Spring 2018 CS 32

Homework 2

Time due: 11:00 PM Tuesday, May 1

1. Write a C++ function named `pathExists`that determines whether or not a there's a path from start to finish in a 10x10 maze. Here is the prototype:

```
bool pathExists(char maze[][10], 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 2-dimensional 10x10 character array of Xs and dots that represents the maze. Each `'X'` represents a wall, and each `'.'` represents a walkway.
   - The starting coordinates in the maze: `sr`, `sc`; the row number and column number range from 0 to 9.
   - The ending coordinates in the maze: `er`, `ec`; the row number and column number range from 0 to 9.

   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 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.
        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 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 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.
}
There was no solution, so return false
```

Here is how a client might use your function:

```
int main()
{
    char maze[10][10] = {
        { 'X','X','X','X','X','X','X','X','X','X' },
        { 'X','.','.','.','.','.','.','.','.','X' },
        { 'X','X','X','X','X','.','X','.','X','X' },
        { 'X','.','.','.','X','.','X','.','.','X' },
        { 'X','.','X','.','.','.','X','.','.','X' },
        { 'X','.','X','X','X','.','X','X','X','X' },
        { 'X','X','X','.','.','.','.','X','.','X' },
        { 'X','.','X','X','.','X','X','X','.','X' },
        { 'X','.','.','.','.','.','X','.','.','X' },
        { 'X','X','X','X','X','X','X','X','X','X' }
    };

    if (pathExists(maze, 6,5, 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 has 10 rows;
- 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`, `sc`, `er`, and `ec` are between 0 and 9;
- `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.doc` or `hw.docx`, 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.doc` or `hw.docx`, 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 boolean expression that consists of the binary boolean operators & (meaning *and*) and | (meaning *inclusive or*), the unary boolean operator ! (meaning *not*), parentheses, and the operands T and F, with blanks allowed for readability. Following convention, ! has higher precedence than &, which has higher precedence than |, and operators of equal precedence associate left to right (so the postfix form of `T|F|T` is `TF|T|`, not `TFT||`, which would be the postfix form of `T|(F|T)`. In evaluating the expressions, T represents the value *true*, and F *false*.

   Here are some examples of valid expressions:

```
T                          evaluates to true
(F)                        evaluates to false
T&(F)                      evaluates to false
T & !F                     evaluates to true
!(F|T)                     evaluates to false
!F|T                       evaluates to true
T|F&F                      evaluates to true
T&!(F|T&T|F)|!!!(F&T&F)    evaluates to true
```

   Here is the function:

```
int evaluate(string infix, string& postfix, bool& result)
  // Evaluates a boolean expression
  //   If infix is a syntactically valid infix boolean expression,
  //   then set postfix to the postfix form of that expression, set
  //   result to the value of the expression, and return zero.  If
  //   infix is not a syntactically valid expression, return 1; in
  //   that case, postfix may or may not be changed, but result must
  //   be unchanged.
```

   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
```

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

int evaluate(string infix, string& postfix, bool& 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
```

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

```
#include <iostream>
#include <string>
#include <stack>
#include <cassert>
using namespace std;
```

if necessary, and append the lines

```
int main()
{
    string pf;
    bool answer;
    assert(evaluate("T| F", pf, answer) == 0  &&  pf == "TF|"  &&  answer);
    assert(evaluate("T|", pf, answer) == 1);
    assert(evaluate("F F", pf, answer) == 1);
    assert(evaluate("TF", pf, answer) == 1);
    assert(evaluate("()", pf, answer) == 1);
    assert(evaluate("T(F|T)", pf, answer) == 1);
    assert(evaluate("T(&T)", pf, answer) == 1);
    assert(evaluate("(T&(F|F)", pf, answer) == 1);
    assert(evaluate("", pf, answer) == 1);
    assert(evaluate("F  |  !F & (T&F) ", pf, answer) == 0
                         &&  pf == "FF!TF&&|"  &&  !answer);
    assert(evaluate(" F  ", pf, answer) == 0 &&  pf == "F"  &&  !answer);
    assert(evaluate("((T))", pf, answer) == 0 &&  pf == "T"  &&  answer);
    cout << "Passed all tests" << endl;
}
```

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

Do not use variables named `and`, `or`, or `not`; the g++ and clang++ compilers treat these as keywords (as the C++ Standard requires) even though Visual C++ doesn't.

(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 and assumes that all operators are binary operators. 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. When converting infix to postfix, a unary operator like ! behaves a lot more like an open parenthesis than like a binary operator.)

By Monday, April 30, 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.doc`, `hw.docx`, 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.)