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. You can download from Board.hpp from `https://tong-yee.github.io/135/f24/Board.hpp`.

   `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;
    void slideUp();
    void slideDown();
    void slideLeft();
    void slideRight();
    void play();

private:
    int numRows;
    int numCols;
    int** panel;
    int emptyCellRow;
    int emptyCellCol;
    int numCorrect; //cell with correct position
};
#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.

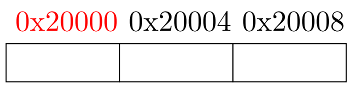
```
int* rowPtr = new int[numCols];
```

i. Warning: cannot write
`int* rowPtr = new int[numCols];` as
`int* rowPtr = new int[3];` *//wrong code*
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.

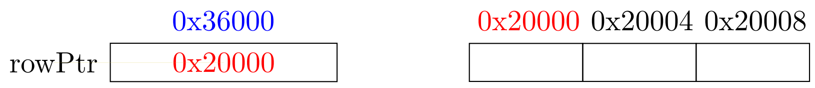
A. **Assume** that `numCols` is 3. **Assume** that the initial address of a dynamically allocated memory is 0x20000, where 0x before 20000 means the number is hexadecimal, whose base is 16. That is, 0x20000 equals $2 * 16^4 = 131072$ in decimal system. A memory address is represented in hexadecimal number.

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.

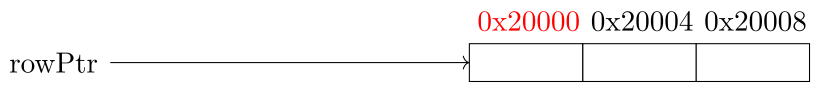


C. Note that the value of each memory slot is not decided yet. That is, expression `new int[numCols]` only allocates memory to hold `numCols` integers. It remains to initialize integers stored in those memories.

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 $i*\texttt{numCols}+j+1$, where $0 \leq i < \texttt{numRows}$ and $0 \leq j < \texttt{numCols}$.

(a) For example, in the following 3 * 3 panel, only number 2 (in bold green) sits in the correct cell, not any other non-empty entry. Hence `numCorrect` is 1. Also, `emptyCellRow` is 2 and `emptyCellCol` is 2.

	col index		
row index	0	1	2
0	8	**2**	7
1	1	6	5
2	4	3	🟦

(b) After sliding down operation, the layout is changed to be the following.

	col index		
row index	0	1	2
0	8	**2**	7
1	1	6	🟦
2	4	3	5

Now `emptyCellRow` is 1 and `emptyCellCol` is 2, `numCorrect` is still 1.

(c) Slide right and here is the layout.

	col index		
row index	0	1	2
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.

   Warning: the following code is wrong. `int` before `numRows` means to the varialbe is a local variable for constructor `Board`, but not data member `numRows`.

```
Board::Board() {
    ~~int~~ numRows = 3;
    ... //omit other code
}
```

   Correct way:

```
Board::Board() {
    numRows = 3;
    ... //omit other code
}
```

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.

```
                         0    1    2
panel   +--------+   +----+----+----+
      0 |        |-->|  1 |  2 |  3 |
        +--------+   +----+----+----+
      1 |        |-->+----+----+----+
        +--------+   |  4 |  5 |  6 |
      2 |        |   +----+----+----+
        +--------+-->+----+----+----+
                     |  7 |  8 |  9 |
                     +----+----+----+
```

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`.

```
           0    1    2
arr      +----+----+----+
      0  |  1 |  2 |  3 |
         +----+----+----+
      1  |  4 |  5 |  6 |
         +----+----+----+
```

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.

```
                         0    1    2
panel   +--------+   +----+----+----+
      0 |        |-->|  1 |  2 |  3 |
        +--------+   +----+----+----+
      1 |        |-->+----+----+----+
        +--------+   |  4 |  5 |  6 |
                     +----+----+----+
```

   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.

```
                        0    1    2    3    4
panel   +---------+   +----+----+----+----+----+
      0 |         |-->|  1 |  2 |  3 |  4 |  5 |
        +---------+   +----+----+----+----+----+
      1 |         |-->+----+----+----+----+----+
        +---------+   |  6 |  7 |  8 |  9 | 10 |
      2 |         |   +----+----+----+----+----+
        +---------+-->+----+----+----+----+----+
                      | 11 | 12 | 13 | 14 | 15 |
                      +----+----+----+----+----+
```

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.

```cpp
#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.

```cpp
#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 ~ 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

1. Define constructors and the destructor in `Board.cpp`.
2. Test codes locally.
   (a) Comment private: line in `Board.hpp` as `//private:`. This is for debug purpose.
   (b) 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;

    for (int row = 0; row < game->numRows; row++) {
        for (int col = 0; col < game->numCols; col++) {
            std::cout << game->panel[row][col];
            if (col < game->numCols-1) //skip the last ,
               std::cout << ",";
        }
        std::cout << std::endl;
    }

    game->~Board();
    std::cout << "After calling destructor, data member panel is " << game->panel <<
   std::endl;

    if (type == 'C') {
       //release dynamically allocated memory for arr
       for (int row = 0; row < numRows; row++) {
            delete[] arr[row];
            arr[row] = nullptr;
        }
        delete[] arr;
        arr = nullptr;
    }

    return 0;
}
```

Explanation of the code is as follows.

i. 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`.

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout<<argv[1] << std::endl;

    return 0;
}
```

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.

ii. 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'.

(c) Run the following command to compile main.cpp and Board.cpp.

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

(d) If there is no compilation errors, run the following command.

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

(e) 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
```

(f) 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
```

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

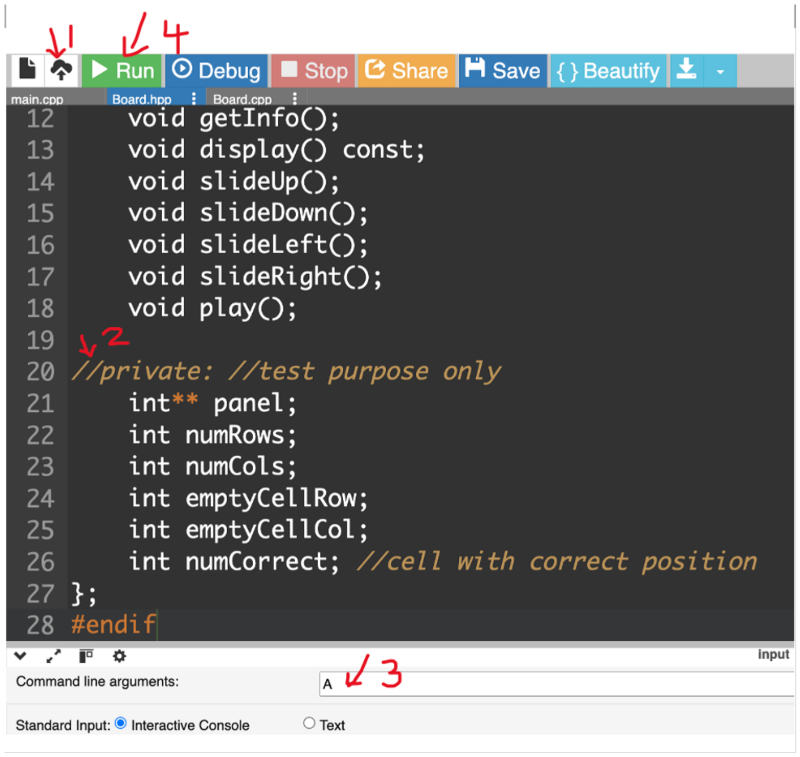
```
./a.out C
```

You should see the following output.

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

3. 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.



```
    void getInfo();
    void display() const;
    void slideUp();
    void slideDown();
    void slideLeft();
    void slideRight();
    void play();

//private: //test purpose only
    int** panel;
    int numRows;
    int numCols;
    int emptyCellRow;
    int emptyCellCol;
    int numCorrect; //cell with correct position
};
#endif
```

Command line arguments: A

Standard Input: [✓ Interactive Console] Text

Figure 1: Test Task A in onlinegdb.com

4. 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.

1. Initialize `numRows` and `numCols` to be valid integers, representing number of rows and number of columns of a two-dimensional array, respectively.

2. Allocate memory to hold a two-dimensional array with `numRows` rows and `numCols` columns, and put the intial address to `panel`.

3. 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.

4. It remains to randomize the elements in the array. This is done in method `randomize`.

5. 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.

6. 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

Must **follow the following steps** to randomize the layout of integers in `panel`, or your code fails gradescope.

Suppose a panel is laid out as follows.

```
                             0    1    2
panel   +---------+    +----+----+----+
      0 |         |-->|  1 |  2 |  3 |
        +---------+    +----+----+----+
      1 |         |-->+----+----+----+
        +---------+    |  4 |  5 |  6 |
                       +----+----+----+
```

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 constant, while for a dynamically allocated one, its number of columns can be a variable.

panel	col index		
row index	0	1	2
0	1	2	3
1	4	5	6

3.1.1 One approach to randomize elements in data member `panel`

In this approach, we do the following:

1. Create a dynamically allocated one-dimensional array or use a vector of integers to hold elements from 1 to `numRows * numCols`.

index in one-dimensional array	0	1	2	3	4	5
element at the index	1	2	3	4	5	6

2. Randomize the layout of integers in the above array. For details, see the following steps. Here is an **illustration** of randomized result.

index in one-dimensional array	0	1	2	3	4	5
element at the index	2	1	3	5	6	4

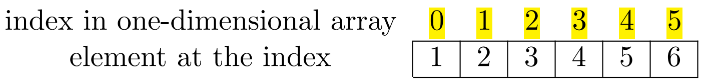
3. Copy the elements back to `panel`, from the top row to the bottom row; in the same row, from left to right.

panel	col index		
row index	0	1	2
0	2	1	3
1	5	6	4

4. If we use a dynamically allocated array in Step 1, need to release dynamically allocated memory. Remember to handle dangling pointer problem.

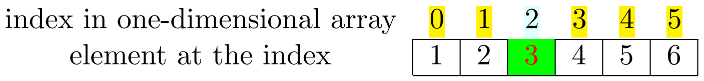
Here are steps to randomize the layout of integers in a one-dimensional array with elements 1, $\cdots$, `numRows * numCols`.

1. From the first index to the last index, place elements 1, $\cdots$, `numRows * numCols` to the array. Assume that `numRows` is 2 and `numCols` is 3. Then we have the following array.


index in one-dimensional array 0 1 2 3 4 5
element at the index 1 2 3 4 5 6

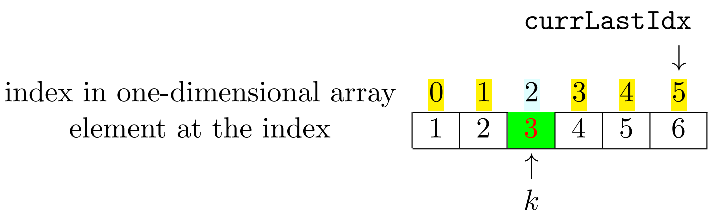
2. Let `currLastIdx` be the current last index of the array. Initialize it to be ________ (you fill in the blank, this expression is related with `numRows` and `numCols`.) In our example, it is 5, since we have `numRows * numCols` elements, and the index starts from 0.

3. Choose a random index from 0 to `currLastIdx`. Assume that **index** 2 is selected. Save the index in variable k. So element 3 **indexed** at 2 is selected.

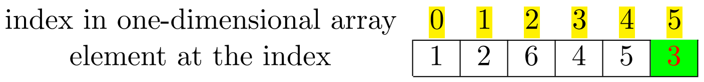

index in one-dimensional array 0 1 2 3 4 5
element at the index 1 2 3 4 5 6

4. Swap the element indexed at k with the element indexed at current last index `currLastIdx`. Doing so would avoid to select that same element again in next round of randomization. In the above example, we get the following layout.

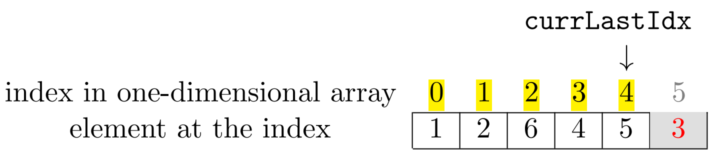
   Before swapping:


currLastIdx
index in one-dimensional array 0 1 2 3 4 5
element at the index 1 2 3 4 5 6
k

   After swapping:

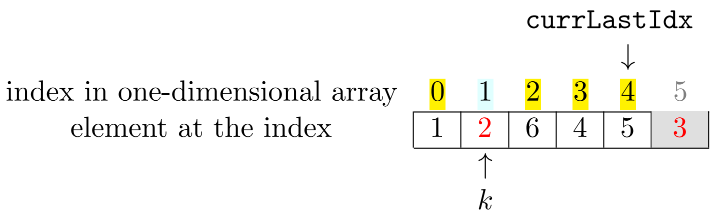

index in one-dimensional array 0 1 2 3 4 5
element at the index 1 2 6 4 5 3

5. Reduce `currLastIdx` by 1. The array looks as follows, as if the last element **were** truncated. So element 3, which is **indexed** at 5 after one randomization, will not be selected again.


currLastIdx
index in one-dimensional array 0 1 2 3 4 5
element at the index 1 2 6 4 5 3

6. Choose a random index from 0 to `currLastIdx`.

   (a) Suppose index 1 is selected. Save it in variable k.


currLastIdx
index in one-dimensional array 0 1 2 3 4 5
element at the index 1 2 6 4 5 3
k

   (b) Swap the element at random index with element at `currLastIdx`.

currLastIdx
↓

index in one-dimensional array	0	1	2	3	4	5
element at the index	1	5	6	4	2	3

7. Reduce `currLastIdx` by 1. So we only need to concentrate on the following one-dimensional array, ignoring the grayed cells.

currLastIdx
↓

index in one-dimensional array	0	1	2	3	4	5
element at the index	1	5	6	4	2	3

8. Repeat Steps 3, 4, and 5 until `currLastIdx` is 0, when randomization is done.

Here is a pseudocode.

```
//Suppose elements 1, ..., numRows * numCols are saved in array temp,
//from the first index to the last index.

declare and initialize currLastIdx to be ...
while (currLastIdx > 0)
begin
    select a random integer in [0, currLastIdx], save in variable k.
    swap temp[k] with temp[currLastIdx]
    decrease currLastIdx by 1
end
```

3.1.2 Another approach to randomize elements in data member `panel`

In the previous approach, we use a one-dimensional array to save the data, randomize them, then copy the randomized data back to data member `panel`. In this approach, we randomize directly in two-dimensional array `panel`, without the need to use a one-dimensional array.

Label elements in data member `panel` from the top row to the last row; in the same row, from left to right. Label the top left element as 0, its right neighbor to be 1, and so on.

Map label k, where $0 \leq k \leq \texttt{numRows} * \texttt{numCols} - 1$, to row index $k/\texttt{numCols}$ and column index $k\%\texttt{numCols}$. For example, when k is 5, the corresponding row index is $5/3 = 1$ and the column index is $5\%3 = 2$.

panel	col index		
row index	0	1	2
0	1	2	3
1	4	5	6

labels of elements	col index		
row index	0	1	2
0	0	1	2
1	3	4	5

Next we will randomize elements in the two-dimensional array as follows.

1. Initialize `currLastIdx` to be ________ (you fill in the blank, this expression is related with `numRows` and `numCols`). In our example, it is $2 * 3 - 1 = 5$.

2. Select a random integer in `[0, currLastIdx]`, where 0 in the first label and 5 is the last label. Suppose 2 is chosen.

3. Map label 2 back to row index `2 / numCols = 2 / 3 = 0` and `2 % numCols = 2 % 3 = 2` in the two-dimensional array.

4. Map `currLastIdx` 5 back to row index `5 / numCols = 5 / 3 = 1` and 5 % numCols = 5 % 3 = 2 in the two-dimensional array.

5. Swap elements at (0, 2) and (1, 2) in the original two-dimensional array `panel`.

   It is like element indexed at (0, 2) in the two-dimensional array is chosen, we swap it with the element at `currLastIdx` to avoid to choose it again.

panel	col index		
row index	0	1	2
0	1	2	6
1	4	5	3

6. Reduce `currLastIdx` by 1.

7. Repeat Steps 2-6 until `currLastIdx` is reduced to be 0.

Here is a pseudocode.

```
declare and initialize currLastIdx to be ...
while (currLastIdx > 0)
begin
    select a random integer in [0, currLastIdx], save in variable k.
    swap panel[k / numCols][k % numCols] with panel[currLastIdx / numCols][currLastIdx %
   numCols]
    decrease currLastIdx by 1
end
```

3.2 Define method `display`

This method prints out the layout of `panel` in tabular format to the screen, if a cell has value `numRows * numCols`, print as an empty cell. Suppose the current value of `panel` is as follows.

panel	col index		
row index	0	1	2
0	2	1	3
1	6	5	4

After calling display method, print out the following to the screen. Note that value `numRows * numCols` is shown as an empty cell.

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

You may call the following function when defining `display` method.

```cpp
void printSeparateLine(int numCols) {
    std::cout << "+";
    for (int col = 0; col < numCols; col++)
        std::cout << "-----+";

    std::cout << std::endl;
}
```

3.3 Define method `valueCorrect`

Value `numRows * numCols` is displayed as an empty cell. This method finds out whether a non-empty cell has the correct value resided. That is, given a cell with row index `row` and column index `col`, if the element residing at the cell equals `row * numCols + col + 1` and the element is in [1, `numRows * numCols -1`], then return true, otherwise, return false.

Suppose the values of data member `panel` is as follows.

panel	col index		
row index	0	1	2
0	2	1	3
1	6	5	4

expected output when the game succeeds	col index		
row index	0	1	2
0	1	2	3
1	4	5	

Note that only numbers 1, 2, $\cdots$, `numRows * numCols -1` are displayed. As a result, even if value `numRows * numCols` resides the bottom right cell with row index `numRows - 1` and column index `numCols - 1`, however, it is shown as an empty cell, thus `valueCorrect(numRows -1, numCols -1)` still returns false.

In the above distribution of `panel`, only elements at (0, 2) and (1, 1) – in green cells – are in their correct positions. That is, valueCorrect(0, 2) returns true, so is valueCorrect(1, 1).

3.4 Define method `getInfo`

In this method, check elements in data member `panel`, count the number of non-empty cells with correct placements, save that number in `numCorrect`, find out the row index of the empty cell and put it in `emptyCellRow`. Do similar thing for `emptyCellCol`.

After a randomization, use this method to initialize data members `numCorrect`, `emptyCellRow`, and `emptyCellCol`. That is, call method `getInfo` in the end of `randomize` method.

3.5 Submission of Task B

Based on code of `Board.cpp` in Task A, do the following steps. Then submit `Board.cpp` to gradescope.

1. In `Board::Board(int** arr, int m, int n)`, the elements in data member `panel` are laid out properly already; no need to randomize anymore. But need to call `getInfo` method to initialize `numCorrect`, `emptyCellRow`, and `emptyCellCol`.

2. Define `valueCorrect` method. You may need to call this method in `getInfo` method to calculate `numCorrect`.

3. Define `getInfo` method. **Call it in `randomize` method**.

4. Define `randomize` method. **Call it in constructors** Board::Board() and Board::Board(int m, int n).

5. Define `display` method.

6. Test locally before uploading to gradescope.

(a) Comment private: line in `Board.hpp` as `//private:`. This is for debug purpose. Need to uncomment when release the product.

(b) Comment all occurrences of `srand` statements in `Board.cpp`.

(c) Upload `Board.hpp` and `Board.cpp` to `https://www.onlinegdb.com/online_c++_compiler`. Note that compilers in different operating systems – linux, Mac, windows – may get different random numbers for `srand` statement. onlinegdb runs in Linux and has the same results in servers of gradescope.

(d) Edit main.cpp in onlinegdb.

```cpp
#include <iostream>
#include "Board.hpp"
#include <cstdlib> //srand

int main() {
    std::cout << "Use srand(2) and default constructor" << std::endl;
    srand(2);
    Board game;
    game.display();
    std::cout << "number of correct cells: " << game.numCorrect << std::endl;
    std::cout << "row of empty cell: " << game.emptyCellRow << std::endl;
    std::cout << "column of empty cell: " << game.emptyCellCol << std::endl;

    std::cout << "\nUse srand(8) and constructor Board(int m, int n)" << std::endl;
    srand(8);
    Board game2(3, 4);
    game2.display();
    std::cout << "number of correct cells: " << game2.numCorrect << std::endl;
    std::cout << "row of empty cell: " << game2.emptyCellRow << std::endl;
    std::cout << "column of empty cell: " << game2.emptyCellCol << std::endl;
    return 0;
}
```

(e) You should get the following output.

```
Use srand(2) and default constructor
+-----+-----+-----+
|  4  |  5  |  1  |
+-----+-----+-----+
|  3  |  2  |     |
+-----+-----+-----+
|  6  |  8  |  7  |
+-----+-----+-----+
number of correct cells: 1
row of empty cell: 1
column of empty cell: 2

Use srand(8) and constructor Board(int m, int n)
+-----+-----+-----+-----+
| 10  |  4  |  1  | 11  |
+-----+-----+-----+-----+
|  2  |     |  7  |  8  |
+-----+-----+-----+-----+
|  9  |  3  |  6  |  5  |
+-----+-----+-----+-----+
number of correct cells: 3
row of empty cell: 1
column of empty cell: 1
```

(f) If everything runs fine, upload `Board.cpp` to gradescope.

4 Task C: define `slideUp`, `slideDown`, `slideLeft`, and `slideRight` methods

Whenever we slide up/down/left/right, empty cell may be changed, so is the number of elements in correct placement. So we need to update data members `emptyCellRow`, `emptyCellCol`, and `numCorrect` in these methods.

4.1 Define method `slideUp`

1. Call element underneath a cell its downstair neighbor.
2. If the empty cell has no downstair neighbor, that is, the empty cell is on the last row already, ie, `emptyCellRow +1` >= `numRows` or `emptyCellRow == numRows -1`, there is nothing to do in slide up operation. Return to the caller.

panel		col index	
row index	0	1	2
0	4	5	1
1	3	2	6
2	8	9	7

display result		col index	
row index	0	1	2
0	4	5	1
1	3	2	6
2	8		7

3. Now the empty cell has a downstair neighbor. Do the following in slide up operation:

   (a) If the downstair neighbor of the empty cell is at its correct location before sliding up, that is, call valueCorrect on this neighbor, the return is true, then after sliding up, this cell is not in the correct position, so `numCorrect` is decreased by 1.

   Before sliding up (the cells where elements are in good positions are in green color):

panel		col index	
row index	0	1	2
0	4	5	1
1	6	9	2
2	7	8	3

display result		col index	
row index	0	1	2
0	4	5	1
1	6		2
2	7	8	3

   After sliding up (the cells where elements are in good positions are in green color):

panel		col index	
row index	0	1	2
0	4	5	1
1	6	9	2
2	7	8	3

display result		col index	
row index	0	1	2
0	4	5	1
1	6	8	2
2	7		3

   (b) Sliding up takes several steps.

i. Downstair neighbor moves up and the empty cell moves down. That is, swap element at the empty cell with its downstair neighbor.

ii. After the above swapping, the value of downstair neighbor resides in the previous empty cell. If, after swapping, the element is in correct position, then `numCorrect` is increased by 1.

A. To test whether an element is in correct position, we can use `valueCorrect` method, which takes row- and column- indices as parameters.

B. Observe that the old downstair neighbor takes up the position of the previous empty cell. Before we update the row- and col- indices for newly empty cell, the row index of old downstair neighbor is `emptyCellRow` and the column index is `emptyCellCol`, but the value `panel[emptyCellRow][emptyCe` is the value of old downstair neighbor after swapping.

C. That is, call `valueCorrect` on `emptyCellRow` and `emptyCellCol`, check the return. If the return is true, then increase `numCorrect` by 1.

Before sliding up (the cells where elements are in good positions are in green color):

panel — col index

row index	0	1	2
0	1	3	2
1	7	8	9
2	4	5	6

display result — col index

row index	0	1	2
0	1	3	2
1	7	8	
2	4	5	6

After sliding up (the cells where elements are in good positions are in green color):

panel — col index

row index	0	1	2
0	1	3	2
1	7	8	9
2	4	5	6

display result — col index

row index	0	1	2
0	1	3	2
1	7	8	6
2	4	5	

iii. Update `emptyCellRow` but not `emptyCellCol` (why??) for the current empty cell.

(c) Call display() method to display panel in tabular format after sliding up.

(d) Warning: you may opt to call `getInfo` method to update values for `emptyCellRow`, `emptyCellCol`, and `numCorrect` after swapping the previous empty cell with its downstair neighbor. This approach works but is not efficient.

i. Reason: method `getInfo` goes through every cell in `panel` to check. It is necessary when we re-arrange elements of `panel` in method `randomize`.

ii. However, in `slideUp` method, only the empty cell and its downstair neighbor are re-arranged, call `getInfo` method is overkill.

4.2 Submission for Task C

Define methods `slideUp, slideDown, slideLeft, and slideRight` in `Board.cpp` from Task B, submit to gradescope.

5 Task D: define `play` method

Skeleton of play method is as follows.

```
#include "Board.hpp"
#include <iomanip> //setw
```

```cpp
#include <cstdlib> //rand, srand
#include <ctime> //rand, srand
#include <iostream> //cout, endl

//TODO: add codes from Task C

void Board::play() {
    display();
    int move = 0;
    while ( ) { //TODO: fill in condition
        char ch = getchar();
        if (ch == 'S' || ch == 's') //STOP
           break;

        if (ch == '\[') { // if the first value is esc
            getchar(); // skip the [
            switch(getchar()) { // the real value
            case 'A':
                // code for arrow up
                move++;
                std::cout << "\nMove " << std::setw(4) << move << ": ";
                std::cout << "Slide UP." << std::endl;
                slideUp();
                break;
            case 'B':
                //TODO: code for arrow down
            case 'C':
                //TODO: code for arrow right

            case 'D':
                //TODO: code for arrow left
           }
        }
    }

    std::cout << "\nCongratulations. Total number of moves is " << move << "." << std::endl;
}
```

Submit `Board.cpp` to gradescope.

6 Wrap up: define BoardTest.cpp and create makefile

1. 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;
}
```

2. Edit a file called `makefile` with the following contents. You can download the file from `https://tong-yee.github.io/135/f24/makefile`.

   makefile includes instructions on how to compile and link multiple files in a project.

```
# 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`.

3. Run make command.

   ```
   make
   ```

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

   ```
   ./shuffle
   ```

5. If you modify any `Board.hpp`, `Board.cpp`, or `BoardTest.cpp`, just run commands in Steps 3 and 4. With a properly defined makefile, only modified source codes are re-compiled and re-linked.

   Using makefile simplifies management of a project with many files.

7 One Solution

In onlinegdb, upload `Board.hpp` and `Board.cpp`. And write the following code in `main.cpp` of onlinegdb.

```cpp
#include <iostream>
#include "Board.hpp"
#include <cstdlib> //srand

int main() {
    srand(2);

    Board game(2, 3);
    game.play();

    return 0;
}
```

Click run button and the output should be as follows.

```
+-----+-----+-----+
|  4  |  2  |  3  |
+-----+-----+-----+
|     |  5  |  1  |
+-----+-----+-----+
^[[D
```

```
Move    1: Slide LEFT.
+-----+----+-----+
|  4  |  2 |  3  |
+-----+----+-----+
|  5  |    |  1  |
+-----+----+-----+
^[[D
Move    2: Slide LEFT.
+-----+----+-----+
|  4  |  2 |  3  |
+-----+----+-----+
|  5  |  1 |     |
+-----+----+-----+
^[[B
Move    3: Slide DOWN.
+-----+----+-----+
|  4  |  2 |     |
+-----+----+-----+
|  5  |  1 |  3  |
+-----+----+-----+
^[[C
Move    4: Slide RIGHT.
+-----+----+-----+
|  4  |    |  2  |
+-----+----+-----+
|  5  |  1 |  3  |
+-----+----+-----+
^[[A
Move    5: Slide UP.
+-----+----+-----+
|  4  |  1 |  2  |
+-----+----+-----+
|  5  |    |  3  |
+-----+----+-----+
^[[C
Move    6: Slide RIGHT.
+-----+----+-----+
|  4  |  1 |  2  |
+-----+----+-----+
|     |  5 |  3  |
+-----+----+-----+
^[[B
Move    7: Slide DOWN.
+-----+----+-----+
|     |  1 |  2  |
+-----+----+-----+
|  4  |  5 |  3  |
+-----+----+-----+
^[[D
```

```
Move    8: Slide LEFT.
+-----+-----+-----+
|  1  |     |  2  |
+-----+-----+-----+
|  4  |  5  |  3  |
+-----+-----+-----+
^[[D
Move    9: Slide LEFT.
+-----+-----+-----+
|  1  |  2  |     |
+-----+-----+-----+
|  4  |  5  |  3  |
+-----+-----+-----+
^[[A
Move   10: Slide UP.
+-----+-----+-----+
|  1  |  2  |  3  |
+-----+-----+-----+
|  4  |  5  |     |
+-----+-----+-----+
Congratulations! Total number of moves: 10
```

8 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.

```
                     (1) +-----+-----+
                         |  1  |  3  |
                         +-----+-----+
                         |     |  2  |
                         +-----+-----+
                           /       \
                     left /         \ down
                         /           \
          (2) +-----+-----+          (3) +-----+-----+
              |  1  |  3  |              |     |  3  |
              +-----+-----+              +-----+-----+
              |  2  |     |              |  1  |  2  |
              +-----+-----+              +-----+-----+
                /       \                  /       \
         right /         \ down       left /         \ up
              /           \              /           \
 (1)   (4) +-----+-----+ (5) +-----+-----+  (1)
           |  1  |     |     |  3  |     |
           +-----+-----+     +-----+-----+
           |  2  |  3  |     |  1  |  2  |
           +-----+-----+     +-----+-----+
              /     \            /     \
       right /       \ up right /       \ up
            /         \        /         \
 (6) +-----+-----+    (2)   (3)      (7) +-----+-----+
     |     |  1  |                       |  3  |  2  |
     +-----+-----+                       +-----+-----+
     |  2  |  3  |                       |  1  |     |
     +-----+-----+                       +-----+-----+
        /     \                             /      \
  left /       \ up                  right /        \ down
      /         \                         /          \
    (4)  (8) +-----+-----+        (9) +-----+-----+  (5)
             |  2  |  1  |            |  3  |  2  |
             +-----+-----+            +-----+-----+
             |     |  3  |            |     |  1  |
             +-----+-----+            +-----+-----+
                /      \                 /     \
          left /        \ down     left /       \ down
              /          \             /         \
(10) +-----+-----+     (6)            (7)     (11) +-----+-----+
     |  2  |  1  |                                 |     |  2  |
     +-----+-----+                                 +-----+-----+
     |  3  |     |                                 |  3  |  1  |
     +-----+-----+                                 +-----+-----+
       /       \                                      /     \
right /         \ down                         right /       \ up
     /           \                                  /         \
   (8)       (12) +-----+-----+                  (12)        (9)
                  |  2  |     |
                  +-----+-----+
                  |  3  |  1  |
                  +-----+-----+
                     /     \
             right /        \ up
                  /          \
                 (11)       (10)
```

9 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.

10 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.