CSE 3150 – Lab 5
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_week_5_lab**. 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 and Displaying 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));
```

Implement the `print_board` function to display the board so that someone can play the game. Additionally, calculate and display the current score (sum of all tiles) using the provided `compute_score` function.

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.
- Display empty cells as dots (.) for clarity.

The printed board should look something like this:

```
Score: 64
4       2       2       .
8       .       .       .
2       .       .       .
16      .       32      .
```

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: Understanding Tile Spawning

The `spawn_tile` function has been provided for you in the starter code. Study how it works:

- It collects all empty positions in a vector of pairs.
- Uses `std::mt19937` with a fixed seed (42) for deterministic behavior during testing.
- Uses `std::uniform_int_distribution` from `<random>` to select a random empty cell.
- Assigns a 2 (90% of the time) or 4 (10% of the time).
- The function uses structured bindings (`auto [r, c]`) to unpack the pair.

Note: The fixed seed ensures that the random number generator produces the same sequence of "random" numbers each time, which is essential for automated testing. In a real game, you would use a time-based or random device seed.

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 valid move, push a copy of the board to the history stack.
- On undo (`u`):
  - Check if the history stack is not empty
  - Set the board to `history.top()`
  - Pop the top element from the stack with `history.pop()`
  - Print the restored board
  - Write the board state to CSV with stage "undo"
  - Use `continue` to skip the rest of the loop

Step 8: 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.

Step 9: Creating a Makefile

We haven't covered Makefiles in class yet, but they are essential build tools in C++ development. You'll use Claude Code to generate one for this project.

Note: Claude Code ($20/month) is a valuable tool for modern software development and is increasingly important for job readiness. If you prefer not to subscribe, partner with a classmate who has it to obtain the Makefile.

Using Claude Code to Generate a Makefile

1. Ask Claude Code what tools you need to install to use `make` on your system

   - **macOS**: You likely already have it via Xcode Command Line Tools
   - **Linux/Ubuntu**: You may need to install `build-essential`
   - **WSL2 (Windows Subsystem for Linux)**: Same as Linux - install `build-essential`

2. Use this example prompt with Claude Code:

   > "Create a Makefile for my C++ 2048 game project. I have solution.cpp that needs to compile to an executable called 'solution'. Use g++ with C++17 standard and include debugging symbols. Also add a 'clean' target to remove compiled files and a 'test' target that runs pytest test_game.py"

3. Claude Code will generate a complete Makefile with:

   - Compilation rules for your solution.cpp
   - Proper compiler flags (-std=c++17, -g for debugging)

- A clean target for removing build artifacts
- A test target for running the pytest tests

Testing Your Code

Once you have your Makefile, run the tests with:

```
make clean
make
make test
```

Your code is correct when all tests pass. Push your final version to GitHub as **solution.cpp** and **Makefile** under the repo **cse3150_week_5_lab**

Optional Challenge: GUI Version with SDL2

This section is optional and not graded.

Want to turn your text-based 2048 game into a graphical application? Use Claude Code to help you create a GUI version using the SDL2 library!

Example Claude Code Prompt

> "Help me convert my C++ 2048 game from solution.cpp into a graphical GUI application using SDL2. Keep the game logic but add: - A window displaying the 4x4 grid - Colored tiles with numbers - Smooth sliding animations for tile movements - Score display - Game over detection Also tell me how to install SDL2 on my system [specify: macOS/Linux/WSL2]"

Claude Code will:

- Guide you through installing SDL2 on your specific platform
- Help you restructure your code to separate game logic from rendering
- Create the graphical interface with proper event handling
- Add visual enhancements like animations and colors

This is excellent practice for:

- Learning GUI programming in C++
- Understanding event-driven programming
- Working with external libraries
- Using AI tools for real-world development tasks