# This is CS50x.

# Problem Set 1: C

**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.

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

## diff pset1 hacker1.

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

## Getting Started.

- Take CS50x.
- Recall that the CS50 Appliance is a "virtual machine" (running an operating system called Fedora, which itself is a flavor of Linux) that you can run inside of a window on your own computer, whether you run Windows, Mac OS, or even Linux itself. To do so, all you need is a "hypervisor" (otherwise known as a "virtual machine monitor"), software that tricks the appliance into thinking that it's running on "bare metal."

  Alternatively, you could buy a new computer, install Fedora on it (i.e., bare metal), and use that! But a hypervisor lets you do all that for free with whatever computer you already have. Plus, the CS50 Appliance is pre-configured for CS50, so, as soon as you install it, you can hit the ground running.

  So let's get a hypervisor and the CS50 Appliance installed on your computer. Head to

https://manual.cs50.net/Appliance#How_to_Install_Appliance

where instructions await. In particular, if running Mac OS, follow the instructions for VMware Fusion. If running Windows or Linux, follow the instructions for VMware Player.

If you run into any problems, head to CS50 Discuss to search or post questions!

- Once you have the CS50 Appliance installed, go ahead and start it (per those same instructions). A small window should open, inside of which the appliance should boot. A few seconds or minutes later, you should find yourself logged in as John Harvard (whose username is **jharvard** and whose password is **crimson**), with John Harvard's desktop before you.

  **If you find that the appliance runs unbearably slow on your PC, particularly if several years old or a somewhat slow netbook, or if you see a hint about "long mode," try the instructions at**

  **https://manual.cs50.net/Virtualization**

  **and let us know via CS50 Discuss if you still need a hand.**

  Feel free to poke around, particularly the CS50 Menu in the appliance's bottom-left corner. You should find the graphical user interface (GUI), called Xfce, reminiscent of both Mac OS and Windows. Linux actually comes with a bunch of GUIs; Xfce is just one. If you're already familiar with Linux, you're welcome to install other software via **Menu > Administration > Add/Remove Software**, but the appliance should have everything you need for now. You're also welcome to play with the appliance's various features, per the instructions at

  https://manual.cs50.net/Appliance#How_to_Use_Appliance

  but this problem set will explicitly mention anything that you need know or do.

- Even if you just downloaded the appliance, ensure that it's completely up-to-date by opening a terminal window, as via **Menu > Programming > Terminal**, typing

  ```
  update50
  ```

  and then hitting Enter on your keyboard. So long as your computer (and, thus, the appliance) has Internet access, the appliance should proceed to download and install any available updates.

- Next, follow the instructions at

  https://manual.cs50.net/Appliance#How_to_Enable_Dropbox

  to configure the appliance to use Dropbox so that your work is automatically backed up, just in case something goes wrong with your appliance. (If you really don't want to use Dropbox, that's fine, but realize your files won't be backed up as a result!) If you don't yet have a Dropbox account, sign up when prompted for the free (2 GB) plan. You're welcome to install Dropbox on your own computer as well (outside of the appliance), per https://www.dropbox.com/install, but no need if you'd rather not; just inside the appliance is fine.

If you're already a Dropbox user but don't want your personal files to be synched into the appliance, simply enable **Selective Sync**, per the CS50 Manual's instructions.

**If Dropbox is blocked in your country, not to worry. Simply skip this step.** Your appliance will still have a folder called **Dropbox**; it just won't be backed up to Dropbox's servers.

- Okay, let's create a folder (otherwise known as a "directory") in which your code for this problem set will soon live. Go ahead and double-click **Home** on John Harvard's desktop (in the appliance's top-left corner). A window entitled **Home** should appear, indicating that you're inside of John Harvard's "home directory" (i.e., personal folder). Then double-click the folder called **Dropbox**, at which point the window's title should change to **Dropbox**. Next select **File > Create New Folder...**, at which point a new folder called **Untitled Folder** should appear. Rename it **hacker1** (in all lowercase, with no spaces). (If the folder's name doesn't seem to be editable, click the **Untitled Folder** once to highlight it, then select **Edit > Rename**, at which point its name should become editable.) Then double-click that **hacker1** folder to open it. The window's title should change to **hacker1**, and you should see an otherwise empty folder (since you just created it). Notice, though, that atop the window are three buttons, **Home**, **Dropbox**, and **hacker1**, that indicate where you were and where you are; you can click buttons like those to navigate back and forth easily.

- Okay, go ahead and close any open windows, then select **Menu > Programming > gedit**. (Recall that the CS50 Menu is in the appliance's bottom-left corner.) A window entitled **Unsaved Document 1 - gedit** should appear, inside of which is a tab entitled **Unsaved Document 1**. Clearly the document is just begging to be saved. Go ahead and type `hello` (or the ever-popular `asdf`) in the tab, and then notice how the tab's name is now prefixed with an asterisk (*), indicating that you've made changes since the file was first opened. Select **File > Save**, and a window entitled **Save As...** should appear. Input `hello.txt` next to **Name**, then click **jharvard** under **Places**. You should then see the contents of John Harvard's home directory. Double-click **Dropbox**, then double-click **hacker1**, and you should find yourself inside that empty folder you created. Now, at the bottom of this same window, you should see that the file's default **Character Encoding** is **Current Locale (UTF-8)** and that the file's default **Line Ending** is **Unix/Linux**. No need to change either; just notice they're there. That the file's **Line Ending** is **Unix/Linux** just means that `gedit` will insert (invisibly) `\n` at the end of any line of text that you type. Windows, by contrast, uses `\r\n`, and Mac OS uses `\r`, but more on those details some other time.

- Okay, click **Save** in the window's bottom-right corner. The window should close, and you should see that the original window's title is now **hello.txt (~/Dropbox/hacker1) - gedit**. The parenthetical just means that **hello.txt** is inside of **hacker1**, which is inside of **Dropbox**, which is inside of **~**, which is shorthand notation for John Harvard's home directory. A useful reminder is all. The tab, meanwhile, should now be entitled **hello.txt** (with no asterisk, unless you accidentally hit the keyboard again).

- Okay, with `hello.txt` still open in `gedit`, notice that beneath your document is a "terminal window," a command-line (i.e., text-based) interface via which you can navigate the appliance's hard drive and run programs (by typing their name). Notice that the window's "prompt" is

```
jharvard@appliance (~):
```

which means that you are logged into the appliance as John Harvard and that you are currently inside of **~** (i.e., John Harvard's home directory). If that's the case, there should be a **Dropbox** directory somewhere inside. Let's confirm as much.

Click somewhere inside of that terminal window, and the prompt should start to blink. Type

```
ls
```

and then 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 John Harvard's home directory, among which is **Dropbox**! Let's open that folder, followed immediately by the **hacker1** folder therein. Type

```
cd Dropbox/hacker1
```

or even

```
cd ~/Dropbox/hacker1
```

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

```
jharvard@appliance (~/Dropbox/hacker1):
```

confirming that you are indeed now inside of **~/Dropbox/hacker1** (i.e., a directory called **hacker1** inside of a directory called **Dropbox** inside of John Harvard's 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.

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:

```
jharvard@appliance (~):
```

Phew, home sweet home. Make sense? If not, no worries; it soon will! It's in this terminal

window that you'll soon be compiling your first program! For now, though, close `gedit` (via **File > Quit**) and, with it, **hello.txt**.

- Incidentally, if the need arises, know that you can transfer files to and from the appliance per the instructions below.

  https://manual.cs50.net/Appliance#How_to_Transfer_Files_between_Appliance_and_Your_Computer

# hello, world!

- Shall we have you write your first program?

  Okay, go ahead and launch `gedit`. (Remember how?) You should find yourself faced with another **Unsaved Document 1**. Go ahead and save the file as `hello.c` (not `hello.txt`) inside of `hacker1`, just as before. (Remember how?) Once the file is saved, the window's title should change to **hello.c (~/Dropbox/hacker1) - gedit**, and the tab's title should change to **hello.c**. (If either does not, best to close `gedit` and start fresh! Or ask for help!)

  Go ahead and write your first program by typing these lines into the file (though you're welcome to change the words between quotes to whatever you'd like):

```
#include <stdio.h>

int main(void)
{
    printf("hello, world!\n");
    return 0;
}
```

  Notice how `gedit` 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 `gedit` to make certain syntax stand out. Had you not saved the file as `hello.c` from the start, `gedit` 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 ctrl-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 its prompt should start blinking. But odds are the prompt itself is just

```
jharvard@appliance (~):
```

  which means that, so far as the terminal window's concerned, you're still inside of John Harvard's home directory, even though you saved the program you just wrote inside of `~/Dropbox/hacker1` (per the top of `gedit`'s window). No problem, go ahead and type

```
cd Dropbox/hacker1
```

or

```
cd ~/Dropbox/hacker1
```

at the prompt, and the prompt should change to

```
jharvard@appliance (~/Dropbox/hacker1):
```

in which case you're where you should be! Let's confirm that `hello.c` is there. Type

```
ls
```

at the prompt 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 0s and 1s that you can now execute. Type

```
./hello
```

at your prompt, followed by Enter, and you should see whatever message you wrote between quotes in your code! Indeed, if you type

```
ls
```

followed by Enter, you should see a new file, `hello`, alongside `hello.c` and `hello.txt`.

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 ctrl-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 `~/Dropbox/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 the greeting you wrote? If not, reach out to CS50 Discuss for help!

Incidentally, if you find `gedit`'s built-in terminal window too small for your tastes, know that you can open one in its own window via **Menu > Programming > Terminal**. You can then alternate between `gedit` and `Terminal` as needed, as by clicking either's name along the appliance's bottom.

Woo hoo! You've begun to program!

# This is CS50 Check.

- Now let's see if the program you just wrote is correct! Included in the CS50 Appliance 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 `~/Dropbox/hacker1` by executing the command below.

  ```
  cd ~/Dropbox/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.

  ```
  check50 2012/hacker1/hello hello.c
  ```

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

```
Compressing... Compressed.
Uploading.... Uploaded.
Checking...... Checked.
:) hello.c exists
:) hello.c compiles
:) prints "hello, world\n"
:) main returns 0
```

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
:| main returns 0
   \ 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"
   \ wasn't expecting "hello, world"
:) main returns 0
```

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 `\n`.

Know that `check50` won't actually record your scores in CS50 Gradebook.

# A Section of Questions.

You're welcome to dive into these questions on your own, but know that they're also covered in Week 1's more-comfortable section!

- Recall that "style" generally refers to source code's aesthetics, the extent to which code is readable (i.e., commented and indented with variables aptly named). Odds are you didn't have to give too much thought to style when writing `hello.c`, given its brevity, but you're about to start writing programs where you'll need to make some stylistic decisions.

  Before you do, read over CS50's Style Guide:

  https://manual.cs50.net/Style

  Then head to

  https://www.edx.org/courses/HarvardX/CS50x/2012/courseware/Week_1/shorts1/

  and watch the short on compilers. Be sure you're reasonably comfortable answering the below when it comes time to submit this problem set's form!

  - 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?

- Using the CS50 Appliance or CS50 Run (which is similar to CS50 Spaces), write a program that prompts the user for a lowercase letter and then converts it to uppercase without using bitwise operations, as per the sample output below, wherein boldfaced text represents some user's input. Consider this problem an opportunity to practice; you won't be asked to submit this program.

  ```
  jharvard@run.cs50.net (~): ./a.out
  a
  A
  ```

- Using the CS50 Appliance or CS50 Run, write another program that prompts the user for a lowercase letter and then converts it to uppercase using bitwise operations, as per the sample output below, wherein boldfaced text represents some user's input. Consider this problem an opportunity to practice; you won't be asked to submit this program.

  ```
  jharvard@run.cs50.net (~): ./a.out
  a
  A
  ```

- Using the CS50 Appliance or CS50 Run, write a program that prompts the user for an non-negative integer and then displays it in binary without leading 0s, as per the sample output below, wherein boldfaced text represents some user's input. Consider this problem an opportunity to practice; you won't be asked to submit this program.

```
jharvard@run.cs50.net (~): ./a.out
50
110010
```

# 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 Nate'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:

3782822463100 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 Nate'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); highlighted in bold is some user's input.

```
jharvard@appliance (~/Dropbox/hacker1): ./credit
Number: 378282246310005
AMEX
```

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

```
jharvard@appliance (~/Dropbox/hacker1): ./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., Nate's phone number), even if numeric:

```
jharvard@appliance (~/Dropbox/hacker1): ./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 2012/hacker1/credit credit.c
```

And if you'd like to play with the staff's own implementation of `credit` in the appliance, 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 `~/Dropbox/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.

```
jharvard@appliance (~/Dropbox/hacker1): ./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 2012/hacker1/mario mario.c
```

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

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