3_Variables Mutability_en

May 2, 2023

1 Variables and mutability

2 Immutable and mutable variables

By default variables are immutable. This is one of many nudges Rust gives you to write your code in a way that takes advantage of the safety and easy concurrency that Rust offers. However, you still have the option to make your variables mutable.

This example shows how the compiler helps you find errors in your programs. Even though compiler errors can be frustrating, they only mean your program isn't safely doing what you want it to do yet; they do not mean that you're not a good programmer! Experienced Rustaceans still get compiler errors. Wrong code sample:

[ ]:
```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
main();
```

It's important that we get compile-time errors when we attempt to change a value that we previously designated as immutable because this very situation can lead to bugs. If one part of our code operates on the assumption that a value will never change and another part of our code changes that value, it's possible that the first part of the code won't do what it was designed to do. In Rust, the compiler guarantees that when you state that a value won't change, it really won't change. That means that when you're reading and writing code, you don't have to keep track of how and where a value might change. Your code is thus easier to reason through.

You can make them mutable by adding `mut` in front of the variable name. In addition to allowing this value to change, `mut` conveys intent to future readers of the code by indicating that other parts of the code will be changing this variable value.

[ ]:
```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

```
main();
```

There are multiple trade-offs to consider when using mutability in addition to the prevention of bugs. For example, in cases where you're using large data structures, mutating an instance in place may be faster than copying and returning newly allocated instances. With smaller data structures, creating new instances and writing in a more functional programming style may be easier to think through, so lower performance might be a worthwhile penalty for gaining that clarity.

3 Constants

Constants aren't just immutable by default—they're always immutable.

You declare constants using the `const` keyword instead of the `let` keyword, and *the type of the value must be annotated.*

[ ]:
```
const MAX_POINTS: u32 = 100_000;
```

Constants can be declared in any scope, including the global scope, which makes them useful for values that many parts of code need to know about. Constants may be set only to a constant expression, not the result of a function call or any other value that could only be computed at runtime.

4 Shadowing

You can declare a new variable with the same name as a previous variable, and the new variable shadows the previous variable. Rustaceans say that the first variable is shadowed by the second, which means that the second variable's value is what appears when the variable is used. We can shadow a variable by using the same variable's name and repeating the use of the let keyword as follows:

[ ]:
```
fn main() {
    let x = 5;

    let x = x + 1;

    let x = x * 2;

    println!("The value of x is: {}", x);
}
main();
```

By using `let`, we can perform a few transformations on a value but have the variable be immutable after those transformations have been completed.

The other difference between `mut` and shadowing is that because we're effectively creating a new variable when we use the `let` keyword again, we can change the type of the value but reuse the same name.

[ ]:
```
let spaces = "   ";
let spaces = spaces.len();
```

5 Exercices

Exercice 1: Correct the following code:

[ ]:
```
let pair = (5, 6);
let (mut x, y) = pair;
x += 5;
y = x;
println!("Coordinates: ({}, {})", x, y); // expecting (10, 10)

let x = x;
x = x + 5;
println!("x: {}", x); // expecting x: 15
```

solution:

[ ]:
```
let pair = (5, 6);
let (mut x, mut y) = pair;
x += 5;
y = x;
println!("Coordinates: ({}, {})", x, y);

let x = x;
let x = x + 5;
println!("x: {}", x);
```