Rust's Memory Management

Clément Lahoche

27.06.2023 Passau

Ownership and Lifetime

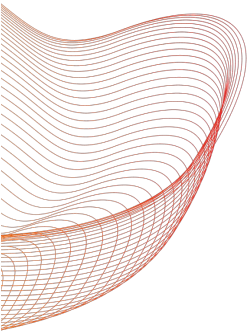
What are Ownership and lifetime?

Memory Management Mechanisms

What is Memory Management ?

History and evolution of Memory Management

How Rust compare to other languages



- form of managing computer memory
- provide ways to dynamically allocate portions of memory to programs at their request
- computer memory is limited > unused allocated memory need to be reuse

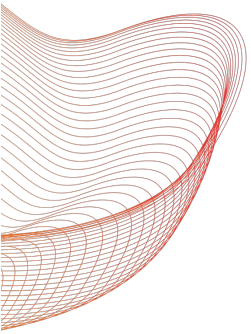
> need to keep track of every byte used by each program

Evolution and History

What is Memory Management ?

History and evolution of Memory Management

How Rust compare to other languages



- Fixed Partitions (1950s)
- Dynamic Partitions (1960s)
- Paging and Segmentation (1970s)
- Virtual Memory (1980s onwards)
  - explicit manual management (C/C++ …)
  - automatic garbage collection (Java, Python …)
  - Ownership and lifetime (Rust)

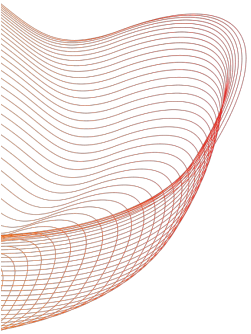
Memory management comparison

What is Memory Management ?

History and evolution of Memory Management

How Rust compare to other languages

Rust uses a system of ownership with a set of rules that the compiler checks at compile time. > Full memory control



- C/C++: manual
- Java: garbage collection
- Python: garbage collection with reference counting
- JavaScript: garbage collection

- Rust: ownership
- Go: garbage collection
- Haskell: concurrent generational garbage collection with laziness

Ownership and lifetime in Rust

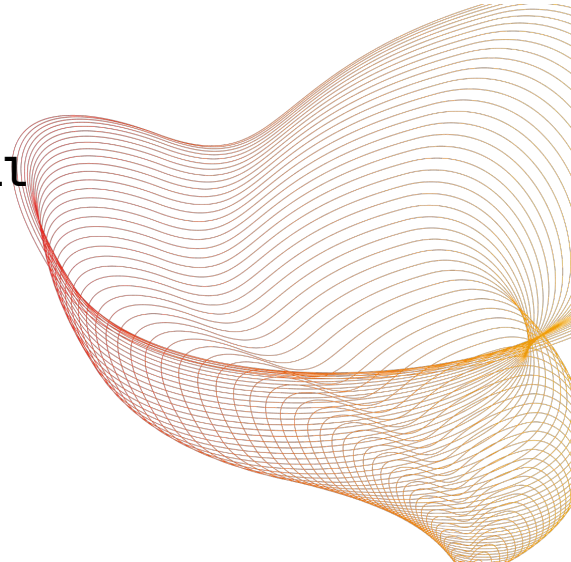
Ownership in rust

1. Each value in Rust has a variable that's called its owner.

2. There can only be one owner at a time.

3. When the owner goes out of scope, the value will be dropped.

> Using ownership in rust allow safe usage of memory without the execution time of a garbage collector.



> Which variable rules the free of the memory

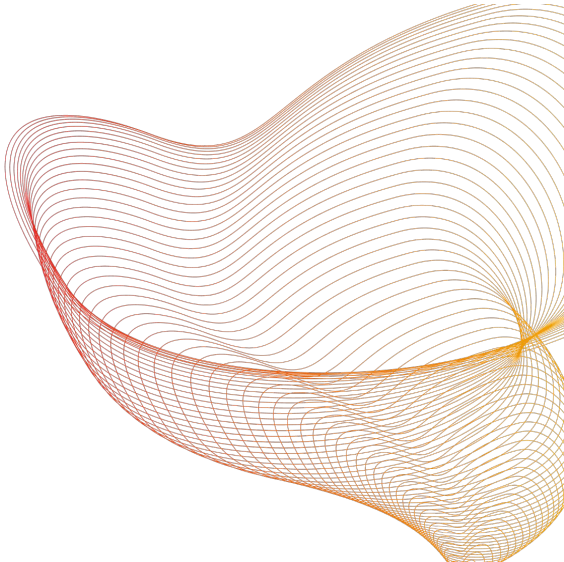
Ownership in Rust

```rust
fn main() {
    let s1 = String::from("hello"); // s1 becomes owner of the memory
    let s2 = s1; // s1's ownership is moved to s2
    //println!("{}, world!", s1); // Compiler error! `s1` does not have ownership anymore
}
```

Lifetime in Rust

- ensure all references are always valid.
- the scope in which a reference is valid is called its lifetime.
- explicit and part of the type system.

> Lifetime define where a reference is valid



Borrowing in Rust

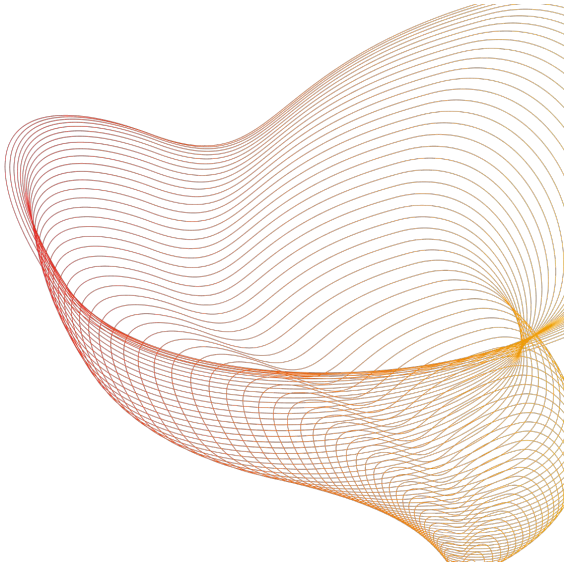
- allows for accessing data without taking ownership over it
- > prevent unnecessary copying of data

Immutable borrowing:

- Several immutable references can exist
- Original data is immutable

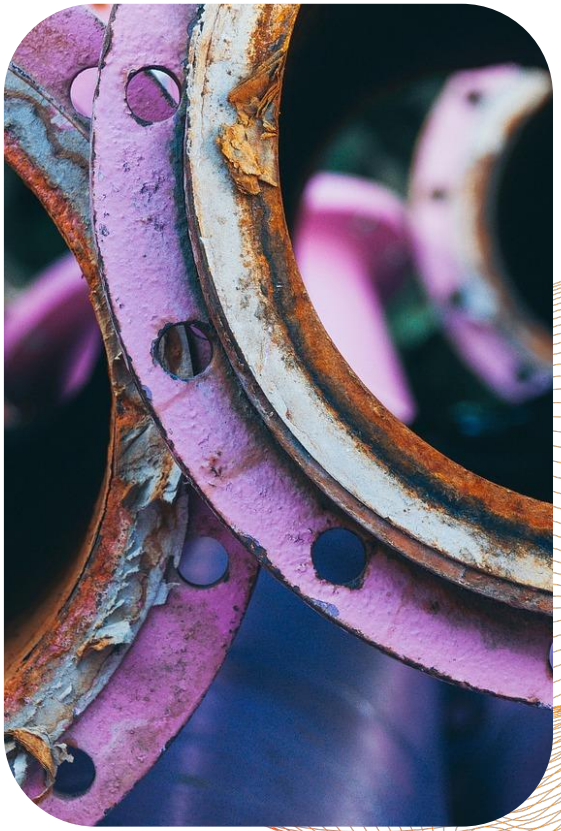
Mutable borrowing:

- Only one mutable reference can exit
- Original data is mutable



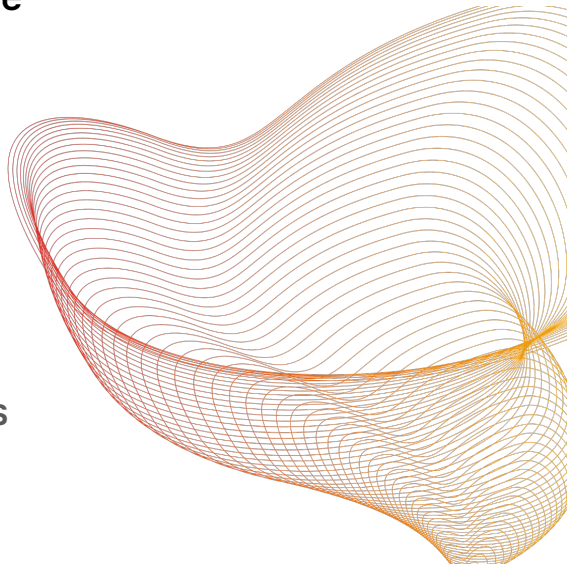
Lifetime in rust

```rust
// 'a is explicit lifetime parameters
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```



Using Both for Secure Memory Management at Compile Time

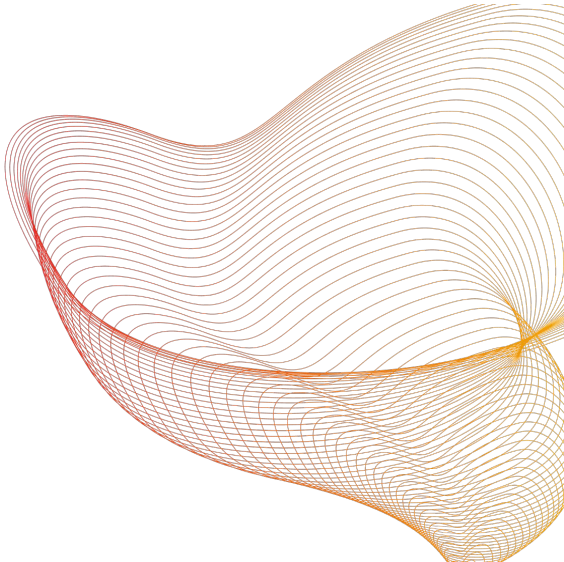
- Ownership and lifetimes work together to ensure a high level of memory safety
- at compile time, many common memory management errors at runtime such as null pointer dereferencing, double free, dangling pointers, etc. are impossible.



> Lifetime automatically check if provided variable is still allocated or not

Using Both for Secure Memory Management at Compile Time

```rust
// This code will compile without any issues
fn main() {
    let s1 = String::from("long string is long");
    let result;
    {
        let s2 = String::from("xyz");
        result = longest(s1.as_str(), s2.as_str());
    }
    println!("The longest string is {}", result);
}
```



A simple example: pointer invalid

C

```c
int* foo() {
    int x = 5;
    return &x;
}
```

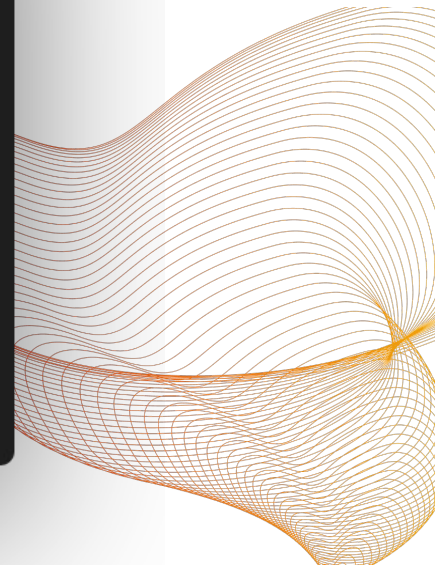
VS

Rust

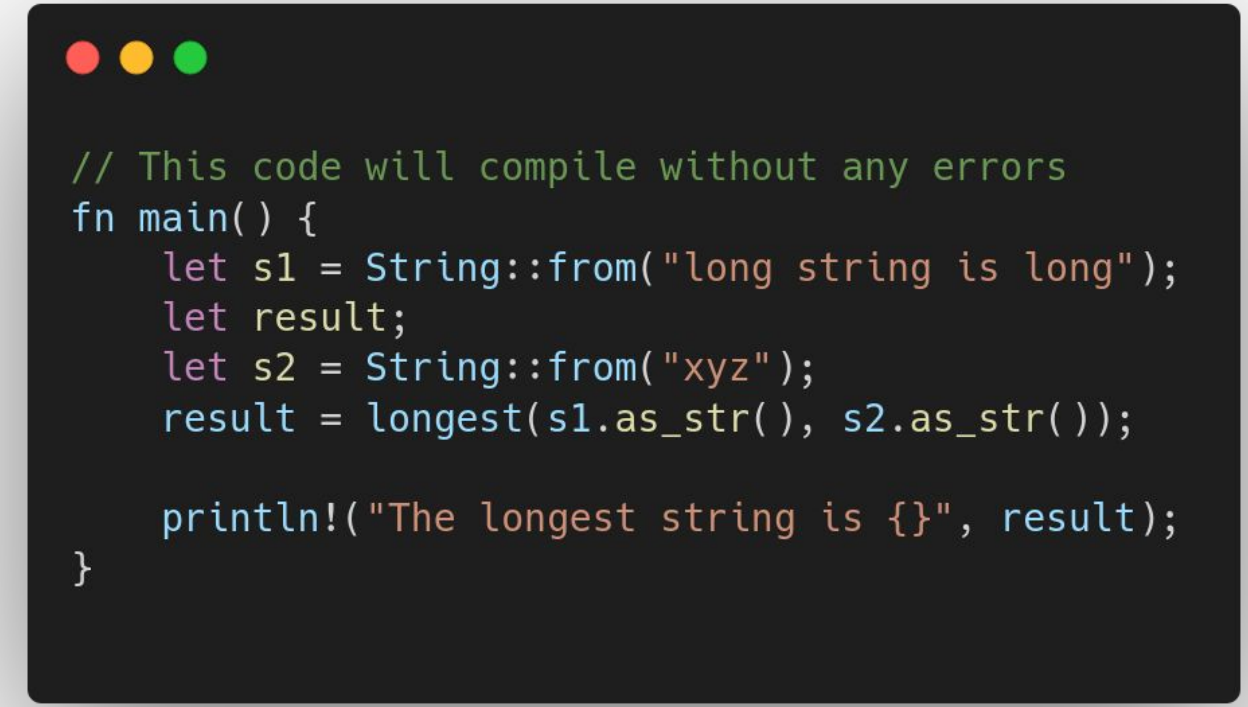
```rust
fn foo() -> &i32 {
    let x = 5;
    &x // Error: `x` does not live long enough
}
```

A more complex example:

```
// This code will NOT compile
fn main() {
    let s1 = String::from("long string is long");
    let result;
    {
        let s2 = String::from("xyz");
        result = longest(s1.as_str(), s2.as_str());
    }
    println!("The longest string is {}", result);
}
```

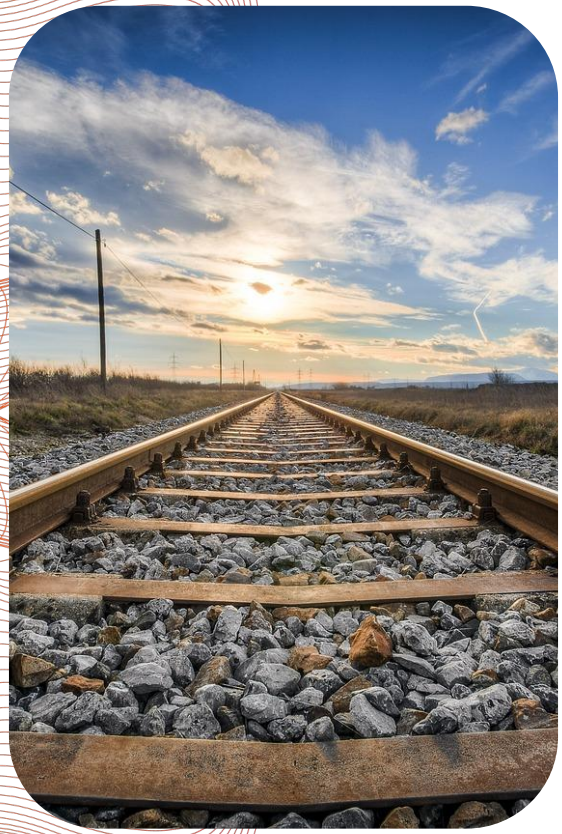


```
⊗ 《onyr ★ kenzael》《rust_programming_paradigm_presentation》》 cargo run
   Compiling rust_programming_paradigm_presentation v0.1.0 (/home/onyr/code/rust_programming_par
error[E0597]: `s2` does not live long enough
  --> src/main.rs:18:39
   |
17 |         let s2 = String::from("xyz");
   |             -- binding `s2` declared here
18 |         result = longest(s1.as_str(), s2.as_str());
   |                                       ^^^^^^^^^^ borrowed value does not live long enough
19 |     }
   |     - `s2` dropped here while still borrowed
20 |     println!("The longest string is {}", result);
   |                                          ------ borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile `rust_programming_paradigm_presentation` due to previous error
```



```rust
// This code will compile without any errors
fn main() {
    let s1 = String::from("long string is long");
    let result;
    let s2 = String::from("xyz");
    result = longest(s1.as_str(), s2.as_str());

    println!("The longest string is {}", result);
}
```

```
〈onyr ★ kenzael〉〈rust_programming_paradigm_presentation〉〉 cargo run
   Compiling rust_programming_paradigm_presentation v0.1.0 (/home/onyr/c
    Finished dev [unoptimized + debuginfo] target(s) in 0.17s
     Running `target/debug/rust_programming_paradigm_presentation`
The longest string is long string is long
```



Conclusion

01 Further reading and resources

- "The Rust Programming Language" book by Steve Klabnik and Carol Nichols.
- "Rust by Example" (an online resource by the Rust community).
- Rust's official documentation (found on their website).

02 Summary of key takeaways

- Rust's memory safety guarantees with no runtime overhead
- Ownership and Lifetimes checked at compile time.
- high degree of control and safety without the runtime cost of a garbage collector.