CMPUT275—Assignment 2 (Winter 2025)

R. Hackman

Due Date: Friday February 14th, 8:00PM

Per course policy you are allowed to engage in *reasonable* collaboration with your classmates. You must include in the comments of your assignment solutions a list of any students you collaborated with on that particular question.

In this assignment and all following assignments you should test against the sample executables. The sample executables try to help you out and print error messages when receiving invalid input (though they do not catch *all* invalid input). You do not have to replicate any error messages printed, unless the assignment specification specifically asks for error messages. These messages are only in the sample executable to help you catch when you write an invalid test case.

Memory Management: In order to complete some of these questions you will be required to use dynamic memory allocation. Your programs must not leak any memory, if you leak memory on a test case then you are considerd to have failed that test case. You can test your program for memory leaks by using the tool `valgrind`.

Compilation Flags: each of your programs should be compiled with the following command:

```
gcc -Wall -Wvla -Werror
```

These are the flags we'll compile your program with, and should they result in a compilation error then your code will not be compiled and ran for testing.

Allowed libraries: `stdlib.h`, `stdio.h`, and `string.h`. No other libraries are allowed.

1. **Credit Card Verification**

   If you've ever attempted to purchas an item online with a credit card, you may have accidentally entered your credit card information incorrectly. Upon entering your credit card information incorrectly you may have immediately been informed that the provided credit card was not valid. Have you ever wondered how, out of all the possible credit card numbers, the online store immediately knew the number you entered was not a valid credit card? This is because credit card numbers are verified with an algorithm known as *Luhn's Algorithm*. This is a very simple *checksum* algorithm that once could choose to use to specify what set of values are valid versus those that are not. Note, Luhn's algorithm does not provide any security it is simply a way to quickly validate if a number is possibly valid, and the major credit card companies all use it to do so (so all valid credit card numbers from these companies meet the checksum determined by Luhn's algorithm).

   Given an "account number" Luhn's algorithm verifies the number is potentially valid with a simple checksum. given an account number of n digits of the form $d_1 d_2 d_3 d_4 ... d_n$ then the following procedure is followed.

   (a) The nth digit is the check digit, and is only used at the end to verify validity.

   (b) Moving left from the nth digit each digit has its value added to a sum. However, we follow a different procedure for every second digit.

(c) The first digit to the left of our check digit has its value doubled, if that product is larger than or equal to 10 then 9 is subtracted from the product to give us the value to add to our sum.

(d) The next digit to the left just has its value added to our sum.

(e) Repeat that process moving left until there are no more digits: doubling one subtracting 9 if necessary, then not doubling the next.

(f) Once the sum is calculated it should be multiplied by 9, the result of this product should have a remainder when divided by 10 that is equal to the check digit.

For example, if we want to check if an account number `34682` is valid, we would do the following procedure:

- Our last digit is our check digit, it is 2.
- The next digit to the left is 8, so since this is the first digit to the left we multiply it by 2 which leaves us with 16. Since 16 is larger than or equal to 10 we subtract 9 from it leaving us with 7 to add to our sum.
- The next digit to the left is 6, since it follows a digit we just doubled we do not double it and simply add 6 to our sum.
- The next digit to the left is 4, since we didn't double the last digit we now double this one leaving us with 8 to add to our sum.
- The next digit to the left is 3, since it follows a digit we just doubled we do not double it and simply add it to our sum.
- So our sum is now 7+6+8+3=24. 24*9=216, the remainder when dividing 216 by 10 is 6, so since our check digit is not 6 this account number is *invalid*.

For this question you must complete the program `luhns.c` which reads digits from the standard input stream until the first non-digit character is received (indicating the end of the account number). Your program should print `Valid` if the account number read in passes Luhn's algorithm and should print `Invalid` if the account number does not pass Luhn's algorithm.

Constraints: For this question you are not allowed to use arrays (on the heap or stack). Use of arrays or pointers of any kind will result in 0 on this question. You do not need an array to solve this question, and part of the challenge of this question is thinking of how to tackle the task without using an array.

Deliverables For this question include in your final submission zip your c source code file named `luhns.c`

2. **Roman Numerals**

Roman numerals refers to a numeral system originating in ancient rome - this numeral system is often used today for various purposes (such as superbowl, the upcoming superbowl being LIX). We are going to write a program that can translate numbers written in a simplified version of the Roman numeral system into their representation in our commonly used Arabic numeral system. The simplified Roman numeral system we will translate will have the following properties and rules.

- Our symbols for representing numbers will be I, V, X, L, C, D, and M representing 1, 5, 10, 50, 100, 500, and 1000 respectively.
- Our numbers are represented by a sequence of these characters, however there are rules about how these form numbers.
- The first rule, is that if a numeral is placed after a larger (or equal) numeral, then its value should be added to our total.
- The second opposing rules, is that if a numeral is placed after a smaller numeral, then the value of the smaller numeral should be deducted from our total.

For example, the number `IV` represents 4, because the `I` immediately precedes the larger numeral `V`, so the 1 that the `I` represents is subtracted from our total while the `V` which represents 5 is added to our total.

This can be tricky though, consider the number `XLI` which represents 41, because the `X` immediately precedes a larger numeral `L` so it is negated, meanwhile the `L` and `I` are not followed by numerals smaller than themselves, so they add themselves to our total for a sum of -10+50+1=41. However, if we consider a similar looking number `XLIV` this number actually represents 44, because the `I` here is actually negative due to immediately preceding the larger numeral `V`, leaving us with a sum of -10+50-1+5=44.

You must write the program `numerals.c` which reads *one* number written in Roman numerals from the standard input stream and then prints out that number in the Arabic numeral system (that is, print out its integer value as we are used to seeing integers).

Deliverables: For this question include in your final submission zip your c source code file named `numerals.c`

3. **Wordl**

In this question you will develop a game called *wordl*, wordl is a word game much like the game wordle. In the game of wordl there is a secret code word that the user is trying to guess, and whenever a user makes a guess the following happens:

- The users guess is printed out colorized with the following rules:
  - If the letter in the guess is not in the code at all then it is printed out as white.
  - If the letter in the guess is the same letter at that position as the code then the letter is printed out as green.
  - If the letter in the guess occurs in the code, but is in the wrong spot then you print it out as yellow *if* the corresponding letter in the code is not matched as green by the previous rule and you have not printed out a number of that character in yellow equal to the number of that character in the code minus the number of green of that character.

For example, if the code was `verge` and you guessed `bevel` then the printed out response would be:

bevel

However, if the code was `bevel` an you guessed `eenie` then the printed out response would be:

eenie

With the important distinction that the last letter `e` was coloured white, despite the fact that the letter `e` appears in the code. This is because there was already a yellow `e` previously printed and there was a green `e` printed, so there were already a coloured `e` to account for both letters `e` in the code.

Your program expects one command line argument which is the code word that must be guessed. Your program then should repeat the following process:

(a) Print the message `"Enter guess: "` and read a string from standard input, that string will be the users guess.

(b) Print out the colourized version of the user's guess followed by a newline.

(c) If the user did not guess the code word correctly, and has not already made 6 guesses, then begin these steps again.

(d) If the user did not guess the code correctly, and has already made 6 guesses then print out the message `"Failed to guess the word: <word>"` followed by a newline, where `<word>` is replaced with the secret code word. Then your program terminates.

(e) If the user does guess the code correctly, then print the message `"Finished in <n> guesses"` followed by a newline, where `<n>` is replaced with the number of guesses the user made.

You may assume the user always gives you a guess exactly equal to the length of the code, and you may assume the code is never more than 12 characters long.

In order to print out colours you will have to print special colour characters, and these are characters that show up in your output, so you have to be careful about when you print them. You should only print out a colour character whne you need to switch colours for the immediate next thing you'd like to print. In order to help with this we have given you some

starter code that uses some global variables as well as a function `setColour`. You should use the function `setColour` and provide as the argument one of the global variables `YELLOW`, `GREEN`, or `WHITE` for those colours. The function makes sure not to print out the colour character if that colour is already set, which will help in ensuring you do not print out erroneous colour characters, but does not guarantee you are printing them exactly only when you need to.

Note: Because the colour characters are invisible (other than changing the colours of the following characters), it is possible for your output to *look* like it matches the sample executable, but not *precisely* match the characters printed by the sample executable. Due to this, it is *very important* that you test by comparing the outputs of your program with the outputs of the sample executable using a tool like diff - which is what your `runTests` should do.

Starter Code: Since we have not discussed using command line arguments yet, the provided starter code also has the beginning of a main which simply checks if the command line argument was properly given, and defines a pointer variable `theWord` to be a pointer to the word which was received as a command line argument. Thus, you do not have to worry about reading the command line argument and can start writing code assuming you have the correct word in that variable.

Deliverables: For this question include in your final submission zip your c source code file named `wordl.c`

How to submit: Create a zip file `a2.zip`, make sure that zip file contains your C source code files `luhns.c`, `numerals.c`, and `wordl.c`. Assuming all three of these files are in your current working directory you can create your zip file with the command

```
$ zip a2.zip luhns.c numerals.c wordl.c
```

Upload your file `a2.zip` to the a2 submission link on eClass.