## Table of Contents



---

# Cash

## tl;dr

Implement a program that calculates the minimum number of coins required to give a user change.

---

```
$ ./cash
Change owed: 0.41
4
```

---

## Background

When using a device like this, odds are you want to minimize the number of coins you're dispensing for each customer, lest you have to press levers more times than are necessary. 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."

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!

# Specification

- Write, in a file called `cash.c` in `~/workspace/pset1/cash/`, 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 `get_float` 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 `get_float` 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.
- You need not try to check whether a user's input is too large to fit in a `float`. Using `get_float` 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, 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`.

# Walkthrough

# Usage

Your program should behave per the example below. Assumed that the underlined text is what some user has typed.

```
$ ./cash
Change owed: 0.41
4
```

```
$ ./cash
Change owed: -0.41
Change owed: -0.41
Change owed: foo
Change owed: 0.41
4
```

# Testing

## Correctness

```
check50 cs50/2018/x/cash
```

## Style

```
style50 cash.c
```

# Staff Solution

```
~cs50/pset1/cash
```

# Hints

- Per the final bullet point of the Specification, above, don't forget to put a newline character at the end of your printout!
- Do beware the inherent imprecision of floating-point values. For instance, `0.1` cannot be represented exactly as a `float`. Try printing its value to, say, `55` decimal places, with code like the below:

  ```
  float f = 0.1;
  printf("%.55f\n", f);
  ```

  And so, before making change, 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 take care to round (https://reference.cs50.net/math/round) your cents (to the nearest penny); don't "truncate" (i.e., floor) your cents!