Problem Set 1: C

due by 7:00 P.M. on Friday, 3 October 2008

Be sure that your code is thoroughly commented
to such an extent that lines' functionality is apparent from comments alone.

Goals.

- Get more 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 – 6 of *Programming in C.*

diff pset1.pdf hacker1.pdf.

- Hacker Edition presents not only ISBN-10s but also ISBN-13s!
- Hacker Edition allows cashier to run out of certain coinage!
- Hacker Edition makes Mario jump two half-pyramids!

Academic Honesty

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed (*e.g.*, by some problem set or the final project). Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, magazine, 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.

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. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

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 staff.

You may even 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.

Grades.

Your work on this problem set will be evaluated along three primary axes.

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)?

Getting Started.

0. For this problem set, we're going to have you use `nice.fas.harvard.edu`, FAS's "New Instructional Computing Environment," on which you should, per Problem Set 0's direction, have an account (*i.e.*, a username and password). Housed in the basement of the Science Center (much like you may be this term), this environment is a cluster of servers running Linux, each of which "mounts" your "home directory," storage space that FAS has allocated to you, so that you can access your files on any server in the cluster. Not only can you use this account to send and receive email (via an address of the form `username@fas.harvard.edu`), you can also use it to, oh, write source code, compile source code into object code, and run programs you've written!

    To "SSH to nice.fas.harvard.edu" means to connect to some (random) server in that cluster via a protocol (a language or program of sorts) called "Secure Shell" in such a way that you can access and control your account from afar, albeit via a fairly arcane interface (a "terminal window"). In fact, the servers in this cluster might very well not even have keyboards and mice, since they're meant to be used from afar by many users at once. Programs like PuTTY and SecureCRT on Windows and Terminal on Mac OS are "SSH clients," programs that implement this SSH protocol and, therefore, allow you to connect in this manner.[1]

    If unfamiliar with this process of SSHing, read over the document entitled **How to SSH to `nice.fas.harvard.edu`**, either for Windows or for Mac OS, available via the link to **Resources** on the course's website.

    Ready? Go ahead and actually SSH to `nice.fas.harvard.edu` using your SSH client of choice. In other words, "log into your FAS account via SSH." In other words still, "connect to `nice.fas.harvard.edu`." Or, better yet, "pull up a terminal window," as the l337 are fond of saying.[2] Henceforth, statements like these pretty much mean the same thing. After providing your username and password, you should reach the "command line" with the so-called "blinking prompt," even though it doesn't always blink. (Depends on your SSH client!) That blinking prompt (or, really, the program that runs by default when you SSH to a server) is called your "shell." A shell is just an interpreter of commands. You type something, it does something.

1. Let's go ahead and restore your FAS account's "dotfiles" (*i.e.*, configuration files) to their original state, lest we assume your account is configured in some way that it is not.[3] At your blinking prompt, go ahead and execute the command below. That is, type the below and then hit Enter.

    ```
    /usr/local/bin/fixdotfiles
    ```

    Assuming you typed the above correctly, you've just run a "shell script" (*i.e.*, a program written not in a compiled language like C but an interpreted language called "C shell") that, per its own

---

[1] Harvard actually has a "site license" for SecureCRT, which is technically commercial software that Harvard pays for on your (and everyone else's) behalf. But we'll generally recommend PuTTY this semester, if only because it's popular and free. But you're welcome to use any SSH client.
[2] `http://en.wikipedia.org/wiki/Leet`
[3] If you'd rather not undo changes that you yourself have made to your dotfiles since obtaining your FAS account, that's fine, you can skip this step. If you never heard of dotfiles before today, though, don't skip this step!

output, will install some "new system default" configuration files into your account.[4] To be even more precise, you just ran a script called `fixdotfiles` that lives in a directory (*i.e.*, folder) called `bin` that lives in a directory called `local` that lives in a directory called `usr` that lives in the server's "root directory" (called `/`). Much like there are conventions in Windows and Mac OS for where programs go (*e.g.*, in "Program Files" on Windows and in "Applications" on Mac OS), similarly are there conventions in Linux. In `/usr/` are programs meant for users.[5] In `/usr/local/` are programs that didn't necessarily come with Linux itself. In `/usr/local/bin/` are programs (*i.e.*, binaries) themselves. But these are just conventions; exceptions abound.

Anyhow, now configure your account for CS 50 specifically by executing the command below. Note that the tilde (~) is likely in your keyboard's top-left corner.

```
~cs50/pub/bin/cs50setup
```

Assuming you typed the above correctly, you've just run another "script" (this one written by us in another interpreted language called "Perl") that alters your shell's configuration.[6] Not only does that script provide you with access to software that the course has installed in its own account on `nice.fas.harvard.edu` (*e.g.*, `gcc`), that script also alters the appearance of your prompt to be more helpful than the default one. But you'll need to log out and back in for these changes to take effect. Do so by executing

```
exit
```

or, if you like longer words,

```
logout
```

at your prompt, and then re-SSH to `nice.fas.harvard.edu`. Upon logging back in, you can confirm that you done good by executing the command below.

```
cs50check
```

If that command is "not found," PEBKAC is probably to blame.[7] For assistance with this process, simply contact the course's staff. If, however, all went according to plan, your prompt should now resemble the below.

```
username@nice (~):
```

Not only does your prompt now remind you who you are, it also makes clear that you're connected to `nice.fas.harvard.edu` and reminds you parenthetically of your "current working directory" (*i.e.*, the folder you currently have open). Upon logging into your account, you are, by default, in your home directory. Ergo the tilde.

---

[4] If you see "Command not found," you typed it incorrectly! Try again!
[5] And, nope, it's not a typo: there's no e in `usr` here! That's efficiency 4 u!
[6] If you see "Command not found," you typed it incorrectly! Try again!
[7] `http://en.wikipedia.org/wiki/PEBKAC`

Now let's make room in your life for CS 50. Execute the command below.[8]

```
mkdir ~/cs50/
```

You've just created in a directory called `cs50` in your home directory (the shorthand for which is a tilde). The code that you write for this problem will ultimately need to reside within this directory for submission.

Next, execute the following command.

```
mkdir ~/cs50/hacker1/
```

Perhaps needless to say, you've just created a directory called `hacker1` within that `cs50` directory. Confirm as much by executing the command below.

```
find ~/cs50/
```

You should see output resembling the "paths" below, where `username` is your FAS username and `u` and `s` are the first and second characters thereof.

```
/home/u/s/username/cs50
/home/u/s/username/cs50/hacker1
```

Now change your current working directory to `~/cs50/hacker1/` by executing the command below.

```
cd ~/cs50/hacker1/
```

To check that you are indeed in `~/cs50/hacker1/`, execute the command below.

```
pwd
```

You should see output resembling the below.[9]

```
/nfs/home/u/s/username/cs50/hacker1/
```

It turns out that `~` is actually shorthand for `/nfs/home/u/s/username/`. Oh and your prompt should now resemble the below.

```
username@nice (~/cs50/hacker1):
```

Handy, eh?

---

[8] Note the difference between `~/cs50` in this command and `~cs50` in that earlier command.
[9] Sometimes, though, /nfs is excluded from such output.

So you now know how to "open folders" at the command line. How do you "close" or "back out" of them? It turns out that `..` (pronounced "dot dot") represents any directory's "parent" directory, the one containing it. Go ahead and execute the below.

```
cd ..
```

Your prompt should now resemble the below.

```
username@nice (~/cs50):
```

Were you to execute that same command again, you'd end up in your home directory. But you can also whisk yourself back to your home directory from anywhere by executing the command below.

```
cd ~
```

In fact, if you ever get lost inside your own account, consider executing the command above so that you can start whatever sequence of steps again from home, sweet home.

It's also worth knowing that `.` (pronounced "dot") represents your current working directory. Go ahead and type the command below.

```
cd .
```

Your prompt should still look the same. Pointless, eh? Trust us, though, `.` does have its uses.

Alright, navigate your way back to `~/cs50/hacker1/`. Remember how?

O hai, Nano!

2. Let's get you warmed up. From within your `~/cs50/hacker1/` directory, go ahead and execute the command below.

```
nano hello.c
```

Proceed to write your own version of "hello, world." It suffices to re-type, nearly character for character, Week 1's `hai1.c`, but do at least replace "O hai, world!" with your own argument to `printf`.

Once done with your recreation, hit ctrl-x to save, followed by Enter, and you should be returned to your prompt. Proceed to execute the command below.

```
gcc hello.c
```

If you've made no mistakes, you should just see another prompt. If you've made some mistake, you'll instead see one or more warning and/or error messages. Even if cryptic, think about what

they might mean, then go find your mistake(s)! To edit `hello.c`, re-execute Nano as before. Once your code is correct and compiles successfully, look for your program in your current working directory by typing the following command.

```
ls
```

You should see output resembling the below.

```
a.out*  hello
```

Actually, some more details would be nice. Go ahead and execute the command below instead.

```
ls -l
```

More than just list the contents of your current working directory, this command lists their sizes, dates and times of creation, and more. The output you see should resemble the below.

```
-rwx------ 1 username student 7077 2008-09-26 18:04 a.out*
-rw------- 1 username student  371 2008-09-26 18:03 hello.c
```

The `-l` is a "switch" that controls the behavior of `ls`. To look up more switches for `ls` (and its documentation in general), execute the command below.

```
man ls
```

You can scroll up and down through in this manual using your keyboard's arrow keys and space bar. In general, anytime you'd like more information about some command, try checking its "man page" by executing `man` followed by the command's name! Let's now confirm that your program does work. Execute the command below.

```
a.out
```

You should see your greeting. Before moving on, let's give your program a more interesting name than `a.out`. Go ahead and execute the following command.[10]

```
gcc -o hello hello.c
```

In this case, `-o` is but a switch for `gcc`. The effect of this switch is to name `gcc`'s output `hello` instead of `a.out`. Let's now get rid of your first compilation. To delete `a.out`, execute the following command.

```
rm a.out
```

If prompted to confirm, hit y followed by Enter.

Welcome to Linux and C.

---

[10] Be careful not to transpose `hello` and `hello.c`, else you'll end up deleting your code!

I S BN Readin Bookz.

3. As you probably know, most any book that you borrow or buy has an International Standard Book Number, otherwise known as an ISBN or ISBN-10, "a 10-digit number that uniquely identifies books and book-like products published internationally."[11] Books published since 2007 might also have an ISBN-13, a 13-digit number with a similar purpose, but more on that in a bit.

    It turns out that the last of an ISBN-10's digits is a "check digit," otherwise known (in binary contexts) as a "checksum," a number related mathematically to its preceding digits. ISBN-10s' digits are supposed to adhere to a formula, not unlike credit card numbers, and this check digit allows you to check whether an ISBN-10's other nine digits are (most likely) valid without having to check, say, a database of books.

    Per the fourth, revised edition of the International ISBN Agency's ISBN Users' Manual, "The check digit is the last digit of an ISBN. It is calculated on a modulus 11 with weights 10-2, using X in lieu of 10 where ten would occur as a check digit."[12]

    O rly? Yes, but what does that mean? The manual elaborates. "This means that each of the first nine digits of the ISBN – excluding the check digit itself – is multiplied by a number ranging from 10 to 2 and that the resulting sum of the products, plus the check digit, must be divisible by 11 without a remainder."

    Okay, better, but still a bit unclear. Let's define the check digit in terms of a formula. Fortunately, thanks to "modular arithmetic," we can simplify the Agency's formal definition using weights ranging from 1 to 9 instead of 10 to 2. In fact, it's really quite simple. If x_1 represents an ISBN-10's first digit and x_{10} its last, it turns out that:[13]

    $$x_{10} = (1 \cdot x_1 + 2 \cdot x_2 + 3 \cdot x_3 + 4 \cdot x_4 + 5 \cdot x_5 + 6 \cdot x_6 + 7 \cdot x_7 + 8 \cdot x_8 + 9 \cdot x_9) \bmod 11$$

    In other words, to compute an ISBN-10's tenth digit, multiply its first digit by 1, its second digit by 2, its third digit by 3, its fourth digit by 4, its fifth digit by 5, its sixth digit by 6, its seventh digit by 7, its eighth digit by 8, and its ninth digit by 9. Take the sum of those products and then divide it by 11. The remainder should be the ISBN-10's tenth digit! If, though, that remainder is 10, the tenth digit should instead be printed as 'X' lest it be confused with a '1' followed by '0'.

    Let's try all this out. Per the course's syllabus, the ISBN-10 for *How Computers Work*, one of the course's recommended books, is 0-7897-3613-6, the tenth digit of which is, obviously, 6. But is the syllabus right? Well, let's first take that sum using the ISBN-10's first nine digits (highlighted in bold):

    $1 \cdot \mathbf{0} + 2 \cdot \mathbf{7} + 3 \cdot \mathbf{8} + 4 \cdot \mathbf{9} + 5 \cdot \mathbf{7} + 6 \cdot \mathbf{3} + 7 \cdot \mathbf{6} + 8 \cdot \mathbf{1} + 9 \cdot \mathbf{3} =$
    $0 + 14 + 24 + 36 + 35 + 18 + 42 + 8 + 27 =$
    204

---

[11] http://www.isbn.org/standards/home/isbn/us/isbnqa.asp
[12] http://www.isbn-international.org/en/userman/download/ISBNmanual.pdf
[13] Normally, we'd start counting from 0 and not 1, but for ISBN-10s, it's simpler not to!

If we now divide that sum by 11, we get $204 \div 11 = 18\ ^6/_{11}$ (*i.e.*, a remainder of 6)! Well that's kind of neat, the ISBN is legit! Actually, also thanks to modular arithmetic, we could just include that tenth digit in our sum and multiply it by 10:

$1 \cdot \mathbf{0} + 2 \cdot \mathbf{7} + 3 \cdot \mathbf{8} + 4 \cdot \mathbf{9} + 5 \cdot \mathbf{7} + 6 \cdot \mathbf{3} + 7 \cdot \mathbf{6} + 8 \cdot \mathbf{1} + 9 \cdot \mathbf{3} + 10 \cdot \mathbf{6} =$
$0 + 14 + 24 + 36 + 35 + 18 + 42 + 8 + 27 + 60 =$
264

If we now divide this sum by 11, we get $264 \div 11 = 24$ with no remainder at all, which is an equivalent way of saying the ISBN-10 is legit! Stated more formally, $0 \equiv 264 \pmod{11}$!

Hopefully those exclamation points make the math more exciting.

Now what about ISBN-13s? First of all, their first three digits must be either 978 or 979. And, per the fifth edition of the ISBN Users' Manual, "Each of the first 12 digits of the ISBN is alternately multiplied by 1 and 3. The check digit is equal to 10 minus the remainder resulting from dividing the sum of the weighted products of the first 12 digits by 10 with one exception. If this calculation results in an apparent check digit of 10, the check digit is 0."[14] More formally, an ISBN-13's check digit is defined as:

$$x_{13} = (10 - ((1 \cdot x_1 + 3 \cdot x_2 + 1 \cdot x_3 + 3 \cdot x_4 + 1 \cdot x_5 + 3 \cdot x_6 + 1 \cdot x_7 + 3 \cdot x_8 + 1 \cdot x_9 + 3 \cdot x_{10} + 1 \cdot x_{11} + 3 \cdot x_{12}) \bmod 10)) \bmod 10$$

Let's try that out. According to Amazon.com, the ISBN-13 for *How Computers Work* is 978-0-7897-3613-0, the last digit of which is, obviously, 0. Let's check whether x_{13} should indeed be 0.

$(10 - ((1 \cdot 9 + 3 \cdot 7 + 1 \cdot 8 + 3 \cdot 0 + 1 \cdot 7 + 3 \cdot 8 + 1 \cdot 9 + 3 \cdot 7 + 1 \cdot 3 + 3 \cdot 6 + 1 \cdot 1 + 3 \cdot 3) \bmod 10)) \bmod 10 =$
$(10 - ((9 + 21 + 8 + 0 + 7 + 24 + 9 + 21 + 3 + 18 + 1 + 9) \bmod 10)) \bmod 10 =$
$(10 - ((130) \bmod 10)) \bmod 10 =$
$(10 - (0)) \bmod 10 =$
$(10) \bmod 10 =$
0

Phew! It should be.

So, computing the check digit for an ISBN-10 or ISBN-13 isn't hard, but it does get a bit tedious by hand. Let's write a program.

In `isbn.c`, write a program that prompts the user for an ISBN-10 or ISBN-13 and then reports (via `printf`) whether the number's legit. So that we can automate some tests of your code, we ask that your program's last line of output be either `YES\n` or `NO\n`, nothing more, nothing less. For simplicity, assume that the user's input will be exactly ten or thirteen decimal digits (*i.e.*, devoid of hyphens and 'X'), the first of which might even be zero(es), as in the case of our recommended book's ISBN-10. But you won't know in advance, until the user's hit Enter, how long the input will be. And do not assume that the user's input will fit in an `int`! Recall, after all, that the largest

---

[14] http://www.isbn-international.org/en/download/2005%20ISBN%20Users%27%20Manual%20International%20Edition.pdf

value that can fit in an `int` is $2^{32} - 1 = 4,294,967,295$ (and, even then, only if declared as `unsigned`). That's only a 10-digit value, and you'd better be able to handle a 13-! Best to use `GetLongLong` from CS 50's library to get users' input, though you may also use `GetString` if you'd like to plow ahead in the syllabus.

Okay, so you've gotten some input. What should you do? Well, realize that this C program, not unlike Scratch projects, can be reduced to the most basic of building blocks. For the sake of discussion, suppose that some variable `x` contains a 10-digit `long long` (with no leading zeroes). How can you get at its tenth (*i.e.*, rightmost) digit? Well how about this?

```
int tenth = x % 10;
```

Do you see why that works? Do not pass Go until it dawns on you why!

How, now, can you get at that same variable's ninth digit? Well, why don't we first get rid of its tenth digit by shifting every other one place to the right?

```
x = x / 10;
```

How about that trick? Do you see why it works? The ninth digit, now, is just:

```
int ninth = x % 10;
```

So I bet there's a pattern here. And odds are you don't need to (*i.e.*, shouldn't) copy/paste lines like the above nine or ten times. Loops are your friend. To be sure, other approaches exist. Proceed as you wish! Perhaps some of these tricks, though, will get you started.

Of course, to use `GetLongLong` (or `GetString`) you'll need to tell `gcc` about CS 50's library. Be sure to put

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

toward the top of `isbn.c`. And be sure to compile your code with a command like the below.

```
gcc -o isbn isbn.c -lcs50
```

Note that `-lcs50` must come at this command's end because of how `gcc` works.

Incidentally, recall that `make` can invoke `gcc` for you and provide that flag for you, as via the command below.

```
make isbn
```

Assuming your program compiled without errors (or, ideally, warnings) via either command, run your program with the command below.

```
isbn
```

Consider the below representative of how your own program should behave when passed a valid ISBN-10 (sans hyphens); highlighted in bold is some user's input.

```
username@nice (~/cs50/hacker1): isbn
ISBN: 0789736136
YES
```

Of course, `GetLongLong` itself will reject an ISBN-10's hyphens (and more) anyway:

```
username@nice (~/cs50/pset1): isbn
ISBN: 0-7897-3613-6
Retry: foo
Retry: 0789736136
YES
```

But it's up to you to catch inputs that are not ISBN-10s (*e.g.*, my phone number), even if ten digits:

```
username@nice (~/cs50/pset1): isbn
ISBN: 6175230925
NO
```

Your program must also work with ISBN-13s in this same fashion:

```
username@nice (~/cs50/hacker1): isbn
ISBN: 9780789736130
YES
```

Test out your program with a whole bunch of inputs, both valid and invalid. (We certainly will!) There's three more ISBN-10s in the syllabus, and way more on Amazon.com, along with ISBN-13s. If your program behaves incorrectly on some inputs (or doesn't compile at all), have fun debugging!

If you'd like to play with the staff's own implementation of `isbn` on `nice.fas.harvard.edu`, you may execute the below.

```
~cs50/pub/solutions/hacker1/isbn
```

Time for Change.

4. "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 fashion accessory.[15]

Of course, the novelty of this thing quickly wears off, especially when some jerk wants to pay for his newspaper with a hundred-dollar 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."[16]

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

---

[15] Description and image from `hearthsong.com`. For ages 5 and up.

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

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 EU's). That is, so long as a cashier has enough of each coin, this largest-to-smallest approach will yield the fewest coins possible.[17] 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?

Unfortunately, finding globally optimal solutions to problems is sometimes easier said than done. Which is precisely why we want you to do it!

Write, in `optimal.c`, a program that first asks the user for four non-negative values, retrieving each with `GetInt` in this order: the (finite) numbers of quarters, dimes, nickels, and pennies that the cashier should be assumed to have at his or her disposal. If the user fails to provide non-negative values, your program should re-prompt the user for appropriate values again and again until the user complies. Your program should next ask the user how much change is owed to some customer and then spit out (via `printf`) the absolute minimum number of coins with which said change can be made. Use `GetFloat` from CS 50's 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 **975** or even **$9.75**. However, if some customer is owed $9 even, assume that your program's input will be **9.00** or just **9** but, again, not **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. Before doing any math, you'll probably want to convert dollars and cents to hundreds of cents (*i.e.*, from a `float` to an `int`) to avoid tiny errors that might otherwise add up! Be careful to round and not truncate your pennies!

---

[17] 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?

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`. If, though, it's not even possible to make exact change, given limited coinage, you should instead output `0\n`. Consider the below representative of how your own program should behave; highlighted in bold is some user's input.

```
username@nice (~/cs50/hacker1): optimal
O hai!
Number of quarters: 10
Number of dimes: 10
Number of nickels: 0
Number of pennies: 10
Change owed: 0.41
5
```

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 (say, n00bs) might experience something more like the below.

```
username@nice (~/cs50/hacker1): optimal
O hai!
Number of quarters: -1
Number of quarters: 10
Number of dimes: 10
Number of nickels: 0
Number of pennies: 10
Change owed: -0.41
Change owed: foo
Retry: 0.41
5
```
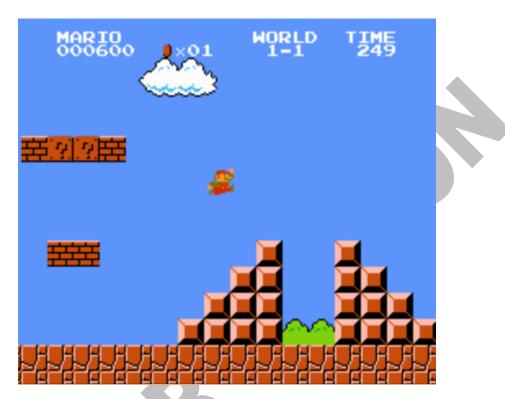
Per these requirements (and the sample above), your code will likely have some sort of loop. If, while testing your program, you find yourself looping forever, remember that you can kill your program (*i.e.*, short-circuit its execution) by hitting ctrl-c (sometimes a lot).

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

If you'd like to play with the staff's own implementation of optimal on nice.fas.harvard.edu, you may execute the below.

```
~cs50/pub/solutions/hacker1/optimal
```

5.   Toward the beginning of World 1-1 in Nintendo's Super Mario Brothers, Mario must hop over two
      "half-pyramids" of blocks as he heads toward a flag pole. Below is a screenshot.

      Write, in a file called `mario.c`, a program that recreates these half-pyramids using asterisks (*)
      for blocks. However, to make things more interesting, first prompt the user for the height of the
      half-pyramids. (The height of the half-pyramids pictured above happens to be 4, the width of
      each half-pyramid 4, with an a gap of size 2 separating them.) Then, generate (with the help of
      `printf` and one or more loops) the desired half-pyramids. Assume that the user's terminal
      window is exactly 80 characters wide by 24 characters tall. (Best to ensure that your own window
      boasts exactly those dimensions.) So that a blinking prompt still fits on the screen after a half-
      pyramid's generation, demand that the user provide a non-negative integer no greater than 23.
      Take care to align your rightmost half-pyramid, no matter its height, in the window's bottom-right
      corner (*i.e.*, 80 characters over and 23 characters down). No need generate the floating bricks,
      cloud, grass, or Mario himself. Just the half-pyramids!

      You're again on your own when it comes time to compile and run and debug this last program!

      If you'd like to play with the staff's own implementation of `mario` on `nice.fas.harvard.edu`,
      you may execute the below.

      `~cs50/pub/solutions/hacker1/mario`

Submitting Your Work.

6.   Ensure that your work is in `~/cs50/hacker1/`. Submit your work by executing the command below.

```
cs50submit hacker1
```

Thereafter, follow any on-screen instructions until you receive visual confirmation of your work's successful submission. You will also receive a "receipt" via email to your FAS account, which you should retain until term's end. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.