Problem Set 1: C

This is CS50. Harvard University. Fall 2015.

Table of Contents

This is the Hacker Edition of Problem Set 1. It cannot be submitted for credit.

Objectives

- Get comfortable with Linux.
- Start thinking more carefully.
- Solve some problems in C.

Recommended Reading

- Pages 1 – 7, 9, and 10 of http://www.howstuffworks.com/c.htm.
- Chapters 1 – 6 of **Programming in C**.

diff pset1 hacker1

- Hacker Edition plays with credit cards instead of coins.
- Hacker Edition demands two half pyramids.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of the course's material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on quizzes is not permitted at all. Collaboration on the course's final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from the course's heads. Acts considered not reasonable by the course are handled harshly. If the course refers some matter for disciplinary action and the outcome is punitive, the course reserves the right to impose local sanctions on top of that outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

If you commit some act that is not reasonable but bring it to the attention of the course's heads within 72 hours, the course may impose local sanctions that may include an unsatisfactory or failing grade for work submitted, but the course will not refer the matter for further disciplinary action except in cases of repeated acts.

Reasonable

- Communicating with classmates about problem sets' problems in English (or some other spoken language).

- Discussing the course's material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code at office hours, elsewhere, or even online, as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past semesters' quizzes and solutions thereto.
- Sending or showing code that you've written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own code online so that others might help you identify and fix a bug.
- Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.
- Whiteboarding solutions to problem sets with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set's problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate a solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz.

- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Getting Started

Recall that CS50 IDE is a web-based "integrated development environment" that allows you to program "in the cloud," without installing any software locally. Underneath the hood is a popular operating system, Ubuntu Linux, that's been "containerized" with open-source software called Docker, that allows multiple users (like you!) to share the operating system's "kernel" (its nucleus, so to speak) and files, even while having files of their own. 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). Anyhow, more on all that another time!

Logging In

Head to cs50.io[1] and log into CS50 IDE. Upon logging in for the very first time, you should be informed that CS50 IDE (aka Cloud9, the software that underlies CS50 IDE) is "creating

[1] https://cs50.io/

your workspace" and "creating your container," which might take a moment. (If you already have an account at c9.io, which is Cloud9's own site, drop sysadmins@cs50.harvard.edu[2] a note, and we'll let you know how to "link" that account.)

Once you see your workspace, which should resemble mine from Week 1, close any tabs that might have been opened for you by default (e.g., **Welcome** and **README.md**). Then decide which "theme" you'd like. By default, CS50 IDE is configured with a theme called "Cloud9 Day." If you'd prefer a darker theme, particularly at night, visit **Support > Welcome Page**, which should open a * Welcome* tab, and then, within that tab, change **Main Theme** to **Cloud9 Classic Dark Theme**, and then close the tab. Feel free to poke around other menus as well to get a sense of CS50 IDE's user interface (UI). No worries if you're not sure what most of the menus do (yet!). If among those more comfortable or just curious, feel free to uncheck **View > Less Comfortable**, which will reveal even more options.

If you have particularly slow, expensive, or infrequent Internet access, see https://cs50.readme.io/v2015/docs/offline for instructions on how to run your own copy of CS50 IDE offline on your own Mac or PC.

Updating

Toward the bottom of CS50 IDE's UI is a "terminal window" (in a tab called **Terminal**), 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. You should find that its "prompt" looks like

```
username:~/workspace $
```

where `username` is your own (automatically assigned) username. Click inside of that terminal window and then type

```
update50
```

followed by Enter to ensure that your workspace is up-to-date. It should take just a few moments for any updates to complete. (Be sure not to close the tab or CS50 IDE itself until they do!)

[2] mailto:sysadmins@cs50.harvard.edu

Hello

Okay, let's create a folder in which your code for this problem set will soon live. Toward CS50 IDE's top-left corner, control-click or right-click **~/workspace**, your workspace's "root", and select **New Folder**. A new folder called **New Folder** should appear; rename it **hacker1**, then hit Enter. (If you misname it or can't seem to edit its name, control-click or right-click the new folder and select **Rename** to try again!)

Next, control-click or right-click that **hacker1** folder that you just created and select **New File**. A new file called **Untilted** should appear; rename it **hello.txt** and then hit Enter. (If you misname it or can't seem to edit its name, control-click or right-click the new file and select **Rename** to try again!) You should see that **hello.txt** is indented immediately beneath **hacker1**, which indicates that the former is inside of the latter. If **hello.txt** doesn't appear to be inside of **hacker1**, simply drag and drop the former into the latter.

Double-click **hello.txt**, and a new tab should appear within CS50 IDE via which you can edit the file. Go ahead and type something simple (e.g., `hello` or the ever-popular `asdf`). Notice that an asterisk (**) should then appear atop the tab, to the left of the tab's name. That asterisk indicates that your file has changed but not yet been saved. Go ahead and save the file via *File > Save** or, more quickly, via command-S (on Macs) or control-S (on PCs). The asterisk should disappear.

Okay, now let's confirm via your terminal window that the file is indeed where it should be and start to familiarize you with that terminal window's command-line interface. As before, your terminal window's prompt should be

```
username:~/workspace $
```

by default, where `username` is, again, your assigned username. The prompt further indicates that `workspace` is the name of your "current working directory" (i.e., the folder that's currently open within that command-line environment). The tilde (`~`), meanwhile, refers to your account's "home directory," in which your `workspace` directory lives, but more on that another time. Note that this `workspace` directory is identical to that **~/workspace** icon in CS50 IDE's top-left corner (inside of which you created **hacker1** earlier).

Okay, let's poke around. Again click somewhere inside of that terminal window and then type

```
ls
```

followed by Enter. That's a lowercase L and a lowercase S, which is shorthand notation for "list." Indeed, you should then see a list of the folders inside of your workspace, among which is `hacker1`! Let's open that folder. Type

```
cd hacker1
```

or even, more verbosely,

```
cd ~/workspace/hacker1
```

followed by Enter to change your directory to `~/hacker1` (ergo, `cd`). You should find that your prompt changes to

```
username:~/workspace/hacker1 $
```

confirming that you are indeed now inside of `~/workspace/hacker1` (i.e., a directory called `hacker1` inside of a directory called `workspace` inside of your home directory). Now type

```
ls
```

followed by Enter. You should see `hello.txt`! Now, you can't click or double-click on that file's name there; it's just text. But that listing does confirm that `hello.txt` is where we hoped it would be. (If not, take another stab at these steps or simply ask classmates or staff for some help!)

Let's poke around a bit more. Go ahead and type

```
cd
```

and then Enter. If you don't provide `cd` with a "command-line argument" (i.e., a directory's name), it whisks you back to your home directory by default. Indeed, your prompt should now be:

```
username:~ $
```

To get back into `hacker1`, type

```
cd workspace
```

and then Enter followed by

```
cd hacker1
```

and then Enter. Alternatively, you can combine both steps into one by typing

```
cd workspace/hacker1
```

followed by Enter. Phew. Make sense? If not, no worries; it soon will! It's in this terminal window that you'll soon be compiling your first program!

Hello, C

First, a hello from Zamyla if you'd like a tour of what's to come, particularly if less comfortable. Note that she's using the CS50 Appliance, the (non-web-based) predecessor of CS50 IDE, but not a problem. Any code she writes within the CS50 Appliance should work the same within CS50 IDE!

https://www.youtube.com/watch?v=HkQD6aw7oDc

Shall we have you write your first program? Inside of your **hacker1** folder, create a new file called **hello.c**, and then open that file in a tab. (Remember how?) Be sure to capitalize the file's name just as we have; files' and folders' names in Linux are "case-sensitive." 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. 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.

Do be sure that you type in this program just right, else you're about to experience your first bug! In particular, capitalization matters, so don't accidentally capitalize words (unless they're between those two quotes). And don't overlook that one semicolon. C is quite nitpicky!

When done typing, select **File > Save** (or hit command- or control-s), but don't quit. Recall that the leading asterisk in the tab's name should then disappear. Click anywhere in the terminal window beneath your code, and be sure that you're inside of `~/workspace/hacker1`. (Remember how? If not, type `cd` and then Enter, followed by `cd workspace/hacker1` and then Enter.) Your prompt should be:

```
username:~/workspace/hacker1 $
```

Let's confirm that `hello.c` is indeed where it should be. Type

```
ls
```

followed by Enter, and you should see both `hello.c` and `hello.txt`? If not, no worries; you probably just missed a small step. Best to restart these past several steps or ask for help!

Assuming you indeed see `hello.c`, let's try to compile! Cross your fingers and then type

```
make hello
```

at the prompt, followed by Enter. (Well, maybe don't cross your fingers whilst typing.) To be clear, type only `hello` here, not `hello.c`. If all that you see is another, identical prompt, that means it worked! Your source code has been translated to object code (0s and 1s) that you can now execute. Type

```
./hello
```

at your prompt, followed by Enter, and you should see the below:

```
hello, world
```

And if you type

```
ls
```

followed by Enter, you should see a new file, `hello`, alongside `hello.c` and `hello.txt`. The first of those files, `hello`, should have an asterisk after its name that, in this context, means it's "executable," a program that you can execute (i.e., run).

If, though, upon running `make`, you instead see some error(s), it's time to debug! (If the terminal window's too small to see everything, click and drag its top border upward to increase its height.) If you see an error like expected declaration or something no less mysterious, odds are you made a syntax error (i.e., typo) by omitting some character or adding something in the wrong place. Scour your code for any differences vis-à-vis the template above. It's easy to miss the slightest of things when learning to program, so do compare your code against ours character by character; odds are the mistake(s) will jump out! Anytime you make changes to your own code, just remember to re-save via **File > Save** (or command- or control-s), then re-click inside of the terminal window, and then re-type

```
make hello
```

at your prompt, followed by Enter. (Just be sure that you are inside of `~/workspace/hacker1` within your terminal window, as your prompt will confirm or deny.) If you see no more errors, try running your program by typing

```
./hello
```

at your prompt, followed by Enter! Hopefully you now see whatever you told `printf` to print?

If not, reach out for help! Incidentally, if you find the terminal window too small for your tastes, know that you can open one in a bigger tab by clicking the circled plus (+) icon to the right of your `hello.c` tab.

Woo hoo! You've begun to program!

CS50 Check

Now let's see if the program you just wrote is correct! Included in CS50 IDE is `check50`, a command-line program with which you can check the correctness of (some of) your programs.

If not already there, navigate your way to `~/workspace/hacker1` by executing the command below.

```
cd ~/workspace/hacker1
```

If you then execute

```
ls
```

you should see, at least, `hello.c`. Be sure it's indeed spelled `hello.c` and not `Hello.c`, `hello.C`, or the like. If it's not, know that you can rename a file by executing

```
mv source destination
```

where `source` is the file's current name, and `destination` is the file's new name. For instance, if you accidentally named your program `Hello.c`, you could fix it as follows.

```
mv Hello.c hello.c
```

Okay, assuming your file's name is definitely spelled `hello.c` now, go ahead and execute the below. Note that `2015.fall.hacker1.hello` is just a unique identifier for this problem's checks.

```
check50 2015.fall.hacker1.hello hello.c
```

Assuming your program is correct, you should then see output like

```
:) hello.c exists
:) hello.c compiles
:) prints "hello, world\n"
```

where each green smiley means your program passed a check (i.e., test). You may also see a URL at the bottom of `check50`'s output, but that's just for staff (though you're welcome to visit it).

If you instead see yellow or red smileys, it means your code isn't correct! For instance, suppose you instead see the below.

```
:( hello.c exists
  \ expected hello.c to exist
:| hello.c compiles
  \ can't check until a frown turns upside down
:| prints "hello, world\n"
  \ can't check until a frown turns upside down
```

Because `check50` doesn't think `hello.c` exists, as per the red smiley, odds are you uploaded the wrong file or misnamed your file. The other smileys, meanwhile, are yellow because those checks are dependent on `hello.c` existing, and so they weren't even run.

Suppose instead you see the below.

```
:) hello.c exists
:) hello.c compiles
:( prints "hello, world\n"
  \ expected output, but not "hello, world"
```

Odds are, in this case, you printed something other than `hello, world\n` verbatim, per the spec's expectations. In particular, the above suggests you printed `hello, world`, without a trailing newline (`\n`).

Know that `check50` won't actually record your scores in CS50's gradebook. Rather, it lets you check your work's correctness *before* you submit your work. Once you actually submit your work (per the directions at this spec's end), CS50's staff will use `check50` to evaluate your work's correctness officially.

Shorts

Curl up with Nate's short on libraries and at least two other shorts for this week.

https://www.youtube.com/watch?v=ED7QtgXDShY&list=PLhQjrBD2T381NKQHUCTezeyCYzbnN4GjC

- What's a pre-processor? How does

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

  relate?
- What's a compiler?
- What's an assembler?
- What's a linker? How does

```
-lcs50
```

  relate?

Hello again, C

Before forging ahead, you might want to review some of the examples that we looked at in Week 1's lectures and take a look at a few more, the "source code" for which can be found under **Lectures** on the course's website. Allow me to take you on a tour, though feel free to forge ahead on your own if you'd prefer. Not to worry if your appliance also looks a bit different from mine.

https://www.youtube.com/watch?v=bQnyxpf0vk0&list=PLhQjrBD2T383fi16gN97XlrTwdxDq2QWZ

Smart Water

Suffice it to say that the longer you shower, the more water you use. But just how much? Even if you have a "low-flow" showerhead, odds are your shower spits out 1.5 gallons of water per minute. A gallon, meanwhile, is 128 ounces, and so that shower spits out 1.5 ×

128 = 192 ounces of water per minute. A typical bottle of water (that you might have for a drink, not a shower), meanwhile, might be 16 ounces. So taking a 1-minute shower is akin to using 192 ÷ 16 = 12 bottles of water. Taking (more realistically, perhaps!) a 10-minute shower, then, is like using 120 bottles of water. Deer Park, that's a lot of water! Of course, bottled water itself is wasteful; best to use reusable containers when you can! But it does put into perspective what's being spent in a shower!

Write, in a file called `water.c` in your `~/workspace/pset1` directory, a program that prompts the user for the length of his or her shower in minutes (as a positive integer, re-prompting as needed) and then prints the equivalent number of bottles of water (as an integer), per the sample output below, wherein underlined text represents some user's input.

```
username:~/workspace/pset1 $ ./water
minutes: what
Retry: oh
```

```
Retry: -1
minutes: 0
minutes: 10
bottles: 120
```

No need to worry about overflow, but do ensure that the user inputs a positive number of minutes!

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2015.fall.hacker1.water water.c
```

And if you'd like to play with the staff's own implementation of `water` within CS50 IDE, you may execute the below.

```
~cs50/hacker1/water
```

Bad Credit

Odds are you have a credit card in your wallet. Though perhaps the bill does not (yet) get sent to you! That card has a number, both printed on its face and embedded (perhaps with some other data) in the magnetic stripe on back. That number is 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, ahem, a quadrillion.)

Now that's a bit of an exaggeration, because credit card numbers actually have some structure to them. American Express numbers all start with 34 or 37; MasterCard numbers all start with 51, 52, 53, 54, or 55; and Visa numbers all 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. (Consider the awkward silence you may have experienced

at some point whilst paying by credit card at a store whose computer uses a dial-up modem to verify your card.) 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.

So what's the secret formula? Well, most cards use an algorithm invented by Hans Peter Luhn, a nice fellow from 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 Daven's AmEx: 378282246310005.

1. For the sake of discussion, let's first underline every other digit, starting with the number's second-to-last digit:
   3<u>7</u>8<u>2</u>8<u>2</u>2<u>4</u>6<u>3</u>1<u>0</u>0<u>0</u>5

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

   7•2 + 2•2 + 2•2 + 4•2 + 3•2 + 0•2 + 0•2

   That gives us:

   14 + 4 + 4 + 8 + 6 + 0 + 0

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

   1 + 4 + 4 + 4 + 8 + 6 + 0 + 0 = 27
2. Now let's add that sum (27) to the sum of the digits that weren't multiplied by 2:
   27 + 3 + 8 + 8 + 2 + 6 + 1 + 0 + 5 = 60
3. Yup, the last digit in that sum (60) is a 0, so Daven'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.

In `credit.c`, 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, and that `main` always return `0`. 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 `GetLongLong` from CS50's library to get users' input. (Why?)

Of course, to use `GetLongLong`, you'll need to tell `clang` about CS50's library. Be sure to put

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

toward the top of `credit.c`. And be sure to compile your code with a command like the below.

```
clang -o credit credit.c -lcs50
```

Note that `-lcs50` must come at this command's end because of how clang works.

Incidentally, recall that `make` can invoke `clang` for you and provide that flag for you, as via the command below.

```
make credit
```

Assuming your program compiled without errors (or, ideally, warnings) via either command, you can run your program with the command below.

```
./credit
```

Consider the below representative of how your own program should behave when passed a valid credit card number (sans hyphens), wherein underlined text represents some user's input.

```
username:~/workspace/pset1 $ ./credit
Number: 378282246310005
AMEX
```

Of course, `GetLongLong` itself will reject hyphens (and more) anyway:

```
username:~/workspace/pset1 $ ./credit
Number: 3782-822-463-10005
Retry: foo
Retry: 378282246310005
AMEX
```

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

```
username:~/workspace/pset1 $ ./credit
Number: 7722574501
INVALID
```

Test out your program with a whole bunch of inputs, both valid and invalid. (We certainly will!) Here are a few card numbers that PayPal recommends for testing:

https://www.paypalobjects.com/en_US/vhelp/paypalmanager_help/credit_card_numbers.htm

Google (or perhaps a roommate's wallet) should turn up more. (If your roommate asks what you're doing, don't mention us.) If your program behaves incorrectly on some inputs (or doesn't compile at all), time to debug!

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2015.fall.hacker1.credit credit.c
```

And if you'd like to play with the staff's own implementation of `credit` within CS50 IDE, you may execute the below.

```
~cs50/hacker1/credit
```

Itsa Mario

Toward the beginning of World 1-1 in Nintendo's Super Mario Brothers, Mario must hop over two "half-pyramids" of blocks as he heads toward a flag pole. Below is a screenshot.

Write, in a file called `mario.c` in your `~/workspace/hacker1` directory, a program that recreates these half-pyramids using hashes (`#`) for blocks. However, to make things more interesting, first prompt the user for the half-pyramids' heights, a non-negative integer no greater than `23`. (The height of the half-pyramids pictured above happens to be `4`, the width of each half-pyramid `4`, with an a gap of size `2` separating them.) Then, generate (with the help of `printf` and one or more loops) the desired half-pyramids. Take care to left-align the bottom-left corner of the left-hand half-pyramid, as in the sample output below, wherein boldfaced text represents some user's input.

```
username:~/workspace/pset1 $ ./mario
Height: 4
   #  #
  ##  ##
 ###  ###
####  ####
```

No need to generate the bricks, cloud, numbers, or text in the sky or Mario himself. Just the half-pyramids! And be sure that main returns `0`.

We leave it to you to determine how to compile and run this particular program!

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 2015.fall.hacker1.mario mario.c
```

And if you'd like to play with the staff's own implementation of `mario` within CS50 IDE, you may execute the below.

```
~cs50/hacker1/mario
```