Problem Set 1: C

This is CS50. Harvard University. Fall 2014.

Table of Contents

Questions? Head to cs50.harvard.edu/discuss[1] or join classmates at office hours[2]!

[1] https://cs50.harvard.edu/discuss
[2] https://cs50.harvard.edu/hours

Objectives

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

Recommended Reading

- Pages 1 – 7, 9, and 10 of http://www.howstuffworks.com/c.htm.
- Chapters 1 – 5, 9, and 11 – 17 of *Absolute Beginner's Guide to C*.
- Chapters 1 – 6 of *Programming in C*.

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.

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

Reasonable

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

Not Reasonable

- Accessing a solution 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.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problem sets.
- Failing to cite (as with comments) the origins of code or techniques that you discover outside of the course's own lessons and integrate into your own work, even while respecting this policy's other constraints.
- Giving or showing to a classmate a solution to a problem set's problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problem sets to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz's questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problem sets online or elsewhere.
- Splitting a problem set's workload with another individual and combining your work.
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.
- Submitting work to this course that you intend to use outside of the course (e.g., for a job) without prior approval from the course's heads.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem set's problem and basing your own solution on it.

Assessment

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 Ubuntu, 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 Ubuntu 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.

Installing

So let's get a hypervisor and the CS50 Appliance installed on your computer. Head to https://manual.cs50.net/appliance/2014/#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 Workstation. **Be sure to download version 2014 of the CS50 Appliance, not 19 or earlier.**

Updating

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 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 `apt-get`, 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/2014/#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 > Accessories > Terminal Emulator**, 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.

Dropboxing

Next, follow the instructions at https://manual.cs50.net/appliance/2014/#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.

Incidentally, if curious how Dropbox itself works, allow us to introduce Thomas Carriero '08 and Alex Allain '06, both former CS50 TFs!

https://www.youtube.com/watch?v=VECV6r9s5SE

File Manager

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 (toward the appliance's top-left corner). A window entitled **jharvard - File Manager** 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 - File Manager**. Next select **File > Create Folder…** in the window's top-left corner, input a name of **pset1**, and then click **Create**. (If you misname the folder, control-click the misnamed folder, select **Rename…**, enter a new name, and click **Rename**.) Then double-click that **pset1** folder to open it. The window's title should change to **pset1 - File Manager**, and you should see an otherwise empty folder (since you just created it).

gedit

Okay, go ahead and close any open windows, then select **Menu > Accessories > gedit**. (Recall that **Menu** is in the appliance's bottom-left corner.) A window entitled **Untitled Document 1 - gedit** should appear, inside of which is a tab entitled **Untitled Document 1**. Clearly the document is just begging to be saved. Go ahead and type `hello` (or the ever-popular `asdf`) on line 1 of the document, 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 **Home** 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 **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 (delightfully annoying) 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" (aka "terminal emulator"), 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 at https://manual.cs50.net/appliance/2014/#how_transfer_files_between_appliance_and_your_computer.

Hello, C

First, a hello from Zamyla if you'd like a tour of what's to come, particularly if less comfortable. Note that her version of the CS50 Appliance might look a bit different from yours, but not a problem.

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

Shall we have you write your first program? 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):

```
#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/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
```

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 object code (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 precisely the below?

```
hello, world
```

If not, reach out 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!

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/pset1` by executing the command below.

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

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 `2014.fall.pset1.hello` is just a unique identifier for this problem's checks.

```
check50 2014.fall.pset1.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.

CS50 Style

In addition to `check50`, the CS50 Appliance comes with `style50`, a tool with which you can evaluate your code's style vis-à-vis CS50's style guide[3]. 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.

If you instead see `java: command not found`, execute `sudo apt-get -y install default-jre-headless` (which will install software that we forgot to install for you!), then try again.

Note that `style50` is still a work in progress (a "beta" version, so to speak), so best to consult CS50's style guide[4] for official guidance.

Shorts

Head to https://cs50.harvard.edu/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 library?
- What role does

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

  play when you write it atop some program?

[3] https://manual.cs50.net/style/
[4] https://manual.cs50.net/style/

- What role does

  ```
  -lcs50
  ```

  play when you pass it as a "command-line argument" to `clang`? (Recall that `make`, the program we've been using to compile programs in lecture, simply calls `clang` with some command-line arguments for you to save you some keystrokes.)

Curl up with at least two other shorts at https://cs50.harvard.edu/shorts/1. Some additional questions may be in your future!

Hello again, C

Before forging ahead, you might want to review some of the examples that we looked at in Week 1's lectures and take a look at a few more, the "source code" for which can be found at https://cs50.harvard.edu/lectures/1. Allow me to take you on a tour, though feel free to forge ahead on your own if you'd prefer. Not to worry if your appliance also looks a bit different from mine.

https://www.youtube.com/watch?v=bQnyxpf0vk0&list=PLhQjrBD2T380JCGC3qD3nGpqt8iIjx2fV

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`.) If the user fails to provide a non-negative integer no greater than `23`, you should re-prompt for the same again. 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 underlined 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.

By contrast, if the user fails to provide a non-negative integer no greater than `23`, your program's output should instead resemble the below, wherein underlined text again represents some user's input. (Recall that `GetInt` will handle some, but not all, re-prompting for you.)

```
jharvard@appliance (~/Dropbox/pset1): ./mario
Height: -2
Height: -1
Height: foo
Retry: bar
Retry: 1
##
```

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 check the correctness of your program with `check50`, you may execute the below.

```
check50 2014.fall.pset1.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/pset1/mario
```

Not sure where to begin? Not to worry. A walkthrough awaits!

https://www.youtube.com/watch?v=z32BxNe2Sfc

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[5] on which we found this here accessory (for ages 5 and up).

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[6] 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."

[5] http://hearthsong.com/
[6] http://www.nist.gov/dads/HTML/greedyalgo.html

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.

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

How few? Well, you tell us. 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. Assume that the only coins available are quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢).

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. 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);
```

Before doing any math, then, 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! Of course, don't just cast the user's input from a `float` to an `int`! After all, how many cents does one dollar equal? And be careful to round[7] and not truncate your pennies!

Not sure where to begin? Not to worry, start with a walkthrough:

https://www.youtube.com/watch?v=9dZzyl7dCuw

Incidentally, 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, wherein underlined text 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!)

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
```

[7] https://cs50.harvard.edu/resources/cppreference.com/stdmath/round.html

```
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 check the correctness of your program with `check50`, you may execute the below.

```
check50 2014.fall.pset1.greedy greedy.c
```

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

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

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.harvard.edu/submit[8], logging in if prompted.
- Click **Submit** toward the window's top-left corner.
- Under **Problem Set 1** on the screen that appears, click **Upload New Submission**.
- On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.

[8] https://cs50.harvard.edu/submit

- Navigate your way to `hello.c`, as by clicking **jharvard**, then double-clicking **Dropbox**, then double-clicking **pset1**, assuming you saved `hello.c` in `~/Dropbox/pset1`. 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 `mario.c` as before. Click it once to select it, then click **Open**.
- Navigate your way to `greedy.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.harvard.edu/submit[9] 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/2014/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.

[9] https://cs50.harvard.edu/submit