Advanced Programming | Final Project

Submission Date: September 15, 2025

Teaching Assistants in Charge: Amir Akbari , Sobhan Baharian

Project:

The final project for this course consists the implementation of a complete game in the C++ language. The game selected for this project is Othello (also known as Reversi), a two-player strategy game played on an 8x8 board. The goal for the players is to place their pieces so as to convert the maximum number of their opponent's pieces to their own color.
This project is designed step-by-step so that you can practice and solidify the important concepts covered throughout the semester in a practical structure.

The key concepts you will practice in this project are:

- Top-down design and modular structuring of the project
- Using classes and object-oriented principles in software design
- Implementing recursive algorithms for game logic
- Using templates and error handling

If you are not familiar with the game of Othello, it is recommended that you first learn its basic rules. You can use the following resources to learn how to play:

- https://www.youtube.com/watch?v=xDnYEOsjZnM
- https://www.wikihow.com/Play-Othello
- https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english

---

Requirements:

1. Designing Base Classes

You must design at least three separate classes with clearly defined responsibilities:

- Board: Responsible for maintaining the game board state, initializing it, and printing it.
- Player: Represents a player with attributes such as piece color (black or white) and turn.
- Game: Manages the overall game flow, including turn control, command processing, and communication between classes.

2. Printing the Board State

At the beginning of the game and after each move, the complete state of the board must be printed as text in the output. It is recommended to use `B` for black pieces, `W` for white pieces and `.` for empty cells.

3. Alternating Turns Between Players

The program must allow players to move in alternating turns. The first player plays with black pieces (B) and the second player with white pieces (W). After each valid move, the turn passes to the other player.

4. Processing Command-Line Commands

The program must receive and execute simple commands from the user. The required commands are as follows:

- new: Start a new game or initialize the board.
- place: Place a piece at a specified position (row x and column y) if that cell is empty.
- save: Save the current game state to a text file.
- load: Load a game state from a file.
- exit: Exit the game.

Invalid inputs (such as positions out of bounds or occupied cells) must be properly detected and reported to the user with an appropriate message.

5. Saving and Loading the Game State

Using text files, you must be able to save the current game state and later resume the game from the same point. The file format must contain sufficient information to reconstruct the board and the current player's turn.

6. Implementing Flipping Logic

After each valid move, all opponent pieces that are surrounded (in horizontal, vertical, or diagonal lines) between the new piece and the current player's pieces must flip to the current player's color.

7. Detecting and Applying Valid Moves

On each turn, the program must detect all valid moves for the player and only allow them to place a piece in those positions. If a player has no valid moves, their turn is automatically skipped, and the turn passes to the opponent. If neither player has any valid moves, the game should end, and the result should be announced.

8. Adding an Undo System

An undo feature must be added to the program. After each move, the current game state (including board layout, player turn, and scores) must be saved so that executing the undo command can revert to previous states. Therefore, the undo command, which is executed as follows, should be added to the list of user commands:

undo

- Executing this command should remove the last move and restore the board to its previous state.
- If there are no moves to revert to, the command should execute without error but have no effect on the game state.

9. Adding a Redo System (Bonus)

The redo functionality can be optionally added to the program as a bonus feature. After executing the undo command, the player can use the redo command to revert to the next move and restore the board state to what it was after that move. The redo command should be added to the list of user commands as follows:

`redo`

- Executing this command should reapply the previously undone move and update the board state accordingly.
- If there are no moves to redo, the command should execute without error but have no effect on the game state.

10. Displaying and Updating Player Scores

After each move, the program must calculate the number of black and white pieces on the board and display them in the output using the following format (an example of score display during gameplay):

`Score - B: 22 | W: 18`

This information should be displayed at the end of the board state, below the line indicating the current turn in the output.

11. Structuring the Project as Multi-file

you are required to separate the project code into distinct files based on responsibility. For example, a suggested structure for the project could be as follows:

- board.hpp / board.cpp: Responsible for the board state and related operations
- game.hpp / game.cpp: Core game logic and managing game flow
- player.hpp / player.cpp: Information related to players (if needed)
- main.cpp: Program entry point and user interaction

The use of include guard or #pragma once to prevent conflicts is mandatory.

12. Writing a Makefile for Project Compilation

You must write a Makefile that manages the modular compilation process of the project. The minimum expected capabilities of the Makefile include:

- Compiling all .cpp files into a final output (named othello)
- Having standard commands make and make clean
- Removing temporary files with the make clean command
- The entire compilation and execution process of the project should be done solely with the make command.

Inputs and Outputs:

This section explains how the user interacts with the program via the command line, as well as the format of the save files. Your program must be able to receive text commands from the user, perform the necessary operations, and display the game state in a clear, textual format at each stage.

1. User Commands

On each turn, the player must enter one of the following commands. The program should parse the command and execute the appropriate action. Each command is explained separately below**:**

`new`

Start a new game. This command initializes the board to its starting state (with the four initial pieces in the center) and sets the turn to the first player.

`place yx`

Place a piece at the specified coordinates. The value y is the column letter (from A to H) and x is the row number (from 1 to 8). If the cell is not empty or the coordinates are out of bounds, an error message should be displayed, and the user should be prompted to enter a new command.

`save <filename.oth>`

Save the current game state to a text file with the specified name. This file must contain the complete information about the board state and the current turn.

`load <filename.oth>`
Load a game state from a file with the specified name. If the file does not exist or its format is incorrect, an error message should be displayed.

`undo`
Revert to the previous move. If there is no previous move, the game state remains unchanged.

`redo`
Revert to the next move. If there is no next move, the game state remains unchanged.

`exit`
End the game execution and exit the program.

2. Text-based Board Output

After executing each command (`new`, `undo`, etc.), the complete current state of the board should be printed.

- Text display of the board after executing the `new` command (the starting turn always belongs to Black color):

```
  A B C D E F G H
1 . . . . . . . .
2 . . . . . . . .
3 . . . . . . . .
4 . . . W B . . .
5 . . . B W . . .
6 . . . . . . . .
7 . . . . . . . .
8 . . . . . . . .
```

- An example of the text-based board display during gameplay after executing a command (e.g., `new`, `place`, `undo` and `redo`):

```
  A B C D E F G H
1 . . B . . B . .
2 . . . B B B W B
3 . . . . B W W B
4 . W W W W W W B
5 . W W W W B W B
6 . . W W . W W B
7 . . . B B . W B
8 . . B . . . . .
Player Turn: W
Score - B: 16 | W: 19
```

If the game ends, the winner should be announced at the end of the output (and the player turn line should be removed).

For example:

```
  A B C D E F G H
1 B B B B B W B W
2 W B B W W W B W
3 B B B B B B B W
4 B B B B B B B W
5 W B B W B B B W
6 W W B W W B B W
7 W W B B W W B W
8 W W W W W W W .
Score - B: 34 | W: 29
Winner: Black
```

3. Handling Invalid Inputs

The program must be able to correctly detect invalid inputs and prevent incorrect execution. If invalid input is received, an appropriate error message should be displayed, and the turn should not change.

Cases that must be handled are as follows:

- Coordinates out of bounds (e.g., `place N2` or `place A9`)
- Coordinates entered in an incorrect format (e.g., `place 8 1` or `place D 5`)
- Attempting to place a piece in a cell that is already occupied
- Commands with incorrect syntax or unrecognized commands (e.g., `savegame .oth` or `move D4`)
- Loading a non-existent file or a file with an incorrect format

In all these cases, the program should report the error with a clear message, preserve the current turn, and prompt the user to enter new input.

4. Save File Format

The save file (with `.oth` format) must contain exactly the same textual output that is printed in the terminal after each turn (the board layout along with the line indicating the player's turn and the line displaying the players' scores). This file should be directly loadable by the program and should be saved in such a way that the game state can be fully reconstructed.

Submission Notes and Instructions:

- Your submission file must be a ZIP file containing all .hpp and .cpp files. Name the ZIP file using the following format:
  `<FirstName>_<LastName>_<StudentNumber>_Project.zip`

  For example:

  `Amirreza_Akbari_810899000_Project.zip`

  Please note that automated tests will execute your files based on this naming convention. Any discrepancy between your naming and the specified format will result in your files not being assessed and loss of marks.

- Your program must compile and run on a Linux operating system using a C++20 compiler (g++20 with standard C++20) and must execute within a reasonable time for the test inputs.

- Adhering to clean and organized coding principles is crucial in program design. Meaningful and consistent naming for variables and functions, logical division of code, writing short functions, avoiding code duplication, and proper indentation will enhance the readability of your code and ensure correct program functionality.

- Websites such as cppreference, cplusplus and Stack Overflow can be helpful resources for solving challenges in this assignment.

- Finally, please note that the grade for this assignment is not solely dependent on the correct execution of the program and the match between your outputs and the automated tests; your in-person presentation and mastery of the submitted code will also impact your final grade.