PS3b: Sokoban

In Part B, we are adding the gameplay mechanics to Sokoban.

- The player can use the WASD and arrow keys to move.
- The `Sokoban` class has a `movePlayer` method that handles the player movement. It must take a single `Direction` enumeration constant as a parameter, chosen from the set `Up, Left, Down, Right`.
- The `Sokoban` class has a `const` `isWon()` method that determines if the player has won the game.
- Walls block movement.
- Boxes can be pushed by the player if there is an open space to be moved into, otherwise they block movement. The player can push only one box at a time.
- The player is trying to push a box into each storage area. When the player pushes a box into each storage area, the game announces that they win.
- Pressing 'R' will reset the level to its original state.

1 Details

For Part A of this assignment, you created a program that loads and displays a Sokoban level. In Part B, you will add the gameplay elements. Name your executable `Sokoban`, so you would run it with e.g.

```
./Sokoban level1.lvl
```

The input file format is the same as for part A.

1.1 Movement

When the player presses the WASD or arrow keys, they move in one of the four cardinal directions (Up, Left, Down, or Right, respectively), assuming that the space is open. Wall spaces block movement. For example, in Figure 1, the player can move down but cannot move left.

Boxes are a little more complicated. If the space on the other side of the box contains a wall or another box, then the box blocks movement like a wall. Otherwise, the box is pushed into the open space and the player moves into the space that the box vacates. For example, in Figure 1, the if the player presses 'D', then both the player and the rightmost box move to the right. However, the player cannot move up since the player cannot push two boxes at the same time.

Since players and boxes move only on whole number of grid spaces, you will want to use integer values rather than floating-point values to store their positions. The boundary of the play area is impassible (just like a wall).

1.2 Try again

Since they can only push boxes and not pull them, it is possible for the player to end up in an unwinable situation. For example, they may have accidentally pushed a box that they need into a corner where it can no longer be moved. The player can press the 'R' key to

Figure 1: The player can move down or push to the right

reset the level back to its original state. When this happens, the player and all boxes are returned to their initial positions when the level was first loaded.

You will likely want to implement this early since it will make play-testing easier.

1.3 Victory

When either all boxes have been placed on storage area spaces (if the level has more storage areas than boxes) or all storage areas contain boxes (if the level has more boxes than storage areas), then the player wins (the two conditions are equivalent if the number of boxes and storage areas are the same). The game should stop disable player movement and display a congratulatory message or image (like "You win!"). The player can still press 'R' to reset the level.

2 Unit Tests

You must write unit tests which will be used to test your Sokoban model (as well as against several implementations provided by the instructor). Since your unit tests will run automatically without user input, you should not be creating a graphical window in your tests (although you could use a `RenderTexture` to draw into if you wanted to test appearance). You will likely want to instantiate a `Sokoban` object in a known state using a level file (either one of the provided ones or one you created) and then issue a series of commands and check that the new state is what is expected.

You should make the tests be one of the first things you do for this part. You can earn points by correctly identifying which instructor implementations are correct and which are incorrect and having the tests will help you identify whether your own implementation is working or not.

For convenience, the reference implementation provides an alternate one-arg constructor that takes a filename, a class constant `TILE_SIZE`, and getters (but not setters) `pixelWidth()` and `pixelHeight()`. All other public members are as specified in the assignment (or the previous part). Neither addition is different between the correct and incorrect instructor implementations, so you are not required to use them in your tests. Remember that the reference implementation does not have any additional helper functions that you may add (which should generally be private).

The reference implementation also provides `undo()` and `redo()` methods for the extra credit.

3 Extra Credit

You can earn extra credit by changing the player image to face the direction of their most recent move and for playing a victory fanfare when the player wins. Additionally, you can add a new feature, such as multiple box colors with different storage areas, locks and keys, or teleporters and a new level demonstrating the feature. Finally, you can incorporate an

undo system to roll back the player's previous moves, allowing them to undo their mistakes. Make sure that your undos handle boxes correctly.

If you do any of the extra credit work, make sure to describe exactly what you did in `Readme-ps4.md`. Be sure to include any resources files in your submission and give the source in your `Readme-ps4.md`.

4 What to turn in

Your makefile should build a program named `Sokoban` and a static library `Sokoban.a` as well as a testing suite `test`.

Submit a zip archive to Gradescope containing:

- Your main file `main.cpp` and test file `test.cpp`
- Your Sokoban (`Sokoban.cpp`, `Sokoban.hpp`) class.
- Your unit tests (`test.cpp`)
- The makefile for your project. The makefile should have targets `all`, `Sokoban`, `Sokoban.a`, `test`, `lint` and `clean`. Make sure that all dependencies are correct.
- Your `Readme-ps4.md` that includes
  1. Your name
  2. Statement of functionality of your program (e.g. fully works, partial functionality, extra credit)
  3. Key features or algorithms used
  4. Any other notes
- Any other source files that you created.
- Any images or other resources not provided (such as for the extra credit)
- A screenshot of program output

Make sure that all of your files are in a directory named `ps4b` before archiving it and that there are no `.o` or other compiled files in it.

5 Grading rubric

Feature	Points	Comment
Unit Tests	6	
	1	Basic movement
	2	Box interactions
	1	Border interatctions
	1	Victory conditions
	1	File Parsing
Autograder	32	Full & Correct Implementation
	2	Contains all code files and builds
	4	Basic movement
	2	Movement with hit detection
	9	Detects win condition with simple paths
	4	Block collision detection
	4	Boundary detection
	4	Handles pass overs correctly
	2	Works for unbalanced numbers of boxes and storage
	1	Self-tests
Drawing	8	
	4	Redraws after movement
	2	Can reset the game
	2	Displays victory notice
Functions	4	
	2	Uses a lambda expression as a parameter
	2	Calls a function from the `algorithm` library
Readme	5	Complete
	2.5	Describes the algorithms or data structures used.
	2.5	Describes how the features were implemented.
Extra Credit	11	
	+2	Player changes direction while moving
	+2	Plays victory sound
	+3	Adds a new feature and describes it
	+4	Can undo moves (+2 for only one move)
Penalties		
	-5	Memory leaks or issues
	-5	Linting problems
	-3	Non-private fields
	-25	Unsourced images or other resources
	-10%	Each day late
Total	55	