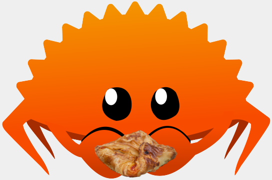
RUSTico

Rust in cryptographic outlook

Davide Carnemolla

Università degli Studi di Catania

2025



Introduction

Introduction: main features

General purpose

Blazingly fast

Memory safe

No Garbage Collector

Tools
(doc, cargo, libraries)

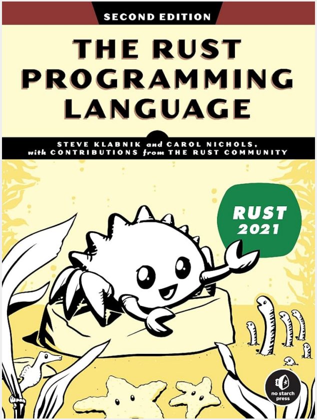
Introduction: history

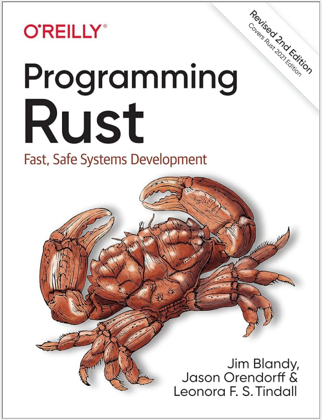
2006	Mozilla creates Rust for the Servo's project
2010	Rust is announced at Mozilla Summit
2012	Alpha release
2013	Version 0.6 release
2015	First stable release (Rust 1.0)
2021	The Rust Foundation is founded

Introduction: who uses it?

- AWS: Firecracker powers Lambda and Fargate
- Google: parts of the Fuchsia operating system
- Linux: 2nd official language for the Kernel!
- CloudFlare: quic / http 3 implementation
- Dropbox: file storage
- Clever Cloud: reverse proxy
- Atlassian, Canonical, Coursera, Chef, Deliveroo, NPM, Sentry...
- Growing ecosystem for embedded development

Introduction: learning Rust


SECOND EDITION
THE RUST PROGRAMMING LANGUAGE
STEVE KLABNIK and CAROL NICHOLS, with CONTRIBUTIONS from THE RUST COMMUNITY
RUST 2021
no starch press


O'REILLY
Revised 2nd Edition
Covers Rust 2021 Edition
Programming Rust
Fast, Safe Systems Development
Jim Blandy, Jason Orendorff & Leonora F. S. Tindall

Online at https://www.rust-lang.org

Getting started

Installing on MacOS/Unix

```
$ curl -sSf https://sh.rustup.rs | sh
```

Installing on Windows

Download and install the `rustup-init.exe from the official website.`

Verifying the installation

```
$ rustc --version
rustc 1.84.0 (9fc6b4312 2025-01-07)
```

Hello Rust

```rust
fn main() {
    println!("Hello Rust");
}
```

Compiling and running

```
$ rustc <file-name.rs>
$ ./file-name
```

Useful extensions for vscode

rust-analyzer

CodeLLDB

Hello, Cargo!

Project Manager

Dependencies Manager

Building

Testing

Benchmarking

Cargo: a new project

Create a new application with cargo

```
$ cargo new <project-name>
```

Create a new library with cargo

```
$ cargo new --lib <project-name>
```

Do you need help?

```
cargo --help
```

Common Programming Concepts

Variables

```
fn main() {
    let x = 5; // immutable variable and type inference
    let mut y = 6; // mutable variable and type inference
    const Y2K: i32 = 2000; // const variables require a known type
    static mut POTATOES: u32 = 0; // This is a mutable static variable
}
```

Variables: Shadowing

```
fn main() {
    let x = 5;
    let x = x + 1;
    {
        let x = x * 2;
        println!("The value of x is: {x}");     // 12
    }
    println!("The value of x is: {x}");     // 6
}
```

Scalar Data Types

Integer Types

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64

Floating-point Types

Length	
32-bit	f32
64-bit	f64

Compound Types: Tuple Type

```
fn main() {
    // implicit declaration using type inference
    let tup = (500, 6.4, 1);
    // explicit declaration
    let tup: (i32, f64, u8) = (500, 6.4, 1);

    println!("The first value is: {tup.0}");

    let (x, y, z) = tup; // destructuring
    println!("The first value is: {y}");
}
```

Compound Types: Array Type

```
fn main() {
    // implicit declaration using type inference
    let a = [1, 2, 3, 4, 5];
    // explicit declaration
    let a: [i32; 5] = [1, 2, 3, 4, 5];

    // Fast init
    let a = [3; 5]; // [3, 3, 3, 3, 3]

    let first = a[0]; // first element of a
}
```

Control Flow: if Expressions

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

Control Flow: using if in a let Statement

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };
    // values must be of the same type

    println!("The value of number is: {number}");  // 5
}
```

Control Flow: Loop

```
fn main() {
    let mut counter: i32 = 0;

    loop {
        if counter == 10 {
            break;
        }

        println!("counter: {}", &counter);
        counter += 1;
    }
}
```

Control Flow: returning values from loops

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");        // 20
}
```

Control Flow: loop labels

```
fn main() {
    let mut count = 0;
    ’counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;
        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break ’counting_up;
            }
            remaining -= 1;
        }
        count += 1;
    }
    println!("End count = {count}");
}
```

Control Flow: while

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{number}!");

        number -= 1;
    }

    println!("LIFTOFF!!!");
}
```

Control Flow: for

```
fn main() {
    // for loop using Range from std
    for number in (1..4).rev() {
        println!("{number}!");
    }

    println!("LIFTOFF!!!");
}
```

Control Flow: Looping Through a Collection

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

Functions

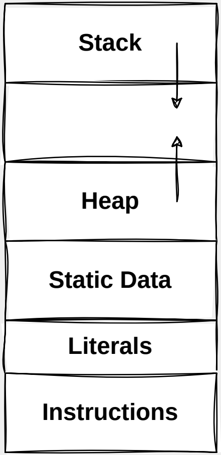
```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1 // expression (without semicolons)
}
```

Understanding Ownership

The process address space



writable; not executable	Stack	Managed "automatically"
writable; not executable	Heap	Managed by programmer
writable; not executable	Static Data	Initialized when process starts
read-only; not executable	Literals	Initialized when process starts
read-only; not executable	Instructions	Initialized when process starts

Ownership rules

1. Each value in Rust 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

The String Type

To illustrate the rules of ownership, we need a data type that can’t be stored on the stack. The String type is a great example.

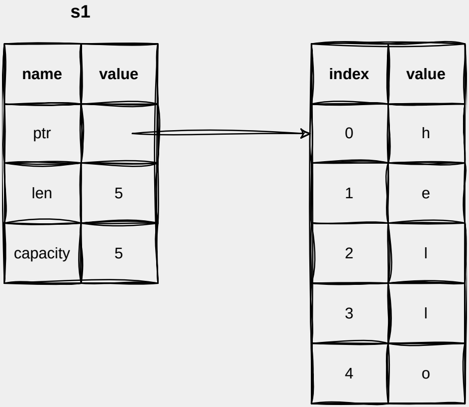
```
let mut s = String::from("hello");

// push_str() appends a literal to a String
s.push_str(", world!");

println!("{s}");     // this will print hello, world!
```

Memory and allocation

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



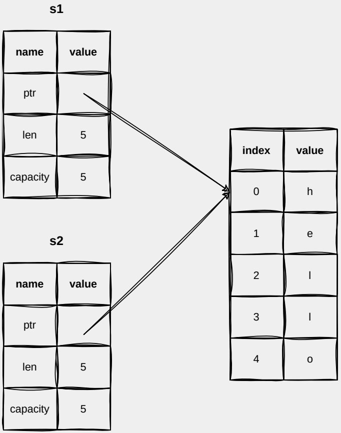
s1

name	**value**
ptr	
len	5
capacity	5

index	**value**
0	h
1	e
2	l
3	l
4	o

Copy of a pointer

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



s1

name	value
ptr	
len	5
capacity	5

s2

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

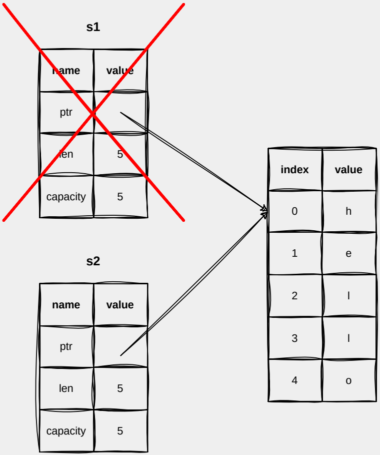
Copy of a pointer: problem

When a variable goes out of scope, Rust automatically calls the drop function and cleans up the heap memory for that variable.

In the previous case, when s2 and s1 go out of scope, they will both try to free the same memory.

This is known as a *double free* error.

The Rust solution



s1

name	value
ptr	
len	5
capacity	5

s2

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

This is called *move*.

Deep copy with Clone

If we do want to deeply copy the heap data of the `String`, not just the stack data, we can use a common method called clone.

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

println!("s1 = {s1}, s2 = {s2}");
```

Stack-Only Data: Copy

Rust has a special annotation called the Copy trait that we can place on types that are stored on the stack.

If a type implements the Copy trait, variables that use it do not move, but rather are trivially copied.

```
let x = 5;
let y = x;

println!("x = {x}, y = {y}");
```

Ownership and Functions (1)

```
fn takes_ownership(some_string: String) {
    // some_string comes into scope
    println!("{some_string}");
} // Here, some_string goes out of scope and drop is called.

fn makes_copy(some_integer: i32) {
    // some_integer comes into scope
    println!(some_integer);
} // Here, some_integer goes out of scope
```

Ownership and Functions(2)

```
fn main() {
    let s = String::from("hello");    // s comes into scope

    takes_ownership(s); // s value moves into the function
    // ...  and so it's  no longer valid here

    let x = 5;  // x comes into scope
    makes_copy(x); // i32 implements Copy so x don't move
    // you can use x here
} // Here, x goes out of scope
```

Return values and Scope (1)

```
fn gives_ownership() -> String {
    let some_string = String::from("yours");
    // some_string comes into scope

    some_string // it moves out to the calling func.
}

fn takes_and_gives_back(a_string: String) -> String {
    // a_string comes into scope

    a_string // a_string moves out to the calling func.
}
```

Return values and Scope (2)

```
fn main() {
    let s1 = gives_ownership(); // gives_ownership moves its return

    let s2 = String::from("hello");    // s2 comes into scope

    let s3 = takes_and_gives_back(s2);
    // s2 is moved into the function which moves
    // its return value into s3
}
```

References and Borrowing

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1); // borrowing

    // s1 is available here
}

fn calculate_length(s: &String) -> usize {
    s.len()
} // The String s is not dropped
```

Mutable references

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

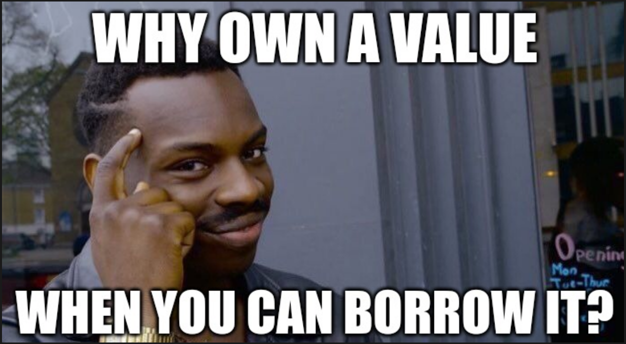
References Rules

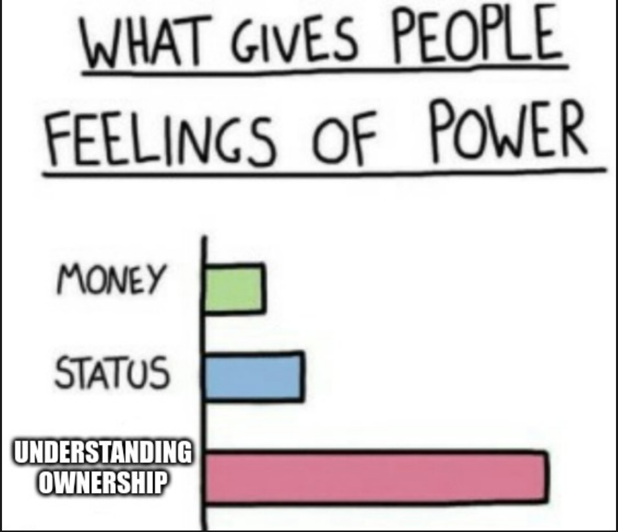
We can summerize the references's rules as follows.

1. At any given time, you can have either one mutable reference or any number of immutable references.
2. References must always be valid (no *dangling* references).

The benefit of having this restriction is that Rust can prevent *data races* at compile time.
A *data race* is similar to a race condition and happens when these three behaviors occur:

1. Two or more pointers access the same data at the same time.
2. At least one of the pointers is being used to write to the data.
3. There's no mechanism being used to synchronize access to the data.


WHY OWN A VALUE
WHEN YOU CAN BORROW IT?
Opening
Mon
Tue-Thur


WHAT GIVES PEOPLE FEELINGS OF POWER
MONEY
STATUS
UNDERSTANDING OWNERSHIP

Struct

Struct

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    }; // immutable
}
```

Derived Traits

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32
}

fn main() {
    let rect1 = Rectangle {
        30,
        50
    };

    println!("rect1 is {:?}",       rect1);
}
```

Defining Methods

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("The area of rect1 is {}",rect1.area());
}
```

Associated Functions

```
impl Rectangle {
    fn square(size: u32) -> Self {  // A constructor
        Self {
            width: size,
            height: size
        }
    }
}
```

“Additional features”

Generics

We use `generics` to create definitions for items like function signatures or structs, which we can then use with many different concrete data types.

Smart Pointers

1. `Box<T>`
2. `Rc<T>` (Reference Counted Smart Pointer)
3. `Arc<T>` (Atomically Reference Counted)
4. `Ref<T>` and `RefMut<T>` (borrowing rules at runtime)

TRAITS (1)

```
struct Sheep { naked: bool, name: &'static str }

trait Animal {
    // Self refers to the implementor type.
    fn new(name: &'static str) -> Self;

    // Method signatures; these will return a string.
    fn name(&self) -> &'static str;
    fn noise(&self) -> &'static str;

    // Traits can provide default method definitions.
    fn talk(&self) {
        println!("{} says {}",      self.name(), self.noise());
    }
}
```

Traits (2)

```
impl Sheep {
    fn is_naked(&self) -> bool {
        self.naked
    }

    fn shear(&mut self) {
        if self.is_naked() {
            // Impl methods can use the implementor’s trait methods.
            println!("{} is already naked...",    self.name());
        } else {
            println!("{} gets a haircut!",    self.name);

            self.naked = true;
        }
    }
}
```

Traits (3)

```
// Implement the ‘Animal‘ trait for ‘Sheep‘.
impl Animal for Sheep {
    // ‘Self‘ is the implementor type: ‘Sheep‘.
    fn new(name: &’static str) -> Sheep {
        Sheep { name: name, naked: false }
    }

    fn name(&self) -> &’static str {
        self.name
    }

    fn noise(&self) -> &’static str {
        if self.is_naked() {
            "baaaaah?"
        } else {
            "baaaaah!"
        }
    }
```

Traits (4)

```
    // Default trait methods can be overridden.
    fn talk(&self) {
        // For example, we can add some quiet contemplation.
        println!("{} pauses briefly... {}", self.name, self.noise());
    }
}
```

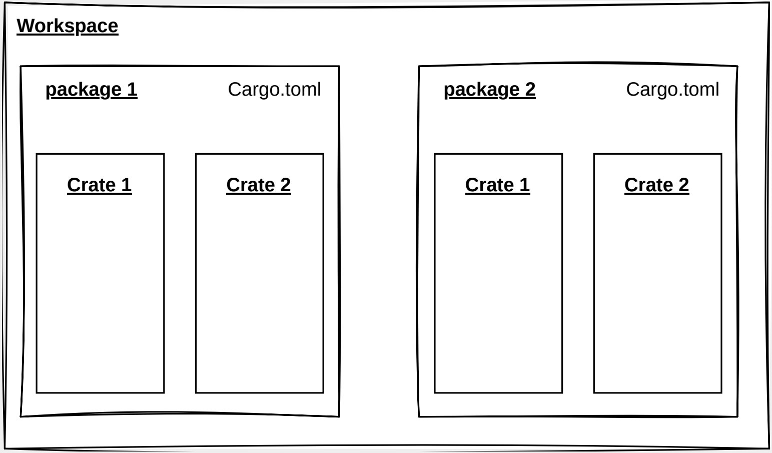
Common collections

Rust's standard collection library provides efficient implementations of:

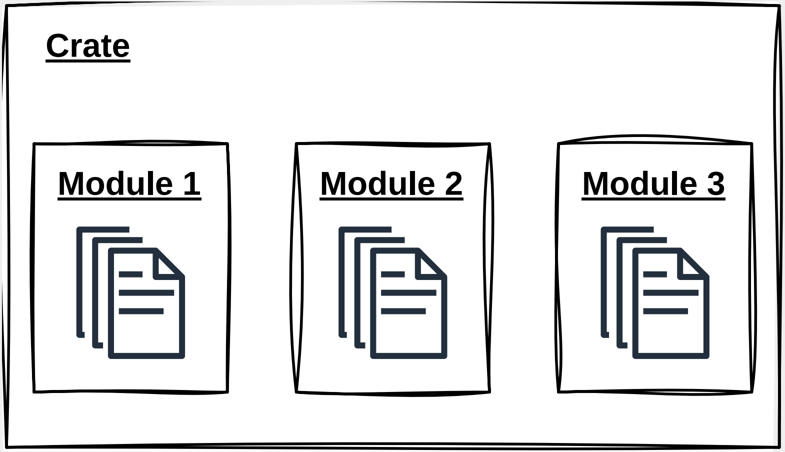
1. Sequences: **Vec**, **VecDeque**, **LinkedList**
2. Maps: **HashMap**, **BTreeMap**
3. Sets: **HashSet**, **BTreeSet**
4. Misc: **BinaryHeap**

Packages, Crates and Modules

Packages and Crates



Modules



CRATES.IO

Cryptographic Libraries

Multi-precision integer

rug
(a well done GMP binding)

Malachite
(a pure Rust implementation)

num_bigint/crypto_bigint
(a slow pure Rust implementation)

Sources: malachite.rs/performance

Pairings

blstrs
(a binding for blst)

zkcrypto/pairing
(a pure Rust implementation)

arkworks-rs
(another pure Rust implementation)

Micro Benchmark

criterion.rs

Divan

(a sophisticated alternative)