# Problem Set 1: C

This is CS50. Harvard University. Fall 2015.

## Table of Contents



# 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 for disciplinary action and the outcome is punitive, the course reserves the right to impose local sanctions on top of that outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

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

## Reasonable

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

## Not Reasonable

- Accessing a solution to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem set’s problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff’s solutions to problem sets.

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

# 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 or not taking the course 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 CS50 IDE is a web-based "integrated development environment" that allows you to program "in the cloud," without installing any software locally. Underneath the hood is a popular operating system, Ubuntu Linux, that's been "containerized" with open-source software called Docker, that allows multiple users (like you!) to share the operating system's "kernel" (its nucleus, so to speak) and files, even while having files of their own. Indeed, CS50 IDE provides you with your very own "workspace" (i.e., storage space) in which you can save your own files and folders (aka directories). Anyhow, more on all that another time!

## Logging In

Head to cs50.io[1] and log into CS50 IDE. Upon logging in for the very first time, you should be informed that CS50 IDE (aka Cloud9, the software that underlies CS50 IDE) is "creating your workspace" and "creating your container," which might take a moment. (If you already have an account at c9.io, which is Cloud9's own site, drop sysadmins@cs50.harvard.edu[2] a note, and we'll let you know how to "link" that account.)

Once you see your workspace, which should resemble mine from Week 1, close any tabs that might have been opened for you by default (e.g., **Welcome** and **README.md**). Then decide which "theme" you'd like. By default, CS50 IDE is configured with a theme called "Cloud9 Day." If you'd prefer a darker theme, particularly at night, visit **Support > Welcome**

[1] https://cs50.io/
[2] mailto:sysadmins@cs50.harvard.edu

**Page**, which should open a **Welcome** tab, and then, within that tab, change **Main Theme** to **Cloud9 Classic Dark Theme**, and then close the tab. Feel free to poke around other menus as well to get a sense of CS50 IDE's user interface (UI). No worries if you're not sure what most of the menus do (yet!). If among those more comfortable or just curious, feel free to uncheck **View > Less Comfortable**, which will reveal even more options.

## Updating

Toward the bottom of CS50 IDE's UI is a "terminal window" (in a tab called **Terminal**), a command-line interface (CLI) that allows you to explore your workspace's files and directories, compile code, run programs, and even install new software. You should find that its "prompt" looks like

```
username@ide50:~/workspace $
```

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

```
update50
```

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

## Hello

Okay, let's create a folder in which your code for this problem set will soon live. Toward CS50 IDE's top-left corner, control-click or right-click **ide50**, your workspace's "root", and select **New Folder**. A new folder called **New Folder** should appear; rename it **pset1**, then hit Enter. (If you misname it or can't seem to edit its name, control-click or right-click the new folder and select **Rename** to try again!)

Next, control-click or right-click that **pset1** folder that you just created and select **New File**. A new file called **Untilted** should appear; rename it **hello.txt** and then hit Enter. (If you misname it or can't seem to edit its name, control-click or right-click the new file and select **Rename** to try again!) You should see that **hello.txt** is indented immediately beneath

**pset1**, which indicates that the former is inside of the latter. If **hello.txt** doesn't appear to be inside of **pset1**, simply drag and drop the former into the latter.

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

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

```
username@ide50:~/workspace $
```

by default, where `username` is, again, your assigned username. The prompt further indicates that `ide50` is the name of your (automatically created) workspace. And it indicates that `workspace` is the name of your "current working directory" (i.e., the folder that's currently open within that command-line environment). The tilde (`~`), meanwhile, refers to your account's "home directory," in which your `workspace` directory lives, but more on that another time. Note that this `workspace` directory is identical to that **ide50** icon in CS50 IDE's top-left corner (inside of which you created **pset1** earlier). A better choice of names for the latter would have been, well, **workspace**; we'll fix that soon!

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

```
ls
```

followed by Enter. That's a lowercase L and a lowercase S, which is shorthand notation for "list." Indeed, you should then see a list of the folders inside of your workspace, among which is `pset1`! Let's open that folder. Type

```
cd pset1
```

or even, more verbosely,

```
cd ~/workspace/pset1
```

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

```
username@ide50:~/workspace/pset1 $
```

confirming that you are indeed now inside of `~/workspace/pset1` (i.e., a directory called `pset1` inside of a directory called `workspace` inside of your home directory). Now type

```
ls
```

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

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

```
cd
```

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

```
username@ide50:~ $
```

To get back into `pset1`, type

```
cd workspace
```

and then Enter followed by

```
cd pset1
```

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

```
cd workspace/pset1
```

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

# Hello, C

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

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

Shall we have you write your first program? Inside of your **pset1** folder, create a new file called **hello.c**, and then open that file in a tab. (Remember how?) Be sure to capitalize the file's name just as we have; files' and folders' names in Linux are "case-sensitive." Proceed to write your first program by typing precisely these lines into the file:

```
#include <stdio.h>

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

Notice how CS50 IDE adds "syntax highlighting" (i.e., color) as you type. Those colors aren't actually saved inside of the file itself; they're just added by CS50 IDE to make certain syntax stand out. Had you not saved the file as `hello.c` from the start, CS50 IDE wouldn't know (per the filename's extension) that you're writing C code, in which case those colors would be absent.

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

When done typing, select **File > Save** (or hit command- or control-s), but don't quit. Recall that the leading asterisk in the tab's name should then disappear. Click anywhere in the terminal window beneath your code, and be sure that you're inside of `~/workspace/pset1`. (Remember how? If not, type `cd` and then Enter, followed by `cd workspace/pset1` and then Enter.) Your prompt should be:

```
username@ide50:~/workspace/pset1 $
```

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

```
ls
```

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

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

```
make hello
```

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

```
./hello
```

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

```
hello, world
```

And if you type

```
ls
```

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

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

```
make hello
```

at your prompt, followed by Enter. (Just be sure that you are inside of `~/workspace/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 whatever you told `printf` to print?

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

Woo hoo! You've begun to program!

## CS50 Check

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

If not already there, navigate your way to `~/workspace/pset1` by executing the command below.

```
cd ~/workspace/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 `2015.fall.pset1.hello` is just a unique identifier for this problem's checks.

```
check50 2015.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.

## Shorts

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

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

## Hello again, C

Before forging ahead, you might want to review some of the examples that we looked at in Week 1's lectures and take a look at a few more, the "source code" for which can be found under **Lectures** on the course's website. Allow me to take you on a tour, though feel free to forge ahead on your own if you'd prefer. (My CS50 Appliance will look a bit different from CS50 IDE, but not to worry.)

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

## Smart Water

Suffice it to say that the longer you shower, the more water you use. But just how much? Even if you have a "low-flow" showerhead, odds are your shower spits out 1.5 gallons of water per minute. A gallon, meanwhile, is 128 ounces, and so that shower spits out 1.5 × 128 = 192 ounces of water per minute. A typical bottle of water (that you might have for a drink, not a shower), meanwhile, might be 16 ounces. So taking a 1-minute shower is akin to using 192 ÷ 16 = 12 bottles of water. Taking (more realistically, perhaps!) a 10-minute shower, then, is like using 120 bottles of water. Deer Park, that's a lot of water! Of course, bottled water itself is wasteful; best to use reusable containers when you can! But it does put into perspective what's being spent in a shower!

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

```
username@ide50:~/workspace/pset1 $ ./water
minutes: 10
bottles: 120
```

For simplicity, you may assume that the user will input a positive integer, so no need for error-checking (or any loops) this time! And no need to worry about overflow!

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

```
check50 2015.fall.pset1.water water.c
```

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

```
~cs50/pset1/water
```

## 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 `~/workspace/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.

```
username@ide50:~/workspace/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.)

```
username@ide50:~/workspace/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 2015.fall.pset1.mario mario.c
```

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

```
~cs50/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, here's "a great way to store change and teach children to add it up. With the press of the lever, the coin changer releases coins on demand. Slots allow you to slide this great toy onto your belt."


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[3] 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."

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

[3] http://www.nist.gov/dads/HTML/greedyalgo.html

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 `~/workspace/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[4] 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.

```
username@ide50:~/workspace/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.

```
username@ide50:~/workspace/pset1 $ ./greedy
O hai! How much change is owed?
-0.41
How much change is owed?
-0.41
How much change is owed?
```

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

```
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 2015.fall.pset1.greedy greedy.c
```

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

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

# How to Submit

## Step 1 of 2

1. When ready to submit, log into CS50 IDE[5].
2. Toward CS50 IDE's top-left corner, within its "file browser" (not within a terminal window), control-click or right-click your `hello.c` file (that's within your **pset1** directory) and then select **Download**. You should find that your browser has downloaded `hello.c`.
3. Repeat for `water.c`.
4. Repeat for `mario.c`.
5. Repeat for `greedy.c`.
6. In a separate tab or window, log into CS50 Submit[6], logging in if prompted.

[5] https://cs50.io/
[6] https://cs50.harvard.edu/submit

7. Click **Submit** toward the window's top-left corner.
8. Under **Problem Set 1** on the screen that appears, click **Upload New Submission**.
9. On the screen that appears, click **Add files…**. A window entitled **Open Files** should appear.
10. Navigate your way to `hello.c`. Odds are it's in your **Downloads** folder or wherever your browser downloads files by default. Once you find `hello.c`, click it once to select it, then click **Open** (or the like).
11. Click **Add files…** again, and a window entitled **Open Files** should appear again.
12. Navigate your way to `water.c` as before. Click it once to select it, then click **Open** (or the like).
13. Navigate your way to `mario.c` as before. Click it once to select it, then click **Open** (or the like).
14. Navigate your way to `greedy.c` as before. Click it once to select it, then click **Open** (or the like).
15. Click **Start upload** to upload all of your files at once to CS50's servers.
16. 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 Submit[7] 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/2015/fall/psets/1/ where a short form awaits. Once you have submitted that form (as well as your source code), you are done! If you end up resubmitting your files (per step 1 of 1), no need to resubmit the form.

This was Problem Set 1.

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