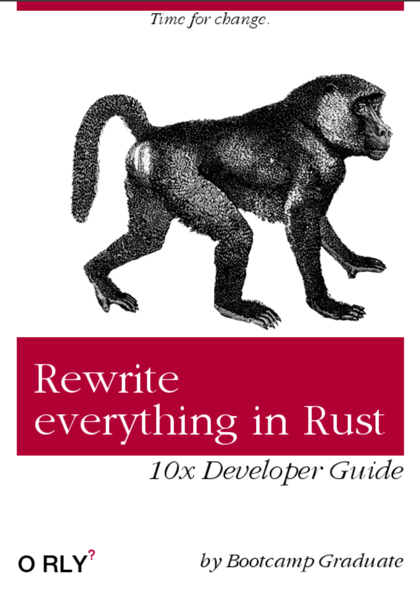
Rust

Jakob Lambert-Hartmann

2025

Introduction



Rust

- Fast like C/C++
- Package Manager like Python and JS
- Functional Programming like Haskell and JS
- Type system like no other

Outline

Memory Management

Memory Management

Stack

- Fast allocation
- Fixed size
- Manages itself

Heap

- Slower allocation
- Dynamic size
- Needs to be managed

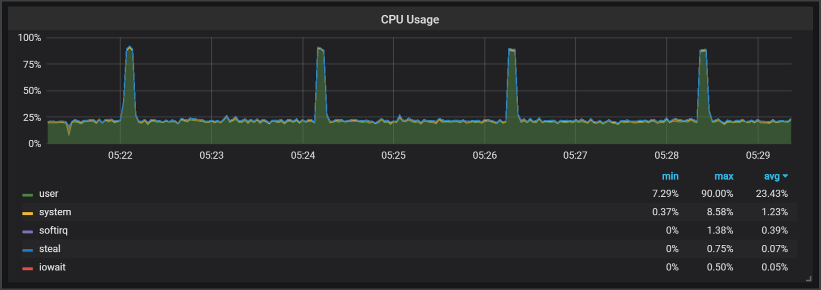
Manual Memory Management

- Programmer responsible for allocating/freeing memory
- Many possibilities of failure:
  - Memory leak
  - Double Free
  - Use After Free

```
// allocate
DataType* data = malloc(sizeof(DataType));
// free
free(data);
```

Garbage Collector

- language manages memory
- periodically checks for heap data without references



source: percona

Rusts Memory Management

Ownership

- Resource Acquisition is Initialization (RAII)
- Memory manages itself
- Three rules required:
    1. Each value has an owner
    2. There can only be one owner at a time
    3. When the owner goes out of scope, the value will be dropped

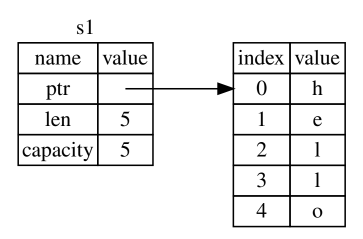
Ownership: Code Snipped

```
fn main() {
    // s1 owner of "hello"
    let s1 = String::from("hello")

}   // s1 goes out of scope -> value "hello" dropped
```

Ownership: Memory Representation

- **Stack:** len, capacity, and ptr
- **Heap:** String data
- when `s1` goes out of scope → free String data



source: rustbook

Ownership: Code snipped, shared data

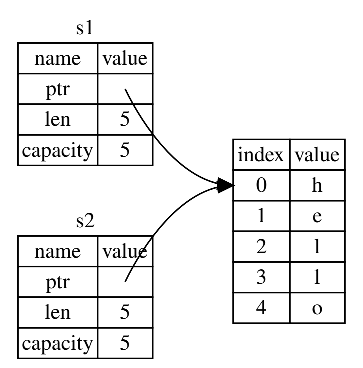
- What happens when two variables use the same data:

```
{
    let s1 = String::from("hello");
    let s2 = s1;
}
```

- Three options →

Ownership: Option 1: Shallow Copy

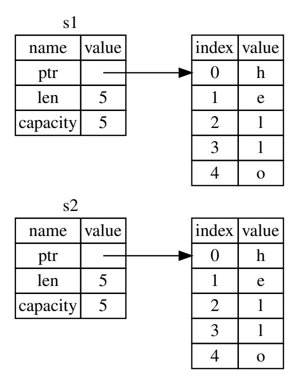
- Stack data copied
- **Only one owner allowed!**
- Double Free when `s1` and `s2` go out of scope



source: rustbook

Ownership: Option 2: Deep Copy

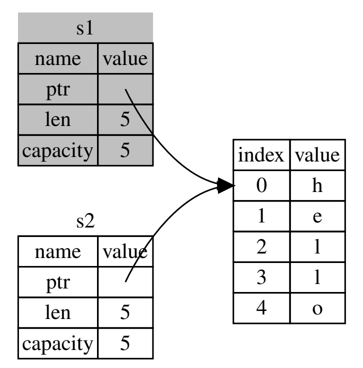
- Stack and heap data copied
- **But copying data can be very expensive!**



source: rustbook

Ownership: Option 3: Move

- `s2` new owner of data
- `s1` is invalidated
- **What Rusts does!**



source: rustbook

Ownership: Move

```
let s1 = String::from("hello");
let s2 = s1;

println!("{s1}, world!");
```

- does not compile: **`s1` is not valid anymore**
- `s1` passed ownership over string data to `s2`

Function Calls: Ownership

```
fn compute_length(s: String) -> usize {
    s.len()
}

fn main(){
    let s1 = String::from("hello");
    compute_length(s1); // moves value
    compute_length(s1); // s1 now invalid
}
```

- function take ownership of their parameters
- Code does not compile:
  - first `compute_lenght` takes ownership of `s1`
  - second `compute_lenght` call not possible: `s1` does not own data anymore

Function Calls: Borrowing

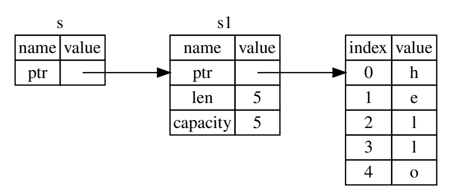
```
fn compute_len(s: &String) {
    s.len()
}

fn main(){
    let s1 = String::from("hello");
    compute_len(&s1);
    compute_len(&s1);
}
```

- functions can return ownership when done with references
- some_function returns ownership of s1 after return

Borrowing: Memory Layout

- multiple references, one owner
- no double free



source: rustbook

Lifetimes

Bind lifetimes of two variables together:

```
fn first_word<'a>(s: &'a str) -> &'a str {
    &s[0..1]
}
```

- references never outlive the value they reference
- **Compiler automatically deduces most lifetimes**

Mutability

Variables immutable by default:

```rust
let immutable   = 1;
let mut mutable = 1;
```

References immutable by default:

```rust
let reference_1 = &variable;
let reference_2 = &variable;
```

If there exists a mutable reference: only reference allowed!

```rust
let immutable_ref = &variable;
let mutable_ref   = &mut variable;
// immutable_ref is now invalid !!!
```

Security Guarantees

A reference always points to valid data:

```
let mut string = String::from("hello");
// immutable reference
let substring = string[0..3]; // reference to "he"

// mutable borrow (reference)
string.clear();

// compilation error: mutable borrow in clear()
println!("{substring}, world!");
```

Error Handling

Types Of Error

1. Logical Errors
2. Compile Errors
3. Runtime Errors
    3.1 Performing invalid arithmetic: division by zero, etc.
    3.2 Accessing invalid data: null pointers, out of bounds, etc.
    3.3 Failing function calls: opening nonexistent file, etc.

Error Handling In C

C Error Handling: Performing invalid arithmetic

It doesn't

```c
int a = 1;
int b = 0;
int c = a / b;
// here be dragons (undefined behaviour)
```

C Error Handling: Accessing invalid data

It doesn't

```
int[2] arr = [1,2];
int el = arr[2];
// here be dragons (undefined behaviour)
```

C Error Handling: Calling a function that fails

It doesn't? Kind of?

```
char* path = 'file/does/not/exist';
FILE* f = fopen(path);
// here be dragons (undefined behaviour)
```

- Return value of function may help detect errors
- Success/Failure values differ from function to function
  - `fopen()` returns `NULL` on failure
  - `system()` returns 0 on success
  - some functions set `errno` some don't

Error Handling In Java

- Helps developers to handle errors
- Uses exceptions:
  1. dividing by zero: `ArithmeticException`
  2. access array out of bounds: `ArrayIndexOutOfBoundsException`
  3. opening nonexistent file: `FileNotFoundException`
- Not all exceptions need to be checked at compile time: `NullpointerException`

Error Handling in Rust

Panic

```
panic!()
```

- terminate program immediately
- used in Rusts error handling to prevent undefined behavior

Option

```
enum Option<T> {
    None,
    Some(T),
}
```

- can hold value of type T
- or nothing

Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- provides additional information on failure

Unpacking Values

```rust
let vec = vec![0,1,3];
let option = vec.get(3);

// only executed when there is a value
if let Some(value) = option {
    println!("vec[3] == {value}");
}

// only executed when there is no value
if let None = option {
    println!("array out of bounds");
}
```

Helper Functions

```
let result = fs::read_to_string("not a file");

// exit program and print information
let content = result.expect("could not read file");

// exit program without message
let content = result.unwrap();

// use default value if error occured
let content = result.unwrap_or("default content");
```

Error Propagation

? operator propagates error or continues with unwrapped value:

```
fn count_words(path: &str) -> Result<usize> {
    // ? returns Err on failure
    let content = fs::read_to_string(path)?;

    // read_to_string did succeed
    // content contains file content
    content.split(" ")
        .count()
}
```

Conclusion

- Memory management is performant, safe, and simple
- Shift of perspective in error handling:
  - C/Java:
    - all variables can be `null`
    - most will never be
  - Rust:
    - variables cannot be null
    - `Option`/`Result` needed when missing value or failure possible
    - check required at compile time

Thank You For Your Attention!

Sources

1. https://doc.rust-lang.org/stable/book/
2. https://www.percona.com/blog/prometheus-2-times-series-storage-performance-analyses/