer) == 1  &&  answer == 999);
    assert(evaluate("a*b", m, pf, answer) == 2  &&
                            pf == "ab*"  &&  answer == 999);
    assert(evaluate("y +o *(   a-u)  ", m, pf, answer) == 0  &&
                            pf == "yoau-*+"  &&  answer == -1);
    answer = 999;
    assert(evaluate("o/(y-y)", m, pf, answer) == 3  &&
                            pf == "oyy-/"  &&  answer == 999);
    assert(evaluate(" a  ", m, pf, answer) == 0  &&
                            pf == "a"  &&  answer == 3);
    assert(evaluate("((a))", m, pf, answer) == 0  &&
                            pf == "a"  &&  answer == 3);
    cout << "Passed all tests" << endl;
}
```

then the resulting file must compile and build successfully when `Map.h` and `Map.cpp` are present, and when executed, produce no output other than `Passed all tests`.

(Tips: In case you didn't already know it, you can append a character `c` to a string `s` by saying `s += c`. You'll have to *adapt* the code from the book, since it doesn't do any error checking. It's possible to do the error checking as you do the infix-to-postfix conversion instead of in a separate step before that; as you go through the infix string, almost all syntax errors can be detected by noting whether it is legal for the current nonblank character to follow the nearest nonblank character before it.)

By Monday, February 3, there will be a link on the class webpage that will enable you to turn in this homework. Turn in one zip file that contains your solutions to the homework problems. The zip file should contain

- `mazestack.cpp`, if you solved problem 1
- `mazequeue.cpp`, if you solved problem 3
- `eval.cpp`, if you solved problem 5
- `hw.docx`, `hw.doc`, or `hw.txt`, if you solved problems 2 and/or 4

Each source file you turn in may or may not contain a main routine; we'd prefer that it doesn't. If it does, our testing scripts will rename your main routine to something harmless. Our scripts will append our own main routine, then compile and run the resulting source file. Therefore, to get any credit, each source file you turn in must at least compile successfully (even though it's allowed to not link because of a missing main routine).

In each source file you turn in, do not comment out your implementation; you want our test scripts to see it! (Some people do this when testing other files' code because they put all their code in one project instead of having a separate project for each of problems 1, 3, and 5.)