CS50 Principles of Good Design

Overview

Design is a very important aspect of programming and product development. Good design differentiates programs that work from programs that work well. Programs with robust, consistent, and nonrepetitive code are generally considered to be well-designed. Other measures of design are program **efficiency** and modularization. In order produce portable, scalable, and reusable code, we must keep design in mind while programming.

Key Terms

- efficiency
- magic numbers
- tradeoffs

Loops and Conditionals

Loops are very powerful and often used in programming. However, as they are somewhat costly, we should make sure we use them efficiently. We can check that we are doing so by asking ourselves the following questions: Are each of my loops essential? Can I combine any loops? And am I taking advantage of every iteration of my loops?

Along similar lines, it's important to use conditionals (if, else if, and else) efficiently. Consider a program that takes in a birth month and outputs a corresponding birthstone. We could implement it by checking if the user inputted `"january"`, then checking if the user inputted `"february"`, and so on until we reach `"december"`. But, if we already know that a user inputted `"january"`, why bother checking any of the other months? In this case, we could improve our program's design by using else if statements or switch statements instead of all if statements.

Constants

Magic numbers are hard-coded constants in code. We consider using them to be bad design since they reduce the scalability and readability of code. Furthermore, making changes to hard-coded values must be done manually. Using variables instead can facilitate making such changes. Additionally, we can use `#define` to define constants that will not change, like the number of letters in the alphabet (26) or the value of a nickel in cents (5). We do this at the beginning of our code with `#define` after our header files and outside of our main function.

Functions

It's typically good design to break code out into functions when needed. For instance, if we were performing the same set of mathematical operations to multiple different values, it might make sense to write the set of operations as a function and simply call that function multiple times. Similarly, it's also a good idea to break really long code into different files, linking between them so they can all work together smoothly. In these ways, we can make code that could otherwise be very tedious and complicated to get through be easier to make sense of.

Tradeoffs

Design is subjective and debatable. What one programmer may think is better design, another might fundamentally disagree with. For instance, someone could write code using an uncommon function that makes the program shorter and more concise. However, a person that had never seen the function before and had to look up its documentation could very well argue that the program was not written clearly.

At right are two different implementations for an algorithm that takes in a number of t-shirts and tells us how many boxes we need to store them, if our options are boxes that fit 12 shirts, 10 shirts, 3 shirts, or 1 shirt. Which is best designed? Well, different programmers could argue in favor of either one, since each come with their own set of **tradeoffs**. What do you think?

```
num_boxes += (shirts_left / 12);
shirts_left %= 12;
num_boxes += (shirts_left / 10);
shirts_left %= 10;
num_boxes += (shirts_left / 3);
shirts_left %= 3;
num_boxes += shirts_left;
```

```
int boxes[] = {12, 10, 3, 1};
for (int i = 0; i < 3; i++)
{
   num_boxes += (shirts_left / boxes[i]);
   shirts_left %= boxes[i];
}
```