Memory safety through Linearity, Ownership & Borrowing

JAKUB SCHWENKBECK

Memory safety is one of the most important tasks in software development. Unsafe handling of memory can lead to errors such as *use-after-free*, *double-free*, or memory leaks. These errors can cause program crashes and unpredictable behavior or even open up critical security vulnerabilities. Guaranteeing memory safety is therefore important to create robust, reliable, and secure applications. This paper explores how Rust leverages type theory to ensure memory-safe code without typical trade-offs.

CCS Concepts: • **Theory of computation** → **Type theory**; **Linear logic**.

ACM Reference Format:
Jakub Schwenkbeck. 2025. Memory safety through Linearity, Ownership & Borrowing . 1, 1 (August 2025), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Common Approaches to Memory Safety

In software development and the development of programming languages, there are different common paradigms to adress memory management. Two of the most prominent approaches are manual memory management and the use of garbage collection (GC).

1.1 Manual Memory Management

Manual memory management requires the programmer to explicitly allocate and release memory. Typical this includes functions such as `malloc()` and `free()` in C, or `new` and `delete` in C++. This method offers developers fine-grained control and can therefore create highly efficient programs. However, it is also error-prone: developers can forget to release memory, causing leaks, or they free memory too early, leading to undefined behavior and possible security issues. The complexity of managing memory manually significantly increases the likelihood of these bugs.

1.2 Garbage Collection

Garbage collection, on the other hand, moves the task of memory management to the runtime system.Well known Languages such as Java, Python, C#, or JavaScript use Garbage Collection which identifies and reclaims unused memory periodically. This reduces the risk of memory-related errors and simplifies the development process. However, it also introduces runtime overhead and can make performance less predictable, since the exact timing of memory reclaims is not in the programmer's control. Furthermore, GC-based languages rely on a runtime environment, which is not ideal for e.g. embedded systems.

2 Theoretical Background of Rust's Type System

The Rust programming language introduces an alternative approach to memory safety that is neither fully manual nor reliant on garbage collection. Instead Rust leverages concepts from type theory such as linearity and affinity, as well as the principle of Resource Acquisition Is Initialization (RAII), to enforce memory safety at compile time without requiring a runtime garbage collector.

2.1 Linearity

The concept of linearity is the idea that each variable should be used exactly once in a program. Enforcing such a restriction ensures exclusive access to resources, thereby preventing common errors such as double usage or data races. While pure linear type systems are rare in practical

2025. ACM XXXX-XXXX/2025/8-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

programming languages, they are studied extensively in theory. For instance, Linear Haskell is an experimental extension that incorporates strict linear types. Rust adapts this principle in a more pragmatic way.

2.2 Affinity

Rust does not implement strict linearity, but rather a relaxed form called *affinity*. Under this rule, variables may be used at most once, meaning that they can be dropped without explicit use. This approach avoids unnecessary errors while still ensuring memory safety. By default, Rust assumes that any variable not explicitly used will be automatically deallocated when it goes out of scope. This affine type system is the base of Rust's guarantees of safe and efficient resource management.

2.3 RAII

A central mechanism is Resource Acquisition Is Initialization (RAII). According to this principle, resources are bound to the lifetime of the objects that manage them. When an object goes out of scope, its destructor—implemented via the `Drop` trait in Rust—is automatically called, ensuring proper release of memory and other resources such as file handles or sockets. This happens deterministically and without the need for a garbage collector, combining efficiency with safety. Its wort mentioning that this concept is not exclusive to Rust as languages like C++ prominently use it too.

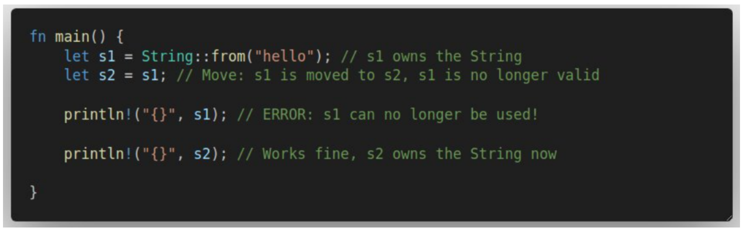
2.4 Stack and Heap Allocation

Rust also provides a clear model for memory allocation. Primitive values and small data structures are allocated on the stack by default, providing fast access and automatic deallocation once they go out of scope. More complex or dynamically sized data structures, such as `String`, `Vec`, or `Box<T>`, are allocated on the heap. In these cases, the stack holds a pointer to the heap data, which is automatically freed when the pointer itself leaves scope.

3 Ownership, Borrowing, and Lifetimes

3.1 Move Semantics

In Rust, when a value is assigned to another variable, ownerships moves. After a move, the previous variable is invalidated, which already prevents common bugs like data races, dangling pointers or double frees!



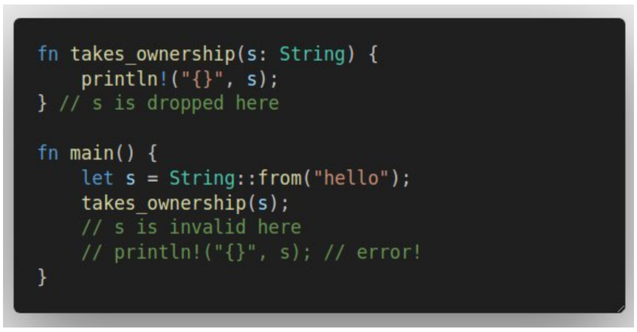
```rust
fn main() {
    let s1 = String::from("hello"); // s1 owns the String
    let s2 = s1; // Move: s1 is moved to s2, s1 is no longer valid

    println!("{}", s1); // ERROR: s1 can no longer be used!

    println!("{}", s2); // Works fine, s2 owns the String now

}
```

Fig. 1. Move Semantics in Rust

3.2 Ownership

Each value has a single owner. This ownership can be transferred by a move. When the owner goes out of scope, the value is dropped and the memory, stack and heap, is automatically freed. Moving ownership from variables invalidates them (as seen in the Move Semantics).

Ownership is e.g. transferred by reassigning variables (Fig. 1.) or when calling functions (Fig. 2.)



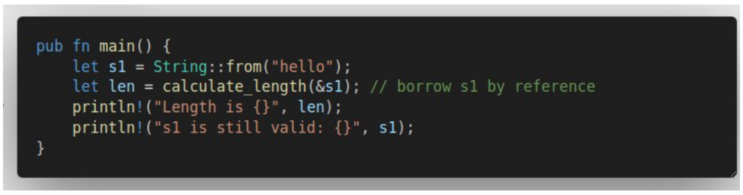
```
fn takes_ownership(s: String) {
    println!("{}", s);
} // s is dropped here

fn main() {
    let s = String::from("hello");
    takes_ownership(s);
    // s is invalid here
    // println!("{}", s); // error!
}
```

Fig. 2. Ownership transfer via function call

3.2.1 Copy and Clone Trait. To still be able to write code practically, some 'primitive types' like integers or booleans implement the so called Copy Trait. Those types are duplicated instead of moved and the ownership stays with both variables after the reassignment. As these types use a very small amount of memory space, it's not critical to duplicate them.
For more complex types, you can use the Clone trait to explicitly create a deep copy of the data. Unlike Copy, which happens automatically, calling .clone() could involve allocating memory or duplicating resources, and therefore needs to be explicitly denotated by the programmmer. Small, trivial types are copied implicitly, while larger or resource-heavy types must be cloned explicitly when a duplication is really intended.

3.3 Borrowing

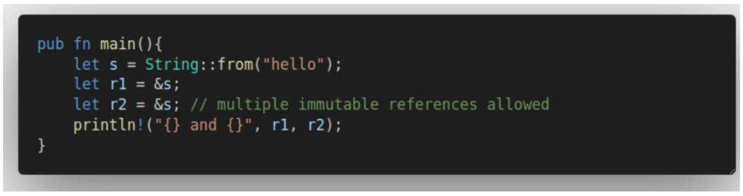
To be able to reuse a variable or resource e.g. after passing it as an argument in function calls, we introduce Borrowing. Borrowing allows the use of a value without taking its ownership by creating references. A reference is its own type and is noted with &. The actual value is accessible by dereferencing with *, but this does not transfer ownership ,it only lets you work with the value the reference points to. This is distinct from raw pointers known in C/C++, as Rust references are always safe and checked by the borrow checker.



```
pub fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1); // borrow s1 by reference
    println!("Length is {}", len);
    println!("s1 is still valid: {}", s1);
}
```

Fig. 3. Example of borrowing not taking ownership

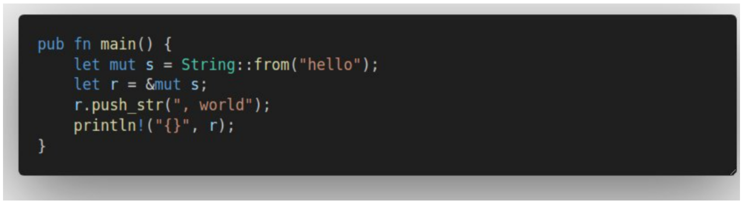
3.3.1 Immutable References. Default references are immutable (notation : &T). There can be many immutable references simultaneously, but data cannot be modified while immutably borrowed. This ensures that data can be safely shared across different parts of a program without unexpected changes. Multiple immutable references can coexist because none of them are allowed to modify the actual value, guaranteeing consistency for all readers.



```
pub fn main(){
    let s = String::from("hello");
    let r1 = &s;
    let r2 = &s; // multiple immutable references allowed
    println!("{} and {}", r1, r2);
}
```

Fig. 4. Multiple immutable references can coexist

3.3.2 Mutable References. To borrow mutably, you use &mut T. Only one mutable reference is allowed at a time to, for example, prevent data races. Mutable references allow you to modify the value. While the mutable reference exists, the owner cannot modify the value either.



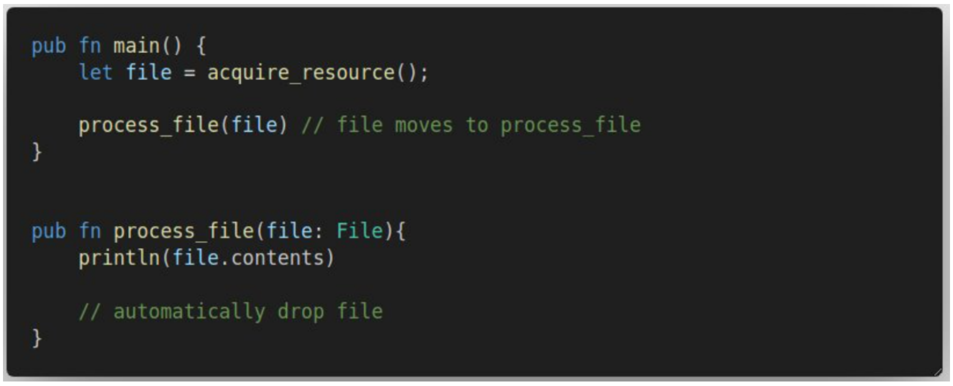
```
pub fn main() {
    let mut s = String::from("hello");
    let r = &mut s;
    r.push_str(", world");
    println!("{}", r);
}
```

Fig. 5. Single mutable reference

The rules are : at any given time, you can have either one mutable reference, or any number of immutable references. References must always be valid, meaning no dangling references are possible.

3.4 All together

The learned rules, together with RAII, enable reliable and clear patterns for memory and resource management. When a value goes out of scope, its memory (or any other resource it owns) is automatically and deterministically released. No need for manual cleanup, no garbage collector pauses — but predictable, safe cleanup every time.

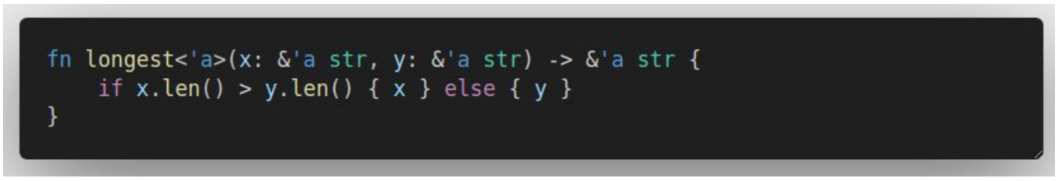


```
pub fn main() {
    let file = acquire_resource();

    process_file(file) // file moves to process_file
}


pub fn process_file(file: File){
    println(file.contents)

    // automatically drop file
}
```

Fig. 6. Schematic example of the concepts in action

3.5 Lifetimes

Lifetimes are Rust's way to track how long references are valid. They prevent dangling references, which are references to invalid memory. Every reference has a lifetime either implicitly or explicitly declared. The Rust Lang uses lifetimes to guarantee memory safety and ensures that a reference does not outlive the data it points to. This also helps the compiler check borrowing rules at compile time. In many cases, the compiler infers lifetimes automatically, however sometimes the compiler needs help, and in those cases we add lifetime parameters 'a. This means both parameters and return value must live at least as long as 'a.



```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Fig. 7. Example of Lifetime annotation

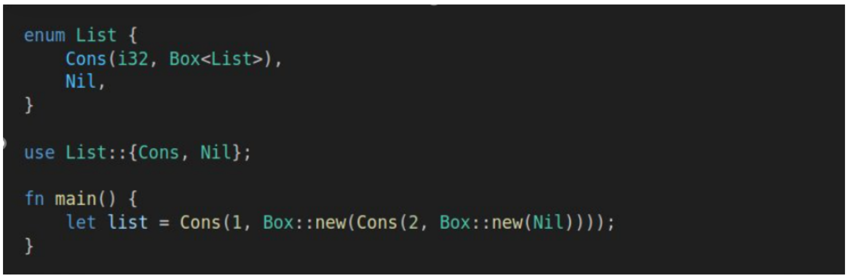
In Fig. 7. we see how the programmer can ensure that both the passed arguments *x* and *y* will live as long as the returned value, which is one of these two strings. If not annotated, we get an ambiguity: the compiler cannot know whether the return value is linked to the lifetime of *x* or of *y*. By adding 'a, we make it clear that the return reference is valid only as long as both inputs are, removing the uncertainty and preventing unsafe code.

4 Smart Pointers and Advanced Concepts

Smart pointers are modern data structures that in difference to 'simple' pointers known from plain C, own and manage memory automatically. They behave just like normal pointer with additional capabilities as automatic cleanup or reference counting.

4.1 Box and Recursive Types

Box⟨T⟩ is a smart pointer in Rust for allocating data on the heap. It stores the actual value on the heap, but owns it on the stack. Like any other type it ensures single ownership and is dropped when out of scope. Why use Box⟨T⟩? You need it when you want to store large data or recursive types, when you want to transfer ownership without copying data, or when you need a known size at compile time but the value is dynamically sized.



```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Nil))));
}
```

Fig. 8. Smart pointers like Box allow recursive types in Rust

Fig. 8. shows how to create recursive tyes in Rust using Box⟨T⟩. Rust needs to know the size of types at compile time. Recursive types like List would have infinite size without indirection. Box adds a heap indirection and makes the enum size-known.

4.2 Unsafe Rust

Sometimes, safe Rust is not enough performance wise and thats where unsafe comes in. It is a way to opt out of some compiler safety checks. Unsafe allows dereferencing raw pointers (`*const T`, `*mut T`), calling unsafe functions or foreign code (FFI), manually managing memory, and implementing low-level abstractions.

5 Conclusion

Rust shows that it is possible to achieve memory safety without needing a GC or meticulous manual management. Its ownership, borrowing, and lifetime system eliminates common memory bugs such as *null pointers*, *use-after-free errors*, and *data races*, while catching mistakes already at compile time rather than at runtime. Deterministic cleanup via RAII avoids unpredictable GC pauses and ensures resources are released as soon as they go out of scope.

The language also provides zero-cost abstractions, meaning that safety checks occur during compilation without runtime overhead. At the same time, programmers have fine-grained control comparable to manual memory management in C or C++, but with fewer risks. Finally, Rust is natural to concurrent programming, meaning: the borrow checker enforces rules that prevent data races by design.

Overall, Rust offers a balance between performance, safety, and expressiveness. Its model of linearity, ownership, and borrowing demonstrates that safe systems programming is possible without sacrificing efficiency.

References

A Online Resources

The following online resources were used for this work:

- https://research.ralfj.de/thesis.html
- https://doc.rust-lang.org/stable/book/
- https://rust-book.cs.brown.edu/
- https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization
- https://btmc.substack.com/p/a-survey-on-memory-management-approaches
- https://smallcultfollowing.com/babysteps/blog/2023/03/16/must-move-types/