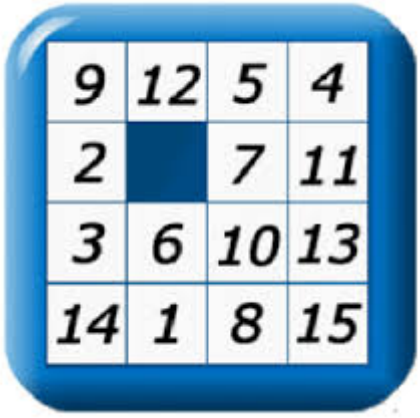
**Contents**



# Assignment 1: Solvability of the NxN sliding tile puzzle




The sliding tile puzzle is a game that requires you to move tiles on a board. The board is NxN, and there are N-1 tiles numbered from 1..N-1 that occupy the board. There is hence 1 location on the board that is empty (referred to as a blank).

There is some (arbitrary) start configuration of the numbered tiles on the board. Starting with this configuration, the aim is to move tiles until some chosen goal configuration is reached, and to do this in the least possible number of moves. You may only move a tile into the blank if the tile neighbours the blank. Moves can be only be in the horizontal and vertical directions (not diagonal).

The sliding-tile puzzle also has other names, such as the 8 Puzzle (for the special case of a 3x3 board) or 15 Puzzle (a 4x4 board) and so on. Sometimes the name N Puzzle is used (indicating an NxN board).

You can play the game online at:

sliding-tile puzzle

In this assignment you are asked to write a C program that determines whether a given puzzle is solvable. (Note that you do not have to actually solve the puzzle.)

There are some conditions that you should strictly adhere to:

1. the program reads text from *stdin* with a format described below (we will be auto-testing your program with our input so you must conform to this format)

2. your program should be able to handle any sized board, starting with 2x2
    - note that the size of the board is determined by the number of tiles on the input
3. if the input is correct and the goal configuration is:
    - reachable from the start configuration, your program should generate the output text *solvable* (to *stdout*)
    - not reachable from the start configuration, your program should generate the output text *unsolvable* (to *stdout*)
4. if the input is not correct, your program should generate an error message (to *stdout*)
5. if a system call fails in your program, the program should generate an error message (to *stderr*)
6. design and programming restrictions:
    - you are not allowed to use any arrays
    - you are not allowed to use linked lists/trees/graphs
    - you should use an ADT to represent the board and operations on the board

## Input format

Two lines of text on *stdin* specifies the start and goal configurations, read from left to right, top to bottom. Each line consists of a sequence of integers, separated by any number (>0) of blanks and/or tabs, that represent the tile numbers, and a single letter *b* to represent the blank space on the board. These integers should of course be in the range 1..N-1 where the board is of size NxN. The first line specifies the start board, the second line the goal board. For example:

```
9 12 5 4 2 b 7 11 3 6 10 13 14 1 8 15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 b 15
```

represents a sliding-tile puzzle on a *4x4* board with the tiles initially placed on the board as shown in the image at the top of page. The goal configuration has the tiles ordered row by row.

## In the case of incorrect input

Checking the correctness of each configuration is vital. For example, an input line may not represent an NxN board, or the blank may be missing, or one or more of the tile numbers 1..N may be missing, or the 2 boards may not be the same size or the input contains something other than a number or *b*. There may be more possibilities.

If the configuration is erroneous, your program must generate an appropriate error message (to *stdout*). Note that it is possible to have more than one error, and in that case you only need to generate a single error message. For example, consider the configuration *1 2 b 1*. There are 2 errors in this configuration: the tile 3 is missing, and the tile 1 is duplicated. It does not matter which error is detected by your program, just as long as the error is correct. When an error is detected and reported, your program should exit gracefully, with status *EXIT_FAILURE*. The text you use in error messages should be informative, but please keep it brief.

## In the case of correct input

If the input is correct, the program should write the following 3 lines to *stdout*:

- the text *start:* followed by the start configuration
- the text *goal:* followed by the goal configuration
- the text *solvable* or *unsolvable* as appropriate

The output for the 4x4 game above is for example:

```
start: 9 12 5 4 2 b 7 11 3 6 10 13 14 1 8 15
goal: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 b 15
solvable
```

The output for a game on a 2x2 board that happens to be unsolvable is:

```
start: 2 1 3 b
goal: 1 2 3 b
unsolvable
```

If the input is correct the program should exit with *EXIT_SUCCESS*.

## Design

You should make an ADT to implement the puzzle. The *client*, which is the main program, calls functions in the ADT to read the input, check for correctness and determine solvability. The interface between the client and the ADT is a header file.

## Marking

Marks will be deducted if you fail any of our tests for incorrect input, or incorrectly determine the solvability of the puzzle. Marks will also be deducted for poor design (e.g. not using an ADT), poor programming practice or violating any of the rules above.

The assignment is worth 10 marks.

## Submission

You should submit exactly 4 files:

1. a *Makefile* that generates the executable *puzzle* (you should use the *dcc* compiler)
2. the C source code of the ADT (call it *boardADT.c*)
3. the header file of the ADT (*boardADT.h*)
4. the main program (*puzzle.c*)

Before submission make sure your *Makefile* is working correctly: the command *make* should generate the target executable *puzzle* using the 3 source files *boardADT.c*, *boardADT.h* and *puzzle.c*.

To submit, simply click on the *Make Submission* button above. You should submit exactly 4 files: *Makefile* and the 3 source files.

## Testing

The following describes a simple *test harness* (is it's called) that will help you organise and run your test cases. The basic idea is to separate your test cases into good boards that are solvable, those that are unsolvable, and bad boards. A test case is simply a file containing puzzle input of course. Carry to the following steps:

1. Create a directory, **Tests** say, to house all your test input. Each file in this directory is a test case. Name the files systematically, so for example, the following input

   ```
   1 2 3 1
   1 2 3 b
   ```

could be called **bad01.inp**. Another 'bad' input file would be called **bad02.inp**. Likewise, create files **sol01.inp**, **sol02.inp** etc to represent solvable boards, and **unsol01.inp**, **unsol02.inp** etc for unsolvable boards. All these files are in the **Tests** directory. You could also make the names even more meaningful by coding in the board size into the name: e.g the prefix of all the *2x2* 'bad' boards could be **bad2**, and of *3x3* boards **bad3** etc, and similarly for **sol** and **unsol**.

2. In the parent directory, which should contain your *puzzle* executable, create a file called **Testrun** containing the following shell script

```
#!/bin/sh
PROG=./puzzle

case $1 in
1) T=Tests/bad* ;;
2) T=Tests/sol* ;;
3) T=Tests/unsol* ;;
esac

if [ A$T != A ]
then
   for i in $T
   do
   echo =================  $i  ==================
   $PROG < $i
   done
else
   echo Usage $0 "[1|2|3]"
fi
```

   which assumes you have used my naming convention in part 1.
3. To run all the *bad-board* tests in one go, simply run the corresponding script:

```
sh Testrun 1
```

   and similarly *sh Testrun 2* and *sh Testrun 3*.

The advantage of the separating into different kinds is that is makes it easy to see that the program is behaving correctly for all the tests with the same kind of input (bad, solvable and unsolvable). It is also easy to add new test input to the *Tests* directory.

You can make this script much fancier of course. You could for example add a $4^{th}$ case to test bigger boards if that is where you want to focus on at a particular stage in the development.

Assignment1 (last edited 2019-06-16 22:14:57 by AlbertNymeyer)