Assignments > Reversi Board Game

Reversi Board Game

In this programming assignment, you are expected to implement a command-line utility for two human players to take turns and play a Reversi board game. The material you have learned in the first two lectures, including the guessing game example and basic programming concepts, will be useful when completing this assignment. This assignment is due at **11:59 p.m.** on **Monday, September 29, 2025**.

Game Rules

Here is a brief description of the Reversi board game. Reversi is played on a 8×8 board (like a chess board or a checkers board) with two players. The game uses pieces that are white on one side, and black on the other side, and as it progresses, these pieces can be *flipped* over to change their colour. One player plays white, the other player plays black, and the players take turns to place their pieces on the board. The game starts with two white and two black pieces placed in advance at the centre. Observe that rows and columns are labelled with letters, ranging from `a` to `h`. If a position is empty, it is represented by a dot (`.`). The initial board configuration in this game is as follows:

```
  abcdefgh
a ........
b ........
c ........
d ...WB...
e ...BW...
f ........
g ........
h ........
```

A *turn* in the game consists of a player placing a piece of his/her own colour on a candidate empty board position, subject to the following two rules:

- There must be a continuous straight line of piece(s) of the opponent's colour in at least one of the eight directions from the candidate empty position (north, south, east, west, and diagonals).
- In the position immediately following the continuous straight line mentioned in the rule above, a piece of the player's own colour must already be placed.

In other words, a player can place a piece on an empty square if the square is adjacent to the opponent's piece and the opponent's pieces are sandwiched between the player's own piece and the piece that the player just placed. After placing a piece at a position that meets the criteria above, all of the lines of the opponent's pieces that meet the criteria above are then flipped to the player's colour.

For example, starting from the initial board configuration above, if the black player plays at position `cd`, the board configuration will become the following after the move:

```
  abcdefgh
a ........
b ........
c ...B....
d ...BB...
e ...BW...
f ........
g ........
h ........
```

The turns alternate between the players, unless one player has no available move, in which case the only player with an available move is allowed to continue to make moves until a move becomes available for the opponent. At this point, the opponent is allowed to take a turn and the alternating turns between the players resumes. The game ends when no more moves can be made by both players, and the player with the most pieces on the board wins the game.

Playing the Game

In this assignment, you will write a command-line utility in Rust that allows two human players to take turns and play the game to completion, assuming that the player with the

black colour always plays first. This utility should start with printing the initial board configuration, and prompting the black player to enter a move, such as the (valid) move `cd`. All moves should be entered with a two-character combination, using lowercase letters to represent rows and columns. Here is a sample run:

```
  abcdefgh
a ........
b ........
c ........
d ...WB...
e ...BW...
f ........
g ........
h ........
Enter move for colour B (RowCol): cd
  abcdefgh
a ........
b ........
c ...B....
d ...BB...
e ...BW...
f ........
g ........
h ........
```

If a move is not valid, the utility should print `Invalid move. Try again.`, and then print the current board again for the player to try another move.

```
Enter move for colour B (RowCol): dd
Invalid move. Try again.
  abcdefgh
a ........
b ........
c ........
d ...WB...
e ...BW...
f ........
g ........
h ........
Enter move for colour B (RowCol):
```

Once the first move is out of the way, the turns proceed following the game rules, alternating between Black and White unless one of the players has no move to make, in which case your program should print a message `W player has no valid move.` (*i.e.*, for the case of the White player) and should prompt the opponent player for another move.

⚠️ When working in this assignment, print the exact messages in the handout rather than customizing them to your own liking. The automarker will look for exact matches to the expected output, and your work will **not** be manually graded. In particular, you should **not** print anything that has not been explicitly mentioned in this handout, as any additional printing will affect the automarker when grading, and we will not be able to give you credits if the automarker fails.

For example, after printing the message `W player has no valid move.` here, you should not print the board again before prompting the opponent player for another move.

After each turn, your utility must print the current board, and must detect whether the game is over. If your utility detects the game is over (i.e. a win or a draw), a message is printed and the program terminates. The specific messages to print are: `White wins by {x} points!`, `Black wins by {x} points!`, or `Draw!`, where `{x}` represents the number of additional pieces that the winner has on the board than the opponent.

Implementation Notes

Similar to the guessing game example covered in our lectures, use the `std::io` module to read input from the user. You may also find it necessary to *flush* the standard output after printing the prompt, but before reading input from the user. To do this, use:

```
io::stdout().flush().expect("Failed to flush stdout.");
```

Just like `gen_range()` needs to bring the `rand::Rng` trait into scope in the guessing game example, `flush()` needs to bring the `Write` trait into scope. We will cover traits in more detail in future lectures. You can do this by adding the following line to your Rust code:

```
use std::io::Write;
```

Another useful trick is to use the `as` keyword to convert a variable to a different type. For example, to convert an `isize`-typed variable `row` to an unsigned type `usize`, you can use:

```
row as usize
```

When you start to work on your assignment, make sure you use the command `cargo new` to create a new project, including a `Cargo.toml` file and a `src/main.rs` file, as well as an initial git repository. Take advantage of the git repository that `cargo new` creates for you, and push your local commits to GitHub. Once your project is managed by git, you can then start writing your code in the `src/main.rs` file. Of course, you can also create additional files in the `src` directory if you need to. You can compile and run your code using the `cargo run` command, as we have introduced in our lectures.

Testing

An initial set of five test cases has been released to you as a compressed archive, named `a1-public-tests.tar.gz`. Run the command `tar zxvf a1-public-tests.tar.gz` to uncompress the archive to a directory. You can use these test cases to test your implementation against the expected output, and revise minor output differences as necessary. To run the test, you can use the following command:

```
cargo run < input.txt > my_output.txt
```

Where `input.txt` is the name of the input test file in test cases provided to you. To compare the output of your program with the expected output, you can use the `diff` command:

```
diff my_output.txt output.txt
```

Notes on AI Tools

This assignment is designed for manual work without AI assistance, and you should **not** use AI tools at all for this assignment. While we recognize that the habit of using AI may be so reflective and strong that — like social media — you cannot think without AI assistance, you need to learn coding in the Rust programming language and this learning experience needs to start from somewhere. This assignment offers an excellent starting point for you to learn from the beginning, and using AI defeats the purpose and completely changes the learning experience. Heed our advice: do **not** use AI and you will enjoy such *"raw"* coding experience much more.

Submission

Submit your project by submitting a `.tar.gz` archive of it using Quercus, under *Assignment 1*. Use the following command to create your `.tar.gz` archive:

```
cd /path/to/parent/directory
tar zcvf 1234567890.tar.gz project_directory
```

```
reversi
    .git
    .gitignore
    Cargo.toml
    src
        main.rs
```

where `1234567890` is your student number, and `project_directory` is the name of the directory containing your project files, such as `reversi`. This is also the project name you initially provided to the `cargo new` command. Make sure that your archive contains all the files necessary to build and run your project, including the `Cargo.toml` file and the `src` directory as shown above. Also, make sure that your project builds and runs correctly after extracted from the archive, using the `cargo run` command.

⚠️ The deadline for this assignment is **Monday, September 29, 2025**, at **11:59pm** Eastern time, and late submissions will **not** be accepted. Do not include any other files and directories, beyond those shown above. For example, always run `cargo clean` to remove your `target/` directory — which contains build artifacts — before you submit your work.

Marking

Your project will be built and tested using the `cargo run` command against a set of 10 test cases, including 5 public test cases that are released to you, and 5 more hidden test cases that we use internally for testing your solutions. The marking will be based on the correctness of your implementation in these test cases, by comparing the output of your code to the correct output. An extra newline character at the end of your output will not affect the automarker.

Make sure that your code is well-commented, easy to read, and that you have followed the Rust naming conventions. You should also make sure that your code is free of warnings when compiled with the `cargo build` or the `cargo check` command.

Last updated on September 13, 2025