This is CS50

CS50's Introduction to Computer Science

OpenCourseWare

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

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Credit

Getting Started

Open VS Code (https://cs50.dev/).

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/1/credit.zip
```

followed by Enter in order to download a ZIP called `credit.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 credit.zip
```

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

```
rm credit.zip
```

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

Now type

```
cd credit
```

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

```
credit/ $
```

If all was successful, you should execute

```
ls
```

and see a file named `credit.c`. Executing `code credit.c` should open the file where you will type your code for this problem set. If not, retrace your steps and see if you can determine where you went wrong!

Credit Cards

A credit (or debit) card, of course, is a plastic card with which you can pay for goods and services. Printed on that card is a number that's also stored in a database somewhere, so that when your card is used to buy something, the creditor knows whom to bill. There are a lot of people with credit cards in this world, so those numbers are pretty long: American Express uses 15-digit numbers, MasterCard uses 16-digit numbers, and Visa uses 13- and 16-digit numbers. And those are decimal numbers (0 through 9), not binary, which means, for instance, that American Express could print as many as 10^15 = 1,000,000,000,000,000 unique cards! (That's, um, a quadrillion.)

Actually, that's a bit of an exaggeration, because credit card numbers actually have some structure to them. All American Express numbers start with 34 or 37; most MasterCard numbers start with 51, 52, 53, 54, or 55 (they also have some other potential starting numbers which we won't concern ourselves with for this problem); and all Visa numbers start with 4. But credit card numbers also have a "checksum" built into them, a mathematical relationship between at least one number and others. That checksum enables computers (or humans who like math) to detect typos (e.g., transpositions), if not fraudulent numbers, without having to query a database, which can be slow. Of course, a dishonest mathematician could certainly craft a fake number that nonetheless respects the mathematical constraint, so a database lookup is still necessary for more rigorous checks.

Luhn's Algorithm

So what's the secret formula? Well, most cards use an algorithm invented by Hans Peter Luhn of IBM. According to Luhn's algorithm, you can determine if a credit card number is (syntactically) valid as follows:

1. Multiply every other digit by 2, starting with the number's second-to-last digit, and then add those products' digits together.
2. Add the sum to the sum of the digits that weren't multiplied by 2.
3. If the total's last digit is 0 (or, put more formally, if the total modulo 10 is congruent to 0), the number is valid!

That's kind of confusing, so let's try an example with David's Visa: 4003600000000014.

1. For the sake of discussion, let's first underline every other digit, starting with the number's second-to-last digit:

   $\underline{4}0\underline{0}3\underline{6}0\underline{0}0\underline{0}0\underline{0}0\underline{0}0\underline{1}4$

   Okay, let's multiply each of the underlined digits by 2:

   1•2 + 0•2 + 0•2 + 0•2 + 0•2 + 6•2 + 0•2 + 4•2

   That gives us:

   2 + 0 + 0 + 0 + 0 + 12 + 0 + 8

   Now let's add those products' digits (i.e., not the products themselves) together:

   2 + 0 + 0 + 0 + 0 + 1 + 2 + 0 + 8 = 13

2. Now let's add that sum (13) to the sum of the digits that weren't multiplied by 2 (starting from the end):

   13 + 4 + 0 + 0 + 0 + 0 + 0 + 3 + 0 = 20

3. Yup, the last digit in that sum (20) is a 0, so David's card is legit!

So, validating credit card numbers isn't hard, but it does get a bit tedious by hand. Let's write a program.

Implementation Details

In the file called `credit.c` in the `credit` directory, write a program that prompts the user for a credit card number and then reports (via `printf`) whether it is a valid American Express, MasterCard, or Visa card number, per the definitions of each's format herein. So that we can automate some tests of your code, we ask that your program's last line of output be `AMEX\n` or `MASTERCARD\n` or `VISA\n` or `INVALID\n`, nothing more, nothing less. For simplicity, you may assume that the user's input will be entirely numeric (i.e., devoid of hyphens, as might be printed on an actual card) and that it won't have leading zeroes. But do not assume that the user's input will fit in an `int`! Best to use `get_long` from CS50's library to get users' input. (Why?)

Consider the below representative of how your own program should behave when passed a valid credit card number (sans hyphens).

```
$ ./credit
Number: 4003600000000014
VISA
```

Now, `get_long` itself will reject hyphens (and more) anyway:

```
$ ./credit
Number: 4003-6000-0000-0014
Number: foo
Number: 4003600000000014
VISA
```

But it's up to you to catch inputs that are not credit card numbers (e.g., a phone number), even if numeric:

```
$ ./credit
Number: 6176292929
INVALID
```

Test out your program with a whole bunch of inputs, both valid and invalid. (We certainly will!) Here are a few card numbers (https://developer.paypal.com/api/nvp-soap/payflow/integration-guide/test-transactions/#standard-test-cards) that PayPal recommends for testing.

If your program behaves incorrectly on some inputs (or doesn't compile at all), time to debug!

Walkthrough

How to Test Your Code

You can also 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/credit
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 credit.c
```

How to Submit

In your terminal, execute the below to submit your work.

```
submit50 cs50/problems/2023/x/credit
```