CMSI 284 Introductory C Exercise
or, Oh Say Can You C

Instructions

Write the requested C programs or functions. If a function is requested, make the full program consist of a test harness that checks the correctness of the function by calling it multiple times with a variety of arguments.

This assignment *must* be submitted using GitHub. Commit your files under the folder *c-intro*.

Mapping to Outcomes and Proficiencies

The overall assignment covers outcomes *2a*, *2b*, *4a*, *4b*, *4c*, *4d*, *4e*, and *4f*. Outcomes *4a* to *4d* apply to all programs. Outcomes *2a* and *2b* apply depending on the type of code that is needed by the exercise. Outcome *4e* depends on your commit frequency, descriptiveness of your messages, and the way you schedule your work. Outcome *4f* depends on whether you submit your code on time.

1. (*2b*) Write a C program, `chord.c`, that takes a command line argument which is the name of a piano key, and writes to standard output the major, minor, dominant 7th, and diminished 7th chords for that key. For simplicity, constrain note names exclusively to sharps.

   For example:

   ```
   $ chord F#
   F#: F# A# C#
   F#m: F# A C#
   F#7: F# A# C# E
   F#dim7: F# A C D#
   ```

2. (*2b*) Write a C program, `interval.c`, that takes two command line arguments which are the names of two piano keys, assumes that the second key is higher than the first

key, and writes to standard output the *interval* between those two keys. The intervals are defined as follows:

Keys Apart	Interval Name
1	minor second
2	major second
3	minor third
4	major third
5	perfect fourth
6	tritone
7	perfect fifth
8	minor sixth
9	major sixth
10	minor seventh
11	major seventh
12	perfect octave

For simplicity, constrain note names exclusively to sharps. Note how, if the same key is given for both arguments, the output should be `perfect octave` because the second key is always assumed to be above the first one.

For example:

```
$ interval F# A
F# to A is a minor third.
```

3. (*2a*, *2b*) Implement a "mad libs" function in its own file, `madlib.c`, with this signature:

```
char* madlib(char* template, char* adjective, char* noun, char* verb)
```

`template` should be a C format string with three `%s` placeholders, one each for the given `adjective`, `noun`, and `verb` strings. `madlib` should create a new string consisting of the `template` with its `%s` placeholders replaced by the three words (*cough* `sprintf` *cough*). For example,

```
madlib("The %s %s likes to %s in the moonlight.",
    "brilliant", "git", "swim")
```

should evaluate to `"The brilliant git likes to swim in the moonlight."`.

Accompany your function file with a test harness, `madlib-test.c`, that uses the C `assert` function to show that your `madlib` function works properly.

4. (*2a*, *2b*) Implement an *ordered* "mad libs" function in its own file, `madlib-by-numbers.c`, with this signature:

   ```
   char* madlib_by_numbers(char* template, int word_count, char* words[])
   ```

   For this variation of the function, `template` should be a string where single digits `0` to `9` may be substituted by the corresponding word in the given `words` array. If a digit exceeds the maximum index in the `words` array, no substitution takes place. As with `madlib`, a new string should be created. For example, given the string array

   ```
   char* words_to_use[] = { "swim", "brilliant", "git" };
   ```

   then the expression

   ```
   madlib_by_numbers("The 1 2 likes to 0 in the 1 moonlight.", 3,
       words_to_use)
   ```

   should evaluate to

   ```
   "The brilliant git likes to swim in the brilliant moonlight."
   ```

   Note `madlib_by_numbers` can do the following: (a) use a word multiple times, and (b) substitute words in any order. Note also that `madlib_by_numbers` does not substitute a double-digit index. e.g., `"12"` in the example above would become `"brilliantgit"`.

   Accompany your function file with a test harness, `madlib-by-numbers-test.c`, that uses the C `assert` function to show that your `madlib_by_numbers` function works properly.

   Protip: Strongly consider defining additional functions in `madlib-by-numbers.c` that perform intermediate computations for you. This not only shortens the lengths of your individual functions, but provides additional units of functionality that you can test separately.

5. (*2a*, *2b*) Use the sample `reverse_range_in_place` function to implement:

   ```
   void reverse_words(char* string)
   ```

   ...which does what you think it does: it reverses the words, in place, within the given string (i.e., the original string is modified into its reversed-words version). To keep things simple, we will define a "word" as any sequence of non-space characters,

including punctuation, numbers, etc. For example, if the string argument pointed to `"Hello my friends!"`, `reverse_words` should change its contents into `"friends! my Hello"`.

You may *not* allocate additional memory, and the only other string function that you may use is `strlen`. Watch out for edge cases such as leading/trailing spaces, multiple spaces in between words, etc.

Organize your source code as follows: place your function in `reverse-words.c` with an accompanying header file `reverse-words.h`. Supply a test harness to demonstrate that your function works as specified; call this `reverse-words-test.c`. The original `reverse_range_in_place` code should not be modified.