HW2 Solution

Problem 1:

```
#include <stack>
#include <string>

using namespace std;

const char WALL = 'X';
const char OPEN = '.';
const char SEEN = 'o';

class Coord
{
  public:
    Coord(int r, int c) : m_row(r), m_col(c) {}
    int r() const { return m_row; }
    int c() const { return m_col; }
  private:
    int m_row;
    int m_col;
};

void explore(string maze[], stack<Coord>& toDo, int r, int c)
{
        if (maze[r][c] == OPEN)
        {
             toDo.push(Coord(r,c));
             maze[r][c] = SEEN;  // anything non-OPEN will do
        }
}

bool pathExists(string maze[], int nRows, int nCols, int sr, int sc, int er, int
ec)
{
    if (sr < 0  ||  sr >= nRows  ||  sc < 0  || sc >= nCols  ||
        er < 0  ||  er >= nRows  ||  ec < 0  || ec >= nCols  ||
        maze[sr][sc] != OPEN  ||  maze[er][ec] != OPEN)
       return false;

    stack<Coord> toDo;
    explore(maze, toDo, sr, sc);

    while ( ! toDo.empty() )
    {
        Coord curr = toDo.top();
        toDo.pop();

        const int cr = curr.r();
```

```
        const int cc = curr.c();

        if (cr == er  &&  cc == ec)
            return true;

        explore(maze, toDo, cr, cc+1);  // east
        explore(maze, toDo, cr-1, cc);  // north
        explore(maze, toDo, cr, cc-1);  // west
        explore(maze, toDo, cr+1, cc);  // south
    }
    return false;
}
```

Problem 2:

```
(5,3) (6,3) (4,3) (4,2) (4,1) (3,1) (2,1) (1,1) (1,2) (3,3) (5,4) (5,5)
```

Problem 3:

Make three changes to the Problem 1 solution:

- Change #include to #include
- Change stack to queue
- Change Coord curr = toDo.top(); to Coord curr = toDo.front();

Problem 4:

```
(5,3) (5,4) (4,3) (6,3) (5,5) (3,3) (4,2) (5,6) (4,5) (4,1) (5,7) (3,5)
```

The stack solution visits the cells in a depth-first order: it continues along a path until it hits a dead end, then backtracks to the most recently visited intersection that has unexplored branches. Because we're using a stack, the next cell to be visited will be a neighbor of the most recently visited cell with unexplored neighbors.

The queue solution visits the cells in a breadth-first order: it visits all the cells at distance 1 from the start cell, then all those at distance 2, then all those at distance 3, etc. Because we're using a queue, the next cell to be visited will be a neighbor of the least recently visited cell with unexplored neighbors.

Problem 5:

```
// eval.cpp

#include "Set.h"  // element type is char
#include <string>
#include <stack>
#include <cctype>
#include <cassert>
using namespace std;

const int RET_OK_EVALUATION               = 0;
const int RET_INVALID_EXPRESSION          = 1;
const int RET_VARIABLE_NO_VALUE           = 2;
const int RET_VARIABLE_CONFLICTING_VALUES = 3;

inline
bool isLowerOrCloseParen(char ch)
{
    return islower(ch) || ch == ')';
}

inline
int precedence(char ch)
  //  Precondition:  ch is in "|&!("
{
    static string ops = "|&!(";
    static int prec[4] = { 1, 2, 3, 0 };
    int pos = ops.find(ch);
    assert(pos != string::npos);  // must be found!
    return prec[pos];
}

bool convertInfixToPostfix(const string& infix, string& postfix)
  // Convert a boolean expression to postfix
  //   If infix is not a syntactically valid infix boolean expression,
  //   the function returns false.  (postfix may or may not be changed.)
  //   Otherwise, postfix is set to the postfix form of that expression,
  //   and the function returns true.
{
    postfix = "";
    stack<char> operatorStack;
    char prevch = '|';  // pretend the previous character was an operator

    for (size_t k = 0; k != infix.size(); k++)
    {
        char ch = infix[k];
        if (islower(ch))
        {
            if (isLowerOrCloseParen(prevch))
                return false;  // invalid expression
            postfix += ch;
        }
        else
        {
```

```
            switch(ch)
            {
              case ' ':
                continue;  // do not set prevch to this char

              case '(':
              case '!':
                if (isLowerOrCloseParen(prevch))
                    return false;  // invalid expression
                operatorStack.push(ch);
                break;

              case ')':
                if ( ! isLowerOrCloseParen(prevch))
                    return false;  // invalid expression
                for (;;)
                {
                    if (operatorStack.empty())
                        return false;  // invalid expression (too many ')')
                    char c = operatorStack.top();
                    operatorStack.pop();
                    if (c == '(')
                        break;
                    postfix += c;
                }
                break;

              case '|':
              case '&':
                if ( ! isLowerOrCloseParen(prevch))
                    return false;  // invalid expression
                while ( ! operatorStack.empty()  &&
                           precedence(ch) <= precedence(operatorStack.top()) )
                {
                    postfix += operatorStack.top();
                    operatorStack.pop();
                }
                operatorStack.push(ch);
                break;

              default:  // bad char
                return false;  // invalid expression
            }
        }
        prevch = ch;
    }

      // end of expression; pop remaining operators

    if ( ! isLowerOrCloseParen(prevch))
        return false;  // invalid expression
    while ( ! operatorStack.empty())
    {
        char c = operatorStack.top();
```

```
        operatorStack.pop();
        if (c == '(')
            return false;  // invalid expression (too many '(')
        postfix += c;
    }
    if (postfix.empty())
        return false;  // invalid expression (empty)

    return true;
}
```

```
int evaluate(string infix, const Set& trueValues, const Set& falseValues, string&
postfix, bool& result)
  // Evaluate a boolean expression
  //   If infix is a syntactically valid infix boolean expression whose
  //   only operands are single lower case letters (whether or not they
  //   appear in the values sets), then postfix is set to the postfix
  //   form of the expression.  If not, postfix might or might not be
  //   changed, result is unchanged, and the function returns 1.
  //
  //   If infix is a syntactically valid infix boolean expression whose
  //   only operands are single lower case letters:
  //
  //      If every operand letter in the expression appears in either
  //      trueValues or falseValues but not both, then result is set to the
  //      result of evaluating the expression (using for each letter in the
  //      expression the value true if that letter appears in trueValues or
  //      false if that letter appears in false values) and the function
  //      returns 0.
  //
  //      Otherwise, result is unchanged and the value the function returns
  //      depends on these two conditions:
  //        at least one letter in the expression is in neither the
  //            trueValues nor the falseValues sets; and
  //        at least one letter in the expression is in both the
  //            trueValues and the falseValues set.
  //      If only the first condition holds, the function returns 2; if
  //      only the second holds, the function returns 3.  If both hold
  //      the function returns either 2 or 3 (and the function is not
  //      required to return the same one if called another time with the
  //      same arguments).
{
      // First convert infix to postfix

    if ( ! convertInfixToPostfix(infix, postfix))
        return RET_INVALID_EXPRESSION;

      // Now evaluate the postfix expression

    stack<bool> operandStack;
    for (size_t k = 0; k != postfix.size(); k++)
```

```
    {
        char ch = postfix[k];
        if (islower(ch))
        {
            bool isTrue  = trueValues.contains(ch);
            bool isFalse = falseValues.contains(ch);
            if (!isTrue && !isFalse)
                return RET_VARIABLE_NO_VALUE;
            if (isTrue && isFalse)
                return RET_VARIABLE_CONFLICTING_VALUES;
            operandStack.push(isTrue);
        }
        else
        {
            bool opd2 = operandStack.top();
            operandStack.pop();
            if (ch == '!')
                operandStack.push(!opd2);
            else
            {
                bool opd1 = operandStack.top();
                operandStack.pop();
                if (ch == '&')
                    operandStack.push(opd1 && opd2);
                else if (ch == '|')
                    operandStack.push(opd1 || opd2);
                else  // Impossible for valid postfix string!
                    return RET_INVALID_EXPRESSION;  // Pretend it's an invalid
expression
            }
        }
    }
    if (operandStack.size() != 1)  // Impossible for valid postfix string!
        return RET_INVALID_EXPRESSION;  // Pretend it's an invalid expression
    result = operandStack.top();

    return RET_OK_EVALUATION;
}
```

```
// Here's an interactive test driver:
//
// #include "Set.h"  // element type is char
// #include <iostream>
// #include <string>
// using namespace std;
//
// int main()
// {
//     Set trues;
//     Set falses;
//     for (char ch = 't'; ch <= 'z'; ch++)
```

```
//         trues.insert(ch);
//     for (char ch = 'a'; ch <= 'f'; ch++)
//         falses.insert(ch);
//     trues.insert('r');
//     falses.insert('r');
//     cout << "Use a-f for false, t-z for true, r for both." << endl;
//     string s;
//     while (getline(cin,s)  &&  s != "quit")
//     {
//         string postfix;
//         bool val;
//         switch (evaluate(s, trues, falses, postfix, val))
//         {
//           case RET_OK_EVALUATION:
//             cout << "Postfix is " << postfix << " and value is "
//                  << (val ? "true" : "false") << endl;
//             break;
//           case RET_INVALID_EXPRESSION:
//             cout << "Malformed expression" << endl;
//             break;
//           case RET_VARIABLE_NO_VALUE:
//             cout << "There's a variable that is neither true nor false" <<
endl;
//             break;
//           case RET_VARIABLE_CONFLICTING_VALUES:
//             cout << "There's a variable that is both true and false" << endl;
//             break;
//           default:
//             cout << "Impossible return code" << endl;
//             break;
//         }
//     }
// }
```