Ordle: A Wordle Solver

Authors:

- Axel Blomén
- Douglas Fjällrud

Design

The program is structured into three main components that work together to solve Wordle puzzles. The first part handles the loading and preparation of the word list. The *read_candidates* function reads all words from a file and filters out those that are not exactly five letters long. At the same time, the words are converted to lowercase, and duplicates are removed. This ensures that the program starts with a clean list of potential solutions.

The second part of the program focuses on filtering words based on user input. Three functors are used to handle different types of requirements:

- *wrong_fn*: Filters out words that contain letters the user has marked as "incorrect".
- *correct_fn*: Ensures that the remaining words have all the green letters at the correct positions.
- *misplaced_fn*: Checks that yellow letters are present in the word but not at the specified incorrect positions.

These functors are applied in the *do_filter* function, which filters the current list of candidates based on the user's criteria. After each step, the number of possible solutions decreases until only one candidate remains, or the user decides to exit.

The third component manages user interaction. The *prompt* function is used to gather input from the user, such as letters that do not appear in the word, green letters with their positions, and yellow letters with their incorrect positions. This input is then used to update the filtering process.

The program consists of three files:

- **main.cc**: Runs the program and uses logic from **ordle.cc**.
- **ordle.cc**: Implements the core logic.
- **ordle.h**: Contains the definitions for the functions and functors in *ordle.cc*.

Build Instructions

Common actions are defined in the provided Justfile and can be viewed with `just -l`

Step-by-Step Instructions

1. **Clean old files**
   To ensure a fresh build, clean up previous build artifacts:

   ```
   make clean
   ```

2. **Generate the words list**
   Create a `words.txt` file using the system dictionary:

   ```
   make words.txt
   ```

3. **Build the project**
   Compile the project in debug mode:

   ```
   make
   ```

4. **Run the program**
   Start the program and specify the `words.txt` file:

   ```
   ./main.elf words.txt
   ```

5. **Release Build**
   To build an optimized version of the program:

   ```
   make RELEASE=1
   ```

6. **Run the tests**

   ```
   make test
   ```

Available Commands

Task	Command
Debug Build	`make`
Release Build	`make RELEASE=1`
Clean Build Artifacts	`make clean`
Generate Words List	`make words.txt`
Lint Code	`make clang-tidy`
Static Analysis	`make cppcheck`
Format Code	`make format`
Watch and Rebuild	`just watch`

User Instructions

1. Start the program with the provided `words.txt` file:

   ```
   ./main.elf words.txt
   ```

2. Follow the program's instructions to input:
    - **Wrong letters**: Letters that do not appear in the solution.
    - **Correct letters (green)**: Letters that are correctly placed along with their positions. Input format: `letter position`.
    - **Misplaced letters (yellow)**: Letters that exist in the word but are incorrectly placed. Input format: `letter position`.

    Example input:

    ```
    Enter wrong letters:
    a e

    Enter correct letters (letter index)*:
    r 1 i 3

    Enter misplaced letters (letter index)*:
    o 2
    ```

3. The program will filter the possible candidates based on your input and display the remaining words after each iteration.
4. Repeat the process until:
    - Only one candidate remains, which is the solution.
    - You decide to exit the program.

Limitations and Drawbacks

- The word list depends on the system dictionary, which might be incomplete.
- The program assumes that the user inputs valid and correctly formatted data.
- There is no robust handling for incorrect input.

Comments

The program is straightforward to use and relies on clear filtering logic. The use of **C++17** and the standard library makes the code efficient and easy to read. A possible improvement could involve better validation of user input to avoid errors.