Wordle program development summary and detailed documentation

By Ericka Florio

Michaelmas term, 2022

Summary

As described in the README for this repository, this program is designed to play and solve the wordle. Wordle is an online game based on five-letter words, where a hidden word is chosen by the program, and the user must attempt to guess this word by entering words of their own. After each guess, the game tells the user which letters in their guess are correct, which letters are correct *and in the right position*, and which letters do not appear in the hidden word at all. The player must then deduce the hidden word from the information provided.

This program is written in C++ and has two main functionalities, which I call "modes": independent mode, where the user supplies the hidden word to the program, and the program attempts to guess this word using the fewest number of "rounds," or iterations; and interactive mode, where the program chooses a word at random, and the user must enter guesses until they can correctly guess the word. This program is designed to take only one external file, the WordleWords.txt file included in the docs/ folder, which contains all five-letter words accepted by the wordle game. The program is run using four main files: *wordle, modes, functions* and *subroutines.* See the README on the git repository for a description of these files and what they contain. Each file also contains a header with a summary of its contents.

In this document, I will outline the structure of this program by describing the steps that the program takes during a run in interactive mode. I will then discuss the development of this work and it's testing protocols, and end with information about how to run the program and report any bugs and errors found.

Program structure: A run in interactive mode

Both interactive mode and independent mode are constructed quite similarly. Independent mode simply consists in the computer providing its own starting word, comparing against the word provided by the user, and determining a new "best guess" using "best word" function. For that reason, I will describe here the overall working of interactive mode only. In this mode, the computer starts by receiving a guess word from the user, either their own starting word or the "best" starting word. The computer then chooses a hidden word from the list of accepted words, using the current clock time to seed the random number generator from the *cstdlib* package. The computer then allocates memory to an array used to store the previous guesses of the user.

> **Important**: in order to allocate only a reasonable amount of memory to the previous guess array, the maximum number of rounds that can be played is set in modes.cpp (currently at 100 rounds). If this number is exceeded, the program will exit and throw the appropriate error. In interactive mode this limit is set in the program files, but in independent mode this value is provided by the user.

If the user has guessed the hidden word on the first try, the program will congratulate them and exit. If not, the main routine is called -- here, the computer initialises four important variables:

- **position** – this is a string of length 5 which holds all the letters that have been guessed and are in the correct position, and places zeroes in those locations which have not been correctly guessed. So, for example, if the user guesses "opens" and the hidden word is "brent" then the position variable for that round will be "00en0".
- **in_word** – this is a string of variable length which holds all letters that have been guessed which are in the hidden word but not in the right position. For instance, if the user guesses "oboes" and the hidden word is "brent" then the in_word variable for that round will be "be".
- **position_complement** – this variable refers to the mathematical definition of a complement as the set of elements not contained by a particular set, because it uses the position and in_word variables to determine which letters in the guess word have been eliminated as being in the hidden word. This variable also stores letters that have been ruled out by previous guesses. So, for instance, if the guess word is "crown" and the hidden word is "brent", then the position_complement will return "cow". Note that the complement is only printed to terminal every round if the debug flag is turned on (see the debugging section below for more details).
- **new_guesses** – this is an array of strings, which contains all the next possible guesses that the user can choose from. It is generated using the new_guess_array function in functions.cpp, which eliminates words from the possible list of guesses using the information in position, in_word and position_complement. This array then forms the basis from which the next best guesses are determined and printed to the terminal. Note that if a word is eliminated as a new guess, it is replaced in the new_guesses array by an empty string.

The program first determines the position, in_word and position_complement variables for a given guess. It then fills in the new_guesses array using the new_guess_array function, which keeps a word in the array if it contains letters in the in_word variable, and eliminates a guess from the array if:

1. it does not contain letters in the right position, according to position,
2. it was previously guessed, or
3. it contains a letter that has been eliminated by current or previous guesses.

The program then ranks the words in new_guesses, returns the best five, then asks the user to guess a new word, and the program loops all over again. The program will exit only if the user guesses the right word, or if the user exceeds the number of allowed rounds.

The "best word" function

The "best word" function, found in functions.cpp, is used throughout the program to rank words from an array of strings. This function is built to return the *nth* best word, that is, it can calculate the second, third, etc. best word in the given array. The function adds up the number of times each letter in the alphabet shows up in each position (1-5) for a given word, then divides this number (calculated for each position) by the total number of words. This gives a sort of "probability measure" for how likely a given letter is to appear in each position in the word. For instance, say the program determines that 1,000 of the accepted words have

the letter “b” in the second position, so the word looks like “0b000.” Then the score for that letter, in that position, would be

$$1{,}000 \div 12947 \cong 0.08.$$

The program then determines a given word's "score" by adding up the probability values for each letter in the word, for the position of that letter. The word with the highest score is said to be the "best" word of the array. Note, however, that this function ranks words based on how common their individual letters are, *not* how common the word itself is. I don't currently have the data to add this feature, but I suspect it would improve the accuracy of the ranking algorithm significantly – at least in *independent* mode, where the user is providing the hidden word. In addition, this algorithm values accuracy over diversity i.e., it does not select for words that have unique letters, which in *interactive* mode may give the user more information from which to make future guesses.

Development

I wrote this program by starting with the wordle file, which contains the overarching structure of the program, and created the other files as the need for more specialized functions arose. I first wrote the architecture which chooses the mode that the user wishes to use, then started off by writing the *independent* function. I first wrote this function explicitly in the modes file, without breaking apart the various subroutines which eliminate guesses based on information in *position, in_word,* etc. However, once I started writing the *interactive* function, I realized that I would need much of the same functionality that the *independent* function used. It was at this point that I broke apart the bulk of the calculations into the functions and subroutines files, using the *independent* and *interactive* functions to control the highest level of calculation. I assigned functions the subroutines files if they did not depend on any other function I had defined – in this way, I could simply import “subroutines” into “functions” and know that all functions would be able to access any other functions they needed to call upon.

Once I had finished writing *independent*, I tested it multiple times in the terminal before moving onto *interactive*. I discovered, however, that I would have been able to address many issues which arose later if I had started by writing *interactive,* since by its nature the independent function prints less information to the terminal while it is running, and so any bugs in how the computer is narrowing down its guesses are by and large hidden from the programmer. Once I began writing *interactive*, I realised there was a need to build up more functionality to help narrow down guesses according to factors such as whether a guess could be eliminated if it contains letters that have been eliminated. It was also during development of *interactive* that I realised the need to store previous guesses and the *position_complement* variable. I also found a need to re-write the logic controlling situations where either the guess word or the hidden word had doubled letters. Once I had fixed the bugs in these areas, I returned to *independent* and revamped this function to include all the functionality in *interactive,* debugging (with more terminal output this time) along the way. It was through this debugging process that I developed many of the techniques I used to design the testing protocols and inbuilt debugging tools, outlined in the next section.

Testing and debugging

Testing protocols

The testing protocol laid out in tools/tests.cpp represents a set of conditions that this program must pass before it can be pushed onto the public git repository. These tests are based upon the debugging technique I used to refine the various functions in subroutines.cpp, where I provided a hidden and guess word of my own and ran just the function under study, to see if it produced the expected output. The tests file thus examines most of the functions in the subroutines.cpp file, focusing on those which do the heart of the calculation of the program and skipping those which directly interact with external files that may not exist (such as *print_to_debug_file*) or which have simple tasks such as printing to the terminal (for example, *print_string_list*).

All the tests involve comparing the output of a function to the value that it must return for the program's logic to work. To do this, I give sample hidden and guess words to each function, where the words have been chosen to produce an expected outcome. Both empty returns (for instance, the combination of "boats" and "growl" for the *letters_in_position* function giving "00000") and partially filled returns (such as the combination in the same function of "bools" and "growl" giving "00o00") have been tested. When running the tests locally (which can be done by calling *make test* in the wordle directory) if a function passes its tests, a message will print to the screen showing this result. If a function fails a test, the program exits, and a message is printed to the screen giving the name of the function that failed and a number indicating which condition the function failed.

> **Important:** If you decide to alter the testing protocol, **please do so very carefully**. The protocol is currently constructed to test each function properly and altering these routines may produce errors where none exist.

Debugging tools

I have added many of the most common features I used to debug this program in as options which the developer may choose to utilise. To turn on all the debugging options, set the *debug* global flag in header.cpp to 1 (the integer). This will, for instance, print the complement for each round to the terminal, and allow the programmer to specify their own hidden word (which I found particularly useful while learning how to calculate *position*, *in_word*, etc.). Turning on the debug flag will also generate a text file at the end of the run, stored in the tools/ directory, which will contain all the information gathered by the program during every run. There also exists a separate flag which can control whether the main, interactive bulk of the program is run -- so for instance, if you would like to run a specific function but skip the intro messages, etc., then you can alter lines 18-32 in wordle.cpp to turn off the *run_main* flag (which is turned off automatically if *debug* is activated).

Final program

As described in the README on the git repository, this program runs using only the basic functionality provided by gcc and compiles with the basic g++ compiler. However, if the user encounters any issues, I have included a Dockerfile in the tools/ directory, which produces a basic Alpine Linux environment in which the code should run without any errors. Any bugs or errors can be reported via Github or to eaf49@cam.ac.uk.