How to design programs

Now that we are armed with functions, let's dive into how to design solutions to problems. We'll propose a function design recipe that you can follow to help you reach solutions to problems. And in following the design recipe, I hope you can get past the problem of staring at a blank screen for far too long.

First, let's answer the question: "Just what is the goal of programming?"

In general, the goal of programming is to create a program to answer a problem. Just as in some of our machine problems, there is a problem statement that we have to convert into a working program. From the problem statement, we extract what is provided and what is asked of us, the programmer. We ask questions such as, "What operations are needed to be done?", or "How do we get from the given to the result?", and so on. Once we answer those questions, we should have enough information to finally solve the problem and implement it as a program.

While there are many ways to solve a problem, good solutions are better. But what are "good solutions", and why should we bother with building good solutions when a good enough answer suffices? Is it about performance? Not necessarily.

Beyond program performance, a good solution is maintainable. A well explained and documented program is easy to look back on and improve or fix, when needed. Since we aren't always able to capture the problem in our solution, being able to go back to change how things work is monumental. And going back to a well documented program is always better than going in blind.

Good programmers create programs not for themselves, but also for the benefit of others. And those "others" also include the future version of the programmer themselves. Don't trust yourself in thinking "Oh, I'll remember this solution even months away, since I wrote it myself." It almost always doesn't work that way, and you'll end up scratching your head at what you wrote yourself merely weeks ago.

Data and information

A program's general purpose is to take in some information, process it, and produce new information. That is, pretty much everything we do with programs is just passing information around and creating useful outputs out of it. Seeing as information is pretty central to programming, we must differentiate between *information* and *data*.

Information are facts about the problem domain. Setting aside programming jargon such as *integers* or *floats*, information is a plain measurement or value. Answers to questions like "how many legs does this table have?" or "what is the rate of interest in this account?" are called information.

To be even more concrete, here are examples of information:

- the interest rate in a savings account: 4.2% p.a.
- a message in a chat application: "What is up, chat?"
- a Pokémon's level in a game of Pokémon: Lv. 60

These are values that are relevent in the problem domain that they are in.

Meanwhile, **data** is information represented in our program. For a program to work with information, it must *convert* that information into data. It is then this data that our program manipulates to get to our solution, which is then converted back into infomation.

Here are couple examples of data:

- a `float` value of `4.2` to represent the interest rate
- a `char[]` array containing "What is up, chat?" to represent a chat message
- an `int` value of `60` to represent the Pokémon's level

The role of the programmer here is to decide on a good data representation of the information. A design decision must be made and justified; for instance, why the decision to use a `float` for interest rate? Why not have two `int` s? One to represent the whole part of the interest rate, and another to represent the decimals? Actually, choosing two `int` s isn't a wrong data representation at all, it is just a different approach to the same problem. These are the questions the programmer must answer in choosing a representation.

Since the data to information conversion is a pretty important part of designing a program, we'll note that decision in a comment that we'll call a "data definition".

```
// We use floats to represent interest rates.
```

Function design recipe

We'll enumerate the function design recipe below. As an example, we'll design a function that converts from a Celsius measurement to Fahrenheit. That is, given a temperature reading in Celsius, convert it into Fahrenheit.

The function design recipe is as follows:

1. Formulate data definitions

   We already know what these are, but this is the first step.

   We know that both Celsius and Fahrenheit readings are measured in degrees. These can also have decimal points, which a `float` can represent.

   ```
   // A Temperature is a float.
   ```

   Here we establish the information (a Temperature) is represented as a `float` .

2. Write down the function signature, statement of purpose, and the function header

   So far, all we're writing here are comments. Not code, just yet.

   A **function signature** is a comment that tells the readers of the design how many and what kind of input our function takes. It also tells the reader what kind of value it returns.

Since we establish that the Temperature is a `float`, our function should accept a `float`.

```
// float -> float
```

This says that our function will accept a `float` and returns another `float` In other words, we can give our function a Temperature (either Celsius or Fahrenheit) and it'll give us back a Temperature representation.

As another example, we can refer to the `strlen` function in C. This function would have the following function signature[1]:

```
// char[] -> int
```

Moving on, we tack on the purpose statement of our function. The **purpose statement** ideally summarizes the purpose of the function in a single line. It answers the question, "what does this function compute?"

For our temperature conversion function, a good purpose statement would be: "converts a Temperature from Celsius to Fahrenheit". That's it, plain and simple.

For `strlen`, it would be "calculates the length of the string, not including the terminator character".

Finally, we add the header. The **function header** is a simplistic function definition. It is here that we give our function a name, along with its parameters.

```c
float celsiusToFahrenheit(float temperature) {
    return 0;
}
```

For now, we return `0`. That is alright, our simplistic function definition doesn't have to be right, yet. We're also still at step 2.

With our function parameters named, we can go back to the purpose statement and use these names instead:

```
// converts `temperature` from Celsius to Fahrenheit
```

Or, for our example function `strlen`:

```c
// calculates the length of `str`, not including the
    terminator character
int strlen(char[] str) {
    return 0;
}
```

Hopefully, this makes the purpose statement a bit clearer in the context of our parameters.

3. Throw in examples of the function result

   So that readers of your design can confirm what they think the function does, we put in examples of function results:

```
// given: 25, expect: 77
// given: 18, expect 64.4
```

[1]Well, not actually... but it effectively is something like this.

As 25 °C converted is 77 °F, this is what we list down as the expected result. The same for 18 °C, which when converted is 64.4 °F.

4. *Take inventory*

   Here we take note of what's given, and what's needed to compute for the result that we want.

   In our temperature conversion problem, we have to think about what the problem gives us: the temperature in Celsius. Then we have to think about how we get from Celsius to Fahrenheit. We know that we are able to simply convert Celsius to Fahrenheit with a formula, so we look for that formula. After a quick search, we get `(celsius * 9/5)+ 32`. This formula will come in handy, when we reach the next step.

5. Implement

   Here we now implement the function. In step 2, we wrote out a simple function declaration that most likely returns the wrong answer. Here in step 5, we must replace that simple declaration with an actual implemention such that it does what it says on the tin.

   ```
   float celsiusToFahrenheit(float temperature) {
       return (temperature * 9 / 5) + 32;
   }
   ```

   Simply using the formula that we found when we took stock of what we have in step 4, this solution seems right!

   Now it's time to test whether it indeed is right in the next step.

6. Test

   We test that we indeed get the right answers based off the examples we provided in step 3.

   ```
   printf("%.1f", celsiusToFahrenheit(25)); // expecting 77
   ```

   ```
   77.0
   ```

   ```
   printf("%.1f", celsiusToFahrenheit(18)); // expecting 64.4
   ```

   ```
   64.4
   ```

   If we got a wrong result here, then we should consider the possibilities of an error being introduced. We should first recheck whether our function examples in step 3 were correct. Next, check whether our function definition is actually correct. In the case our function definition is wrong, this is called a *bug* in our code. And finally, we should consider the possibility that both our function examples *and* our function definition is wrong.

Putting all these together, we get the final product of our design process: a well-designed function.

```
// A Temperature is a float.

/* float -> float
 * converts `temperature` from Celsius to Fahrenheit
 * given: 25, actual: 77
 * given: 18, actual: 64.4
 */
float celsiusToFahrenheit(float temperature) {
    return (temperature * 9 / 5) + 32;
}
```