Isso é CS50

Introdução do CS50 à Ciência da Computação

OpenCourseWare

Doar (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.clubhouse.com/@davidjmalan) (https://www.facebook.com/dmalan)
(https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://orcid.org/0000-0001-5338-2522)
(https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan)
(https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/)
(https://twitter.com/davidjmalan)

Wordle50

For this problem, you'll implement a program that behaves similarly to the popular Wordle (https://www.nytimes.com/games/wordle/index.html) daily word game.

```
$ ./wordle 5
This is WORDLE50
You have 6 tries to guess the 5-letter word I'm thinking of
Input a 5-letter word: crash
Guess 1: crash
Input a 5-letter word: scone
Guess 2: scone
Input a 5-letter word: since
Guess 3: since
You won!
```

Getting Started

Open VS Code (https://code.cs50.io/).

Start by clicking inside your terminal window, then execute `cd` by itself. You should find that its "prompt" resembles the below.

```
$
```

Click inside of that terminal window and then execute

```
wget https://cdn.cs50.net/2022/fall/psets/2/wordle.zip
```

followed by Enter in order to download a ZIP called `wordle.zip` in your codespace. Take care not to overlook the space between `wget` and the following URL, or any other character for that matter!

Now execute

```
unzip wordle.zip
```

to create a folder called `wordle`. You no longer need the ZIP file, so you can execute

```
rm wordle.zip
```

and respond with "y" followed by Enter at the prompt to remove the ZIP file you downloaded.

Now type

```
cd wordle
```

followed by Enter to move yourself into (i.e., open) that directory. Your prompt should now resemble the below.

```
wordle/ $
```

If all was successful, you should execute

```
ls
```

e veja um arquivo chamado `wordle.c`, bem como `5.txt`, `6.txt` e `7.txt`. `8.txt` A execução `code wordle.c` deve abrir o arquivo onde você digitará seu código para este conjunto de problemas. Se não, refaça seus passos e veja se consegue determinar onde errou! Se você tentar compilar o jogo agora, ele o fará sem erros, mas ao tentar executá-lo, você verá este erro:

```
Error opening file 0.txt.
```

É normal, porém, como você ainda não implementou parte do código, precisamos fazer com que essa mensagem de erro desapareça!

Fundo

As probabilidades são, se você é um usuário do Facebook, pelo menos um de seus amigos postou algo parecido com isto, especialmente no início de 2022, quando estava na moda:

Se sim, seu amigo jogou Wordle e está compartilhando os resultados desse dia! A cada dia, uma nova "palavra secreta" é escolhida (a mesma para todos) e o objetivo é adivinhar qual é a palavra secreta em seis tentativas. Felizmente, como existem mais de seis palavras de cinco letras no idioma inglês, você pode obter algumas pistas ao longo do caminho, e a imagem acima mostra a progressão de seu amigo por meio de suas suposições, usando essas pistas para tentar se concentrar no palavra correta. Usando um esquema semelhante ao do jogo Mastermind (https://en.wikipedia.org/wiki/Mastermind_(board_game)) , se depois de adivinhar aquela letra ficar verde, significa que não só aquela letra está na palavra secreta daquele dia, mas também está na posição correta. Se ficar amarelo, significa que a letra adivinhada aparece *em algum lugar*na palavra, mas não naquele ponto. As letras que ficam cinzas não estão na palavra e podem ser omitidas de suposições futuras.

Vamos terminar de escrever um programa chamado `wordle` que nos permite recriar este jogo e jogá-lo em nosso terminal. Faremos algumas pequenas alterações no jogo (por exemplo, a maneira como ele lida com uma letra que aparece duas vezes em uma palavra não é a mesma que o jogo real, mas, para simplificar, vamos errar no lado da facilidade de compreensão em vez de uma interpretação perfeitamente fiel), e usaremos texto vermelho em vez de cinza para indicar letras que não estão na palavra. No momento em que o usuário executa o programa, ele deve decidir, fornecendo um argumento de linha de comando, qual é o comprimento da palavra que deseja adivinhar, entre 5 e 8 letras.

Aqui estão alguns exemplos de como o programa deve funcionar. Por exemplo, se o usuário omitir totalmente um argumento de linha de comando:

```
$ ./wordle
Usage: ./wordle wordsize
```

Se o usuário fornecer um argumento de linha de comando, mas não estiver no intervalo correto:

```
$ ./wordle 4
Error: wordsize must be either 5, 6, 7, or 8
```

Veja como o programa pode funcionar se o usuário fornecer uma chave de `5`:

```
$ ./wordle 5
This is WORDLE50
You have 6 tries to guess the 5-letter word I'm thinking of
Input a 5-letter word:
```

Nesse ponto, o usuário deve digitar uma palavra de 5 letras. Obviamente, o usuário pode ser teimoso e devemos garantir que ele siga as regras:

```
$ ./wordle 5
This is WORDLE50
You have 6 tries to guess the 5-letter word I'm thinking of
Input a 5-letter word: wordle
Input a 5-letter word: computer
Input a 5-letter word: okay
```

```
Input a 5-letter word: games
Guess 1: games
Input a 5-letter word:
```

Notice that we didn't even count any of those invalid attempts as guesses. But as soon as they made a legitimate attempt, we counted it as a guess and reported on the status of the word. Looks like the user has a few clues now; they know the word contains an `a` and an `e` somewhere, but not in the exact spots they appear in the word `games`. And they know that `g`, `m`, and `s` don't appear in the word at all, so future guesses can omit them. Perhaps they might try, say, `heart` next! ❤️

Specification

Design and implement a program, `wordle`, that completes the implementation of our Wordle50 clone of the game. You'll notice that some large pieces of this program have already been written for you—you are not allowed to modify any of those parts of the program. Instead, your work should be constrained to the seven `TODO`s we've left behind for you to fill in. Each one of those parts solves a specific problem, and we recommend you tackle them in order from 1 to 7. Each numbered `TODO` corresponds to the same item in the below list.

1. In the first `TODO`, you should ensure the program accepts a single command-line argument. Let's call it k for the sake of discussion. If the program was not run with a single command-line argument, you should print the error message as we demonstrate above and return `1`, ending the program.
2. In the second `TODO`, you should make sure that k is one of the acceptable values (5, 6, 7, or 8), and store that value in `wordsize`; we'll need to make use of that later. If the value of k is not one of those four values exactly, you should print the error message as we demonstrate above and return `1`, ending the program.

After that, the staff has already written some code that will go through and open the word list for the length of word the user wants to guess and randomly selects one from the 1000 options available. Don't worry about necessarily understanding all of this code, it's not important for purposes of this assignment. We'll see something similar though in a later assignment, and it will make a lot more sense then! This is a good place to stop and test, before proceeding to the next `TODO`, that your code behaves as expected. It's always easier to debug programs if you do so methodically!

3. For the third `TODO`, you should help defend against stubborn users by making sure their guess is the correct length. For that, we'll turn our attention to the function `get_guess`, which you'll need to implement in full. A user should be prompted (as via `get_string`) to type in a k-letter word (remember, that value is passed in as a parameter to `get_guess`) and if they supply a guess of the wrong length, they should be re-prompted (much like in Mario) until they provide exactly the value you expect from them. Right now, the distribution code doesn't do that, so you'll have to make that fix! Note that unlike the real Wordle, we actually don't check that the user's guess is a real word, so in that sense the game is perhaps a little bit easier. All guesses in this game should be in **lowercase** characters, and it is acceptable for you to assume that the user will not be so stubborn as to provide anything other than lowercase characters when making a guess. Once a legitimate guess has been obtained, it can be `return`ed.
4. Next, for the fourth `TODO`, we need to keep track of a user's "score" in the game. We do this both on a per-letter basis—by assigning a score of 2 (which we `#define`d as `EXACT`) to a letter in the correct place, 1 (which we `#define`d as `CLOSE`) to a letter that's in the word but in the wrong place, or 0 (which we `#define`d as `WRONG`)—and a per-word basis, to help us detect when we've potentially

triggered the end of the game by winning. We'll use the individual letter scores when we color-code the printing. In order to store those scores, we need an array, which we've called `status`. At the start of the game, with no guesses having taken place, it should contain all 0s.

This is another good place to stop and test your code, particularly as it pertains to item 3, above! You'll notice that at this point, when you finally enter a legitimate guess (that is to say, one that's the correct length), your program will likely look something like the below:

```
Input a 5-letter word: computer
Input a 5-letter word: games
Guess 1:
Input a 5-letter word:
```

That's normal, though! Implementing `print_word` is `TODO` number 6, so we should not expect the program to do any processing of that guess at this time. Of course, you can always add additional `printf` calls (just make sure to remove them before you submit) as part of your debugging strategy!

5. The fifth `TODO` is definitely the largest and probably most challenging. Inside of the `check_word` function, it's up to you to compare each of the letters of the `guess` with each of the letters of the `choice` (which, recall, is the "secret word" for this game), and assign scores. If the letters match, award `EXACT` (2) points and `break` out of the loop—there's no need to continue looping if you already determined the letter is in the right spot. Technically, if that letter appears in the word twice, this could result in a bit of a bug, but fixing that bug overcomplicates this problem a bit more than we want to now, so we're going to accept that as a feature of our version! If you find that the letter is in the word but not in the right spot, award `CLOSE` (1) points but don't `break`! After all, that letter might later show up in the right spot in the `choice` word, and if we `break` too soon, the user would never know it! You don't actually need to explicitly set `WRONG` (0) points here, since you handled that early in Step 4. Ultimately though, you should also be summing up the total score of the word when you know it, because that's what this function is supposed to ultimately return. Again, don't be afraid to use `debug50` and/or `printf`s as necessary in order to help you figure out what the values of different variables are at this point – until you implement `print_word`, below, the program won't be offering you much in the way of a visual checkpoint!
6. For the sixth `TODO` you will complete the implementation of `print_word`. That function should look through the values you populated the `status` array with and print out, character by character, each letter of the `guess` with the correct color code. You may have noticed some (scary-looking!) `#define`s at the top of the file wherein we provide a simpler way of representing what's called an ANSI color code (https://en.wikipedia.org/wiki/ANSI_escape_code#Colors), which is basically a command to change the font color of the terminal. You don't need to worry about how to implement those four values (`GREEN`, `YELLOW`, `RED`, and `RESET`, the latter of which simply returns to the terminal's default font) or exactly what they mean; instead, you can just use them (the power of abstraction!). Note as well that we provide an example in the distribution code up where we print some green text and then reset the color, as part of the game's introduction. Accordingly, you should feel free to use the below line of code for inspiration as to how you might try to toggle colors:

```
printf(GREEN"This is WORDLE50"RESET"\n");
```

Of course, unlike our example, you probably don't want to print a newline after each character of the word (instead, you just want one newline at the end, also resetting the font color!), lest it end up looking like the below:

```
Input a 5-letter word: games
Guess 1: g
a
m
e
s
Input a 5-letter word:
```

7. Finally, the seventh `TODO` is just a bit of tidying up before the program terminates. Whether the main `for` loop has ended normally, by the user running out of guesses, or because we broke out of it by getting the word exactly right, it's time to report to the user on the game's outcome. If the user did win the game, a simple `You won!` suffices to print here. Otherwise, you should print a message telling the user what the target word was, so they know the game was being honest with them (and so that you have a means to debug if you look back and realize your code was providing improper clues along the way!)

How to Test Your Code

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2023/x/wordle
```

Execute o abaixo para avaliar o estilo do seu código usando `style50`.

```
style50 wordle.c
```

Como enviar

Em seu terminal, execute o abaixo para enviar seu trabalho.

```
submit50 cs50/problems/2023/x/wordle
```