Problem Set 1: C

due by noon on Thu 9/20

Per the directions at this document's end, submitting this problem set involves submitting source code as well as filling out a Web-based form, which may take a few minutes, so best not to wait until the very last minute, lest you spend a late day unnecessarily.

Goals.

- 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 – 5, 9, and 11 – 17 of *Absolute Beginner's Guide to C*.
- Chapters 1 – 6 of *Programming in C*.

Academic Honesty

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed in writing by the course's instructor. Collaboration in the completion of problem sets is not permitted unless otherwise stated by some problem set's specification.

Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or posted online) or lifting material from a book, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student or soliciting the work of another individual. Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Nor may you provide or make available solutions to problem sets to individuals who take or may take this course in the future. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the course's instructor or preceptor.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the course's instructor or preceptor.

You may turn to the Web for instruction beyond the course's lectures and sections, for references, and for solutions to technical difficulties, but not for outright solutions to problems on problem sets or your own final project. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course's lectures and sections (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

All forms of academic dishonesty are dealt with harshly. If the course refers some matter to the Administrative Board and the outcome for some student is *Admonish*, *Probation*, *Requirement to Withdraw*, or *Recommendation to Dismiss*, the course reserves the right to impose local sanctions on top of that outcome for that student that may include, but not be limited to, a failing grade for work submitted or for the course itself.

Fine Print.

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 Pass/Fail or for a letter grade, must ordinarily submit this and all other problem sets to be eligible for a passing grade (*i.e.*, Pass or A to D–) unless granted an exception in writing by the course's instructor or preceptor. No more than one late day may be spent on this, or any other, problem set.

Getting Started.

☑ Take CS50.

☐ 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.[1] If running Windows or Linux, follow the instructions for VMware Player.[2]

If you run into any problems, head to `cs50.net/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 at `cs50.net/discuss` if you still need a hand.

[1] If you mentioned on Problem Set 0's form that you're a Mac user, we may have emailed you a "VMAP" username and password already to save you a step. If not, simply apply per the instructions, and we'll follow up within 24 hours. Email `sysadmins@cs50.net` to inquire if you start the problem set quite close to its deadline.
[2] You won't need a "VMAP" account to download VMware Player.

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

```
sudo yum -y update appliance50
```

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.[3] 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.

☐ 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 **pset1** (in all lowercase, with no spaces).[4] Then double-click that **pset1** folder to open it. The window's title should change to **pset1**, 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 **pset1**, that indicate where you were and where you are; you can click buttons like those to navigate back and forth easily.

[3] If you really don't want to use `dropbox.com`, that's fine, but realize your files won't be backed up as a result!
[4] 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.

☐ 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 **pset1**, 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/pset1) - gedit**. The parenthetical just means that **hello.txt** is inside of **pset1**, 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 **pset1** folder therein. Type

```
cd Dropbox/pset1
```

or even

```
cd ~/Dropbox/pset1
```

followed by Enter to change your directory to **~/Dropbox/pset1** (ergo, `cd`). You should find that your prompt changes to

```
jharvard@appliance (~/Dropbox/pset1):
```

confirming that you are indeed now inside of **~/Dropbox/pset1** (*i.e.*, a directory called **pset1** 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 **pset1**, just as before. (Remember how?) Once the file is saved, the window's title should change to **hello.c (~/Dropbox/pset1) - 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):[5]

```
#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/pset1** (per the top of gedit's window). No problem, go ahead and type

```
cd Dropbox/pset1
```

or

```
cd ~/Dropbox/pset1
```

at the prompt, and the prompt should change to

```
jharvard@appliance (~/Dropbox/pset1):
```

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

```
ls
```

[5] Do type in this program keystroke by keystroke inside of the appliance; don't try to copy/paste from the PDF! Odds are copy/paste won't work yet anyway until you install "Guest Additions," but more on those some other time!

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/pset1** 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.net/discuss` for help! In fact, if you log into `cs50.net/discuss` within the appliance itself (via **Menu > Internet > Google Chrome**), you can even attach your code to your post; just take care to flag it as private.

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!

A Section of Questions.

You're welcome to dive into these questions on your own, but know that they'll also be explored in 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.cs50.net/shorts/`

and watch the 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 library?
- ☐ What role does
  `#include <cs50.h>`
  play when you write it atop some program?
- ☐ What role does
  `-lcs50`
  play when you pass it as a "command-line argument" to `clang`?

☐ Using the CS50 Appliance or `cs50.net/spaces`, write a program that converts a temperature in Celsius to Fahrenheit, as per the sample output below, wherein boldfaced text represents some user's input. No matter how the user inputs the temperature in Celsius, take care to display Fahrenheit to one decimal place. No need to worry about floating-point imprecision. Consider this problem an opportunity to practice; you won't be asked to submit this program.

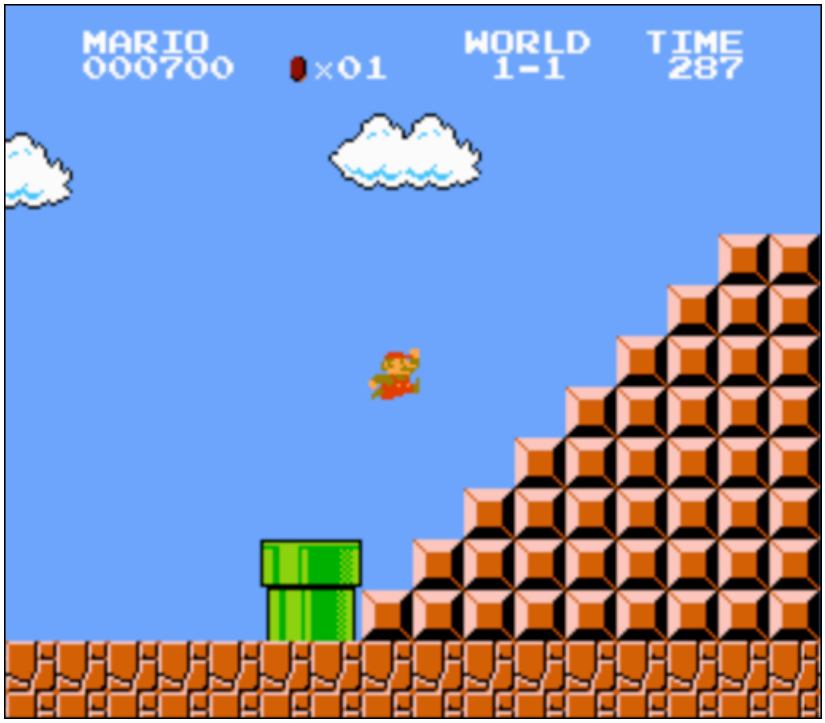
```
jharvard@run.cs50.net (~): ./a.out
C: 100
F: 212.0
```

☐ Using the CS50 Appliance or `cs50.net/spaces`, write a program that prompts the user for two integers, a non-negative numerator and a positive denominator, and then displays numerator ÷ denominator as a percentage to 2 decimal places, as per the sample output below, wherein boldfaced text represents some user's input. If the user fails to provide the desired input, the program should re-prompt the user until the user cooperates. Consider this problem another opportunity to practice; you won't be asked to submit this program.

```
jharvard@run.cs50.net (~): ./a.out
non-negative numerator: -50
non-negative numerator: 50
positive denominator: 0
positive denominator: 100
50.00%
```

Itsa Mario.

☐ Toward the end of World 1-1 in Nintendo's Super Mario Brothers, Mario must ascend a "half-pyramid" of blocks before leaping (if he wants to maximize his score) toward a flag pole. Below is a screenshot.



Write, in a file called `mario.c` in your `~/Dropbox/pset1/` directory, a program that recreates this half-pyramid using hashes (`#`) for blocks. However, to make things more interesting, first prompt the user for the half-pyramid's height, a non-negative integer no greater than 23. (The height of the half-pyramid pictured above happens to be 8.) Then, generate (with the help of `printf` and one or more loops) the desired half-pyramid. Take care to align the bottom-left

corner of your half-pyramid with the left-hand edge of your terminal window, as in the sample output below, wherein boldfaced text represents some user's input.

```
jharvard@appliance (~/Dropbox/pset1): ./mario
Height: 8
        ##
       ###
      ####
     #####
    ######
   #######
  ########
#########
```

Note that the rightmost two columns of blocks must be of the same height. No need to generate the pipe, clouds, numbers, text, or Mario himself. Just the half-pyramid! And be sure that main returns `0`.

To compile your program, remember that you can execute

```
make mario
```

or, more manually,

```
clang -o mario mario.c -lcs50
```

after which you can run your program with the below.

```
./mario
```

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

```
~cs50/pset1/mario
```

Time for Change.

☐ Speaking of money, "counting out change is a blast (even though it boosts mathematical skills) with this spring-loaded changer that you wear on your belt to dispense quarters, dimes, nickels, and pennies into your hand." Or so says the website on which we found this here accessory.[6]

Of course, the novelty of this thing quickly wears off, especially when someone pays for a newspaper with a big bill. 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."[7]

What's all that mean? Well, suppose that a cashier owes a customer some change and on that cashier's belt are levers that dispense quarters, dimes, nickels, and pennies. Solving this "problem" requires one or more presses of one or more levers. Think of a "greedy" cashier as one who wants to take, with each press, the biggest bite out of this problem as possible. 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.

[6] Description and image from `hearthsong.com`. For ages 5 and up.
[7] `http://www.nist.gov/dads/HTML/greedyalgo.html`

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.[8]

How few? Well, you tell us. Write, in a file called `greedy.c` in your `~/Dropbox/pset1/` directory, a program that first asks the user how much change is owed and then spits out the minimum number of coins with which said change can be made. Use `GetFloat` from the CS50 Library to get the user's input and `printf` from the Standard I/O library to output your answer.

We ask that you use `GetFloat` 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. And you need not try to check whether a user's input is too large to fit in a `float`. But you should check that the user's input makes cents! Er, sense. Using `GetFloat` 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. Incidentally, do beware the inherent imprecision of floating-point values.[9] Before doing any math, you'll probably want to convert the user's input entirely to cents (*i.e.*, from a `float` to an `int`) to avoid tiny errors that might otherwise add up![10] Be careful to round and not truncate your pennies![11]

So that we can automate some tests of your code, we ask that your program's last line of output be only the minimum number of coins possible: an integer followed by `\n`. Consider the below representative of how your own program should behave; highlighted in bold is some user's input.

```
jharvard@appliance (~/Dropbox/pset1): ./greedy
O hai!  How much change is owed?
0.41
4
```

By nature of floating-point values, that user could also have inputted just **`.41`**. (Were they to input **`41`**, though, they'd get many more coins!)

[8] By contrast, suppose that a cashier runs out of nickels but still owes some customer 41¢. How many coins does that cashier, if greedy, dispense? How about a "globally optimal" cashier?

[9] For instance, `0.01` cannot be represented exactly as a `float`. Try printing its value to, say, 50 decimal places, with code like the below:

```
float f = 0.01;
printf("%.50f\n", f);
```

[10] Don't just cast the user's input from a `float` to an `int`! After all, how many cents does one dollar equal?

[11] `https://www.cs50.net/resources/cppreference.com/stdmath/round.html`

Of course, more difficult users might experience something more like the below.

```
jharvard@appliance (~/Dropbox/pset1): ./greedy
O hai!  How much change is owed?
-0.41
How much change is owed?
-0.41
How much change is owed?
foo
Retry: 0.41
4
```

Per these requirements (and the sample above), your code will likely have some sort of loop. If, while testing your program, you find yourself looping forever, know that you can kill your program (*i.e.*, short-circuit its execution) by hitting ctrl-c (sometimes a lot).

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

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

```
~cs50/pset1/greedy
```

How to Submit.

In order to submit this problem set, you must first execute a command in the appliance and then submit a (brief) form online.[12]

☐ Open a terminal window (as via **Menu > Programming > Terminal** or within gedit) then execute:

```
cd ~/Dropbox/pset1
```

Then execute:

```
ls
```

At a minimum, you should see `hello.c`, `greedy.c`, and `mario.c`. If not, odds are you skipped some step(s) earlier! If you do see those files, you are ready to submit your source code to us. Execute:

```
submit50 ~/Dropbox/pset1
```

and follow the on-screen instructions. That command will essentially upload your entire `~/Dropbox/pset1` directory to CS50's servers, where your TF will be able to access it. The

[12] This one's much shorter than Problem Set 0's!

command will inform you whether your submission was successful or not. And you may inspect your submission at `cs50.net/submit`.

You may re-submit as many times as you'd like; we'll grade your most recent submission. But take care not to submit after the problem set's deadline, lest you spend a late day unnecessarily or risk rejection entirely.

If you run into any trouble at all, let us know via `cs50.net/discuss` and we'll try to assist! Just take care to seek help well before the problem set's deadline, as we can't always reply right away!

☐ Head to the URL below where a short form awaits:

`https://www.cs50.net/psets/1/`

Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 1.