Number Shuffle Project

Tong Yi

Implement number shuffle game in `https://www.artbylogic.com/puzzles/numSlider/numberShuffle.htm`.

Warning:

1. This is copyrighted materials; you are not allowed to upload to the Internet.
2. Our project is more complicate than similar projects in the Internet and uses a different approach.
   (a) Ask help only from teaching staff of this course.
   (b) Use solutions from ChatGPT or online tutoring websites like, but not limited to, chegg.com violates academic integrity and is not allowed.

1 Files of the Project

We use Object-oriented Programming approach.

1. Create directory `numShuffle` to hold codes of number shuffle project **if** you have not done so. Said differently, you only need to run the following command once.

   ```
   mkdir numberShuffle
   ```

2. Move to the above directory.

   ```
   cd numberShuffle
   ```

3. Create `Board.hpp` with the following contents. **Warning:** do not write `Board.hpp` as `board.hpp`. C++ is a case-sensitive language.

   `Board.hpp` is the header file of Board class that **declares** data members and operations (aka methods) on those data members.

```
#ifndef Board_H
#define Board_H

class Board {
public:
    Board(); //3 * 3 board
    Board(int m, int n); //m * n board
    Board(int** arr, int m, int n); //m * n board where data is stored in a 2-dimensional
    array
    ~Board(); //destructor
    void randomize();
    void getInfo(); //find out emptyCellRow, emptyCellCol, and numCorrect from panel
    bool valueCorrect(int row, int col) const;
    void display() const;
```

```
14      void slideUp();
15      void slideDown();
16      void slideLeft();
17      void slideRight();
18      void play();
19
20  private:
21      int numRows;
22      int numCols;
23      int** panel;
24      int emptyCellRow;
25      int emptyCellCol;
26      int numCorrect; //cell with correct position
27  };
28  #endif
```

4. Your main task is to implement `Board.cpp`, which **defines** constructors, the destructor, and methods declared in Board.hpp.

   (a) Note that, in `Board.hpp`, data members are declared but not yet initialized. The data members are initialized in constructors.

   (b) Similarly, constructors, the destructor, and methods are declared (have function header) in `Board.hpp` but not defined (no function body).

1.1 Explanation of Board.hpp

A board is represented by a two dimensional array of integers. Key methods are slide left/right/up/down and play.

1.1.1 Include guard

```
1  #ifndef Board_H
2  #define Board_H
3  ...
4  #endif
```

Lines 1, 2, and 4 in the above code is called include guard. With it, even if `#include "Board.hpp"` is used more than once, since `Board_H` is defined already after the first occurrence of `#include "Board.hpp"`, the contents in ... are only declared once.

The details of data members, constructors and methods in Board class of the game are discussed as follows.

1.2 Data members

1. Variable `numRows` is an integer representing the number of rows of the board in number shuffling game.
2. Variable `numCols` is an integer representing the number of columns of the board.
3. The board is represented by variable `panel` of `int**` type.

   (a) An int pointer `int*` saves the initial address of an array of integers, which represents a row. The following is an example of using `int*` type variable.

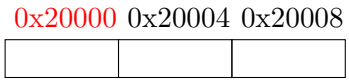
```
1 int* rowPtr = new int[numCols];
```

i. Warning: cannot write
`int* rowPtr = new int[numCols];` as
`int* rowPtr = new int[3];`
otherwise, `rowPtr` points to a memory block holding only 3 integers. However, `numCols` is a **VARIABLE** initialized when a constructor is called.

ii. Expression `new int[numCols]` returns the initial address of a dynamically allocated space that holds `numCols` integers.

A. **Assume** that `numCols` is 3. **Assume** that the initial address of a dynamically allocated memory is 0x20000, where 0x before 20000 means the number is hexadecimal, whose base is 16. That is, 0x20000 equals 2 * 16^4 = 131072 in decimal system. A memory address is represented in hexadecimal number.

B. Each integer takes 4 bytes. As a result, the address of the second integer is 0x20000 + 4 = 0x20004, and the address of the third integer is 0x20000 + 4*2 = 0x20008.



C. Note that the value of each memory slot is not decided yet. That is, expression `new int[numCols]` only allocates memory to hold `numCols` integers. It remains to initialize integers stored in those memories.

D. Statement `int* rowPtr = new int[numCols];` puts that initial address to `rowPtr`. After the above statement, `int*` variable `rowPtr`, whose value is the address of an int, is set to be 0x20000. **Assume** the address of `rowPtr` is 0x36000.

It is equivalent to let `rowPtr` point to the dynamically allocated memory, illustrated as follows.

Each variable has a name, occupies some memory, and has a value.

variable	**memory of the variable**	**variable value**
rowPtr	0x36000	0x20000
the first element in the array	0x20000	not initialized yet
the second element in the array	0x20004	not initialized yet
the third element in the array	0x20008	not initialized yet

How to access the value of a memory?

address of a memory	**value of the memory**
rowPtr	*rowPtr, aka rowPtr[0], the leftmost element.
rowPtr+1	*(rowPtr + 1), aka rowPtr[1], the second element to the left
rowPtr+2	*(rowPtr + 2), aka rowPtr[2], the third element to the left

- Expression `rowPtr + n`, where `n` ≥ 0, is **not** to add `n` literally to `rowPtr`. Instead, it means to add n * size of the type pointed by rowPtr.
  In this example, `rowPtr` is `int*` type, and the type pointed by `rowPtr` is `int`.
  Each `int` type takes 4 bytes. Suppose `rowPtr` is 0x20000. Then `rowPtr + 1` is 0x20004.

- In general, if `rowPtr` is the address of the first element of the array, then `rowPtr + n` is the address of the `n`th element of the array.

(b) Observe that `rowPtr` is a variable of `int*` type, which saves the address of one individual integer or the initial address of an array of integers.

If we use an array of `int*` – distinct from `int` – type, the first array element saves the address of elements in the first row, the second array element saves the address of the elements of the second row, and so on. Then we have a panel of `numRows`* `numCols` integers.

The initial address of an array of `int*` type can be saved in an `int**` variable.

Type `int**` is a pointer to int pointers, which saves the initial address of an array of int pointers `int*`. You may think `int**` as an array of one-dimensional array, which is actually a two-dimensional array.

4. To track changes of panel after each slide operation, declare `emptyCellRow`, `emptyCellCol`, and `numCorrect`. More details will be covered in Task B.

5. Variable `emptyCellRow` is the row index of the empty cell in the panel.

6. Variable `emptyCellCol` is the column index of the empty cell in the panel.

7. Variable `numCorrect` is the number of non-empty entries sitting in the correct cell, that is, for a cell at the ith row and jth column, its value should be $i*$`numCols`$+j+1$, where $0 \leq i <$ `numRows` and $0 \leq j <$ `numCols`.

   For example, in the following 3 * 3 panel, only number 2 (with # mark right to it) sits in the correct cell, not any other non-empty entry. Hence `numCorrect` is 1.

```
  \ col index   0     1     2
row index    +-----+-----+-----+
           0 |  8  |  2# |  7  |
             +-----+-----+-----+
           1 |  1  |  6  |  5  |
             +-----+-----+-----+
           2 |  4  |  3  |     |
             +-----+-----+-----+
```

   In the above example, `emptyCellRow` is 2 and `emptyCellCol` is 2. After sliding down operation, the layout is changed to be the following.

```
  \ col index   0     1     2
row index    +-----+-----+-----+
           0 |  8  |  2# |  7  |
             +-----+-----+-----+
           1 |  1  |  6  |     |
             +-----+-----+-----+
           2 |  4  |  3  |  5  |
             +-----+-----+-----+
```

   Now `emptyCellRow` is 1 and `emptyCellCol` is 2, `numCorrect` is still 1. Slide right and here is the layout.

```
  \ col index   0     1     2
row index    +-----+-----+-----+
```

```
0 |  8  |  2# |  7  |
  +-----+-----+-----+
1 |  1  |     |  6# |
  +-----+-----+-----+
2 |  4  |  3  |  5  |
  +-----+-----+-----+
```

Now `emptyCellRow` is 1 and `emptyCellCol` is 1, `numCorrect` is changed to 2, since number 6 is in correct position, besides number 2.

2 Task A: Define constructors and destructors in Board.cpp

The purpose of constructor is to initialize data members. A class may have multiple constructors. Different constructors have different parameter lists. Each constructor has exactly the same name as class, no return type, not even void.

2.1 The default constructor `Board()`

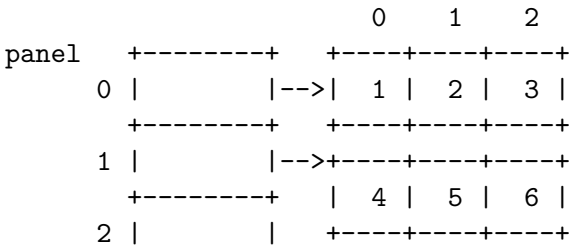
The default constructor does not take any parameter. It does the following:

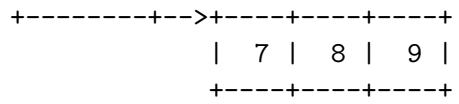
1. Set data members `numRows` and `numCols` to be 3.
2. Dynamically apply for space to hold a `numRows` * `numCols` integer array. Put the initial address in data member `panel`.
3. Set the elements of `panel` to be 1, 2, $\cdots$, `numRows * numCols`. The numbers are placed from the top row to the bottom row, and in each row, from left to right.
4. In Task A, we do not randomize the placement of numbers yet.

The default constructor is called when a user does not know or bother with the details of a class and just want to create an object of the class. It is like to order a hamburger skipping the details of choosing the ingredients in its meat-, vegetable-, and bread- layers (aka data members of a hamburger class). Such a "typical" (or default) hamburger object may contain beef, lattice, and wheat bread (aka values for the corresponding data members), created by the default hamburger maker (aka the default constructor of hamburger class). No input parameter is taken.

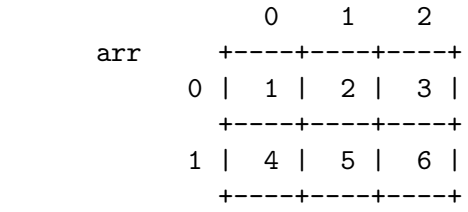
After calling the default constructor, `numRows` and `numCols` are set to be 3 and `panel` is the initial address of a dynamically allocated two-dimensional array with 3 rows, each row has 3 integers. The layout of panel is as follows.

Note that `int*` is a pointer normally has 8 bytes in 64-bit operating system and `int` has 4 bytes. Row indices are shown in vertical direction, starting from 0. Column indice are displayed in horizontal direction, starting from 0. `panel` is an array of `int*` type, and `int*` can be illustrated as a pointer pointing to an array of integers.





In each row, the elements of integers are placed in consecutive memory, however, the elements of each adjacent rows may not be in consecutive spaces. This is a difference between a dynamically allocated two-dimensional array and a static allocated two dimensional array`int arr[][3] = { {1, 2, 3}, {4, 5, 6} };` Here is an illustration of `arr`.



2.2 A nondefault constructor `Board(int m, int n)`

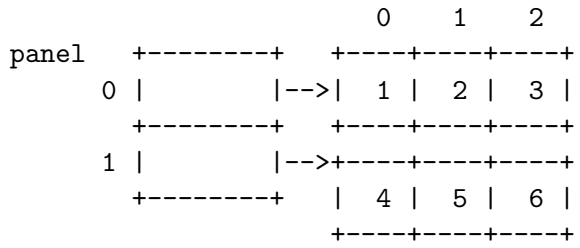
1. If both given parameters `m` and `n` are larger than or equal to 2, use `m` and `n` to set data members `numRows` and `numCols`, respectively, otherwise, set data members `numRows` and `numCols` to be 3.

2. Dynamically apply for space to hold a `numRows` * `numCols` integer array. Put the initial address in data member `panel`.

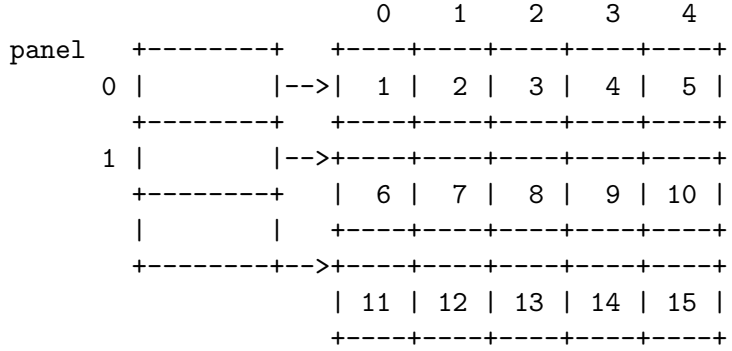
3. Set the elements of `panel` to be 1, 2, $\cdots$, `numRows * numCols-1`. The numbers are placed from the top row to the bottom row, and in each row, from left to right.

4. In Task A, we do not randomize the placement of numbers yet.

   After calling Board(2, 3), `numRows` is 2 and `numCols` is 3 and `panel` is the initial address of a dynamically allocated two-dimensional array with 2 rows, each row has 3 integers. The layout of panel is as follows.



   After calling Board(3, 5), `numRows` is 3 and `numCols` is 5 and `panel` is the initial address of a dynamically allocated two-dimensional array with 3 rows, each row has 5 integers. The layout of panel is as follows.



Related: a default hamburger is one with beef, lettuce and wheat bread. For simplicity, we may assume that meat, vegetable, and bread layers of a hamburger is a string.

Contents of header file of Hamburger class, `Hamburger.hpp`, are as follows.

```
#ifndef Hamburger_H
#define Hamburger_H
#include <string>

class Hamburger{
public:
    Hamburger();
    Hamburger(std::string meat, std::string vegetable, std::string bread);
    std::string getMeat() const;
    std::string getVegetable() const;
    std::string getBread() const;
    int getCalories() const;
    void setMeat(std::string meat); //change meat layer of the current hamburger
    void setVegetable(std::string vegetable);
    void setBread(std::string bread);

private:
    std::string meat;
    std::string vegetable;
    std::string bread;
};
#endif
```

Contents of source code of Hamburger class, `Hamburger.cpp`, are as follows.

```
#include "Hamburger.hpp"

Hamburger::Hamburger() : Hamburger("beef", "lettuce", "wheat bread")
//Hamburger("beef", "lettuce", "wheat bread") means to call
//Hamburger(std::string meat, std::string vegetable, std::string bread)
//with parameters "beef", "lettuce", and "wheat bread",
//it is like to call a Hamburger maker specifying the contents of
//meat-, vegetable-, and bread-layers.
{

}

Hamburger::Hamburger(std::string meat, std::string vegetable, std::string bread) {
    //TODO: initialize data members meat, vegetable and bread by
    //the corresponding given parameters in the constructor, correspondingly.
}

//omit methods of Hamburger class.
```

2.3 A nondefault constructor `Board(int** arr, int m, int n)`

1. If both given parameters `m` and `n` are larger than or equal to 2, use `m` and `n` to set data members `numRows` and `numCols`, respectively, otherwise, set data members `numRows` and `numCols` to be 3.

   Set the elements of `panel` to be 1, 2, $\cdots$, `numRows * numCols-1`. The numbers are placed from the top row to the bottom row, and in each row, from left to right. No randomize is needed at this step.

2. Otherwise, dynamically apply for space to hold a `numRows` * `numCols` integer array. Put the initial address in data member `panel`.

3. Set the elements of `panel` to be the same arrangement as that of given parameter `arr`.

You may notice that there are a lot of common codes among those constructors. A better way is to define `Board(int m, int n)`. Then use constructor delegate to define `Board()` and `Board(int** arr, int m, int n)`.

```
//TODO: fill in ? in the parentheses.
//Hint: what are the values of numRows and numCols for a default Board object?
Board::Board() : Board(?, ?) {
    //Question: after calling Board(?, ?) to create a Board object with
    //? * ? two-dimensional array,
    //is there any additional thing to do in the default constructor?
}

Board::Board(int m, int n) {
    //TODO: Write your codes here
}

//TODO: fill in ?? and ??? in the parentheses.
//Hint: what are the values of numRows and numCols
//      for a Board object with m rows and n columns?
Board::Board(int** arr, int m, int n) : Board(??, ???) {
    //TODO: after panel saves the address of a dynamically allocated
    //m by n two dimensional array, how to set the values of panel
    //to be those of arr?
}
```

2.4 The destructor

The purpose of destructor is to release the dynamically allocated memory allocated to an object. The destructor has the same name as class, with $\sim$ in front of it. No return type, not even void. No parameter.

Normally we do not need to call the destructor explicitly, when an object is no longer needed – for example, out of its definition scope – C++ will call the destructor automatically.

2.5 Finish Task A

- Define constructors and the destructor in `Board.cpp`.
- Test codes locally.
  - Comment private: line in `Board.hpp` as `//private:`. This is for debug purpose.

– Edit main.cpp as follows.

```cpp
#include <iostream>
#include "Board.hpp"
//g++ -std=c++11 Board.cpp main.cpp
//test default constructor using
//./a.out A or ./a.out 'A'
//./a.out B or ./a.out 'B'
//./a.out C or ./a.out 'C'

int main(int argc, const char *argv[]) {
    if (argc != 2) {
       std::cout << "Need 'A'-'C' in parameters" << std::endl;
       return -1;
    }

    //unit-testing for constructors and the destructor
    char type = *argv[1];
    std::string prompt;
    Board *game;
    int numRows = 3;
    int** arr;
    if (type == 'A') {
       prompt = "default constructor,";
       game = new Board;
    }
    else if (type == 'B') {
        prompt = "Board game(3, 5);";
        game = new Board(3, 5);
    }
    else if (type == 'C') {
        prompt = "Board game(arr, 3, 3);";
        const int NUM_COLS = 3;
        int brr[][NUM_COLS] = { {3, 9, 8}, {5, 7, 2}, {1, 6, 4} };
        arr = new int*[numRows];
        for (int row = 0; row < numRows; row++) {
            arr[row] = new int[NUM_COLS];
            for (int col = 0; col < NUM_COLS; col++)
                arr[row][col] = brr[row][col];
        }
        game = new Board(arr, 3, 3);
    }

    std::cout << "After " << prompt
         << " data member numRows is " << game->numRows << std::endl;
    std::cout << "After " << prompt
         << " data member numCols is " << game->numCols << std::endl;
    std::cout << "After " << prompt
         << " data member panel is " << std::endl;
```

```
48
49      for (int row = 0; row < game->numRows; row++) {
50          for (int col = 0; col < game->numCols; col++) {
51              std::cout << game->panel[row][col];
52              if (col < game->numCols-1) //skip the last ,
53                  std::cout << ",";
54          }
55          std::cout << std::endl;
56      }
57
58      game->~Board();
59      std::cout << "After calling destructor, data member panel is " << game->panel <<
        std::endl;
60
61      if (type == 'C') {
62         //release dynamically allocated memory for arr
63         for (int row = 0; row < numRows; row++) {
64              delete[] arr[row];
65              arr[row] = nullptr;
66          }
67          delete[] arr;
68          arr = nullptr;
69      }
70
71      return 0;
72  }
```

Explanation of the code is as follows.

* normally we use
  `int main()`, when such file is compiled and runnable, we use
  `./a.out`
  In our version, the first parameter is number of parameters, the second parameter is an array of char* (aka string in C), which represents the parameters.
  `int main(int argc, char* argv[])`
  Suppose the following contents are saved in `test.cpp`.

```
1  #include <iostream>
2
3  int main(int argc, char* argv[]) {
4      std::cout<<argv[1] << std::endl;
5
6      return 0;
7  }
```

  If run the following command, the phrase "Hello, world" is put in argv[1], the second parameter.

```
g++ test.cpp
./a.out "Hello, world"
```

  Output "Hello, world" without quotes.

* In the above `main.cpp`, we test default constructor when command parameter is 'A', non-default

constructor `Board(int, int)` when command parameter is 'B', and non-default constructor `Board(int**, int, int)` when command parameter is 'C'.

- Run the following command to compile main.cpp and Board.cpp.

  ```
  g++ -std=c++11 main.cpp Board.cpp
  ```

- If there is no compilation errors, run the following command.

  ```
  ./a.out A
  ```

- You should be able see something like the following.

  ```
  After default constructor, data member numRows is 3
  After default constructor, data member numCols is 3
  After default constructor, data member panel is
  1,2,3
  4,5,6
  7,8,9
  After calling destructor, data member panel is 0x0
  ```

  In Linux, the output of the last line is

  ```
  After calling destructor, data member panel is 0
  ```

- If you test non-default construtor Board(int m, int n) using

  ```
  ./a.out B
  ```

  You should see the following output.

  ```
  After Board game(3, 5); data member numRows is 3
  After Board game(3, 5); data member numCols is 5
  After Board game(3, 5); data member panel is
  1,2,3,4,5
  6,7,8,9,10
  11,12,13,14,15
  After calling destructor, data member panel is 0x0
  ```

- If you test non-default construtor Board(int m, int n) using

  ```
  ./a.out C
  ```

  You should see the following output.

```
1 After Board game(arr, 3, 3); data member numRows is 3
2 After Board game(arr, 3, 3); data member numCols is 3
3 After Board game(arr, 3, 3); data member panel is
4 3,9,8
5 5,7,2
6 1,6,4
7 After calling destructor, data member panel is 0x0
```

- Or you can test the code in `https://www.onlinegdb.com/online_c++_compiler`.

  Upload main.cpp, Board.hpp (comment private: line) and Board.cpp to onlinegdb. In the textbox right to **Command line arguments:**, enter A or B or C or D.

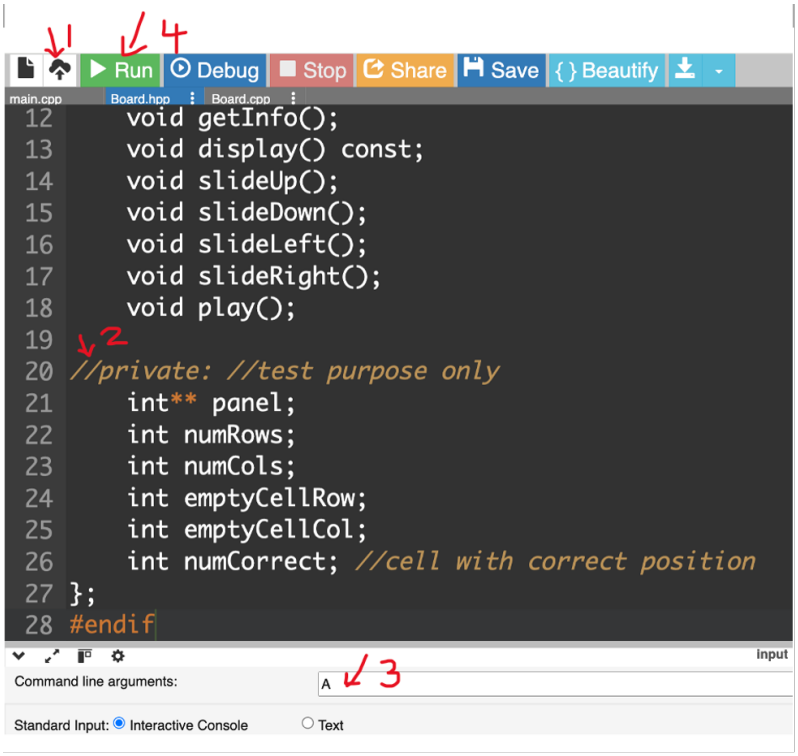


Figure 1: Test Task A in onlinegdb.com

- If the code runs well in local computer, upload `Board.cpp` to gradescope.

3 Task B: define randomize, getInfo, display, and valueCorrect methods

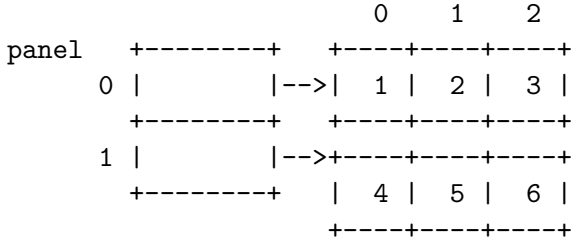
In Task A, we write codes for constructors and the destructor.

- Initialize `numRows` and `numCols` to be valid integers, representing number of rows and number of columns of a two-dimensional array, respectively.
- Allocate memory to hold a two-dimensional array with `numRows` rows and `numCols` columns, and put the intialize the address to `panel`.
- Put integers from 1 to `numRows * numCols` to the array from the top row to the bottom row, and for the same row, from left to right.
- It remains to randomize the elements in the array. This is done in method `randomize`.
- After randomization, need to find out the row and column indices of the empty cell and store them in `emptyCellRow` and `emptyCellCol` data members. In our project, integer `numRows * numCols` resides in the empty cell.
- Furthermore, need to calculate the number of **non-empty** cells with correct value placed in. That is, at row index i and column index j, where $0 \leq i < numRows$ and $0 \leq j < numCols$, its value is $1 \leq i * numCols + j + 1 \leq numRows * numCols$.

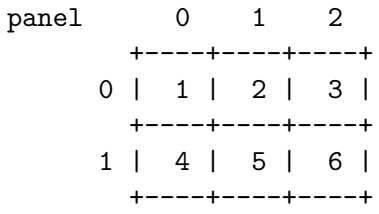
3.1 Method randomize

You must **follow the steps** to randomize the layout of these integers, otherwise, your code cannot pass gradescope.

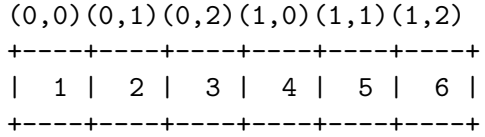
Suppose a panel is laid out as follows.



We find out `panel[0][0]` to be 1, `panel[0][1]` to be 2, $\cdots$, `panel[1][1]` to be 5, and `panel[1][2]` to be 6. So `panel` as a dynamically allocated 2-dimensional array can be TREATED as the following statically allocated 2-dimensional array. The difference is, for a statically allocated 2-dimensional array, the number of columns must be a const(ant), however, for a dynamically allocated one, its number of columns can be a variable initialized by users.



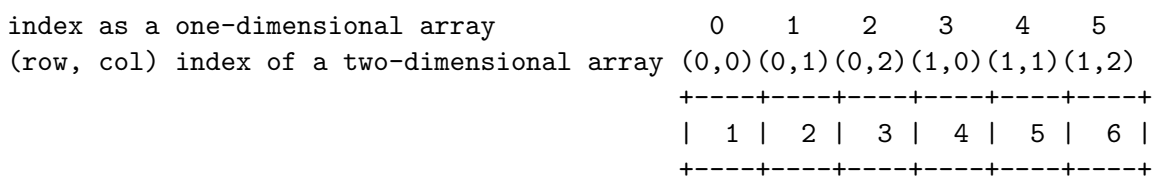
We can imagine the above elements are linearized as laying out elements row by row. (i, j) for row index i and column index j, where $0 \leq i <$ `numRows` and $0 \leq j <$ `numCols`.



We can map (i, j) to $i *$ `numCols` $+ j$. Reason:

- BEFORE the FIRST element at row index i, there are $i *$ `numCols` integers.
- At ith row, there are j integers BEFORE the element at column index j.

So it is like we have a one-dimensional array as follows.



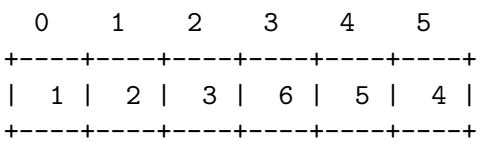
This is how we linearize that one-dimensional array.

1. Let `currLastIdx` to be the current last index of the array. Initialize it to be `numRows * numCols -1`.

2. Choose a random index from 0 to `currLastIdx`. Assume that 3 is selected.

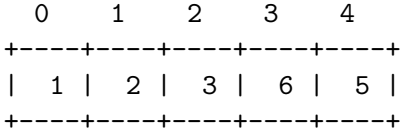
3. Swap the element indexed at k with the current last index. In the above example, we get the following setting. It is like 4 is selected.

```
index as a one-dimensional array
```



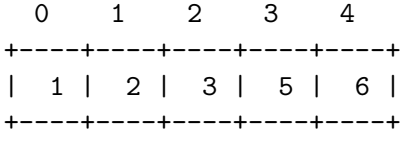
4. Reduce `currLastIdx` by 1. So the array looks as follows, as if the last element is truncated. So number 4 will not be selected again.

```
index as a one-dimensional array
```



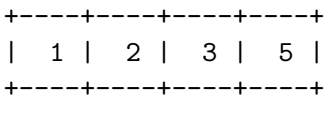
5. Choose a random index from 0 to `currLastIdx`. Suppose 3 is selected again. Swap the element at random index with element at `currLastIdx`.

```
index as a one-dimensional array
```



6. Reduce `currLastIdx` by 1. So we only need to concentrate on the following one-dimensional array.

```
index as a one-dimensional array
```



7. Repeat Steps 2 and 3 until `currLastIdx` is 0. That is, there is no more randomization is needed.

4 Update number of matched elements after each move

Roman is not built in one day. Trace changes in each move.

5 Wrap up: define BoardTest.cpp and create makefile

- Create `BoardTest.cpp` with the following contents. The purpose of `BoardTest.cpp` is to test constructors and methods defined in `Board.cpp`.

```cpp
#include "Board.hpp"
#include <iostream>
#include <string>
using namespace std;

int main() {
    //TODO: declare a board object called game using its default constructor

    //TODO: call play method of Board object game.

    return 0;
}
```

- Edit a file called `makefile` with the following contents.

```
# This is an example Makefile for number shuffle project.
# This program uses Board and BoardTest modules.
# Typing 'make' or 'make run' will create the executable file.
#

# define some Makefile variables for the compiler and compiler flags
# to use Makefile variables later in the Makefile: $()
#
#  -g    adds debugging information to the executable file
#  -Wall turns on most, but not all, compiler warnings
#
# for C++ define  CC = g++
CC = g++ -std=c++11
#CFLAGS  = -g -Wall

# typing 'make' will invoke the first target entry in the file
# (in this case the default target entry)
# you can name this target entry anything, but "default" or "all"
# are the most commonly used names by convention
#
all: run

# To create the executable file shuffle (see -o shuffle), we need the object files
# BoardTest.o and Board.o:
run:  BoardTest.o Board.o
$(CC) -o shuffle BoardTest.o Board.o

# To create the object file BoardTest.o, we need the source
# files BoardTest.cpp, Competition.h
BoardTest.o:  BoardTest.cpp
$(CC) -c BoardTest.cpp

# To create the object file Board.o, we need the source files
```

```
# Board.cpp.
# By default, $(CC) -c Board.cpp generates Board.o
Board.o:  Board.cpp
$(CC) -c Board.cpp

# To start over from scratch, type 'make clean'.  This
# removes the executable file, as well as old .o object
# files and *~ backup files:
#
clean:
$(RM) shuffle *.o *~
```

According to the command in this `makefile`,

`$(CC) -o shuffle BoardTest.o Board.o`

The generated runnable file is called `shuffle`, which appears after `-o`.

- Run make command.

  `make`

- If there is no error in the above command, run the following command, where dot (.) means current directory.

  `./shuffle`

6 One Solution

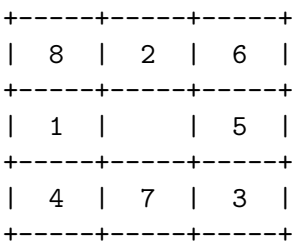
If you use `srand(2)` in a constructor in Mac, you would get the following layout.

8	2	6
1	7	5
4	3	

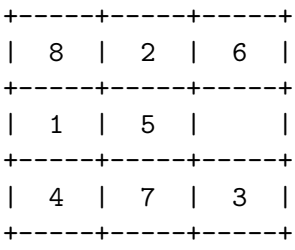
The optimal solution takes 22 moves. One such solution is shown as follows.

```
+-----+-----+-----+
|  8  |  2  |  6  |
+-----+-----+-----+
|  1  |  7  |  5  |
+-----+-----+-----+
|  4  |  3  |     |
+-----+-----+-----+

1. slide right

+-----+-----+-----+
|  8  |  2  |  6  |
+-----+-----+-----+
|  1  |  7  |  5  |
+-----+-----+-----+
|  4  |     |  3  |
+-----+-----+-----+
```

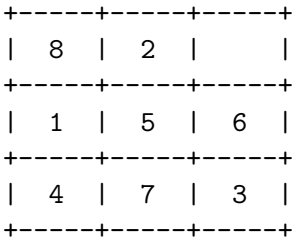
2. slide down



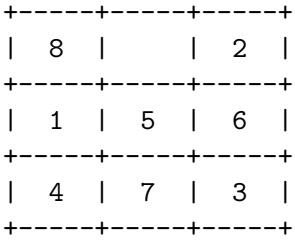
3. slide left



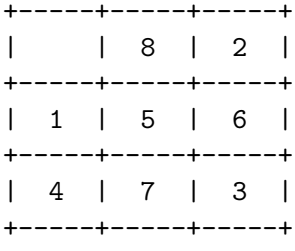
4. slide down



5. slide right



6. slide right



7. slide up

1	8	2
	5	6
4	7	3

8. slide up

1	8	2
4	5	6
	7	3

9. slide left

1	8	2
4	5	6
7		3

10. slide down

1	8	2
4		6
7	5	3

11. slide left

1	8	2
4	6	
7	5	3

```
+-----+-----+-----+
```

12. slide up

```
+-----+-----+-----+
|  1  |  8  |  2  |
+-----+-----+-----+
|  4  |  6  |  3  |
+-----+-----+-----+
|  7  |  5  |     |
+-----+-----+-----+
```

13. slide right

```
+-----+-----+-----+
|  1  |  8  |  2  |
+-----+-----+-----+
|  4  |  6  |  3  |
+-----+-----+-----+
|  7  |     |  5  |
+-----+-----+-----+
```

14. slide down

```
+-----+-----+-----+
|  1  |  8  |  2  |
+-----+-----+-----+
|  4  |     |  3  |
+-----+-----+-----+
|  7  |  6  |  5  |
+-----+-----+-----+
```

15. slide down

```
+-----+-----+-----+
|  1  |     |  2  |
+-----+-----+-----+
|  4  |  8  |  3  |
+-----+-----+-----+
|  7  |  6  |  5  |
+-----+-----+-----+
```

16. slide left

```
+-----+-----+-----+
|  1  |  2  |     |
+-----+-----+-----+
|  4  |  8  |  3  |
+-----+-----+-----+
```

```
|  7  |  6  |  5  |
+-----+-----+-----+
```

17. slide up

```
+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  4  |  8  |     |
+-----+-----+-----+
|  7  |  6  |  5  |
+-----+-----+-----+
```

18. slide up

```
+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  4  |  8  |  5  |
+-----+-----+-----+
|  7  |  6  |     |
+-----+-----+-----+
```

19. slide right

```
+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  4  |  8  |  5  |
+-----+-----+-----+
|  7  |     |  6  |
+-----+-----+-----+
```

20. slide down

```
+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  4  |     |  5  |
+-----+-----+-----+
|  7  |  8  |  6  |
+-----+-----+-----+
```

21. slide left

```
+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  4  |  5  |     |
```

```
+-----+-----+-----+
|  7  |  8  |  6  |
+-----+-----+-----+
```

22. slide up

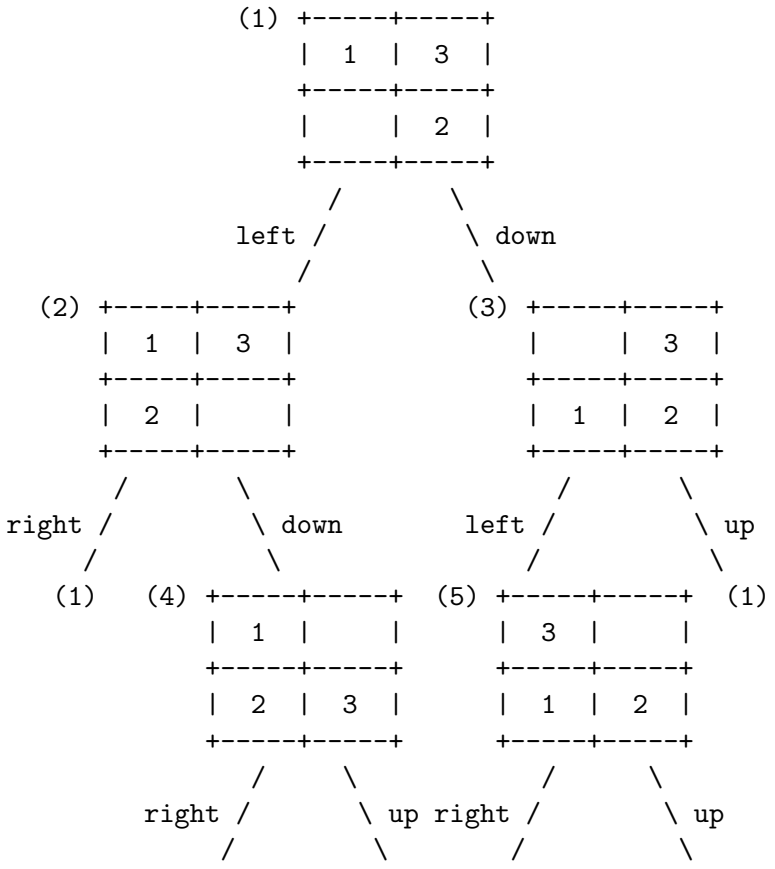
```
+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  4  |  5  |  6  |
+-----+-----+-----+
|  7  |  8  |     |
+-----+-----+-----+
```

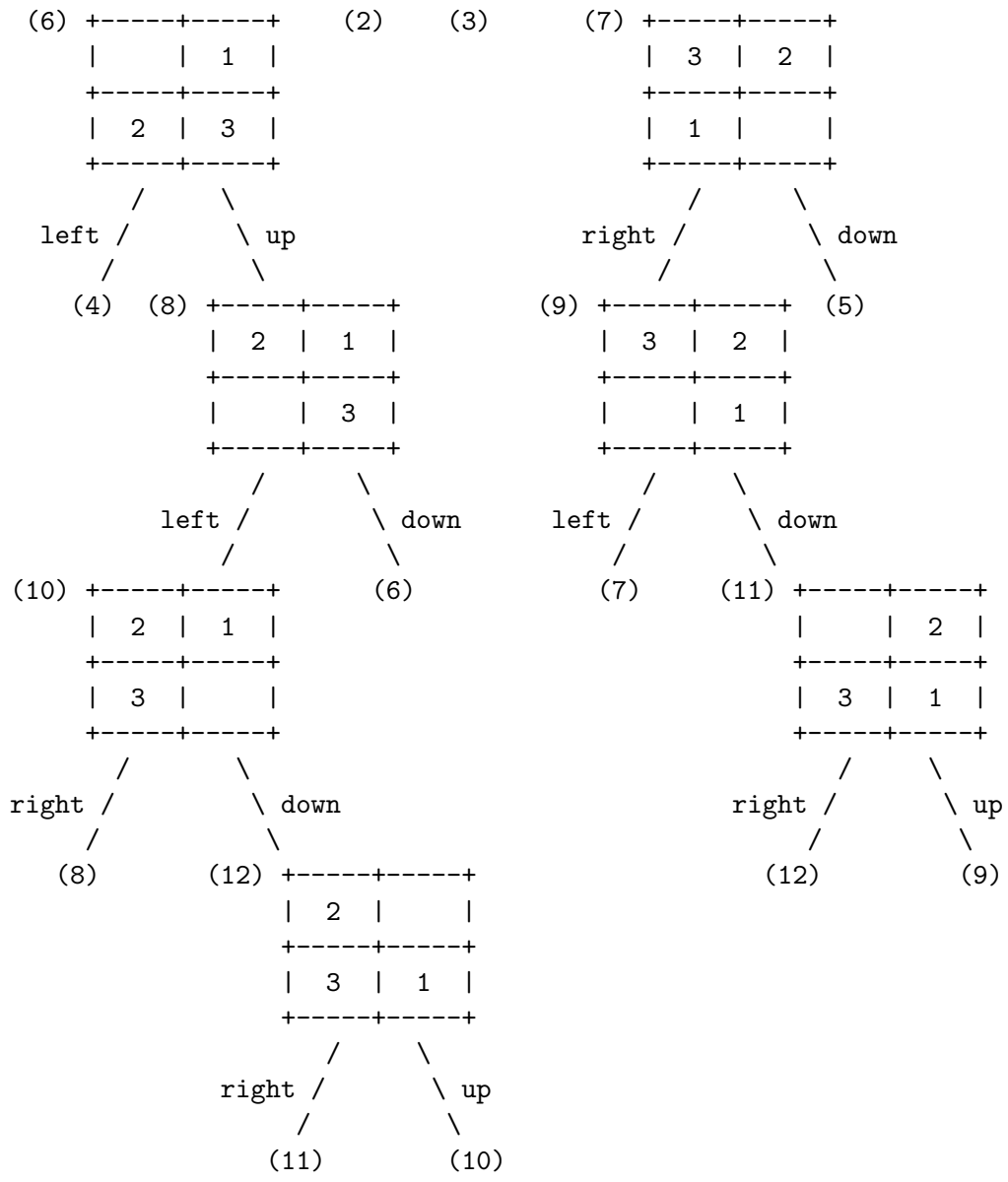
7 Optional: not every puzzle can be solved

Example 1: there is no way to slide left or right to put 1 and 2 to the leftmost two positions.

```
+-----+-----+-----+
|  2  |  1  |     |
+-----+-----+-----+
```

Example 2: here is an illustration for an unsolvable 2 x 2 puzzle. After an eligible move, if a layout is not shown before, it is drawn and labeled, otherwise, just list the label with the redundant layout.





8 Optional: solvable and optimization

Use Breadth First Search (BFS), introduced in CS 235, we can do the following:

- Find out whether a puzzle is solvable or not.
- If solvable, find a solution with the minimum number of moves.

9 Optional: public vs. private methods

If we do not want users to call the methods of a class, we may set them to be private. For example, methods randomize, getInfo, valueCorrect in Board class. It is like, for login method of BankAccount class may call verification method to test whether a username and a password are correct or not. However, for users, login method can be called directly, but not verification method. So login method is set to be public while verification method is set to be private.