CSE 3150 – Lab 4
Building the 2048 Game in C++

By: Dr. Justin Furuness

Overview

In this lab, you will implement a simplified version of the game **2048**. You will start from provided starter code and incrementally add functionality until your program passes the given test suite.

By the end, you will:

- Understand how to manipulate 2D arrays in C++.
- Use Standard Template Library (**STL**) tools such as `std::vector`, `std::stack`, and `std::algorithm`.
- Apply iterators, the `auto` keyword, and adapters effectively.
- Implement game mechanics such as shifting, merging, and undo.
- Verify your program using automated tests.

Your work must be committed to a file called **solution.cpp** in your GitHub repository **cse3150_lab_4**. Your submission is complete when all tests pass.

Game Rules Recap

The game is played on a 4×4 board:

- Each turn, the player chooses a direction: left (`a`), right (`d`), up (`w`), or down (`s`).

- Tiles slide in that direction (if possible). Adjacent tiles of the same value **merge** into one tile of double the value.
- After each move, a new tile (usually a 2) spawns in an empty cell.
  - Note: If you attempt to slide left for example, but every cell was already slid to the left, then the move desn't count and no new tiles spawn. Moves only count when something changes on the board.
- The player can type `u` to undo the last move.
- The player can type `q` to quit.

Starter Files

You are given two files:

- `starter.cpp` — Skeleton code for the game (already configured to write board state to CSV).
- `test_game.py` — Pytest tests that validate your implementation.

You will build your solution by editing and renaming `starter.cpp` into **solution.cpp**.

Step 1: Representing the Board

The board is a 4×4 grid of integers. In C++, this can be stored as:

```
std::vector<std::vector<int>> board(4, std::vector<int
   >(4, 0));
```

Key C++ tools for this step:

- Use `std::vector` instead of raw arrays for flexibility.
- Access rows with `board[i]` and individual cells with `board[i][j]`.
- Use range-based `for` loops with `auto&` to traverse rows and cells.

At the bottom of the print board function there is a function call that writes the board to the CSV. This is already implemented for you so do not change it - it is meant to output the board for testing purposes in our file that we will run with pytest.

Step 2: Shifting and Compressing Tiles

To shift tiles left, right, up, or down, we first need to *compress* them by removing zeros. A recommended STL approach is to use the copy_if function. Start with an empty vector compressed, and then using copy_if and a back_inserter and a lambda, you can copy the row to your compressed vector. copy_if also comes with the ability to filter as you copy, read the documentation for how to do this and pass in a lambda to filter out the zeros.

```
std::vector<int> compressed;
```

This uses:

- `std::copy_if` from `<algorithm>` to filter tiles.
- A `std::back_inserter` from `<iterator>` to append efficiently.
- A lambda expression to keep only non-zero entries.

After compression, pad with zeros to return the row to length 4.

Step 3: Merging Tiles

When two adjacent tiles are equal, merge them (multiplying the first value by two and setting the second value to zero):

Then call the compress step again to remove the zeros created by merging. This ensures the "no double merge" rule is respected.

Step 3: Moving the tiles left

Start with move_left since it is the easiest. For each row in your board, compress the row and merge the row, and set the row to this new merged value. If any new row is different from any old row, then the move was valid and the function should return true. Otherwise, if the board didn't change at all, the function should return False.

Try the game out now and try moving left.

Step 4: Moving the tiles right

Now let's move on to move right. This one should be fairly simple to your left function, except we just need to reverse our vector. Do so using the std::reverse, and look up the documentation on how to do this. Then compress and merge the row, and reverse the row again back to it's original direction! Implement the rest of the logic similar to moving left.

Try the game out now and try moving both left and right.

Step 5: Spawning New Tiles

After each valid move:

- Collect empty positions in a vector of pairs.
- Use `std::mt19937` and `std::uniform_int_distribution` from `<random>` to select a random empty cell.
- Assign a 2 (90% of the time) or 4 (10%).
- We haven't covered this in class, and even if we did, who could possibly remember this crazy syntax? This is a perfect place to use AI since this is a syntax heavy one-off. Prompt chat GPT or another AI well to complete this function.

Step 6: Moving the tiles up and down

Now we need to move the tiles up and down. You can do this in a fairly similar way to move left, except this time just create a new vector that will represent the column itself. Implement these two functions. Now you should be able to move in any direction!

Step 7: Implementing Undo

To support undo:

- Use the adapter `std::stack` from `<stack>` to store previous board states.
- Before applying a move, push a copy of the board.
- On undo (`u`), restore from the stack if not empty, print the board, and continue

Step 8: Implementing Score

To support this:

- Modify the stub for the score function to take instead a template for a reference class Board and make computing the score more abstract.
- Calculate the score properly by merely adding up every tile on the board.

Step 9: Putting It All Together

At this point everything should be working and you should not need to implement anything further. Your `main()` loop should:

1. Print the current board and (already handled in starter code) write it to CSV.
2. Read a character command from input.
3. Use a `switch` statement for actions (`a`, `d`, `w`, `s`, `u`, `q`).
4. For moves: push the board to history, compress, merge, compress again, then spawn a new tile.
5. For undo: restore from the stack.
6. For quit: break the loop.

Testing Your Code

Run the provided tests with:

```
pytest test_game.py
```

Your code is correct when all tests pass. Push your final version to GitHub as **solution.cpp** under the repo cse3150_week_4_lab