**Homework 2**

**Time due: 9:00 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:
2. bool pathExists(char maze[][10], int sr, int sc, int er, int ec);
3. // Return true if there is a path from (sr,sc) to (er,ec)
4. // 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 determine if there is a path from maze[sr][sc] to maze[er][ec] that includes only walkways, no walls. The path may turn only north, east, south, and west, but not diagonally. You must use a stack data structure.

Your solution will use the following simple class, 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.)

The stack you must use is a *stack of Coord*s. You may either write your own stack class, or use the stack class in the C++ Standard Library (pp. 309-311 in the textbook discuss that class). Here's an example of the relevant functions in the library's stack class:

#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:*

*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 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.*

*}*

*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'}

};

if (pathExists(maze,1,1,8,8))

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;
  + 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.)

1. Given the algorithm, main function, and maze shown at the end of problem 1, what are the first 9 (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).

1. Now convert your pathExists function to use a queue instead of a stack, making the fewest changes to achieve this. You can use the C++ Standard Library's queue class (described in pp. 364-367 of the textbook) or write your own, if you like:
2. #include <queue>
3. using namespace std;
5. int main()
6. {
7. queue<Coord> coordQueue; // declare a queue of Coords
9. Coord a(5,6);
10. coordQueue.push(a); // enqueue item at back of queue
11. coordQueue.push(Coord(3,4)); // enqueue item at back of queue
12. ...
13. Coord b = coordQueue.front(); // look at front item
14. coordQueue.pop(); // remove the front item from queue
15. if (coordQueue.empty()) // Is the queue empty?
16. cout << "empty!" << endl;
17. cout << coordQueue.size() << endl; // num of elements
18. }

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.)

1. Given the same main function and maze as are shown at the end of problem 1, what are the first 9 (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).

1. Implement this function that evaluates an infix boolean expression that consists of the operands T and F, parentheses, the binary boolean operators & and |, and the unary boolean operator ! (with blanks allowed for readability). Following convention, ! has higher precedence than &, which has higher precedence than |. In evaluating the expressions, T represents the value *true*, and F *false*. Here are some examples of valid expressions:
2. T evaluates to true
3. (F) evaluates to false
4. T&(F) evaluates to false
5. T & !F evaluates to true
6. !(F|T) evaluates to false
7. !F|T evaluates to true
8. T|F&F evaluates to true
9. T&!(F|T&T|F)|!!!(F&T&F) evaluates to true

Here is the function:

int evaluate(const string& infix, string& postfix, bool& result)

// Evaluates a boolean expression

// Precondition: infix is an infix boolean expression consisting of

// the symbols T and F, the operators |, &, and !, and parentheses,

// with embedded blanks allowed for readability.

// Postcondition: If infix is a syntactically valid infix boolean

// expression, postfix is set to the postfix form of that

// expression, result is set to the value of the expression, and

// the function returns zero. If infix is a malformed expression,

// the return value is 1. (In that case, postfix may or may not be

// changed and but result must be unchanged.)

Adapt the algorithms presented on [pp. 311-316 of the textbook](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/2/bookcode.html) 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.

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(const string& infix, string& postfix, bool& result)

{

your expression evaluation code

}

implementations of any 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 postfix;

bool answer;

assert(evaluate("T| F", postfix, answer) == 0 && postfix == "TF|"

&& answer);

assert(evaluate("T|", postfix, answer) == 1);

assert(evaluate("F F", postfix, answer) == 1);

assert(evaluate("(T&(F|F)", postfix, answer) == 1);

assert(evaluate("F | !F & (T&F) ", postfix, answer) == 0

&& postfix == "FF!TF&&|" && !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.

(Tips: In case you didn't already know it, you can append a character c to a string s by saying s += c. As you go through the infix string, almost all syntax errors can be detected by noting whether the current nonblank character is allowed 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() 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).