21.y — Chapter 21 project

ALEX DECEMBER 28, 2023

A tip o' the hat to reader Avtem for conceiving of and collaborating on this project.

Project time

Let's implement the classic game 15 Puzzle (https://en.wikipedia.org/wiki/15_puzzle)!

In 15 Puzzle, you begin with a randomized 4×4 grid of tiles. 15 of the tiles have numbers 1 through 15. One tile is missing.

For example:

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

In this puzzle, the missing tile is in the upper-left corner.

Each turn of the game, you pick one of the tiles that is adjacent to the missing tile, and slide it into the spot occupied by the missing tile.

The goal of the game is to slide tiles around until they are in numerical order, with the missing tile in the bottom right corner:

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

You can play a few rounds on this site (https://15puzzle.netlify.app/). It will help you to understand how this game works and how it should be implemented.

In our version of the game, each turn the user will enter a single letter command. There are 5 valid commands:

- w - slide tile up
- a - slide tile left
- s - slide tile down
- d - slide tile right
- q - quit game

Because this is going to be a longer program, we'll develop it in stages.

One more thing: at each step, we'll present two things: a goal and tasks. The goal defines the outcome that the step is trying to achieve, along with any additional relevant information. The tasks provide detail and hints about how to implement the goal.

The tasks will initially be hidden from view, to encourage you to see if you can complete each step using just the goal and sample output or sample program. If you are unsure how to start, or are feeling stuck, you can unhide the tasks. They should help get you moving forward.

> Step #1

Because this is going to be a bigger program, let's start with a design exercise.

> **Author's note**
>
> If you don't have much experience designing programs up-front, you may find this a bit difficult. That's expected. It's less important that you get it right, and more important that you participate and learn.
>
> We will go into more detail on all of these items in subsequent steps, so if you are feeling completely lost, feel free to skip this step.

Goal: Document the key requirements for this program, and plan how your program will be structured at a high level. We will do this in three parts.

A) What are the top-level things that your program needs to do? Here are a few to get you started:

Board things:

- Display the game board
- ...

User things:

- Get commands from user
- ...

Show Solution (javascript:void(0))

B) What primary classes or namespaces will you use to implement the items outlined in Step 1? Also, what will your main() function do?

You can create a diagram, or use two tables like this:

Primary class/namespace/main	Implements top-level items	Members
class Board	Display the game board ...	...
function main	Main game logic loop ...	...

Show Solution (javascript:void(0))

C) (extra credit) Can you think of any helper classes or capabilities that will make implementing the above easier or more cohesive?

Show Solution (javascript:void(0))

If you had a hard time with this exercise, that's okay. The goal here was mainly to get you thinking about what you're going to do before you start doing it.

Now, it's time to get implementing!

> Step #2

Goal: Be able to display individual tiles on the screen.

Our game board is a 4×4 grid of tiles that can slide around. Therefore, it will be useful to have a `Tile` class that represents one of the numbered tiles on our 4×4 grid or the missing tile. Each tile should be able to:

- Be given a number or be set as the missing tile
- Determine whether it is the missing tile.
- Draw to the console with the appropriate spacing (so the tiles will line up when the board is displayed). See the sample output below for an example indicating how tiles should be spaced.

Show Tasks (javascript:void(0))

The following code should compile and produce the output result you can see below the code:

```
int main()
{
    Tile tile1{ 10 };
    Tile tile2{ 8 };
    Tile tile3{ 0 }; // the missing tile
    Tile tile4{ 1 };

    std::cout << "0123456789ABCDEF\n"; // to make it easy to see how many spaces are in the next line
    std::cout << tile1 << tile2 << tile3 << tile4 << '\n';

    std::cout << std::boolalpha << tile1.isEmpty() << ' ' << tile3.isEmpty() << '\n';
    std::cout << "Tile 2 has number: " << tile2.getNum() << "\nTile 4 has number: " << tile4.getNum() << '\n';

    return 0;
}
```

Expected output (pay attention to the white spaces):

```
0123456789ABCDEF
 10   8       1
false true
Tile 2 has number: 8
Tile 4 has number: 1
```

Show Solution (javascript:void(0))

> Step #3

Goal: Create a solved board (4×4 grid of tiles) and display it on the screen.

Define a `Board` class that will represent 4×4 grid of tiles. A newly created `Board` object should be in the solved state. To display the board, first print `g_consoleLines` (defined in code snippet below) empty lines and then print the board itself. Doing so will ensure that any prior output is pushed out of view so that only the current board is visible on the console.

Why initiate the board in the solved state? When you buy a physical version of these puzzles, the puzzles typically start in the solved state -- you have to manually mix them up (by sliding tiles around) before trying to solve them. We will mimic that process in our program (we'll do the mixing up in a future step).

Show Tasks (javascript:void(0))

The following program should run:

```
    // Increase amount of new lines if your board isn't
    // at the very bottom of the console
    constexpr int g_consoleLines{ 25 };

    // Your code goes here

    int main()
    {
        Board board{};
        std::cout << board;

        return 0;
    }
```

and output the following:

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

Show Solution (javascript:void(0))

> Step #4

Goal: In this step, we'll allow the user to repeatedly input game commands, handle invalid input, and implement the quit game command.

These are the 5 commands our game will support (each of which will be input as a single character):

- 'w' - slide tile up
- 'a' - slide tile left
- 's' - slide tile down
- 'd' - slide tile right
- 'q' - quit game

When the user runs the game, the following should occur:

- The (solved) board should be printed to the console.
- The program should repeatedly get valid game commands from the user. If the user enters an invalid command or extraneous input, ignore it.

For each valid game command:

- Print `"Valid command: "` and the character the user input.
- If the command is the quit command, also print `"\n\nBye!\n\n"` and then quit the app.

Because our user input routines do not need to maintain any state, implement them inside a namespace named `UserInput`.

Show Tasks (javascript:void(0))

The output of the program should match the following:

```
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15
w
Valid command: w
a
Valid command: a
s
Valid command: s
d
Valid command: d
f
g
h
Valid command: q


Bye!
```

Show Solution (javascript:void(0))

> Step #5

Goal: Implement a helper class that will make it easier for us to handle directional commands.

After implementing the prior step, we can accept commands from the user (as characters 'w', 'a', 's', 'd', and 'q'). These characters are essentially magic numbers in our code. While it's fine to handle these commands in our `UserInput` namespace and function `main()`, we don't want to propagate them throughout our whole program. For example, the `Board` class should have no knowledge of what 's' means.

Implement a helper class named `Direction`, which will allow us to create objects that represent the cardinal directions (up, left, down, or right). `operator-` should return the opposite direction, and `operator<<` should print the direction to the console. We will also need a member function that will return a Direction object containing a random direction. Lastly, add a function to namespace `UserInput` that converts a directional game command ('w', 'a', 's', or 'd') to a Direction object.

The more we can use `Direction` instead of directional game commands, the easier our code will be to read and understand.

Show Tasks (javascript:void(0))

Finally, modify the program you wrote in the prior step so that the output matches the following:

```
  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15
Generating random direction... up
Generating random direction... down
Generating random direction... up
Generating random direction... left

Enter a command: w
You entered direction: up
a
You entered direction: left
s
You entered direction: down
d
You entered direction: right
q


Bye!
```

Show Solution (javascript:void(0))

> Step #6

Goal: Implement a helper class that will make it easier for us to index the tiles in our game board.

Our game board is a 4×4 grid of `Tile`, which we store in two-dimensional array member `m_tiles` of the `Board` class. We will access a given tile using its {x, y} coordinates. For example, the top left tile has coordinate {0, 0}. The tile to the right of that has coordinate {1, 0} (x becomes 1, y stays 0). The tile one down from that has coordinate {1, 1}.

Since we'll be working with coordinates a lot, create a helper class named `Point` that stores an {x, y} pair of coordinates. We should be able to compare two Point objects for equality and inequality. Also implement a member function named `getAdjacentPoint` that takes a Direction object as a parameter and returns the Point in that direction. For example, `Point{1, 1}.getAdjacentPoint(Direction::right)` == `Point{2, 1}`.

Show Tasks (javascript:void(0))

Save your `main()` function from the prior step, as you'll need it again in the next step.

The following code should run and print `true` for every test-case:

```
    // Your code goes here

    // Note: save your main() from the prior step, as you'll need it again in the next step
    int main()
    {
        std::cout << std::boolalpha;
        std::cout << (Point{ 1, 1 }.getAdjacentPoint(Direction::up)    == Point{ 1, 0 }) << '\n';
        std::cout << (Point{ 1, 1 }.getAdjacentPoint(Direction::down)  == Point{ 1, 2 }) << '\n';
        std::cout << (Point{ 1, 1 }.getAdjacentPoint(Direction::left)  == Point{ 0, 1 }) << '\n';
        std::cout << (Point{ 1, 1 }.getAdjacentPoint(Direction::right) == Point{ 2, 1 }) << '\n';
        std::cout << (Point{ 1, 1 } != Point{ 2, 1 }) << '\n';
        std::cout << (Point{ 1, 1 } != Point{ 1, 2 }) << '\n';
        std::cout << !(Point{ 1, 1 } != Point{ 1, 1 }) << '\n';

        return 0;
    }
```

Show Solution (javascript:void(0))

> Step #7

Goal: Add the ability for players to slide the tiles on the board.

First, we should take a closer look at how sliding tiles actually works:

Given a puzzle state that looks like this:

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

When the user enters 'w' on the keyboard, the only tile that can go up is tile `2`.

After moving the tile, the board looks like this:

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

So, essentially what happened is we swapped the empty tile with tile `2`.

Let's generalize this procedure. When the user enters a directional command, we need to:

- Locate the empty tile.
- From the empty tile, find the adjacent tile that is in the direction opposite of the direction the user entered.
- If the adjacent tile is valid (it's not off the grid), swap the empty tile and adjacent tile.
- If the adjacent tile is not valid, do nothing.

Implement this by adding a member function `moveTile(Direction)` to the class `Board`. Add this to your game loop from step 5. If the user successfully slides a tile, the game should redraw the updated board.

Show Tasks (javascript:void(0))

Show Solution (javascript:void(0))

> Step #8

Goal: In this step, we'll finish our game. Randomize the initial state of the game board. Also, detect when user wins, so after that we can print a win message and quit the game.

We need to be careful about how we randomize our puzzle, because not every puzzle is solvable. For example, there is no way to solve this puzzle:

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

If we just blindly randomize the numbers in the puzzle, there is a chance that we will generate such an unsolvable puzzle. With a physical version of the puzzle, we'd randomize the puzzle by sliding tiles in random directions until the tiles were sufficiently mixed. The solution for such a randomized puzzle is to slide each tile in the opposite direction that it was slid to randomize it in the first place. Thus, randomizing puzzles this way always generates a solvable puzzle.

We can have our program randomize the board in the same way.

Once the user has solved the puzzle, the program should print `"\n\nYou won!\n\n"` and then exit normally.

Show Tasks (javascript:void(0))

Here is the full solution for our 15 puzzle game:

Show Solution (javascript:void(0))

Next lesson

22.1 Introduction to smart pointers and move semantics

Back to table of contents

Previous lesson

21.x Chapter 21 summary and quiz
