Chapters 3&4

julia dreiling

10.03.25

Chp3 Overview

Topics Addressed:

- Variable declaration
  - Consts
  - *Mutability vs Shadowing*
- Data Types
  - Type annotation
- Function declaration and use
  - *Statements vs Expressions*
- Control Flow
  - If-else
  - Loops; for, while, loop

Mutability vs Shadowing

Variables are by default, immutable. The `mut` keyword used to indicate that they can be rebound to a different value of the same type as the initial value.

Shadowing, is in essence re-declaring a variable. The only keyword at play here is `let`. This allows us to change the type of value that variable is bound to as well as what the value is.

Mutability vs Shadowing

Code example:

```
let a = 5;
let mut b = 6;
```

Now a cannot be assigned any value without shadowing.
b can be assigned a new value of type int.

This is valid rust code:

```
b = 7; // rebound value for b
let a = “apple”; // shadowed a
```

This is not:

```
b = “banana”; // you cannot change type
a = 9; // immutable var
```

Statements vs Expressions

Rust makes a distinction between statements and expressions, which allows it to have some funky behavior.

What falls into these two categories?

Statements:

- Function Declarations
- Variable Declarations

Expressions:

- Calling functions
- Most normal operations

Statements vs Expressions

Example Code:

```
let apples = [“fuji”, “gala”,“evercrisp”];
fn pick_an_apple(count : int32) {
    let apple_type = apples[count];
    println!(“You picked a {}!”, apple_type);
    count++;
}
let mut apple_count = {
                        let x = -2;
                        x + 2
                        } ;
pick_an_apple(apple_count);
```

Statements vs Expressions

A weirder code example:

```
fn venli() -> char{
    'a'
}
```

compiles

```
fn venli() -> char{
    'a';
}
```

doesn't

WHY?

Statements vs Expressions

```
fn venli() -> char{
    'a'
}
```

`'a'` is an expression, and because it's at the end of the function, it implicitly is returned.

```
fn venli() -> char{
    'a';
}
```

`'a';` is a statement, and thus returns *nothing*, leaving the function looking for a return value.

Long story short:

Statements do NOT return a value.

Expressions do.

Chapter 4 Overview

Topics Addressed:

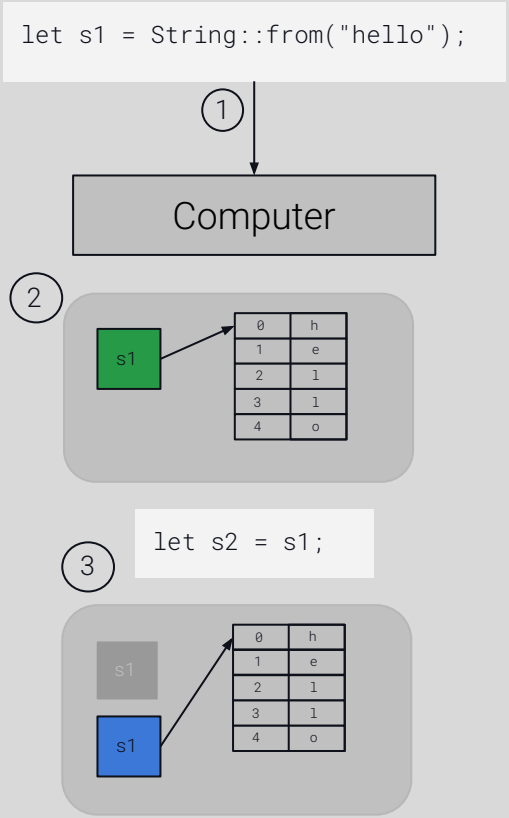
- *Ownership*
  - Value vs Variable
  - Implicit `drop` call
  - Copy Trait
- *References*
  - Mutability
- Slice
  - Strings, String Literals, and Arrays

Ownership

To handle memory management, Rust implements the concept of “ownership” of values/memory. Values/data can only have one owner. This is so that when the owner goes out of scope, that memory will be freed only once. Memory is freed via a `drop` call, which Rust implicitly will enact when the scope of a variable ends.

Ownership

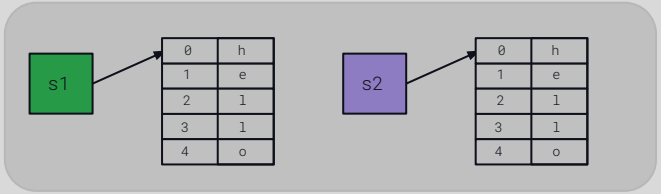
1. Program requests space on the heap for something that doesn't have a copy trait.
2. That memory is allocated on the heap and the variable name is bound to that part of memory
3. When ownership of data changes hands, the previous owner is no longer a valid variable to call within the system, and the new owner's scope and life is responsible for that data.



Ownership

What if you want the value but not the memory it is at? Clones!*

```
let s1 = String::from("hello");
let s2 = s1.clone();
```

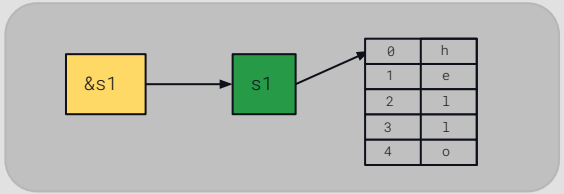


*Also data types annotated with the trait `Copy` automatically do this when you try to assign them to another variable's value.

References

In other words: what if we wanted to borrow something, rather than own it?

```
let s1 = String::from("hello");
// now use &s1 to reference this
value rather than taking ownership of
it
```



These references are inherently immutable like all other variables within Rust.

As soon as the variable passes out of scope though, its reference is no longer valid and cannot be used (thus avoiding dangling references).

References

What if you want to muck around with your value you're borrowing?

You must set:

1. Initial variable as mutable AND
2. Reference as mutable

```
//1
let mut s = String::from("hello");
//2
change(&mut s);
```

To avoid a data race, you can only have ONE mutable reference to some data within a scope.

https://www.youtube.com/watch?v=k1cjQDTD_ww

Summary

Variables: defined by let, default immutable, can be mutable.

Const: defined by const, inherently immutable.

Shadowed variables: use the same name as a previous variable, overshadowing it in the view of the program. Allows type changing.

Values: bound to variables, stored in heap or stack depending on their traits.

Ownership: who has the pointer to the place in memory.

Reference: indicated by & symbol, default immutable, only one mutable reference per scope, colloquially “borrowing” a value.