Cash - CS50x 2022

cs50.harvard.edu/x/2022/psets/1/cash/

Cash

▶ Did you start this problem in 2021?

Getting Started

Open VS Code.

Start by clicking inside your terminal window, then execute `cd` by itself. You should find that its “prompt” resembles the below.

```
$
```

Click inside of that terminal window and then execute

```
wget https://cdn.cs50.net/2021/fall/psets/1/cash.zip
```

followed by Enter in order to download a ZIP called `cash.zip` in your codespace. Take care not to overlook the space between `wget` and the following URL, or any other character for that matter!

Now execute

```
unzip cash.zip
```

to create a folder called `cash`. You no longer need the ZIP file, so you can execute

```
rm cash.zip
```

and respond with “y” followed by Enter at the prompt to remove the ZIP file you downloaded.

Now type

```
cd cash
```

followed by Enter to move yourself into (i.e., open) that directory. Your prompt should now resemble the below.

```
cash/ $
```

If all was successful, you should execute

```
ls
```

and see a file named `cash.c`. Executing `code cash.c` should open the file where you will type your code for this problem set. If not, retrace your steps and see if you can determine where you went wrong!

Greedy Algorithms

When making change, odds are you want to minimize the number of coins you're dispensing for each customer, lest you run out (or annoy the customer!). 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 in that cashier's drawer are quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢). The problem to be solved is to decide which coins and how many of each to hand to the customer. Think of a "greedy" cashier as one who wants to take the biggest bite out of this problem as possible with each coin they take out of the drawer. 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!

Implementation Details

In `cash.c`, we've implemented most (but not all!) of a program that prompts the user for the number of cents that a customer is owed and then prints the smallest number of coins with which that change can be made. Indeed, `main` is already implemented for you. But notice how `main` calls several functions that aren't yet implemented! One of those functions, `get_cents`, takes no arguments (as indicated by `void`) and returns an `int`. The rest of the functions all take one argument, an `int`, and also return an `int`. All of them currently return `0` so that the code will compile. But you'll want to replace every `TODO` and `return 0;` with your own code. Specifically, complete the implementation of those functions as follows:

- Implement `get_cents` in such a way that the function prompts the user for a number of cents using `get_int` and then returns that number as an `int`. If the user inputs a negative `int`, your code should prompt the user again. (But you don't need to worry about the user inputting, e.g., a `string`, as `get_int` will take care of that for you.) Odds are you'll find a `do while` loop of help, as in `mario.c`!
- Implement `calculate_quarters` in such a way that the function calculates (and returns as an `int`) how many quarters a customer should be given if they're owed some number of cents. For instance, if `cents` is `25`, then `calculate_quarters` should return `1`. If `cents` is `26` or `49` (or anything in between, then `calculate_quarters` should also return `1`. If `cents` is `50` or `74` (or anything in between), then `calculate_quarters` should return `2`. And so forth.
- Implement `calculate_dimes` in such a way that the function calculates the same for dimes.
- Implement `calculate_nickels` in such a way that the function calculates the same for nickels.
- Implement `calculate_pennies` in such a way that the function calculates the same for pennies.

Note that, unlike functions that only have side effects, functions that return a value should do so explicitly with `return`! Take care not to modify the distribution code itself, only replace the given `TODO`s and the subsequent `return` value!
Note too that, recalling the idea of abstraction, each of your calculate functions should accept any value of `cents`, not just those values that the greedy algorithm might suggest. If `cents` is 85, for example, `calculate_dimes` should return 8.

▶ Hint

Your program should behave per the examples below.

```
$ ./cash
Change owed: 41
4

$ ./cash
Change owed: -41
Change owed: foo
Change owed: 41
4
```

How to Test Your Code

For this program, try testing your code manually–it's good practice:

- If you input `-1`, does your program prompts you again?
- If you input `0`, does your program output `0`?
- If you input `1`, does your program output `1` (i.e., one penny)?
- If you input `4`, does your program output `4` (i.e., four pennies)?
- If you input `5`, does your program output `1` (i.e., one nickel)?
- If you input `24`, does your program output `6` (i.e., two dimes and four pennies)?
- If you input `25`, does your program output `1` (i.e., one quarter)?
- If you input `26`, does your program output `2` (i.e., one quarter and one penny)?
- If you input `99`, does your program output `9` (i.e., three quarters, two dimes, and four pennies)?

You can also execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2022/x/cash
```

▶ Is `check50` failing to compile your code?

And execute the below to evaluate the style of your code using `style50`.

```
style50 cash.c
```

How to Submit

In your terminal, execute the below to submit your work.

```
submit50 cs50/problems/2022/x/cash
```