
PROGRAMMING
WITH
RUST
DONIS MARSHALL

Programming with Rust

Code Listing 8.6. An example of copy semantics

```
let width = 10;
let height = width;
println!("W {} : H {}", width, height);
```

Clone Trait

Implement the Clone trait when a deep copy is required.

The String type, for example, implements the Clone trait. Why? Strings include a pointer to the underlying string, which is on the heap. For this reason, the String type does not support copy semantics. If allowed, this would create problematic dependencies. Instead, the String type implements a deep copy using the Clone trait. Here are the general steps for cloning Strings:

1. For the target String, allocate memory on the heap for the String buffer.
2. Copy the buffer from the originating String into the new buffer.
3. Update the target String to reference this memory.

The result of this are Strings with the same value, `stringa` and `stringb`, at different locations in memory (see Figure 8.3).

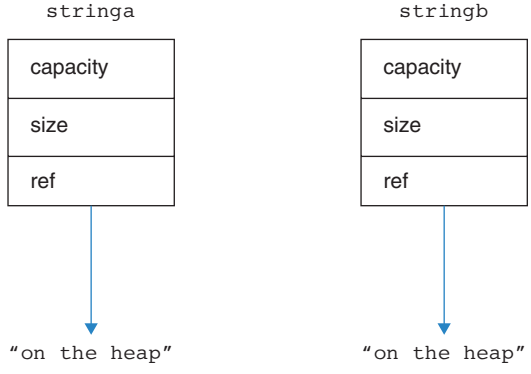

Figure 8.3. Showing result of cloning stringb from stringa

Cloning must be invoked explicitly with the `clone` function, which is a member of the Clone trait. In Listing 8.7, the `stringa` variable is cloned to initialize `stringb`.

Code Listing 8.7. Cloning a String

```
let stringa = String::from("data");
let stringb = stringa.clone();
println!("{} {}", stringa, stringb)
```

Copy Trait

You can assign the Copy trait to types that support a shallow copy.

Here is the Copy trait:

```
pub trait Copy: Clone { }
```

Notice that the Clone trait is a supertrait of the Copy trait.

Structs support move semantics by default. An assignment transfers ownership. See Listing 8.8.

Code Listing 8.8. When assigned, structs are moved by default.

```
struct Transaction {
    debit: bool,
    amount: f32,
}

fn main() {
    let t1=Transaction{debit: true, amount: 32.12};
    let t2=t1;  // moved
}
```

The Copy trait can be applied to structs if each field supports copy semantics. This is a trivial implementation and can be done with the `#[derive]` attribute, as shown in Listing 8.9.

Code Listing 8.9. Applying the Copy trait using the derive attribute

```
#[derive(Copy, Clone)]
struct Transaction {
    debit: bool,
    amount: f32,
}

let t2 = t1;  // copied
```

Let's add a `description` field, as a String type, to `Transaction`, as shown next. This would appear to be a reasonable addition to our structure. However, this means the struct no longer supports copy semantics. This is because the added String field does not support copy semantics.

```
struct Transaction {
    description: String,
    debit: bool,
    amount: f32,
}
```

However, references support copy semantics. In Listing 8.10, we have modified the structure to change the String to &String. The struct now supports copy semantics, which is provided with the `#[derive]` attribute.

Code Listing 8.10. This is a struct that supports copy semantics.

```
#[derive(Copy, Clone)]
struct Transaction<'a> {
    description: &'a String,
    debit: bool,
    amount: f32,
}
 . . .
let t2 = t1; // copied
```

You can implement the Copy trait manually, not using the `#[derive]` attribute. As a marker trait, there is no implementation for the Copy trait. The existence of the Copy trait instructs Rust to perform a bitwise copy. However, the Clone trait must be implemented as the supertrait. This means implementing the `clone` function. The function should return the current object by value, which is consistent with a bitwise copy. See Listing 8.11 for the implementation of the Copy trait for `Transaction`.

Code Listing 8.11. Explicit implementation of the Copy trait

```
impl Copy for Transaction {}

impl Clone for Transaction {
    fn clone(&self) -> Transaction {
        *self  // return current object by value
    }
}

struct Transaction {
    debit: bool,
    amount: f32,
}
```

Clone Trait

The Clone trait implements a deep copy for types that do not support a bitwise copy.

For the Clone trait, you must implement the clone function, which returns the cloned value. The implementation must also remove any dependencies caused by pointers.

For struct, assuming all the fields are cloneable, you can simply add the `#[derive]` attribute for the Clone trait. This attribute calls the `clone` method on each field.

Every member of the `Transaction` type is cloneable. This means you can apply the `#[derive(Clone)]` attribute. See Listing 8.12.

Code Listing 8.12. Applying the `Clone` trait to Transaction

```
#[derive(Clone)]
    struct Transaction {
        description: String,
        debit: bool,
        amount: f32,
    }

...
    let t1 = Transaction {
        description: String::from("ATM"),
        debit: true,
        amount: 32.12,
    };
```

If desired, you can explicitly implement the Clone trait for `Transaction`. In our implementation, the description field is reset when cloning a `Transaction`. The other fields are just copied for cloning. See Listing 8.13 for the implementation of the trait.

Code Listing 8.13. Explicit implementation of the Clone trait

```
struct Transaction {
    description: String,
    debit: bool,
    amount: f32,
}

impl Clone for Transaction {
    fn clone(&self) -> Transaction {
        let temp = Transaction {
            description: String::new(),
            debit: self.debit,
            amount: self.amount,
        };
        return temp;
    }
}
```

Summary

Ownership is a seminal feature of Rust programming and enforces safe and secure programming techniques for memory management. This prevents many of the common memory-related problems that occur in other languages.

The borrow checker enforces the rules of ownership at compile time, preventing memory errors from seeping into binaries and occurring at runtime.

Move semantics is the default behavior. When you assign a value, ownership is moved. If a reference is assigned, a borrow occurs and ownership is not transferred.

Alternatively, types that have the Copy trait support copy semantics. Copy semantics is implicit and performs a bitwise copy, which is a shallow copy. This is appropriate for scalar types.

You can apply the Copy trait with the `#[derive(Copy, Clone]` attribute. However, you can manually implement either trait. It is more common to implement the Clone trait.

You implement the Clone trait for types that require a deep copy. The actual implementation of the trait depends on the type.

Lifetimes also provide memory safeness. Ownership and lifetimes together provide a complete solution for memory management in Rust. The conversation on lifetimes is next.

9

Lifetimes

An important promise from Rust to developers is memory safeness, which is enforced within the ownership model. Ownership and borrowing are two of the three pillars in the ownership model. Lifetimes is the final pillar to complete our conversation about the subject. Despite this laudable goal, there is a general misunderstanding of lifetimes within the Rust community. Most Rust developers are satisfied knowing just enough about lifetimes to prevent the borrow checker from complaining. The objective of this chapter is to provide clarity on this important topic. Lifetimes should be a benefit, not something to ignore.

Lifetimes is another unique feature within the Rust language. Other languages, such as C, C++, and Java, do not have a lifetime feature, or something similar. For most developers, this means a lack of familiar context as a starting point to understanding lifetimes. No worries. You will learn both the context and syntax of lifetimes in this chapter.

The principal objective of lifetimes is to prevent dangling references. *That's it!* Keep this in mind especially when you're deep in the vagaries of lifetimes. What is a dangling reference? A dangling reference occurs when a reference outlives a borrowed value. At that time, the reference points to invalid memory.

Listing 9.1 shows an example of a dangling reference.

Code Listing 9.1. `ref1` **becomes a dangling reference**

```
fn main() {
    let ref1;
    { // ---------------------  inner block (start)
        let num1=1;
        ref1=&num1;
    } // ---------------------  inner block (end)
    println!("{}", ref1);  // dangling reference
}
```

Here, `ref1` is a reference and it borrows `num1`, which is then dropped at the end of the inner block. Therefore, the reference to `num1` in the `println!` macro outlives the borrowed value—the very definition of a dangling reference. If allowed to continue, the program is no longer memory safe.