Problem Set 1: C

---

due Thu 9/19 at noon

Questions? Head to cs50.net/discuss (https://www.cs50.net/discuss) or join classmates at office hours (https://www.cs50.net/ohs)!

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 (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 to the Administrative Board and the outcome is Admonish, Probation, Requirement to Withdraw, or Recommendation to Dismiss, 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.

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 on CS50 Discuss or elsewhere 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 in CS50 Vault 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.
- 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 your 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.
- Redeeming or attempting to redeem someone else's code for a late day.
- 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.

Scores

Your work on this problem set will be evaluated along four axes primarily.

Scope

To what extent does your code implement the features required by our specification?

Correctness

To what extent is your code consistent with our specifications and free of bugs?

Design

To what extent is your code written well (i.e., clearly, efficiently, elegantly, and/or logically)?

Style

To what extent is your code readable (i.e., commented and indented with variables aptly named)?

All students, whether taking the course SAT/UNS or for a letter grade, must ordinarily submit this and all other problem sets to be eligible for a satisfactory grade unless granted an exception in writing by the course's heads.

Getting Started

- 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/19/#how_to_install_appliance (https://manual.cs50.net/appliance/19/#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.

- 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 (https://manual.cs50.net/virtualization) and let us know if you still need a hand.

  Feel free to poke around, particularly the 50 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/19/#how_to_use_appliance (https://manual.cs50.net/appliance/19/#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/19/#how_to_synchronize_files_with_dropbox (https://manual.cs50.net/appliance/19/#how_to_synchronize_files_with_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 (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.

- 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 **New Folder** under the gear icon in the window's top-right corner, 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, control-click (i.e., click while holding your keyboard's control key) the **Untitled Folder** once, then select **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 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 **Unicode (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 at https://manual.cs50.net/appliance/19/#how_transfer_files_between_appliance_and_your_computer (https://manual.cs50.net/appliance/19/#how_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");
}
```

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 precisely the below?

```
hello, world
```

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. Note that `2013.hacker1.hello` is just a unique identifier for this problem's checks.

  ```
  check50 2013.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.

This is CS50 Style.

- In addition to `check50`, the CS50 Appliance comes with `style50`, a tool with which you can evaluate your code's style. To run it on, say, `hello.c`, execute the below:

```
style50 hello.c
```

You should see zero or more lines of suggestions. Yellow smileys indicate warnings that you should consider addressing. Red smileys indicate errors that you should definitely address.

Late Day

- Alright, odds are it took bit of time to get to this point, even if `hello` itself didn't. Let's give you a shot at a late day. Assuming you got `hello` to work and `check50` outputted only green smileys, go ahead and open a terminal window via **Menu > Programming > Terminal**. A big, black window should open (bigger than the one embedded in `gedit`). Navigate your way to `~/Dropbox/hacker1`, as with

  ```
  cd Dropbox/hacker1
  ```

  and then re-run `check50` as follows:

  ```
  check50 -c 2013.hacker1.hello hello.c
  ```

  Notice the addition of `-c` this time, which is a "flag" or a "switch" (aka a "command-line argument") that signifies that you'd like `check50` to output a "coupon code" for you instead of some smileys. (The coupon code will be 32 hexadecimal digits. Remember those?) Go ahead and highlight the code that you see, select **Edit > Copy**, then visit https://www.cs50.net/coupons/1 (https://www.cs50.net/coupons/1) using Chrome inside of the appliance, paste the code where prompted, as with control-v on your keyboard, or type it manually, and then click **Redeem**. You should be rewarded with an extension of 24 hours. If the coupon's not accepted, be sure that your program's indeed correct, as by re-running `check50` without `-c` as follows:

  ```
  check50 2013.hacker1.hello hello.c
  ```

  If you don't see green smileys, and only green smileys, best to fix any bugs! Re-run `check50` with `-c` when ready to generate a new coupon. You may (re-)submit coupons as many times as you'd like, but you'll be rewarded with no more than one 24-hour extension per problem set.

Shorts

- Head to https://www.cs50.net/shorts/1 (https://www.cs50.net/shorts/1) and curl up with Nate's short on libraries. 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?

- Curl up with at least two other shorts at https://www.cs50.net/shorts/1 (https://www.cs50.net/shorts/1). Some additional questions may be in your future!

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:

   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>5</u>

   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., Lauren'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 (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.

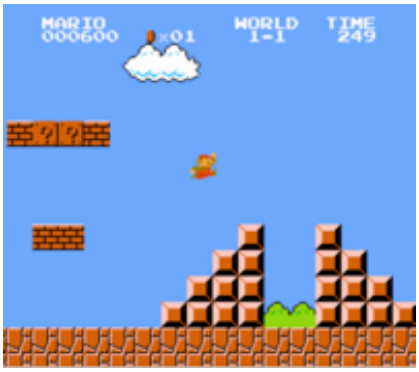
```
check50 2013/hacker1/credit credit.c
```

And if you'd like to play with the staff's own implementation of `credit` in the appliance, you may execute the below **starting Tue 9/17**.

```
~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 2013.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 **starting Tue 9/17**.

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

How to Submit

Step 1 of 2

- When ready to submit, open up Chrome *inside* of the appliance (not on your own computer) and visit cs50.net/submit (https://www.cs50.net/submit), logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **pset1** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
- Navigate your way to `hello.c`, as by clicking **jharvard**, then double-clicking **Dropbox**, then double-clicking **hacker1**, assuming you saved `hello.c` in `~/Dropbox/hacker1`. Once you find `hello.c`, click it once to select it, then click **Open**.
- Click **Add files…** again, and a window entitled **Open Files** should appear again.
- Navigate your way to `credit.c` as before. Click it once to select it, then click **Open**.
- Navigate your way to `mario.c` as before. Click it once to select it, then click **Open**.
- Click **Start upload** to upload all of your files at once to CS50's servers.
- On the screen that appears, you should see a window with **No File Selected**. If you move your mouse toward the window's lefthand side, you should see a list of the files you uploaded. Click each to confirm the contents of each. (No need to click any other buttons or icons.) If confident that you submitted the files you intended, consider your source code submitted! If you'd like to re-submit different (or modified) files, simply return to cs50.net/submit (https://www.cs50.net/submit) and repeat these steps. You may re-submit as many times as you'd like; we'll grade your most recent submission, so long as it's before the deadline.

Step 2 of 2

- Head to https://forms.cs50.net/2013/fall/psets/1/ (https://forms.cs50.net/2013/fall/psets/1/) where a short form awaits. Once you have submitted that form (as well as your source code), you are done!

  This was Problem Set 1.

Last updated 2013-10-30 17:28:27 EDT