Assignment #2 Sample Solutions

Question 1: A C++ Class for Slide Puzzles

Write a C++ class SlidePuzzle that (crudely) represents a slide puzzle. If you do not know what a slide puzzle is, then, for example, see here:

http://www.proprofs.com/games/puzzle/sliding/

First of all, you need a constructor that creates a puzzle of default size 3 by 3, but can create puzzles of any size, with the restriction that they have to have at least one row and at least one column:

SlidePuzzle(int rows, int columns)

The puzzle is initialized in such a way that the tiles are numbered in English reading direction, i.e., the tile in the upper left corner has the number 1, the one to its right the number 2, and so on. The last position (in the lower right corner) is empty.

Of course you also need a destructor:

~SlidePuzzle()

You then need to define an enumeration type move that can assume the four values UP, DOWN, LEFT, and RIGHT. For example, a RIGHT move indicates that you are sliding the tile to the left of the empty position to the right so that it fills the empty position (and leaves a new one).

Then you need a function that tells you whether for the current puzzle a given move (UP, DOWN, LEFT, or RIGHT) is legal:

bool isLegalMove(move m)

The actual execution of a move is done through a function makeMove:

bool makeMove(move m)
It returns true if the move is legal and was thus executed, and returns false if the move is illegal and was thus not executed.

Then we need a function getTile that returns the number of the tile at a given row and column. If the position is empty, it returns zero:

```cpp
int getTile(int row, int column)
```

There also has to be a function getEmptyPosition whose arguments are references to integers holding the number of a row and the number of a column. The function fills the referenced variables with the current coordinates of the empty position:

```cpp
void getEmptyPosition(int &row, int &column)
```

Finally, there will be a reset function that moves all tiles to their starting positions as described above.

```cpp
void reset()
```

You only need to provide the class declaration and the function definitions. No other functions such as a main function are required. However, you should write one for yourself to test the SlidePuzzle class.

Please put the code in a hw2 subdirectory in your cs410 folder. If your variable names etc. are intuitive, no further memos or readmes are necessary.
#include <cassert>

enum move {UP, DOWN, LEFT, RIGHT};

class SlidePuzzle
{
public:
    SlidePuzzle(int rows = 3, int columns = 3);
    ~SlidePuzzle() { freeMemory(); }
    bool isLegalMove(move m);
    bool makeMove(move m);
    int getTile(int row, int column);
    void getEmptyPosition(int &row, int &column);
    void reset();

private:
    int **puzzle;
    int height, width;
    void allocateMemory();
    void freeMemory();
};

SlidePuzzle::SlidePuzzle(int rows, int columns) : height(rows), width(columns)
{
    allocateMemory();
    reset();
}

void SlidePuzzle::allocateMemory()
{
    assert(height > 0 && width > 0);
    puzzle = new int*[height];       // create array of pointers to integers
    assert(puzzle != 0);
    for (int i = 0; i < height; i++)
    {
        puzzle[i] = new int[width];   // for each pointer, create array of integers
        assert(puzzle[i] != 0);
    }
}

void SlidePuzzle::freeMemory()
{
    for (int i = 0; i < height; i++)
        delete[] puzzle[i];          // delete arrays of integers
    delete[] puzzle;                // delete array of pointers to integers
}
void SlidePuzzle::reset()
{
    int counter = 1;
    for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++)
            puzzle[i][j] = counter++;
    puzzle[height - 1][width - 1] = 0;
}

bool SlidePuzzle::isLegalMove(move m)
{
    int eRow, eColumn;
    getEmptyPosition(eRow, eColumn);
    switch (m)
    {
        case UP: return (eRow < height - 1);
        case DOWN: return (eRow > 0);
        case LEFT: return (eColumn < width - 1);
        case RIGHT: return (eColumn > 0);
    }
}

bool SlidePuzzle::makeMove(move m)
{
    int eRow, eColumn;
    getEmptyPosition(eRow, eColumn);
    switch (m)
    {
        case UP: if (eRow < height - 1)
        { 
            puzzle[eRow][eColumn] = puzzle[eRow + 1][eColumn];
            puzzle[eRow + 1][eColumn] = 0;
            return true;
        }
        return false;
        
        case DOWN: if (eRow > 0)
        { 
            puzzle[eRow][eColumn] = puzzle[eRow - 1][eColumn];
            puzzle[eRow - 1][eColumn] = 0;
            return true;
        }
        return false;
        
        case LEFT: if (eColumn < width - 1)
        { 
            puzzle[eRow][eColumn] = puzzle[eRow][eColumn + 1];
            puzzle[eRow][eColumn + 1] = 0;
            return true;
        }
        return false;
    }
case RIGHT: if (eColumn > 0)
{
    puzzle[eRow][eColumn] = puzzle[eRow][eColumn - 1];
    puzzle[eRow][eColumn - 1] = 0;
    return true;
}
return false;
}

int SlidePuzzle::getTile(int row, int column)
{
    assert(row >= 0 && row < height && column >= 0 && column < width);
    return puzzle[row][column];
}

void SlidePuzzle::getEmptyPosition(int &row, int &column)
{
    for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++)
            if (puzzle[i][j] == 0)
            {
                row = i;
                column = j;
                return;
            }
}
Question 2: Technical Debt and Test-Driven Development

Our guest instructor Nam Chu Hoai provided some links to texts relating to the subjects of his presentations:

http://firstround.com/review/forget-technical-debt-heres-how-to-build-technical-wealth
http://agiledata.org/essays/tdd.html
http://martinfowler.com/articles/is-tdd-dead/
http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

Please take a look at these texts and then briefly answer the following questions:

(a) How can you avoid accruing technical debt?

- Focus on core components,
- refactor frequently,
- solid product management

(b) What are the trade-offs between testing at the unit test layer vs. the integration/acceptance test layer?

Unit tests are faster to execute, easier to write, less dependent on external components and identify issues concretely.

Integration/Acceptance tests are more appropriate for asserting correctness for a larger part of the system, but can be flakier and more expensive to run/write.