IC221 Lab: Strings and Command Line Arguments
100 points total. Spring AY22.

See solutions at https://github.com/umbertofontana/systems-programming

Learning Objectives
- Write programs that manipulate C strings
- Perform basic pointer manipulation and arithmetic in C
- Process command line arguments in C
- Convert between C strings and other data types
- Implement in C a simple algorithm described in pseudocode

Test Script `./test.sh`

Submission
`palindrome.c`
`check.c`

Part 1: String Manipulation (30 Points)

For this part of the lab, you will complete a small program to check if the input provided by the user is a palindrome. A palindrome is a string that is the same forward and backwards. Palindromes may be of odd or even length but, for this exercise, not empty.

There are two main ways for testing if a string is a palindrome:
- Iterate from forward to back, and from back to forward, and check that they are the same.
- Create a copy of the string, in reverse, and check that the copy matches the original.

You will implement both methods using C strings (character arrays). The main portion of the program, which is provided, reads the input and calls the two palindrome check functions.

Note that the input is read into the string `str` and may be substantially shorter than the maximum length, 1024. The input is also passed to the `check1()` and `check2()` functions without a length argument. You should make sure that you use `strlen()` somewhere.

Running `check2()` requires making a copy of the string; don't forget that C strings must be NULL terminated. If you do not NULL terminate the string, `strcmp()` will not work properly.

Complete the palindrome program using two different checks, `check1()` and `check2()`, using two distinct algorithms:
- `check1()` uses two iterators, one from the start and one from the end of the string, to check if the forward and reverse methods read the same
- `check2()` copies the string to a new string, in reverse, and compares the original and reversed versions using `strcmp()`.

Sample output:

```
$ ./palindrome
Enter a string:
racecar
Palindrome according to check 1
Palindrome according to check 2

$ ./palindrome
Enter a string:
madamimadam
Palindrome according to check 1
Palindrome according to check 2

$ ./palindrome
Enter a string:
amanaplanacanalpanama
Palindrome according to check 1
Palindrome according to check 2

$ ./palindrome
Enter a string:
notapalindrome
NOT a palindrome according to check 1
NOT a palindrome according to check 2
```

Part 2: Credit Card Checking with Luhn's Algorithm (70 points)

For this part of the lab, you will implement a checksum for credit card numbers. A checksum is a method to verify that a number meets certain properties. It is an important principle in networking and other areas, and you will implement the checksum used to check if a credit card number is valid. Note that just because a number passes the checksum, it doesn't make it a truly valid credit card number in the real world, it just means that the number may be used as a credit card number.

Credit card numbers are checked using Luhn's Algorithm. You can find a description on wikipedia, but it is highlighted below. It occurs in three basic steps:

1) For the 16 digit card number, double *every other* digit, starting with the first:

```
6  4  7  4  5  6  2  3  8  9  9  7  2  7  5  6  <- number
|     |     |     |     |     |     |     |
v     v     v     v     v     v     v     v
12    14    10    4     16    18    4     10    <- double every other digit
```

2) If the double of a digit is greater than 9, replace it with the sum of its two digits. For example, 12 would be replaced by 3, because 12 is greater than 10 and 1+2=3. Example:

```
6  4  7  4  5  6  2  3  8  9  9  7  2  7  5  6  <- number
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
v  |  v  |  v  |  v  |  v  |  v  |  v  |  v  |  v  |
12 | 14 | 10 |  4 | 16 | 18 |  4 | 10 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
v  v  v  v  v  v  v  v  v  v  v  v  v  v  v  v
3  4  5  4  1  6  4  3  7  9  9  7  4  7  1  6  <- sum of digits if >9
```

3) Take the sum of the remaining numbers

```
3 + 4 + 5 + 4 + 1 + 6 + 4 + 3 + 7 + 9 + 9 + 7 + 4 + 7 + 1 + 6 = 80
```

If that number is a multiple of 10, then the original card number passes the checksum:

```
80%10 = 0 <- PASS
```

Your program should take a 16 digit number on the command line and print VALID or INVALID:

```
$ ./check 6474562389972756
VALID
$ ./check 6474562389972757
INVALID
```

Your program should detect three error conditions: (1) No argument provided; (2) Invalid length of credit card number; (3) Invalid numbers in credit card. Example output:

```
$ ./check
ERROR: require credit card number
$ ./check 12345
ERROR: Invalid credit card number: Bad Length
$ ./check 12345a7890123456
ERROR: Invalid credit card number: Bad number 'a'
```

To perform the error checks, consider that the command line arguments are a string, and each number is currently a character. The numeric value of that character is not the same as its integer value, but they are still ordered. For example, this will check for a valid digit:

```
if ( c >= '0' && c <= '9' )
  printf("Valid digit!\n");
else
  printf("Invalid digit!\n");
```

Finally, because the checksum occurs over integers and not strings, you will need to convert each individual digit into a integer. Consider creating an array for the credit card number that can store all these digits:

```
int ccnum[16];
```

You can use the function `atoi()` to perform the conversion from string to integer, but you'll need to first isolate the character of the number as its own string.

```
char num[2];
num[0] = '8';
num[1] = '\0'; //null terminate
int i = atoi(num); //convert to integer
```

Requirements

- Your program must print "VALID" for a valid credit card number that passes Luhn's checksum and "INVALID" for a credit card number that does not pass the checksum.
- You must take the credit card number as the command line argument and check for the following the errors and print the following error statements:
- No argument provided:
  `ERROR: require credit card number` printed to *stderr*
- Credit card is not the right length:
  `ERROR: Invalid credit card number: Bad Length` printed to *stderr*
- A non-digit appears in the number (the "%c" should be the invalid character):
  `ERROR: Invalid credit card number: Bad number '%c'` printed to *stderr*
- Your program should return the following *exit codes* on completion:
  - return 0: valid credit card number
  - return 1: input error (no argument, incorrect length, non-digit)
  - return 2: invalid credit card number

(Extra Credit: 5 points): Write and submit a bash script `rand_creditcard.sh` that will find a random, valid credit card number by using the `check` program. Sample output:

```
$ ./rand_creditcard.sh
1002947242955100
$ ./rand_creditcard.sh
8113723297573564
```