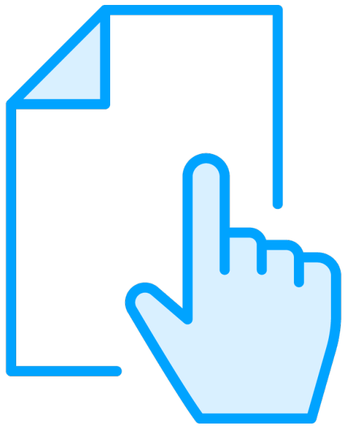
Ownership and Borrowing

Zachary Bennett
Lead Software Developer

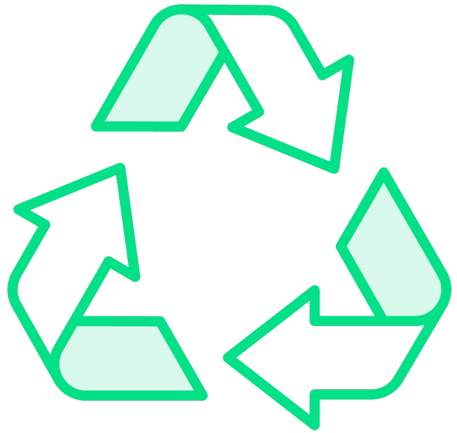
@z_bennett_ | zachbennettcodes.com

Approaches to Memory Management



Manual Memory Management

In this category, programming languages require the developer to manage memory.



Garbage Collected

Languages in this category use a garbage collector – a program that is responsible for managing memory

Ownership gives us safety - without a Garbage Collector!

The Rule of Three

Every value is owned

One owner at a time

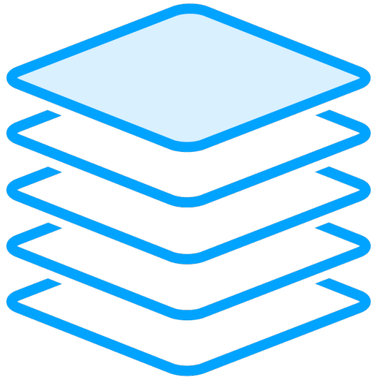
When the owner leaves scope, what is owned gets dropped!

Scope

Scope is the range within a program for which an item (a value) is valid.

Citation: Rust Docs, “What is Ownership”, https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

The Stack and the Heap



The Stack

Memory that is immediately available to your program and that is ordered (last-in first-out)

The Heap

A chunk of memory that is broadly available to your program. Your program must request what it needs

The Stack vs. the Heap

Stack	VS	Heap
Fast – the memory allocator does not need to request access		Slower – must request access and follow a pointer
Contains ordered values - LIFO		Values are not necessarily ordered
Think ”compile-time” - everything on the stack has a known, fixed-size		Memory is allocated at runtime

```
// Example of simple stack-based scope


{


  let my_number = 1;


}


println!(“{}”, my_number);
```

- Let’s imagine this code resides within a “main” function
- Defining a scope using curly braces.
- Within this scope, “my_number” owns this value.
- This will not compile! The value pointed to by the “my_number” variable is no longer in scope. The value has been “dropped”.

Stack-based ownership is simple

```
// Ownership on the heap


{


  let my_string = String::from(“Hello”);


}
```

- **Allocate memory on the heap for a String**

- **The ”my_string” variable is no longer valid – behind-the-scenes, Rust calls the “drop” function which frees up this memory!**

- **Easy - no need to manually free the memory**

```
// Moves/Copies – transferring ownership

// On the stack

{

  let a = 1;

  let b = a;


}

// On the heap

{

  let str_a = String::from(“Hello”);

  let str_b = str_a;

}
```

- Variables ”a” and “b” can both be used as these are simple values that are stored on the stack. Here, “b” is a copy (think shallow-copy) of “a”.

- Here the “str_a” variable has stack data that points to data on the heap. The stack data is copied to “str_b”. The heap data is left untouched. This is known as a “move”.

- Use of the “str_a” variable is now invalid.

What about deep copies?

```
// Cloning – making deep copies




{
  let str_a = String::from(“Hello”);
  let str_b = str_a.clone();
}
```

- Sometimes we need to make a copy of data on the heap.
- We can do this on data structures that implement the ”Clone” trait.
- Here we make a deep copy of a string.
- The variables ”str_a” and “str_b” are both valid and both reference deeply-cloned data on the heap.

Demo

Edge cases and “gotchas”

Ownership in Rust

- Moving
- Copying
- Cloning

References and Slices

Categories

References

Immutable or mutable links back to some piece of data

Slices

A special reference that points to a subset of data

Rules with References

One mutable reference or any amount of immutable references

References HAVE to be valid

```
// References and borrowing

fn concat(a: &mut String) {

  a.push_str(“bar”)

}


let foo = String::from(“foo”);

concat(&foo); // We can’t do this!


let mut other_foo = String::from(“foo”);

concat(&mut other_foo);
```

◀ **Function that takes a String reference**

◀ **Immutable reference**

◀ **Mutable reference**

Slice – a reference to a contiguous sequence

```
// Slices – strings and arrays


let foobar = String::from(“foobar”);

let foo = &foobar[0..4];

let bar = &foobar[3..]


let my_arr = [ 1, 2, 3 ];

let one = &my_arr[..1];

let my_mut_arr = [ 1, 2, 3];

let my_mut_slice = &mut my_mut_arr[..1];

my_mut_slice[0] = 2;

// my_mut_arr now looks like: [2, 2, 3]
```

- **String variable**
- **A string slice that references the string “foo”**
- **A string slice that references the string “bar”**
- **Arrays have slices too!**
- **Mutable array slice**
- **Mutate by an index (unsafe - this is just an example)**

Demo

References

- Immutable
- Mutable
- Slices as partial references