CS 3424 Data Structures Spring 2008

Assignment 2

Assignment Date: 2/12/08 Due Date: 2/21/08 Early Date: 2/17/08

Description: Maze Solver with thanks to **Dr. K. Ganesan**

In this program, you will write a program to solve a maze problem. A maze can be thought of as an m x n grid of cells. The row numbers are 0..m-1 and the column number are 0..n-1. Each grid cell is like a room with 4 sides: north, east, south, and west. A side of the room is either a complete wall or a wall with an open door (entry way). We can represent a room with 4 bits nesw (using clockwise starting from north) where n e s w are bit values for the corresponding direction. A bit value of 1 indicates that there is an open entry way on that side of the room and a bit value of 0 indicates that there is no way (i.e. complete wall). Thus 1001 indicates that north and west has doors and other two sides don't. We can associate a position with each bit starting from the rightmost bit. Thus, north has bit position 3, east has bit position 2, south has bit position 1, and west has bit position 0. We can represent this with a single number (in this case 9) in the range 0..15. Assume that there is light in each room so you can see (Is this sentence important for programming? :-)). A mouse is initialized placed in some room and a cheese is placed in another (far from the mouse). Can you help the mouse eat the cheese?
You will solve this problem using two different implementations. The first version uses a stack and the second one uses a queue. Use the stack and queue classes from the STL. Your application will have two versions, one using stack and one using queue. (Note that the application itself is using a different ADT as opposed to an ADT using a different implementation. There is a difference!! So, you need to rewrite the application).
Build a Maze ADT that can be used to load a maze from a file, solve it, and print the solution. The main program uses the Maze ADT to solve the problem. You need two versions of the Maze ADT, one using stack and the other using queue. Call them maze-stk.h[cpp] and maze-q.h[cpp].
The format of the maze file is as follows:

```
m n
mouserow mousecolumn
cheeserow cheesecolumn
room[0][0]      room[0][1]     room[0][2]    ... room[0][n-1]
room[1][0]      room[1][1]     room[1][2]    ... room[1][n-1]
:
room[m-1][0]    room[m-1][1]   room[m-1][2]  ... room[m-1][n-1]
```

where each room[i][j] is a number in the range 0..15 representing the 4 bits. Look at a room, and start writing the bit vector starting from west and going counter clockwise. Then convert it to a single number. (west is worth 8, south is 4, east is 2, north is 1).
The maze ADT should have a member function to solve this maze. Its output is either "Maze has no solution." or it prints a path for the mouse to take to reach the cell with cheese. The solution is a sequence of letters from the set {N E W S} which, if followed, will get the mouse to the cheese. The letter indicates the direction in which the mouse should move.

Algorithm idea:

The way you find the solution is as follows: For each square we want to keep track of the following information:

```
visited   -- 1 ==> the cell has been visited. 0 => not visited.
parent    -- parent (i.e. previous cell) if the cell has been visited.
             For the first cell (mouse's initial position) there is no parent.
             You can use -1 for the parent row and parent column.
direction -- The direction (i.e. the letter) from parent to this cell.
```

Note that the solution path can have at most n^2 letters.
We start with the cell in which the mouse is placed. Initialize all the fields for this square. We place it in the stack or queue, whichever we are using. Then we start a loop to explore the maze. The loop stops when we hit the cell with cheese or when the stack or queue has no items in it (i.e. empty). In each iteration of the loop, we take a cell from the stack or queue, mark it visited. If it is the cheese cell, then stop the search. If not, we continue. It should already have the parent and direction pointers set when it was originally placed in stack or queue. After marking, examine the room to see which doors are available and determine the neighboring cells. For each neighbor cell that has NOT been visited, set the parent and direction to use the current cell we are examining and then place the neighbor cell in stack or queue.
When the search stops, examine the cheese square to see if it was visited. If not, there is no solution. If it was, we can backtrack using the direction all the way to the initial square (If you did visit the cheese square, the path will take you back to starting cell. You can count on it.) and store all the letters in a solution array (vector) and print that array (vector).
The maze should have a constructor function to load the maze from a file. You can use a two-dimensional vector of Cell Structures where each cell structure stores information about a cell. The constructor and destructor functions for the vector class take care of the dynamic allocation and deallocation. The maze class itself does not have any dynamically allocated storage and hence the destructor function can be empty. The Maze ADT can have a member function called Solve to solve the maze. The function Solve will need stack or queue locally to help the search. Each stack or queue item is a structure with row and column numbers. Note that the stack object is local to this function and not a member of the maze class itself. The maze class will also have a function PrintSolution that will check the two dimensional vector to see if the solution exists. If not, it prints an appropriate solution and returns. If the solution exists, it should form the solution using a local one-dimensional vector of characters and then print this solution.

The class definition (You need to fill in all the necessary comments) for the maze can be:

```
#include <string>
typedef struct
{
    short int doorEncoding; // Range 0..15.
            // north = doorEncoding & 0x01
            // east  = doorEncoding & 0x02
            // south = doorEncoding & 0x04
            // west  = doorEncoding & 0x08
            // Note that the result of these operations is not
            // necessarily 1, but some non-zero value if there
            // is in that particular direction. 0 else.
    bool     visited;
    int      parentRow;
    int      parentCol;
    char     direction; // From parent.
} MazeCell;
typedef struct
{
    int row;
    int col;
} CellPosition; // useful as StackItem or QueueItem.
class Maze
{
public:
    Maze(const string& fileName); // Load from file.
    void Solve(); // Solve the maze.
    void PrintSolution();
private:
    vector<vector<MazeCell> > maze; // maze square
    int rows;                       // number of rows
    int cols;                       // number of columns
    int mouseRow;                   // row position of mouse
    int mouseCol;                   // col position of mouse
    int cheeseRow;                  // row position of cheese
    int cheeseCol;                  // col position of cheese
    int squaresVisited;
};
```

maze is a vector of vectors. Unlike two dimensional arrays where each row has same number of columns, in a vector of vectors, each row vector is independent of the others and hence they all can have different column sizes. But, in our case they will all have the same size. Note that there is space between the two > symbols in maze declaration. Without the space, your code will not compile as the compiler will interpret >> as insertion operator.
The design given here is just a suggestion. Feel free to add other functions as necessary. For example, you can think of having some private functions such as bool North(int row, int col) etc.

How do we initialize the maze?

```
Maze::Maze(const string& fileName):
           maze(0),   // init maze
           rows(0),
           cols(0),
           mouseRow(0),
           mouseCol(0),
           cheeseRow(0),
           cheeseCol(0),
           squaresVisited(0)
{
    // declare variable to open file etc.
    // read m and n and init rows and cols.
    // read and init mouse and cheese positions.
    // Now we know the size of maze. let us init.
    // Reserve space for rows (= n) many rows.
    maze.reserve(n);
    // reserve space for cols (= n) many columns in each row.
    for (int rowNum = 0; rowNum < rows; rowNum++)
    {
        maze[rowNum].reserve(cols);
    }
    // Now we can use maze[0][0] ... maze[n-1][m-1].
    // Read and initialize all the cells.
}
```

Write a main program to test your maze ADT. I will provide you with 5 different maze files for inputs. Since your program involves testing maze with stack as well as queue, the file mazetest.cpp can include maze-stack.h or maze-queue.h. So, comment out one and use the other. This way you can submit only one copy of mazetest.cpp file.

Documentation
You will need to write Design and Test Documentation. Design documentation for the maze ADT, test documentation for your 5 trials.

Code
You must turn in your Source Code, which should include both the header file(s) and implementation file(s).

General Information
ALL output must begin with your name, course, date, and assignment number. You may discuss the problem with other students and other students may help you to debug your code, but the idea for the solution and the final preparation of the assignment MUST BE DONE ALONE. Any code that is based on code from a textbook should be attributed to the original source.

Remember--if you are having problems, ask questions during class or come see me during my office hours!!

Grading of Programs:

Program assignments will be graded heavily for correct results, but emphasis will also be placed upon accurate and neat documentation as well as effective and proper use of the programming language.

Your final project should be typed and in the following format:

1) A cover page
2) A print out of the short key for the assignment
3) **Source Code**
   a) maze-stk.h
   b) maze-stk.cpp
   c) maze-q.h
   d) maze-q.cpp
   e) mazetest.cpp
   Remember all code needs to be documented and include:
   - description of your ADT
   - a short description of each procedure/function (preconditions/postconditions)
   - run time performance analysis
   - all methods must have descriptions
   - all methods should include comments as to what the method is doing
4) **Testing Outputs**
   You will have 10 outputs; 5 each with each data structure (i.e. 5 with stack, 5 with queue).
   maze1.txt maze2.txt maze3.txt maze4.txt maze5.txt