This is CS50x

OpenCourseWare

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.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 1

For this problem set, you'll use CS50 IDE, a cloud-based programming environment. This environment is similar to CS50 Sandbox and CS50 Lab, the programming environments that David discussed during lecture.

What to Do

1. Go to ide.cs50.io (https://ide.cs50.io) and click "Sign in with GitHub" to access your CS50 IDE.
2. Submit Hello
3. Submit one of:
   - this version of Mario if feeling less comfortable
   - this version of Mario if feeling more comfortable
4. Submit one of:
   - Cash if feeling less comfortable
   - Credit if feeling more comfortable

If you submit both Marios, we'll record the higher of your two scores. If you submit both of Cash and Credit, we'll record the higher of your two scores.

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Try out any of David's programs from class via Week 1's source code.
- If you see any errors when compiling your code with `make`, focus first on fixing the very first error you see, scrolling up as needed. If unsure what it means, try asking `help50` for help. For instance, if trying to compile `hello`, and

  ```
  make hello
  ```

  is yielding errors, try running

  ```
  help50 make hello
  ```

  instead!

This is CS50x

OpenCourseWare

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.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Hello

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/hello/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

Getting Started

CS50 IDE is a web-based "integrated development environment" that allows you to program "in the cloud," without installing any software locally. Indeed, CS50 IDE provides you with your very own "workspace" (i.e., storage space) in which you can save your own files and folders (aka directories).

Logging In

Head to ide.cs50.io (https://ide.cs50.io) and click "Sign in with GitHub" to access your CS50 IDE. Once your IDE loads, you should see that (by default) it's divided into three parts. Toward the top of CS50 IDE is your "text editor", where you'll write all of your programs. Toward the bottom of is a "terminal window" (light blue, by default), a command-line interface (CLI) that allows you to explore your workspace's files and directories, compile code, run programs, and even install new software. And on the left is your "file browser", which shows you all of the files and folders currently in your IDE.

Start by clicking inside your terminal window. You should find that its "prompt" resembles the below.

```
~/ $
```

Click inside of that terminal window and then type

```
mkdir ~/pset1/
```

followed by Enter in order to make a directory (i.e., folder) called `pset1` in your home directory. Take care not to overlook the space between `mkdir` and `~/pset1` or any other character for that matter! Keep in mind that `~` denotes your home directory and `~/pset1` denotes a directory called `pset1` within `~`.

Here on out, to execute (i.e., run) a command means to type it into a terminal window and then hit Enter. Commands are "case-sensitive," so be sure not to type in uppercase when you mean lowercase or vice versa.

Now execute

```
cd ~/pset1/
```

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

```
~/pset1/ $
```

If not, retrace your steps and see if you can determine where you went wrong.

Now execute

```
mkdir ~/pset1/hello
```

to create a new directory called `hello` inside of your `pset1` directory. Then execute

```
cd ~/pset1/hello
```

to move yourself into that directory.

Shall we have you write your first program? From the *File* menu, click *New File*, and save it (as via the *Save* option in the *File* menu) as `hello.c` inside of your `~/pset1/hello` directory. Proceed to write your first program by typing precisely these lines into the file:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

Notice how CS50 IDE adds "syntax highlighting" (i.e., color) as you type, though CS50 IDE's choice of colors might differ from this problem set's. Those colors aren't actually saved inside of the file itself; they're just added by CS50 IDE to make certain syntax stand out. Had you not saved the file as `hello.c` from the start, CS50 IDE wouldn't know (per the filename's extension) that you're writing C code, in which case those colors would be absent.

Listing Files

Next, in your terminal window, immediately to the right of the prompt (`~/pset1/hello/ $`), execute

```
ls
```

You should see just `hello.c` ? That's because you've just listed the files in your `hello` folder. In particular, you *executed* (i.e., ran) a command called `ls` , which is shorthand for "list." (It's such a frequently used command that its authors called it just `ls` to save keystrokes.) Make sense?

Compiling Programs

Now, before we can execute the `hello.c` program, recall that we must *compile* it with a *compiler* (e.g., `clang`), translating it from *source code* into *machine code* (i.e., zeroes and ones). Execute the command below to do just that:

```
clang hello.c
```

And then execute this one again:

```
ls
```

This time, you should see not only `hello.c` but `a.out` listed as well? (You can see the same graphically if you click that folder icon again.) That's because `clang` has translated the source code in `hello.c` into machine code in `a.out` , which happens to stand for "assembler output," but more on that another time.

Now run the program by executing the below.

```
./a.out
```

Hello, world, indeed!

Naming Programs

Now, `a.out` isn't the most user-friendly name for a program. Let's compile `hello.c` again, this time saving the machine code in a file called,

```
submit50 cs50/problems/2020/x/hello
```

more aptly, `hello`. Execute the below.

```
clang -o hello hello.c
```

Take care not to overlook any of those spaces therein! Then execute this one again:

```
ls
```

You should now see not only `hello.c` (and `a.out` from before) but also `hello` listed as well? That's because `-o` is a *command-line argument*, sometimes known as a *flag* or a *switch*, that tells `clang` to output (hence the `o`) a file called `hello`. Execute the below to try out the newly named program.

```
./hello
```

Hello there again!

Making Things Easier

Recall that we can automate the process of executing `clang`, letting `make` figure out how to do so for us, thereby saving us some keystrokes. Execute the below to compile this program one last time.

```
make hello
```

You should see that `make` executes `clang` with even more command-line arguments for you? More on those, too, another time!

Now execute the program itself one last time by executing the below.

```
./hello
```

Phew!

Getting User Input

Suffice it to say, no matter how you compile or execute this program, it only ever prints `hello, world`. Let's personalize it a bit, just as we did in class.

Modify this program in such a way that it first prompts the user for their name and then prints `hello, so-and-so`, where `so-and-so` is their actual name.

As before, be sure to compile your program with:

```
make hello
```

And be sure to execute your program, testing it a few times with different inputs, with:

```
./hello
```

Walkthrough

Hints

Don’t recall how to prompt the user for their name?

Recall that you can use `get_string` as follows, storing its *return value* in a variable called `name` of type `string`.

```
string name = get_string("What is your name?\n");
```

Don’t recall how to format a string?

Don’t recall how to join (i.e., concatenate) the user’s name with a greeting? Recall that you can use `printf` not only to print but to format a string (hence, the `f` in `printf`), a la the below, wherein `name` is a `string`.

```
printf("hello, %s\n", name);
```

Use of undeclared identifier?

Seeing the below, perhaps atop other errors?

```
error: use of undeclared identifier 'string'; did you mean 'stdin'?
```

Recall that, to use `get_string`, you need to include `cs50.h` (in which `get_string` is *declared*) atop a file, as with:

```
#include <cs50.h>
```

How to Test Your Code

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/2020/x/hello
```

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

```
style50 hello.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you’ll see asterisks (`*`) instead of the actual characters in your password.

This is CS50x

OpenCourseWare

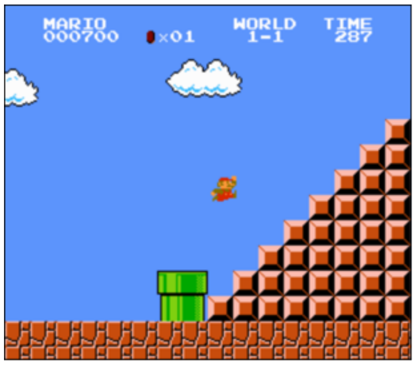
David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Mario

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/mario/less/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

World 1-1

Toward the end of World 1-1 in Nintendo's Super Mario Brothers, Mario must ascend right-aligned pyramid of blocks, a la the below.



Let's recreate that pyramid in C, albeit in text, using hashes (#) for bricks, a la the below. Each hash is a bit taller than it is wide, so the pyramid itself is also be taller than it is wide.

```
       #
      ##
     ###
    ####
   #####
  ######
 #######
########
```

The program we'll write will be called `mario`. And let's allow the user to decide just how tall the pyramid should be by first prompting them for a positive integer between, say, 1 and 8, inclusive.

Here's how the program might work if the user inputs `8` when prompted:

```
$ ./mario
Height: 8
       #
      ##
     ###
    ####
   #####
  ######
 #######
########
```

Here’s how the program might work if the user inputs `4` when prompted:

```
$ ./mario
Height: 4
   #
  ##
 ###
####
```

Here’s how the program might work if the user inputs `2` when prompted:

```
$ ./mario
Height: 2
 #
##
```

And here’s how the program might work if the user inputs `1` when prompted:

```
$ ./mario
Height: 1
#
```

If the user doesn’t, in fact, input a positive integer between 1 and 8, inclusive, when prompted, the program should re-prompt the user until they cooperate:

```
$ ./mario
Height: -1
Height: 0
Height: 42
Height: 50
Height: 4
   #
  ##
 ###
####
```

How to begin? Let’s approach this problem one step at a time.

Pseudocode

First, create a new directory (i.e., folder) called `mario` inside of your `pset1` directory by executing

```
~/ $ mkdir ~/pset1/mario
```

Add a new file called `pseudocode.txt` inside of your `mario` directory.

Write in `pseudocode.txt` some pseudocode that implements this program, even if not (yet!) sure how to write it in code. There's no one right way to write pseudocode, but short English sentences suffice. Recall how we wrote pseudocode for finding Mike Smith (https://docs.google.com/presentation/d/17wRd8ksO6QkUq906SUgm17AqcI-Jan42jkY-EmufxnE/edit?usp=sharing). Odds are your pseudocode

will use (or imply using!) one or more functions, conditions, Boolean expressions, loops, and/or variables.

▶ **Spoiler**

Prompting for Input

Whatever your pseudocode, let's first write only the C code that prompts (and re-prompts, as needed) the user for input. Create a new file called `mario.c` inside of your `mario` directory.

Now, modify `mario.c` in such a way that it prompts the user for the pyramid's height, storing their input in a variable, re-prompting the user again and again as needed if their input is not a positive integer between 1 and 8, inclusive. Then, simply print the value of that variable, thereby confirming (for yourself) that you've indeed stored the user's input successfully, a la the below.

```
$ ./mario
Height: -1
Height: 0
Height: 42
Height: 50
Height: 4
Stored: 4
```

▶ **Hints**

Building the Opposite

Now that your program is (hopefully!) accepting input as prescribed, it's time for another step.

It turns out it's a bit easier to build a left-aligned pyramid than right-, a la the below.

```
#
##
###
####
#####
######
#######
########
```

So let's build a left-aligned pyramid first and then, once that's working, right-align it instead!

Modify `mario.c` at right such that it no longer simply prints the user's input but instead prints a left-aligned pyramid of that height.

▶ **Hints**

Right-Aligning with Dots

Let's now right-align that pyramid by pushing its hashes to the right by prefixing them with dots (i.e., periods), a la the below.

```
.......#
......##
.....###
....####
...#####
..######
```

```
..######
.#######
########
```

Modify `mario.c` in such a way that it does exactly that!

▶ **Hint**

How to Test Your Code

Does your code work as prescribed when you input

- `-1` (or other negative numbers)?
- `0` ?
- `1` through `8` ?
- `9` or other positive numbers?
- letters or words?
- no input at all, when you only hit Enter?

Removing the Dots

All that remains now is a finishing flourish! Modify `mario.c` in such a way that it prints spaces instead of those dots!

How to Test Your Code

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/2020/x/mario/less
```

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

```
style50 mario.c
```

▶ **Hint**

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/mario/less
```

This is CS50x

OpenCourseWare

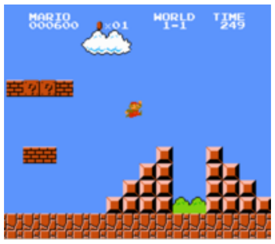
David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Mario

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/mario/more/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

World 1-1

Toward the beginning of World 1-1 in Nintendo's Super Mario Brothers, Mario must hop over adjacent pyramids of blocks, per the below.



Let's recreate those pyramids in C, albeit in text, using hashes (#) for bricks, a la the below. Each hash is a bit taller than it is wide, so the pyramids themselves are also be taller than they are wide.

```
   #  #
  ##  ##
 ###  ###
####  ####
```

The program we'll write will be called `mario` . And let's allow the user to decide just how tall the pyramids should be by first prompting them for a positive integer between, say, 1 and 8, inclusive.

Here's how the program might work if the user inputs `8` when prompted:

```
$ ./mario
Height: 8
       #  #
      ##  ##
     ###  ###
    ####  ####
   #####  #####
  ######  ######
 #######  #######
########  ########
```

Here's how the program might work if the user inputs `4` when prompted:

```
$ ./mario
Height: 4
   #  #
  ##  ##
 ###  ###
```

```
####  ####
```

Here’s how the program might work if the user inputs `2` when prompted:

```
$ ./mario
Height: 2
 #  #
##  ##
```

And here’s how the program might work if the user inputs `1` when prompted:

```
$ ./mario
Height: 1
#  #
```

If the user doesn’t, in fact, input a positive integer between 1 and 8, inclusive, when prompted, the program should re-prompt the user until they cooperate:

```
$ ./mario
Height: -1
Height: 0
Height: 42
Height: 50
Height: 4
   #  #
  ##  ##
 ###  ###
####  ####
```

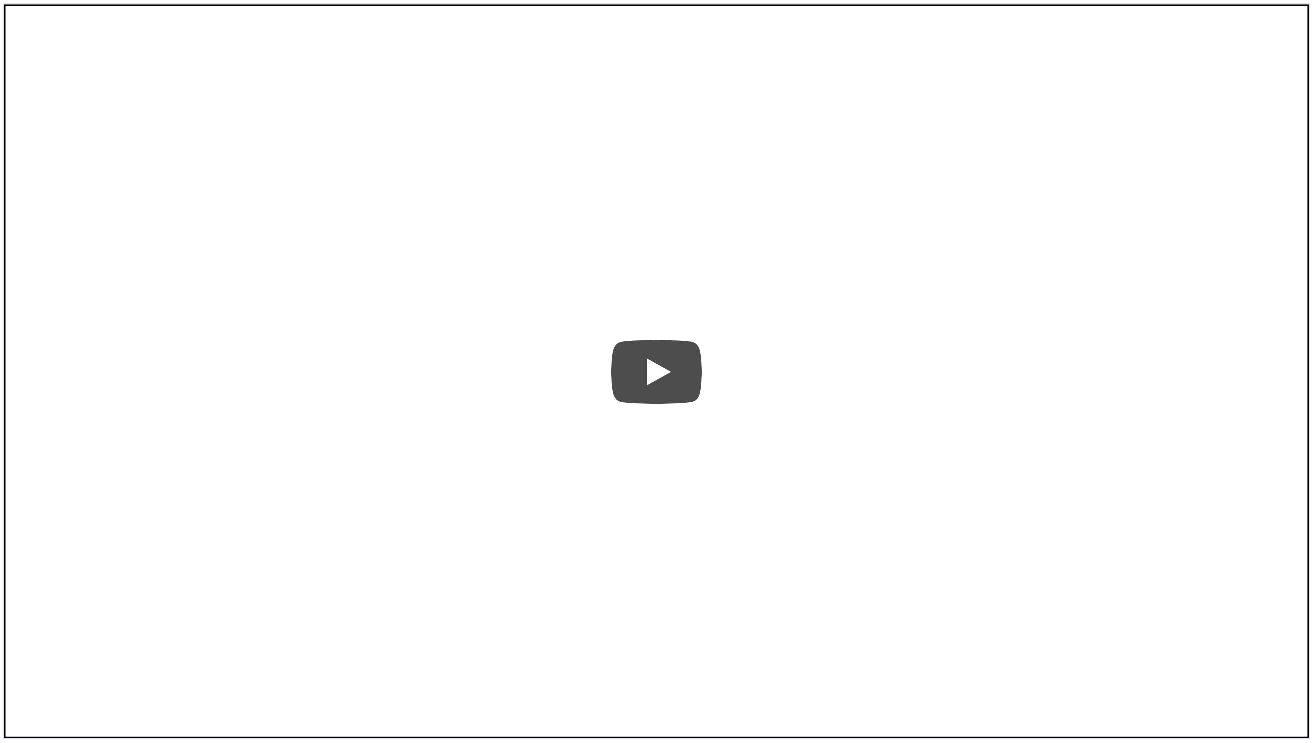
Notice that width of the “gap” between adjacent pyramids is equal to the width of two hashes, irrespective of the pyramids’ heights.

Create a new directory called `mario` inside of your `pset1` directory by executing

```
~/ $ mkdir ~/pset1/mario
```

Create a new file called `mario.c` inside your `mario` directory. Modify `mario.c` in such a way that it implements this program as described!

Walkthrough



How to Test Your Code

Does your code work as prescribed when you input

- `-1` (or other negative numbers)?
- `0`?
- `1` through `8`?
- `9` or other positive numbers?
- letters or words?
- no input at all, when you only hit Enter?

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/2020/x/mario/more
```

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

```
style50 mario.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/mario/more
```

This is CS50x

OpenCourseWare

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.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Cash

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/cash/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

Greedy Algorithms

When making change, odds are you want to minimize the number of coins you're dispensing for each customer, lest you run out (or annoy the customer!). Fortunately, computer science has given cashiers everywhere ways to minimize numbers of coins due: greedy algorithms.

According to the National Institute of Standards and Technology (NIST), a greedy algorithm is one "that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems."

What's all that mean? Well, suppose that a cashier owes a customer some change and in that cashier's drawer are quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢). The problem to be solved is to decide which coins and how many of each to hand to the customer. Think of a "greedy" cashier as one who wants to take the biggest bite out of this problem as possible with each coin they take out of the drawer. For instance, if some customer is owed 41¢, the biggest first (i.e., best immediate, or local) bite that can be taken is 25¢. (That bite is "best" inasmuch as it gets us closer to 0¢ faster than any other coin would.) Note that a bite of this size would whittle what was a 41¢ problem down to a 16¢ problem, since 41 - 25 = 16. That is, the remainder is a similar but smaller problem. Needless to say, another 25¢ bite would be too big (assuming the cashier prefers not to lose money), and so our greedy cashier would move on to a bite of size 10¢, leaving him or her with a 6¢ problem. At that point, greed calls for one 5¢ bite followed by one 1¢ bite, at which point the problem is solved. The customer receives one quarter, one dime, one nickel, and one penny: four coins in total.

It turns out that this greedy approach (i.e., algorithm) is not only locally optimal but also globally so for America's currency (and also the European Union's). That is, so long as a cashier has enough of each coin, this largest-to-smallest approach will yield the fewest coins possible. How few? Well, you tell us!

Implementation Details

Implement, in a file called `cash.c` in a `~/pset1/cash` directory, a program that first asks the user how much change is owed and then prints the minimum number of coins with which that change can be made.

- Use `get_float` to get the user's input and `printf` to output your answer. Assume that the only coins available are quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢).
    - We ask that you use `get_float` so that you can handle dollars and cents, albeit sans dollar sign. In other words, if some customer is owed $9.75 (as in the case where a newspaper costs 25¢ but the customer pays with a $10 bill), assume that your program's input will be `9.75` and not `$9.75` or `975`. However, if some customer is owed $9 exactly, assume that your program's input will be `9.00` or just `9` but, again, not `$9` or `900`. Of course, by nature of floating-point values, your program will likely work with inputs like `9.0` and `9.000` as well; you need not worry about checking whether the user's input is "formatted" like money should be.
- You need not try to check whether a user's input is too large to fit in a `float`. Using `get_float` alone will ensure that the user's input is indeed a floating-point (or integral) value but not that it is non-negative.
- If the user fails to provide a non-negative value, your program should re-prompt the user for a valid amount again and again until the user complies.
- So that we can automate some tests of your code, be sure that your program's last line of output is only the minimum number of coins possible: an integer followed by `\n`.
- Beware the inherent imprecision of floating-point values. Recall `floats.c` from class, wherein, if `x` is `2`, and `y` is `10`, `x / y` is not precisely two tenths! And so, before making change, you'll probably want to convert the user's inputted dollars to cents (i.e., from a `float` to an `int`) to avoid tiny errors that might otherwise add up!
- Take care to round your cents to the nearest penny, as with `round`, which is declared in `math.h`. For instance, if `dollars` is a `float` with the user's input (e.g., `0.20`), then code like

  ```
  int cents = round(dollars * 100);
  ```

  will safely convert `0.20` (or even `0.200000002980232238769531250`) to `20`.

Your program should behave per the examples below.

```
$ ./cash
Change owed: 0.41
4
```

```
$ ./cash
Change owed: -0.41
Change owed: foo
Change owed: 0.41
4
```

Walkthrough

How to Test Your Code

Does your code work as prescribed when you input

- `-1.00` (or other negative numbers)?
- `0.00` ?
- `0.01` (or other positive numbers)?
- letters or words?
- no input at all, when you only hit Enter?

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/2020/x/cash
```

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

```
style50 cash.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you’ll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/cash
```

This is CS50x

OpenCourseWare

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.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Credit

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/credit/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

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:

   4003600000000014

   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!

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 a file called `credit.c` in a `~/pset1/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). 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
```

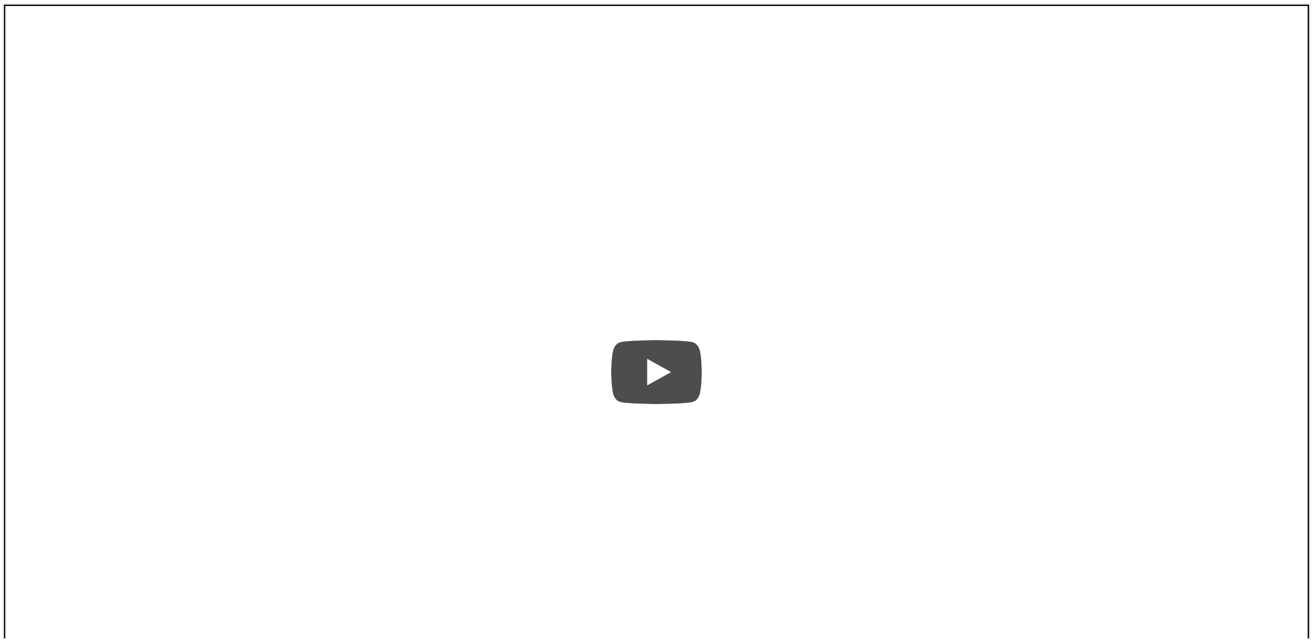
But it's up to you to catch inputs that are not credit card numbers (e.g., a phone number), even if numeric:

```
$ ./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/docs/classic/payflow/payflow-pro/payflow-pro-testing/#credit-card-numbers-for-testing) 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/2020/x/credit
```

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

```
style50 credit.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/credit
```