**Who is the owner: Rust Move Semantics**

[[](https://medium.com/@technocrat21?source=post_page-----9f2fa71859e7--------------------------------)](https://medium.com/@technocrat21?source=post_page-----9f2fa71859e7--------------------------------)

[[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----9f2fa71859e7--------------------------------)](https://medium.com/coderhack-com?source=post_page-----9f2fa71859e7--------------------------------)

[Amay B](https://medium.com/@technocrat21?source=post_page-----9f2fa71859e7--------------------------------)

·

Follow

Published in

[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----9f2fa71859e7--------------------------------)

·

9 min read

·

Oct 3

39

1



Photo by [Timon Studler](https://unsplash.com/@derstudi?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

In Rust, types have “move semantics” by default. This means that when a value is assigned to a new variable or passed as an argument to a function, the value is moved rather than copied. The original variable is invalidated, and only the new variable can be used to access the value.

Move semantics allow for a few key benefits:

1. Performance: Moves do not require a copy of the value, so they are very fast. This allows Rust to provide zero-cost abstractions without penalizing performance.
2. Memory efficiency: Without moves, copying a value would require allocating space for a full copy whenever a value is passed around or reassigned. Moves avoid these extra allocations.
3. Uniqueness: Moves enforce unique ownership of a value, ensuring that only one variable directly owns the value at a given time. This property powers many of Rust’s memory safety guarantees.

For example, consider this simple Rust program:

struct Foo {  
 x: i32,   
 y: String,   
}  
  
fn main() {  
 let mut f = Foo { x: 1, y: String::from("Hello") };  
 do\_something(f);  
  
 println!("f.x: {}, f.y: {}", f.x, f.y);  
}  
  
fn do\_something(f: Foo) {  
 println!("f.x: {}, f.y: {}", f.x, f.y);  
}

When we call do\_something(f), the value of f is moved into the function argument f. So inside do\_something(), we can use f to access the value, but outside the call, f is no longer valid! If we try to use it, we'll get a compile-time error.

This shows how moves allow us to have unique ownership — either f or the function argument f owns the value, but not both. In the next sections, we'll explore moves in Rust in more detail.

**Move by default**

In Rust, types have “move semantics” by default. This means that when you assign a value to a new variable or pass it into a function, the value will be *moved* rather than copied.

For example, say we have a struct like this:

struct Foo {  
 message: String  
}

If we assign f1 to a new variable f2, the String value will be moved out of f1 and into f2:

let mut f1 = Foo {  
 message: String::from("Hello")  
};  
  
let f2 = f1;  
  
println!("f1: {}", f1.message); // This is invalid!  
println!("f2: {}", f2.message); // Prints "Hello"

The f1 variable is now invalid because ownership of the message value has moved into f2. This happens for all types in Rust by default.

To demonstrate this further, we can have a function take ownership of a value:

fn take\_foo(foo: Foo) {  
 println!("Inside function: {}", foo.message);  
}  
  
let f1 = Foo {  
 message: String::from("Hello again")   
};  
  
take\_foo(f1);  
  
println!("f1: {}", f1.message); // This is invalid!

Again, the f1 variable is invalidated after we call take\_foo() and move its value. The ownership of message moves into the function call.

This move-by-default model enables Rust to have memory safety guarantees without requiring garbage collection.

**Move into function arguments**

When we pass a struct instance as an argument to a function, the struct is moved into the function. This means the original variable that was passed is no longer valid and cannot be used again.

For example:

struct Toy {  
 name: String  
}  
  
fn take\_toy(toy: Toy) {  
 println!("Taking ownership of {}", toy.name);  
}  
  
fn main() {  
 let toy = Toy {  
 name: String::from("Buzz Lightyear")  
 };  
  
 take\_toy(toy);  
  
 println!("toy is : {:?}", toy); // Compilation error! `toy` is no longer valid  
}

Here we define a Toy struct with a String name field. We then define a take\_toy() function that takes a Toy struct as an argument. Within the function, we print a message saying we've taken ownership of the toy.

In the main() function, we create a Toy instance and pass it into the take\_toy() function. After this call, we then try to print the toy variable again, but this results in a compilation error. This is because the toy instance was moved into the take\_toy() function, invalidating toy.

The ownership of the struct was moved into the function, so we could no longer use the original variable. This demonstrates how moves affect function arguments in Rust.

To fix this, we’d need to return the Toy from the take\_toy() function so ownership comes back out, or we'd need to pass a reference to the Toy instead of moving it.

**Returning values and move**

When a function returns a struct, the value moves out of the function and the function is invalidated. For example:

fn return\_struct() -> Vec<i32> {  
 let vec = Vec::new();  
 vec.push(1);  
 vec.push(2);  
 vec  
}  
  
fn main() {  
 let vec = return\_struct();  
 println!("{:?}", vec); // Prints [1, 2]  
}

Here, the Vec is returned from the function, moving the value out. If we tried to use vec again within the function, it would no longer be valid.

fn return\_struct() -> Vec<i32> {  
 let mut vec = Vec::new();  
 vec.push(1);  
 vec.push(2);  
  
 let vec2 = vec; // Error! `vec` has been moved.  
  
 vec //this use is not valid since vec already moved.  
}

This would result in a compiler error, as vec has been moved and invalidated.

To demonstrate this further, we can return a struct with a String field:

struct ReturnStruct {  
 message: String  
}  
  
fn return\_message() -> ReturnStruct {  
 let mut return\_struct = ReturnStruct {  
 message: String::from("Hello!"),  
 };  
  
 return\_struct.message = String::from("Goodbye!");  
  
 return\_struct  
}  
  
fn main() {  
 let ret = return\_message();  
 println!("{}", ret.message); // Prints "Goodbye!"  
}

Here, the ReturnStruct is returned from the function, moving the value out. The modifications we made to return\_struct within the function are moved out, and we see the modified value “Goodbye!” printed.

However, if we tried to use return\_struct again within the function, it would be invalid:

fn return\_message() -> ReturnStruct {  
 let mut return\_struct = ReturnStruct {  
 message: String::from("Hello!"),  
 };  
  
 return\_struct.message = String::from("Goodbye!");  
  
 let return\_struct2 = return\_struct; // Error! `return\_struct` has been   
 // moved.  
  
 return\_struct  
}

This would result in a compiler error as return\_struct has been moved.

**Cloning to prevent move**

To prevent a move from occurring and invalidating an original variable, we can call the .clone() method to perform a *deep copy*, creating a new, independent instance of the type.

For example, say we have a struct like this:

struct Foo {  
 bar: String  
}

If we call a function that takes Foo as an argument, the Foo value will be moved into the function, invalidating our original variable:

fn take\_foo(mut foo: Foo) {  
 foo.bar = "Changed!".to\_string();  
}  
  
let mut foo = Foo {  
 bar: "Hello".to\_string()  
};  
  
take\_foo(foo);  
  
println!("foo.bar: {}", foo.bar); // Will not compile! `foo` was invalidated by the move

To prevent this, we can call .clone() to perform a deep copy before passing the struct into the function:

fn take\_foo(mut foo: Foo) {  
 foo.bar = "Changed!".to\_string();  
}  
  
let mut foo = Foo {  
 bar: "Hello".to\_string()   
};  
  
take\_foo(foo.clone());  
  
println!("foo.bar: {}", foo.bar); // Prints "Hello"

Now we have two independent instances of Foo, and our original foo remains valid after the function call.

Using .clone() allows us to choose move semantics or copy semantics explicitly in Rust. We have granular control over the memory footprint of our programs by choosing when deep copies should (and should not!) occur.

**Using the Move Keyword**

The move keyword can be used to force a move to occur, even when a copy would normally happen. This is useful in a few situations.

For example, say we have an Option<String> and we want to pass it into a function that will always move the inner String out of the Option, invalidating the Option. We can use move to force this:

fn consume\_string(option: Option<String>) {  
 let string = option.move(); // Forces a move  
 println!("{}", string);  
}  
  
let mut opt = Some(String::from("Hello"));  
consume\_string(opt);  
println!("{:?}", opt); // None

Here, even though Option<T> is a copy type, the move() call will move the inner String out of the Option, consuming it.

The move keyword can also be used when you have a closure that borrows from the environment, to force those borrows to be moved into the closure:

fn do\_something() {  
 let mut x = String::from("Hello");  
 let mut y = String::from("World");  
  
 let closure = move || {  
 println!("{} {}", x, y);  
 };  
  
 closure();  
 println!("x: {}", x); // This will not compile! x has been moved  
}

The move closure will move the references to x and y into the closure, invalidating them outside the closure.

Using the move keyword appropriately is an important part of understanding move semantics in Rust. It gives you more control over moving values when copies would normally occur.

**Copy Types**

Certain types in Rust implement the Copy trait, which means their values are simply copied when moved, leaving the source intact. Types that implement Copy include primitive types like integers, floats, booleans, characters, etc.

For example, if we have a variable x with a value of 5 (an integer) and we assign y = x, this will copy the value of x into y. x will retain the value 5 even after the assignment.

let x = 5;  
let y = x;  
  
println!("x is still {}", x); // Prints `x is still 5`

This is different from types that do not implement Copy, such as Strings. For example:

let mut x = String::from("Hello");  
let y = x;   
  
println!("x is {}", x); // Prints `x is `

Here, assigning y = x performed a *move* on x, invalidating it. x is no longer valid after the move.

Any type composed entirely of Copy types is also Copy. For example:

struct Point {  
 x: i32,  
 y: i32  
}  
  
let origin = Point { x: 0, y: 0 };   
let point = origin;  
  
println!("origin is still {:?}", origin);   
// Prints `origin is still Point { x: 0, y: 0 }`

Here, the Point struct is composed only of i32 fields, so it implements Copy and the assignment to point simply copies the values, leaving origin valid.

To summarize, types that implement Copy in Rust have their values copied on assignment, rather than moved. This leaves the source value intact after the assignment.

**Additional Examples and Exercises**

Now that we’ve covered the basics of move semantics in Rust, let’s go through some additional examples and exercises to help strengthen your understanding.

**Example 1: A Custom Copy Type**

Let’s say we want to create a custom Copy type. We can do this by making a trivial struct and implementing the Copy trait for it:

struct Foo;  
  
impl Copy for Foo {}

Now, if we have two Foo values, assigning one to the other will perform a copy, leaving the original intact:

let x = Foo;  
let y = x;  
  
println!("x is still valid: {:?}", x);  
println!("y is also valid: {:?}", y);

This will print:

x is still valid: Foo   
y is also valid: Foo

*Exercise:*

*Create your own trivial Copy type and show an example of it performing a copy on assignment.*

**Example : Returning a Moved Value**

Here’s an example of a function that returns a String, moving its value out:

fn get\_string() -> String {  
 let mut s = String::from("Hello");  
 s.push\_str(" world!");  
 s   
}  
  
fn main() {  
 let s = get\_string();  
 println!("{}", s);  
 // Prints "Hello world!"  
  
 println!("{:?}", s);  
 // Prints "Hello world!"  
  
 let s2 = s; // Move occurs here, invalidating `s`  
 println!("{:?}", s);  
 // Prints "" - `s` is now invalid!  
}

We call get\_string(), which returns a String by value, moving it out. We store that in s and print it, showing it has the expected value. Then, when we assign s to s2, a move occurs, invalidating s. We see this when we try to print s again and get an empty string.

**Summary**

In this article, we explored the core concept of *move semantics* in Rust. Move semantics enable Rust to have zero-cost abstractions without a garbage collector.

We saw that by default, assigning or passing a value to another variable or function will *move* the value, invalidating the source. This prevents accidental copying of large values, improving performance.

However, we can use the .clone() method to force a *deep copy* of a value, preventing a move from occurring. We also saw the move keyword, which will force a move even when a copy might normally occur.

Certain types implement the Copy trait, meaning they are small enough to be cheaply copied instead of moved. Primitive types like isize and char are examples of Copy types.

Move semantics power many of Rust’s idioms and enable efficient, zero-cost abstractions. Understanding exactly when values will be moved and when they will be copied is an important part of writing idiomatic Rust code.

I hope this article provides a solid overview of the move semantics in Rust! Let me know if you have any other questions.

I hope this article has been helpful to you! If you found it helpful please support me with 1) click some claps and 2) share the story to your network. Let me know if you have any questions on the content covered.

Feel free to contact us at coderhack.com(at)xiv.in