Problem 2-3: Old Friends

This is CS50. Harvard University. Fall 2014.

Table of Contents

Questions? Feel free to head to CS50 on Reddit[1], CS50 on StackExchange[2], or the CS50 Facebook group[3].

Objectives

- Revise old programs to make use of new found knowledge
- Interact with users at the command line
- Become familiar with functions and libraries

Recommended Reading

- Pages 11 – 14 and 39 of http://www.howstuffworks.com/c.htm.
- Chapters 6, 7, 10, 17, 19, 21, 22, 30, and 32 of *Absolute Beginner's Guide to C*.
- Chapters 7, 8, and 10 of *Programming in C*.

[1] https://www.reddit.com/r/cs50
[2] http://cs50.stackexchange.com
[3] https://www.facebook.com/groups/cs50

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 problems is not permitted (unless explicitly stated otherwise) 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 or writing to others, but you may not view theirs, so long as you and they respect this policy's other constraints. Collaboration on quizzes and tests is not permitted at all. Collaboration on the 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 your instructor. If a violation of this policy is suspected and confirmed, your instructor reserves the right to impose local sanctions on top of any disciplinary outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

Reasonable

- Communicating with classmates about 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, such 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 years' quizzes, tests, 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 solutions to problems online so that others might help you identify and fix a bug or other issue.
- 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 problems or your own final project.
- Whiteboarding solutions to problems 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 before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff's solutions to problems.
- Failing to cite (as with comments) the origins of code, writing, 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 when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual's work during a quiz or test.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problems 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 problems online or elsewhere.
- Splitting a problem's workload with another individual and combining your work (unless explicitly authorized by the problem itself).

- 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.
- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a 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)?

To obtain a passing grade in this course, all students must ordinarily submit all assigned problems unless granted an exception in writing by the instructor.

Well-Distributed

Here's some good news for you right off the bat: you've seen almost everything we're going to talk about in this problem once before. We're going to revisit three old programs and rewrite them so that instead of taking input from the user while the program is running, they will instead accept input from the user at the command-line, before the program is even run.

Even better, to ensure everyone is on a level playing field while solving this problem, you'll be downloading some "distribution code" (otherwise known as a "distro"), written by us, and make modifications to it. You're welcome of course to use the code you wrote for a prior problem (if it worked!) and re-work it for this problem, but if you struggled with the problems we'll be reimplementing this time around, know that we will otherwise be supplying you with fully-functional code. All you have to do is change the way that the user inputs data.

Start off by opening up CS50 IDE and then type

```
update50
```

within a terminal window to make sure your workspace is up-to-date. If you somehow closed your terminal window (and can't find it!), make sure that **Console** is checked under the **View** menu, then click the green, circled plus (+) in CS50 IDE's bottom half, then select **New Terminal**. If you need a hand, do just ask via the channels noted at the top of this specification.

Next, navigate to your `unit2` directory, as with

```
cd ~/workspace/unit2
```

Keep in mind that `~` denotes your home directory, `~/workspace` denotes a directory called `workspace` therein, and `~/workspace/unit2` denotes a directory called `unit2` within `~/workspace`. Your prompt should now resemble the below.

```
username@ide50:~/workspace/unit2 $
```

Now, at the prompt, type:

```
wget http://cdn.cs50.net/ap/1516/problems/2/3/friends.zip
```

to download a ZIP of this problem's distro into your workspace (with a command-line program called `wget`). You should see some output followed by:

```
'friends.zip' saved
```

Confirm that you've indeed downloaded `friends.zip` by executing

```
ls
```

and then run

```
unzip friends.zip
```

to unzip the file. If you then run `ls` again, you should see that you have a newly unzipped directory called `friends` as well. Proceed to execute

```
cd friends
```

followed by

```
ls
```

and you should indeed see a few old friends!

```
hello.c  fahrenheit.c  pennies.c
```

How nice to see them again!

Lastly, have a look at Christopher's short video on command-line arguments. Since we'll be converting all three of the programs listed above to accept command-line arguments (none of them currently do!), this video should come in handy.

https://www.youtube.com/watch?v=X8PmYwnbLKM
If you happen to see (and are confused by!) `char *` in this and other shorts, know for now that `char *` simply means `string`. But more on that soon!

Hello, again!

In Problem 1-2[4] you were asked to write `hello`, a program that very simply printed the message `hello, world\n` to the screen when run. It's not too much of a leap to extend

[4] http://cdn.cs50.net/ap/1516/problems/1/2/1-2.html

this program to say hello to a specific person by asking for the user to type a name at the prompt instead, so the program behaves like this.

```
username@ide50:~/workspace/unit2/friends $ ./hello
Your name: Zamyla
hello, Zamyla!
```

In fact, the distro you downloaded contains a file, `hello.c`, with exactly this behavior. What we want, though, is a program that has this behavior instead:

```
username@ide50:~/workspace/unit2/friends $ ./hello Zamyla
hello, Zamyla!
```

See the slight difference? Instead of prompting the user for information **after** the program has started running, we collect the desired information from the user **before** they run the program, and then use that information once the program has started. How do we do so?

Recall that our programs are capable of knowing information about what the user typed at the command line by modifying the way we write the start of our `main` function. Instead of

```
int main(void)
```

if we start `main` off by typing

```
int main(int argc, string argv[])
```

we then have access to two special variables that we can use inside of `main`. First is `argc`, which is an integer variable that tells us how many things the user typed in at the command line, and second is `argv`, which is an array of strings representing exactly what the user typed.

Knowing this, and from the information in Christopher's short, can you now modify `hello.c` so that it prints out the name provided at the command line, instead of collecting a `string` from the user after the program has started?

One more wrinkle. How do you make sure the user in fact did provide you with one (and only one) additional argument, so that you can print it out? Well remember that's what our new friend `argc` can manage for us. If the user doesn't supply a command-line argument, best to terminate the program and have them try again. One way to accomplish this might be to have this near the top of our code:

```
if (argc != 2)
{
    printf("Usage: ./hello <name>\n");
    return 1;
}
```

Note what this accomplishes? We check to make sure that the user has supplied the proper number of command-line arguments (2). If not, we tell the user how they *should* run the program, and then we `return 1;`, which is our way of indicating that our program finished running, but not successfully. We use nonzero return values from `main`, also known as *exit codes*, to report back to the system that something went awry.

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

```
check50 1516.unit2.hello hello.c
```

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

```
~cs50/unit2/hello <name>
```

Chill Out

In Problem 1-3[5] you were asked to write `fahrenheit`, a program that asked the user for a temperature in Celsius (which they provided via `GetFloat`) and which then printed out the equivalent temperature on the Fahrenheit scale.

```
username@ide50:~/workspace/unit2/friends $ ./fahrenheit
C: 0
F: 32.0
```

[5] http://cdn.cs50.net/ap/1516/problems/1/3/1-3.html

In `fahrenheit.c` you will find a fully-functioning version of the code you were tasked with writing in that problem. Convert that program so that it accepts the Celsius temperature from the command line instead.

```
username@ide50:~/workspace/unit2/friends $ ./fahrenheit 0
F: 32.0
```

There’s a catch, though.

Just because the user types a real number at the prompt, that doesn’t mean their input will be automatically stored in a `float`. Actually, it will be stored as a `string` that just so happens to look like an `float`; after all, remember the data type of `argv`? It’s an array where each element is a `string`! And so you’ll need to convert that `string` to an actual `float`. As luck would have it, a function, `atof`[6], exists for exactly that purpose! Here’s how you might use it:

```
float celsius = atof(argv[1]);
```

Notice, this time, we’ve declared `celsius` as an actual `float` so that you can do some arithmetic with it. Incidentally, you can assume that the user will only type real numbers at the command line.

Because `atof` is declared in `stdlib.h`, you’ll want to `#include` that header file atop your own code. And, as with `hello.c` earlier, you’ll want to make sure the user provides exactly the correct number of command-line arguments to your program before doing any calculations, returning 1 should they fail to.

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

```
check50 1516.unit2.fahrenheit fahrenheit.c
```

If you’d like to play with the staff’s own implementation of `fahrenheit`, you may execute the below.

[6] https://reference.cs50.net/stdlib.h/atof

```
~cs50/unit2/hello <temperature>
```

Makin' Bank

In Problem 1-6[7] you were asked to write `pennies`, a program that demonstrated the power of exponentiation by showing how much money you would have if a person gave you *x* pennies on a particular day, and then doubled the amount they gave you every day for a period of *y* days.

```
username@ide50:~/workspace/unit2/friends $ ./pennies
Days in month: 31
Pennies on first day: 1
$21474836.47
```

As you might expect, we'd now like the program to work as follows.

```
username@ide50:~/workspace/unit2/friends $ ./pennies 31 1
$21474836.47
```

Notice a few differences with this program from the previous two. How many command line arguments does *this* one accept? And also know that in `stdlib.h` alongside of the function `atof` exists another, `atoi`, that converts a `string` to an `int` in much the same way that `atof` converts a `string` to a `float`.

Incidentally, you can assume that the user will only type integers at the command line; there's no need for you to anticipate a rogue user this time around!

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

```
check50 1516.unit2.pennies pennies.c
```

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

[7] http://cdn.cs50.net/ap/1516/problems/1/6/1-6.html

```
~cs50/unit2/pennies <days> <starting amount>
```

This was Problem 2-4.