1 Introduction
In this programming assignment you will implement the game AI for a sliding tile game. You will use an A* search algorithm to suggest a sequence of moves to solve the puzzle should the user desire.

This programming assignment should be performed INDIVIDUALLY. You may re-use portions of code from previous examples and labs provided in this class, but any other copying of code is prohibited. We will use software tools that check all students’ code and known Internet sources to find hidden similarities.

2 What you will learn
After completing this programming assignment you will:
1. Be able to create your own classes and interfaces between them
2. Utilize a heap data structure to store potential moves as you perform your solution search
3. Use operator overloading on user defined classes
4. Use friend functions to access private data members
5. Understand search algorithms, specifically A*
6. Perform memory management and understand memory ownership

3 Background Information and Notes
Gameplay: This assignment will allow a user to select the size of the game board, though we will require it to be square. Valid sizes are 4 (2x2), 9 (3x3), 16 (4x4), etc. Further, we will define the SOLVED gameboard to be as shown below with the blank tile at the upper-left corner. Internally, represent that tile with the value: 0.

```
1 2
3 4 5
6 7 8
```

The main gameplay will create the gameboard and mix it. Since randomly shuffling tiles may lead to an unsolvable gameboard, we will perform a series of random legal moves on the solved gameboard to create a scrambled starting board. After creating the starting puzzle board, your program shall allow the user to take turns selecting a tile to move OR asking for a cheat (which should return a correct and shortest move sequence solution). If the user selects a tile to move (which must be
a tile neighboring the blank spot), update the board and iterate to the next turn (i.e. display the board and prompt for a move). If the user asks for a cheat (entering ‘-1’ at the prompt) perform the A* search and output the solution moves (and number of expansions performed during the search [discussed later in this handout]).

A* Search Algorithm: The A* search algorithm starts with a given gameboard and generates a search tree of possible successive moves until it reaches the goal state. In order to minimize the search time (i.e. reduce the number of moves explored), A* always explores the move with the smallest sum, $f=g+h$ [$g$=moves from the start state (depth in the search tree); $h$=heuristic approximation of moves to the goal state]. Provided your heuristic always underestimates moves to the goal (i.e. is optimistic) and is consistent (obeying the triangle inequality) A* will produce the minimum pathway from start to goal state. Review the lecture slides discussed in class for more about this algorithm and its implementation.

Heuristics: Two of the simplest heuristics to use in the problem are: a.) # of tiles out of place and b.) Sum of Manhattan distance of each actual tile (i.e. exclude the blank tile) to its correct location. A third option is no heuristic at all (i.e. $h=0$ which degenerates to a BFS). In this program the user can choose which heuristic to apply at the command line.

The search starts by placing the start state in a priority queue, PQ (implemented as a heap). It then extracts the lowest f-valued move from PQ, checks if it is the solution, and if so, stops and retraces its steps to find the solution. If it is not a solution, new successor states are produced from the one just extracted. These states (or possible moves) are then scored and added to PQ. These moves in PQ that have not been extracted are called the “open list”. Each state (possible move) should have a link (pointer, etc.) to its parent state that generated it so that when a solution state is found, the sequence of moves that led from the start to the goal can be reconstructed.
Command line arguments: Your program should allow the user to pass some command line arguments that specify key initial values that include:

- Dimension of one side of the board [Total size = dim*dim].
- Number of initial scrambling moves [# of moves to try when scrambling the board before the game starts]
- Seed [Value to use to seed the random number generator that will pick which tiles to move for the initial scrambling].
- Heuristic [0 = BFS (h=0), 1 = Tiles-Out-of-Place, 2=ManhattanDistance]

```
puzzle 3 5 400 2
```

Enter tile number to move or -1 for a cheat: -1
Try this sequence:
5 2 1
(Expansions = 6)

```
1 2 5
3 4
6 7 8
```

Enter tile number to move or -1 for a cheat: 5
1 2
3 4 5
6 7 8

Enter tile number to move or -1 for a cheat:
**Class definitions:** To design this program we want to create classes that model the problem appropriately. First, we would want to model the game board (class Board). This object would store the tile configuration and provide services such as moving a tile, checking if the board is solved, offering potential moves, etc. We would likely want to be able to print out a Board to an ostream and provide comparison operators such as < (this may seem strange but will be explained later).

Next we would want an object representing a search state / possible move (class PuzzleMove). You might say, “Isn’t that just a Board?” In fact, we need some additional meta-data like g and h scores, links to the previous state that led us to this one, etc. These objects encapsulate which tile was moved, the resulting Board after the move, the appropriate data for the A* search such as g and h values as well as a previous state pointer so we can retrace our steps when we find a solution. This class also defines a comparison operator for how to compare two PuzzleMoves by defining relational operators like <. **Note:** For the heap/priority queue we will want to compare states based on their f-score. For the closed list which tracks unique states we would want to compare based on the board/tile locations. We have provided two special helper structs in the PuzzleMove class that will be used for these tasks. You do NOT need to modify them. They will just utilize the appropriate < operators for your Board (tile location comparison) and PuzzleMove class (f score comparison).

Another object may be a simple abstraction and implementation of a heuristic computation object. This class would take in a tile arrangement and produce a heuristic score. In this program it should compute the Manhattan distance, but could easily be modified to compute some other heuristic, if desired.

Finally a solver class would be the “glue” that implements the actual A* algorithm, tracks some stats, and stores the solution (class PuzzleSolver). This class would contain a copy of the starting board, a PuzzleHeuristic object, and a list of moves representing the solution. It would provide a function such as run() that would find the sequence of moves to solve the puzzle, and other accessor functions to get the solution, statistics like how many expansions were performed, etc.
**Board class**: Implement the specified member functions in board.h. We have provided one Board constructor to allocate the board and scramble it using the appropriate parameters from the command line. Please do not modify this code as it will allow all of us to reproduce starting game boards from program run to program run (making it easier for you and us to test and debug). You are welcome to look over the code which:

- Seeds the random number generator with the provided seed argument
- Makes exactly numInitMoves (realize one move might cancel the previous, so you may need to put in a number greater than first anticipated to scramble the board well)
  - It would be a good exercise for you to understand the provided code to scramble the board in the constructor. The code attempts to:
    - Find the location of the blank tile
    - Generate a random number between 0-3 [i.e. rand() % 4]
    - Let 0 mean attempt to move North, 1 mean attempt to move West, 2 mean attempt to move South, and 3 mean attempt to move East.
    - If the neighbor of the blank tile in that direction exists, then swap it and the blank tile, otherwise proceed to the next attempt
Be sure you implement a copy constructor that performs a deep copy of the Board data and a corresponding destructor. Implement the potentialMoves() which returns a std::map (map<int, Board*>). This function should determine the potential tiles to move and enter that tile number as a key in the map with the associated value being a new Board object representing the state if that move was actually performed (this new Board object should be dynamically allocated and perform a deep copy of the previous board with the updated tileMove).

Also, implement an ostream operator<< that will print out the board in a nice square format with a 3 character field width for each tile value (use the setw() ostream manipulator).

Also implement a less-than operator. This operator is required if we want to have a set or map using Boards as a key value. If you think for a second, it is unclear how one Board would be less-than another. Let us consider the boards’ tile array like a string of digits and compare them according to sorted (alphabetic) order (i.e. 134025678 > 034125678 and 123456708 < 123456780).

**PuzzleMove struct**: Write a puzzle move class that has the following data members:

```cpp
int tileMove; // tile moved to reach the Board b
Board *b; // Pointer to a board representing
// the updated state
int g; // distance from the start board
int h; // heuristic distance to the goal
PuzzleMove *prev; // Pointer to parent PuzzleMove
```

You may leave data members public in this class only. This class represents a single potential move storing the tile that would be moved and the resulting Board. There will be two different constructors. The first constructor is for use when we first start the search and have the original game board. This PuzzleMove will have no parent. The second constructor is for potential moves in the search where PuzzleMoves have a ‘parent’ that led us to generate this next move. You should be able to set the g score and the previous pointer using this parent state. The h score will be produced by the PuzzleHeuristic object which is used to score the Board of that move. Implement the less-than operator for this class based on the f score (g + h). The less-than operator code should check if one PuzzleMove’s f value is less than another or if their values are the same, then break the tie based on the h score.

**PuzzleHeuristic class**: Has no data members, just an abstract member function: compute(...). Write a derived ManhattanHeuristic, OutOfPlaceHeuristic, and BFSHeuristic class. (Putting them all in the same file, puzzle_heur.h/.cpp is fine.)

**PuzzleSolver class**: Stores a copy of the original Board. It does not run the search right away. The search should be executed when run() is called. Run takes in a PuzzleHeuristic object pointer that can be used in this function to score
PuzzleMoves. In run, you can declare local variables for the open-list (Instructor provided Heap class) (typedef’d as PuzzleMoveHeap for you in the class declaration) and a closed-list (C++ set) of PuzzleMoves (also typedef’d for you as PuzzleMoveSet) to be used to track unique board configurations so that we only enter a new PuzzleMove into the open-list if it represents a board configuration that has not been entered before. Essentially, the PuzzleMoveSet is what we call the “closed set”. Run will perform the A* search described with the pseudocode below:

```
Add the start state (start move) to the open_list and closed_list
while(open_list is not empty and no goal state has been found)
    minstate = remove min. f-value state from open_list
    if minstate is goal state then trace solution path backwards & stop
    Generate next potential boards
    For each potential boards, b
        Make a successor PuzzleMove, s, from the board b
        if the move, s, is not in the closed list
            Compute f value for s
            Add s to the open_list
            Add this move to the closed_list as well
            Increment number of expansions
        Else
            Cleanup
    Cleanup and return appropriate values
```

The graphic on the following page attempts to show the objects, memory allocation and process of one iteration of the A* algorithm. It may be confusing but reference it as you start to write your A* algorithm…it may be useful.
Figure 5: When run() grabs a PuzzleMove from the open-list it will call potentialMoves() of the corresponding board and get a map <tile,Board*> in return. run() then creates PuzzleMove objects from these pairs and places them in the open-list and closed list if they are not already in the closed list. The closed list helps us know what PuzzleMoves we’ve generated so we don’t generate duplicates.

Run() should track how many expansions were performed (i.e. potential moves added to the priority queue during the search).

puzzle.cpp (main program): Your main program should allocate a start board and initialize it using the command line parameters to compute the starting configuration. You can enter your main gameplay loop of taking turns querying the user for a move or a cheat. When a cheat is requested then and only then should you instantiate a PuzzleSolver, run the puzzle solver, and print out the necessary results and continue the game play loop. Later the user should be able to request another cheat and have another PuzzleSolver instance run using the game board’s current state. Be sure to use the Board’s ostream operator to make printing out the board move convenient.

Memory Management: A big educational goal of this project is to exercise your skill at managing memory appropriately. To that end we will not use (nor are you allowed to use) vectors and deques in our Board class, but will instead use raw C/C++ dynamically allocated arrays.

An important concept is memory ownership. It is fine for several "client" objects to each have a pointer to a dynamically allocated object (effectively sharing it). But one client object must be the owner of that dynamic memory and thus be responsible for deallocating it when it is no longer needed (i.e. deleting it in its destructor). Other client objects may have a pointer but when they are...
deallocated, they do NOT in turn deallocate what the pointer is point to. In that case, we'd say the object does NOT *own* what the pointer is point to.

Take care that if you have multiple objects sharing a pointer to a single thing that you decide who the one “owner” will be so that you don't try to deallocate the same memory twice. In addition, take care that an owning object does not mix pointers to dynamically-allocated objects and objects that live on the stack. Since an owning object needs to deallocate heap-based objects and your code is unable to distinguish pointers to heap-based objects vs. stack-based objects, you generally must ensure that you only have pointers to heap-based objects.

Also try to use good encapsulation where the owning object will delete the memory it owns in its destructor and not make some external code do it. Finally, be sure to make deep-copies where needed.

4 Requirements

Implement the specified class definitions “as-is”. You should not alter the following files:

- board.h, puzzle_solver.h, puzzle_heur.h, heap.h

Your program shall meet the following requirements for features and approach:

Milestone A

1. Implement the Board class per requirements listed earlier in this document (Be sure the board prints out in appropriately sized, neatly formatted columns based on the size of the board [at most we will give a size 5x5 board as a test case] such that the columns are perfectly aligned for every row).
2. Implement the game play in puzzle.cpp by iteratively printing out the board and querying the user for a tile to move. The user input of 0 should cause the program to quit, -1 should cause the A* algorithm to be run to display a cheat (though for this milestone you don’t have to implement this), and valid tile values (those neighboring the blank spot) should cause your program to attempt to move that tile. Invalid tile values can be ignored and cause you to start the next turn. Exit automatically when the game is solved by the user. [For now don't implement the cheat].
3. Test your basic manual tile puzzle game.

Milestone B

4. Implement the PuzzleMove class (add or modify member functions and data as necessary). You will need to create your own puzzle_move.cpp
5. Implement ManhattanHeuristic, OutOfPlaceHeuristic, and BFS heuristic derived classes from the PuzzleHeuristic base class. You will need to create your own puzzle_heuristic.cpp file.

Milestone C

6. Implement the PuzzleSolver class (make your own puzzle_solver.cpp) and the run() function to perform the A* search algorithm
a. Using the heap class we’ve provided in “heap.h” (no corresponding heap.cpp is needed). It has been typedef’d as PuzzleMoveHeap in puzzle_solver.h for your open list
b. Use the std::set<PuzzleMove*> class typedef’d as PuzzleMoveSet in puzzle_solver.h for you closed list
c. Store the solution of tiles to move a deque<int> class
7. Update your puzzle.cpp main() to now implement the cheat (when they type -1).

Milestone D
8. In the main() gameplay, format the results of the cheat solution as shown in Figure 3 including the number of expansions that were performed
9. Be sure you deallocate (free/delete) any memory you dynamically allocate (run $ valgrind --tool=memcheck --leak-check=yes ./puzzle ... to check)
10. Go back and use the other heuristics to make sure it works? Which one performs better in general?

5 Procedure
Perform the following.
1. Create a ‘puzzle’ directory:
   $ mkdir puzzle
   $ cd puzzle
2. Copy the skeleton files to your local directory:
   $ wget http://ee.usc.edu/~redekopp/ee355/code/puzzle.tar
   $ tar xvf puzzle.tar
   This will download the appropriate header files.
3. Start by writing the basic puzzle game without the cheat option (Milestone A)...This should only require the Board class. Here are some initial boards that should be produced when you start the program on the VM (if you don’t use the VM you may get different board configurations due to alternate srand() implementations):

<table>
<thead>
<tr>
<th>./puzzle 2 15 76</th>
<th>./puzzle 3 20 1711 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 3</td>
<td>2 4 8</td>
</tr>
<tr>
<td></td>
<td>6 5 7</td>
</tr>
<tr>
<td>./puzzle 4 20 1711 2</td>
<td>./puzzle 5 20 1711 2</td>
</tr>
<tr>
<td>4 1 2 3</td>
<td>5 1 2 3 4</td>
</tr>
<tr>
<td>5 9 6 7</td>
<td>6 11 7 8 9</td>
</tr>
<tr>
<td>8 14 11</td>
<td>10 16 12 13 14</td>
</tr>
<tr>
<td>12 10 13 15</td>
<td>15 22 18 19</td>
</tr>
<tr>
<td></td>
<td>20 17 21 23 24</td>
</tr>
</tbody>
</table>

Final Product
4. Complete the PuzzleMove, PuzzleHeuristic, and PuzzleSolver classes integrate the cheat option into your main gameplay.
5. Test your games and feel free to post your command line arguments along with the start board and cheat solution on the Piazza discussion boards so that you can all cross-validate your answers.

6. Comment your code as you write your program documenting what abstract operation a certain for, while, or if statement is performing as well as other decisions you make along the way that feel particular to your approach. We have provided a few test cases below.

6 Lab Report

Analyze the effectiveness of different heuristics. For each problem, you should use a board of size 4 (4x4) and 1711 as your seed value. Using each heuristic in turn, run your program with increasing number of initial moves. Start with 20 and work up in increments of 20 until you reach 100 (i.e. 20, 40, 60, 80, 100) and then continue in increments of 50 (150, 200, 250, 300, etc.). Keep re-running your program with increasing values of initial moves UNTIL the time it takes to get the cheat code is 30 seconds or more (you can Ctrl-C to quit your program if you don’t get the cheat in 30 seconds and then move on to the next heuristic). For successful cheats (i.e. found in under 30 seconds), record the number of expansions required to find the solution.

- In under 30 seconds, which initial move values and how many expansions did each solution require using BFS?
- In under 30 seconds, which initial move values and how many expansions did each solution require using Tiles-Out-of-Place?
- In under 30 seconds, which initial move values and how many expansions did each solution require using Manhattan Distance?

Submit your answers in a text file named `perf.txt` in a nice table format showing the number of expansions as the value of each cell if the run took under 30 seconds. Once it took more than 30 seconds leave the rest of the entries in the column blank.

Sample Table Organization in perf.txt

<table>
<thead>
<tr>
<th>initMoves</th>
<th>BFS</th>
<th>TilesOutOfPlace</th>
<th>Manhattan</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7 Submission
Submit your code on our website. Submit each of your files separately (puzzle.cpp, puzzle_heur.h/.cpp, puzzle_solver.h/.cpp, board.h/.cpp, puzzle_move.h/.cpp heap.h, Makefile, perf.txt)

8 Test Cases
Feel free to perform different runs and post them on Piazza for others to verify. Paste your command line and then solution and number of expansions.

Tiles Out Of Place Heuristic:
./puzzle 3 30 1537 1
3 1 2
6 4 5
7 8

Enter tile number to move or -1 for a cheat: -1
Try this sequence:
8 7 6 3
(Expansions = 7)

./puzzle 4 50 1537 1
1 8 2 3
9 4 7 10
5 6 11
12 13 14 15

Enter tile number to move or -1 for a cheat: -1
Try this sequence:
9 4 7 10 11 6 10 7 8 1 4 8 5 10 6 11 7 6 10 9 8 4
(Expansions = 32451)

Manhattan Heuristic:
./puzzle 3 30 1537 2
3 1 2
6 4 5
7 8

Enter tile number to move or -1 for a cheat: -1
Try this sequence:
8 7 6 3
(Expansions = 7)
Enter tile number to move or -1 for a cheat: -1
Try this sequence:
9 4 7 10 11 6 10 7 8 1 4 8 5 10 6 11 7 6 10 9 8 4
(Expansions = 1585)

BFS Heuristic

Enter tile number to move or -1 for a cheat: -1
Try this sequence:
8 7 6 3
(Expansions = 47)
## Rubric

### Student Name: ________________________________

<table>
<thead>
<tr>
<th>Item</th>
<th>Outcome</th>
<th>Score</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiles successfully</td>
<td>Yes / No</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><strong>Code Design</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Classes broken into .h and .cpp files</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Board class adheres to definition</td>
<td>Yes / No</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>• PuzzleMove class adheres to definition</td>
<td>Yes / No</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>• PuzzleSolver class adheres to definition</td>
<td>Yes / No</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>• Heuristics adhere to definition &amp; are correctly computed</td>
<td>Yes / No</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td><strong>Gameplay</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Correct output format of boards of various sizes</td>
<td>Yes / No</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>• Correct output format of cheat solution</td>
<td>Yes / No</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>• Gameplay takes place in turns until board is solved</td>
<td>Yes / No</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>• User can quit with option 0</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><em><em>A</em> Algorithm implementation</em>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Heap correctly used to pop moves in correct order</td>
<td>Yes / No</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>• Set used to track board moves</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>• Dynamic memory is allocated and freed correctly (no leaks)</td>
<td>Yes / No</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>• Expansions update code is correct</td>
<td>Yes / No</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>Sample Gameplay</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Correct: Test 1-5 (and Notes)</td>
<td>____ / 15</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td><strong>Correct performance discussion in report</strong></td>
<td>Yes / No</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

SubTotal: 50

Late Deductions

Total: 50