MP4

MP 4 - Wheel of Fortune

Your task this week is to implement some of the logic and process the user inputs for a modified version of the Wheel of Fortune which uses the names of your favorite characters.

To play the game, the user first enters a "seed" value, which is an integer. This seed value initializes a pseudo-random number generator. Using the pseudo-random number generator, 4 strings will be chosen from the string pool given below. You can assume the length of the string < 10.

The player then has 10 guesses to get the correct sequence of the strings. The game will terminate if the player guesses all 4 strings correctly or all 10 guesses are exhausted. Guesses are typed into the terminal. After each guess, the player is given feedback on the number of *perfect matches* and *misplaced matches*, which is printed to the screen. The number of guesses remaining and the score of the current guess are also displayed in the feedback.

The player will get 1000 points for each perfect guess and 100 points for each imperfect guess. The game will keep track of the current best score and will display it at the end.

The guess strings that appear in the same place in the solutions are called *perfect matches*. The strings that appear somewhere in the solution code but in a different place in the solution are called *misplaced matches*.

The objective for this week is for you to gain some experience with basic I/O, to implement code using library functions, practice using pointers, and to solve a problem that requires moderately sophisticated reasoning and control logic. You should use arrays and for loops to solve the problem, instead of hard coding with if statements!

Terms the guessing game (pool):

> "Vader", "Padme", "R2-D2", "C-3PO", "Jabba", "Dooku", "Lando", "Snoke"

Here's an example of the output of the game when the solution is {"Vader", "Padme", "Jabba", "Snoke"}

NOTED: solution for 12321 is not "Vader", "Padme", "Jabba", "Snoke". It is an example.

> Please enter a seed: 12321
>
> Valid term to guess:

```
    Vader Padme R2-D2 C-3PO Jabba Dooku Lando Snoke

Guess 1

Enter your guess: Vader R2-D2 Dooku Jabba

With guess 1, you got 1 perfect matches and 1 misplaced matches.

Your score is 1100 and current max score is 1100.

Guess 2

Enter your guess: Vader Jabba Lando Snoke

With guess 2, you got 2 perfect matches and 1 misplaced matches.

Your score is 2100 and current max score is 2100.

Guess 3

Enter your guess: Vader Padme Jabba Snoke

With guess 3, you got 4 perfect matches and 0 misplaced matches.

Your score is 4000 and current max score is 4000.

You guessed correctly in 3 guesses.

The solution was Vader Padme Jabba Snoke.
```

The list of character names can be found in prog4.c.

Examples of valid strings are: "Vader", "R2-D2", "Dooku". Examples of invalid strings are: "a string", "abc", "Dijk", "yoyoyo".

NOTED: the score here is not the score for MP. It is a score for this game. Even if you guess more than 10 times, you will not lose any points for this MP as long as both your program and return value for each function is correct.

The Pieces

This week you will write all your code in **prog4.c**.

Let's discuss each of these files in a little more detail:

prog4.h - This header file provides function declarations and descriptions of the functions that you must write for this assignment.

prog4.c - The source file for your code. Function headers for all functions are provided to help you get started. The functions are detailed below and will be reviewed in lab.

main.c - This file uses `#include` to pull in prog4.h, and provides the logic to execute the game.

test.c - This file uses `#include` to pull in prog4.h, and runs a number of unit tests on each function you will write. See the Testing section.

In lab, we will discuss pseudo-random number generation and the idea of error-checking user input.

Some Details

Global variables

You will note that there are some static variables defined near the top of prog4.c which are global variables.

These are variables that are allocated outside of any specific functions in the run-time stack. If you check prog4.h you will also notice the same declaration with keyword "extern" in front. The extern definition in the header file tells the compiler that these variables are global and can be also accessed by any functions outside prog4.h. Therefore, these static variables can be called in any function in prog4.c and main.c.

In your code, use the global array solutions generated by `start_game` (see below). The variable `guess_number` should be used to track which guess the user is on (starting at 1 and possibly going to 10). The variable `max_score` should be used to track the current max score.

Pseudo-random Number Generation

For this MP, we need to generate a randomized array of strings for this game to keep the game interesting. We don't have the ability to generate truly random numbers in C. Instead, we generate something called *pseudo-random numbers*, which are numbers that look random but are in fact calculated using a mathematical function.

Generating *pseudo-random numbers* requires an initial value, which we call the "seed".

Given this seed, the number generator will create a sequence of numbers from **srand** and **rand** function.

Please check the ref for example:

srand function: https://en.cppreference.com/w/c/numeric/random/srand (https://en.cppreference.com/w/c/numeric/random/srand)

rand function: https://en.cppreference.com/w/c/numeric/random/rand (https://en.cppreference.com/w/c/numeric/random/rand)

This is important, as it allows us to test our code by setting the seed value to be the same each time.

The srand and rand are used to produce sequences of pseudo-random numbers. The function definitions for these are

`void srand (unsigned int seed);`

`int rand();`

The function *srand* accepts an integer as an argument and sets the seed. It returns nothing and generates no random numbers.

The function *rand* generates a random integer between 0 and the value `RAND_MAX` and returns it.

Example usage would be:

```
srand(12321);
int x = rand();
int y = rand();
```

After execution, the value of x should be 162313199 and the value of y 1042925187.

Random number generation is operating system dependent! Be sure to do your testing on an EWS machine, though you may develop your code on another machine.

Processing User Input

The user will be typing characters into the terminal, for instance, "Vader R2-D2 Jabba Lando". The code is already given in main.c to process this input and store them as a *string*.

A string is an array of characters typed in by the user. You will use the function `sscanf` (http://www.cplusplus.com/reference/cstdio/sscanf/ (http://www.cplusplus.com/reference/cstdio/sscanf/)) to turn the characters into integers and recognize if there is an error.

The behavior of `sscanf` is almost identical to `scanf`, but with a string as an input. Some example code for processing the inputs will be discussed in lab and is provided in the comments of the `set_seed` and `make_guess` functions.

What You Need to Do

You will implement these functions:

1. `set_seed`
2. `start_game`
3. `make_guess`

You will then need to test your code to make sure it is working. You should read the descriptions of the functions in the comments of the source file before you begin coding to get a feel for the project.

`int set_seed (const char seed_str[]);`

INPUT: String `seed_str` (an array of ASCII characters, the qualifier const indicates that it cannot be changed). The user has typed characters into the terminal, which are stored by main.c into `seed_str` before calling `set_seed`

RETURN: Returns 1 if the user entered exactly one integer in the string, Returns 0 if the user enters no integers, more than one integer, or anything other than an integer.

1. If the user enters exactly one integer, set_seed sets the this integer as the seed value as the argument for Return 1.
2. If it is not a valid string (a 0 will be returned), ***srand*** **is not called** and an **error message is printed.**

srand should be called exactly once!

DETAILS:

The function receives a string (an array of characters) as its input. This string contains characters typed in by the user. You will check to see if this string contains one integer number. If it does, this number will be used as the seed for pseudo-random number generation. The "const " qualifier means that the routine is not allowed to change the contents of the string. You need to check if the string contains only one integer number, which we will use to seed a random number generator (use the function *srand* to set the seed).

The return value from the function `set_seed` indicates whether the input string did, in fact, correspond to a number. When the string represents a single integer number (and only a number), the `set_seed` should call `srand` (using the integer as an argument), then return 1. Otherwise, the function should not call srand, print **"set_seed: invalid seed\n"**, and return 0.

You will find the function `sscanf` helpful here (**http://www.cplusplus.com/reference/cstdio/**

sscanf/ (http://www.cplusplus.com/reference/cstdio/sscanf/)) to check the string for a number and for any formatting errors. Check the comments for the function in prog4.c, an example function call to `sscanf` is provided for you (`sscanf (seed_str, "%d%1s", &seed, post)`). This call reads one integer (and stores it in seed) and one string (and stores it in the post) from the string `seed_str`. If the user input is correct, one integer is read into seed and nothing is read into the post.

`void start_game ();`

INPUT: None.

RETURN: None.

SIDE EFFECTS:

1. Generate a solution of 4 random terms. The static strings at the 4 char pointers (`extern char solutions[4][10]`) should be filled with the random strings chosen from the pool array.
2. Init `max_score = -1`, `guess_number = 1`.

DETAILS:

The `start_game` function selects the solution strings at random from the pool array.

You will generate some random integers and use the random integer as an index into the pool array to select the random strings.

To ensure consistency between your program's output and ours, you **must use the algorithm shown below** for generating the solution code.

- **Step 1:** Starting with the first value in the solution code sequence, generate a random integer in the range 0 to 7 as index using a single call to `rand()` and a single modulus (%) operator
- **Step 2:** Use the random value you got as an index and copy the string from **"pool"** to **"solutions"**
- **Step 3:** Repeat steps 1-2 for the all 4 terms as our solutions (in order)
- **Step 4:** Set the static variable guess_number to 1
- **Step 5**: Set the static variable max_score to -1

Be sure not to call `srand` outside of the `set_seed` function and **not to call rand outside of the `start_game` function**. Calling either of these disrupts the sequence of random numbers and will cause your output to differ from ours.

For c strings, you can find some useful functions from here: (remember to include string.h in your code) http://www.cplusplus.com/reference/cstring/ (http://www.cplusplus.com/reference/cstring/)

`int make_guess (const char guess_str[]);`

RETURN: Returns 1 if the user entered a valid guess (exactly 4 valid strings, separated by spaces), Return 2 if guess is valid and the game should end (all 4 perfect guesses), Returns 0 if the guess is invalid (user enter more than 4 terms, or anything not in the pool).

SIDE EFFECTS:

1. If the guess is valid, the number of ***perfect*** and ***misplaced*** matches are calculated and printed, and the `guess_number` is incremented.
2. If the guess is invalid, it prints an error message and does not increment the `guess_number`.
3. Adjust `max_score` if needed.

DETAILS:

This function needs to

1. compare a player's guess with the solution code.
2. compute the number of *perfect* and *misplaced* matches.
3. check if the guess is valid (by calling helper function `IsValid`) or not.
4. be called by main.c every time the user types a guess.
5. calculate the score and substitute `max_score` if the current score is larger than the `max_score`.

The inputs to this routine is a string `guess_str` (containing the player's typed input). Your routine must validate the string in the same way that we did for `set_seed`.

An example call to `sscanf` is included in the comments, using a format string of "%s%s%s%s%1s" to read 4 strings and possible extra strings (if there is extra garbage strings are entered).

A valid string contains exactly 4 strings (and no extra garbage at the end). You can check the validity of each string using the provided `IsValid` function.

If the string is invalid, your routine must print an error message, **"make_guess: invalid guess\n"**, then return 0. Don't worry about the values of the integer pointers in this case.

Examples of valid strings: "Vader R2-D2 Dooku Jabba" " Vader Lando Snoke Jabba" "Vader R2-D2 C-3PO Lando".

Examples of invalid strings: "1" "Vader R2-D2 Jabba " "Vader R2-D2 Jabba Snoke Snoke Lando Jabba test" "aefeVader R2- D2 Jabba Snoke "

Print to stdout in the format of **"With guess 1, you got 1 perfect matches and 0 misplaced**

matches.\nYour score is 1000 and current max score is 1000.\n", substituting the values you've computed. Do not adjust the word "matches" for the subject-verb agreement. After printing, increment the static variable for the `guess_number`.

To compute the number of perfect and misplaced matches, you can follow this algorithm:

```
for each element at position i :
    if (guess [i] == solution[i]) :
        mark solution[i] as matched
        mark guess[i] as matched
        increase perfect match
for each unmatched guess at the position i :
    for each unmatched solution at position j :
        if (guess[i] == solution[j]) :
            mark solution[j] as matched
            mark guess[i] as matched
            increase misplaced match
```

Let's consider some examples.

Assume that the solution code is "**Vader R2-D2 Dooku Lando**".

If the player guesses **`"Vader Padme R2-D2 Lando"`**:

Then, we will find "R2-D2" is misplaced. (mark `solution[1]` matched because R2-D2 == `solution[1]`)

As a result, there are 2 perfect matches and 1 misplaced match.

Perfect matches are always paired (marked as matched) before misplaced matches, so there is never any ambiguity.

If you are in doubt, use the test code that we have provided to check your answers.

Some examples-

Solution: "Vader Vader Dooku Dooku" Guess: "Vader Dooku Vader Dooku"; 2 perfect, 2 misplaced

Solution: "Vader Vader Dooku Dooku" Guess: "Dooku Dooku Padme Dooku"; 1 perfect, 1 misplaced

Solution: "Vader Vader Vader Vader" Guess: "Vader Vader Vader Vader"; 4 perfect, 0 misplaced

Solution: "Vader Vader Dooku Dooku" Guess: "Vader Vader Vader Dooku"; 3 perfect, 0 misplaced

Solution: "Vader Vader Dooku Dooku" Guess: "Padme Padme Lando Lando"; 0 perfect, 0 misplaced

Specifics

- Your code must be written in C and must be contained in the prog4.c file provided to you – we will not grade files with another name.
- You must **NOT** modify any other files. We will only grade your prog4.c file.
- You must implement `set_seed`, `start_game`, and `make_guess` correctly.
- Don't change the function definitions. You may write additional functions if you wish.
- Your function's return values and outputs must match the solution exactly for full credit (see the Building and Testing).
- Your code must be well-commented. You may use either C-style (`/* can span multiple lines */`) or C++ style (`// comment to the end of line`) comments, as you prefer. Follow the commenting style of the code examples provides in class, in the textbook, and on Canvas.

Building and Testing

We suggest that you begin by developing your code on the Linux machines in the lab (remote access is fine). Definitely do your final testing on the EWS machines, or the pseudo-random numbers may not match. All operations described here should be run in the MP4 folder.

You should test your program before handing in your solution. We have provided you a set of tests. You should also get in the habit of writing your own tests, and you may want to think about how you would modify test.c to further test your program. To compile the program, we compile main.c by typing

```
gcc -g -std=c99 -Wall -Werror main.c prog4.c -o mp4
```

Noted that if you get any warning message, please fix it before you submit your code!

If successful, the compiler produces an executable called `mp4`, which you can execute by typing "`./mp4`" (no quotes). You can also run the code with the gdb debugger (or a GUI interface that uses it, such as DDD) by typing:

```
gdb mp4
```

Here is a quick reference for gdb (gdb quick reference (https://wiki.illinois.edu/wiki/display/ece220/gdb%2Bquick%2Breference)).

Unit Tests

You can run the tests as parts, or *units* of your code. In this case, it makes sense to write some code to test each function independently before running the whole thing. Although we've provided

some tests for you (in test.c), you should start thinking about writing your own testing code. This is a critical testing and debugging skill for a programmer to develop.
You can run the tests as

```
gcc -g -std=c99 -Wall -Werror test.c prog4.c -o test4
```

./test4 0 (this is for `set_seed`)

./test4 1 (this is for `start_game`)

./test4 2 (this is for `make_guess`)

Testing the Whole Thing

Please use the following command if your terminal says permission denied.

```
chmod +x test4
```

```
chmod +x mp4
```

To get full points, you need to match all cases exactly, as specified below.

Intro- you got 5 points if you had an intro paragraph in prog4.c. If you put the intro somewhere else, we may have missed it and you may submit a regrade. If you're missing 5 points (for example 95/100), check that you have an intro paragraph.

Errors and warnings- If you had a very low score (say 10/100), your code likely had compilation errors or warnings. If you had compilation warnings, you may submit a regrade request but you will lose your five style points. Be very careful to submit code without warnings in the future.

Remember that these tests contain only a few test cases- they do not 100% guarantee that your code is working when you test the whole thing.

To add more test cases, you can alter the test.c files to test your program more thoroughly.

Grading Rubric

Functionality (90%)

- 20% - `set_seed` function works correctly
- 20% - `start_game` function works correctly
- 40% - `make_guess` function works correctly

- 10% - all outputs match exactly for hidden cases

Style (5%)

- 5% - code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Comments, clarity, and write-up (5%)

- 5% - introductory paragraph explaining what you did (even if it's just the required work)

Note that some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.