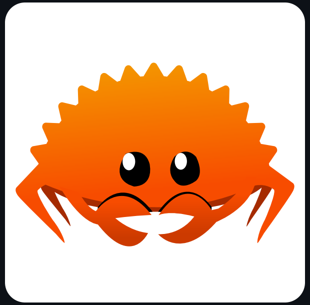
Introduction to the Rust programming Language



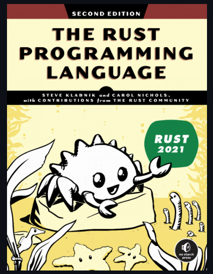
Following along The Rust Book from the official source

by: **Simon Lalonde**

For: **IFT-769** (Theoritical concepts CS)

📖 **Project overview -** Going through "The Rust Programming Language"

The Rust Programming Language by Steve Klabnik and Carol Nichols



Book overview:

- Official guide to the Rust programming language
- Covers the basics (syntax, types, functions) + toolchain
- Advanced and Rust-specific features:
  - Ownership, borrowing, lifetimes
  - Unique error handling
  - Concurrency

📚 **Theoretical concepts** - Key topics covered

1. Common Programming Concepts (*variables, types, control flow*)
2. Understanding Ownership (*memory management*)
3. Structs, Enums and Pattern Matching
4. Containers/Collections
5. Error Handling
6. Generics, Traits and Lifetimes
7. OO features
8. Smart pointers and Concurrency

Klabnik, Steve, and Carol Nichols. The Rust Programming Language. 2nd ed., No Starch Press.

Rust Overview

Written by **Graydon Hoare in 2006**, Rust is a systems programming language focused on safety, speed, and concurrency. Backed by **Mozilla** and now the Rust Foundation.

Currently known projects

TODO

Predicted use cases

TODO

Rust Programming Language. 2024. Official Website

📝 Pros and Cons of Rust

PROS:

- **Memory safety:** No null pointers, dangling pointers, or buffer overflows
- **Error handling**: With the `Result` and `Option` types
- **Concurrency**: Safe and efficient with the ownership system
- **Performance**: Comparable to C/C++ with zero-cost abstractions
- **Ecosystem**: Growing with a strong community and package manager (**Cargo**)
- **Helpful compiler**: Provides detailed error messages and warnings

CONS:

- **Learning curve**: Ownership, borrowing, and lifetimes can be challenging
- **Tooling and prevalence**: Not as mature as other languages (C/C++, Python, etc.)
- **Syntax**: Can be verbose and complex compared to other languages

⚙️ Installation and setup

Installation:

1. Install Rust using `rustup` (Rust toolchain installer)

Included toolchain:

- `rustc`: Rust compiler
- `rustup`: Rust toolchain manager
- `rustfmt`: Rust code formatter
- `cargo`: Rust package manager and build tool

Package and library management

- **Crates** are Rust packages that can be shared and reused
- Managed with **Cargo**, the Rust package manager

🏞️ **Development environment** - Toolchain overview

Env setup and features:

- Easy install: `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`
- Rustup for managing toolchains: `rustup update`
- Included formatter: `rustfmt --check src/main.rs` (dry-run mode)
- Cargo for building and managing projects: `cargo new project_name`
- Quality of life with `rust-analyzer`: LSP, build/debug IDE support etc.

🏞️ **Development environment** - Cargo features

Useful Cargo commands when building a project:

- `cargo build` or `cargo run` to compile and run the project. Use `--release` flag for compilation with optimizations inside `target/release/`
- `cargo check`: Check the project for errors without building
- `cargo doc`: Generate documentation for the project
- `cargo clean`: Remove build artifacts
- `cargo update`: Update dependencies
- `cargo fmt`: Format the code according to the Rust style guidelines
- `cargo test`: Run tests in the project

🛠️ **Practical project #0 -** Guessing game

Great way to introduce to the development environment and basic concepts of Rust:

- Common programming concepts (types, funcs, control flow)
- Use of another crate (rand) inside the project
- I/O, String manipulation, error handling
- Compiler warnings and error messages
- `rust-analyzer` compiler FE for IDE support

Klabnik, Steve, and Carol Nichols. The Rust Programming Language. 2nd ed., No Starch Press.

🚀 Demo Time!

Simple guessing game CLI app 🎲 (Basics and dev environment features)

🔎 **Demo reminders** - P#0 (Guessing game)

- `Result` type with `.expect()` for error handling
- `cargo doc --open` to generate and view documentation
- `cargo fmt` to format the code
- Type annotations and `let` for variable declaration

📝 Variables and mutability

Variables are immutable by default

```
let x = 5;              // immutable variable
x = 10;                 //!ERROR!

let mut y = 10;         // mutable variable
y = 15;                 // Will compile (overwrite with same type)
let y = "hello";        // Will compile (shadowing)
```

Constants are always immutable within the scope

```
const MAX_POINTS: u32 = 100_000;
```

🧮 Statically typed + type inference

`rust-analyzer` provides type hints and suggestions

```rust
let secret_num = rand::thread_rng().gen_range(1..101); // Will infer i32 type
```

Explicit type annotations can or must be used

```rust
let mut num: String = String::new(); // Can be annotated or inferred
num = "42".to_string();

let guess = guess.trim().parse().expect("Please enter a number");  // Wont Compile
let guess: u32 = guess.trim().parse().expect("Please enter a number"); // Will compile
```

🧮 **Data types** - Scalars

Data type	Size	Specifity
int	8-128 bits	signed/unsigned
float	32/64 bits	simple/double precision
char	4 bytes	unicode
bool	1 byte	true/false

🧮 **Data types** - Compound

Data type	Size	Elements	Example	Access
tuple	fixed	mixed types	`(1, "hello", 3.14)`	`tuple.0`
array	fixed	same type	`[1, 2, 3, 4, 5]`	`array[0]`
vec	dynamic	same type	`vec![1, 2, 3, 4, 5]`	`vec[0]`

Access safety with runtime bounds checking. If using `array[10]` will panic at runtime instead of *undefined behavior like in C/C++*

⚙️ Functions - `main`

Functions are defined with the `fn` keyword. All programs start with a `main` function

```rust
fn main() {
    println!("Hello, world!");

    say_hello_back();
}

fn say_hello_back() {
    println!("Hello back!");
}
```

⚙️ **Functions** - Parameters and return

Function signatures and use:

- Parameters must have type annotations
- Return type must be specified with `->`
- Functions can return multiple values with tuples

```rust
fn main() {
    let num_sum = add(5, 10);
    println!("The sum is: {}", num_sum);
}

fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

📜 Statements

- `let` is a statement, and `x + y` is an expression.
- Compared to C/C++, var assignment is an expression in Rust and **does not** return a value
- Statements must end with a semicolon `;`

```
let x = 5;              // statement
let y = z = 10;         // ERROR! z = 10 does not return a value
```

💡 Expressions

- Expressions **evaluate** to a value (*func calls, operations, blocks*)
- No `;` at the end of expressions
- **Blocks** `{}` are expressions and can be used to create new scopes + return values

```rust
fn main() {
    let x = 5; // whole line is statement, 5 is expression
    let y = {
        let x = 3;
        x + 1
    }; // an expression
    println!("The value of y is: {}", y); // Prints 4!
}
```

🧰 **Control Flow** - Conditionals

if/else: (Only takes boolean expressions)

```
// Classic if/else if/else
let mut condition = false;
if number < 5 {
    println!("Too small!");
} else if number > 5 {
    println!("Too big!");
} else {
    println!("Just right!");
    condition = true;
}

// Assignement with if/else
let result = if condition { 5 } else { 6 };
```

🧰 **Control Flow** - Loops overview

3 types of loops in Rust: `loop`, `while` and `for`

- `loop`: Infinite loop until `break` or return
- `while`: Loop while condition is true
- `for`: Loop over an iterator

```rust
// Conditional loop
let mut counter = 0;
while counter < 10 {
    println!("counter = {counter}");
    counter += 1;
}
```

🧰 Control Flow - Loop labels

Loop labels can be used to distinguish nested loops (*break* and *continue*)

```rust
fn main() {
    let mut count = 0;
    'counting_up: loop { // Label the outer loop
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;    // Break the outer loop
            }
            remaining -= 1;
        }
        count += 1;
    }
    println!("End count = {count}");
}
```

🧰 **Control Flow** - Collection with `for`

No need for manual indexing, `for` loops iterate over collections

```
let collection = [10, 20, 30, 40, 50];
for element in collection {
    println!("The value is: {element}");
};
```

🔎 **Ranges**, use the `..` operator

```
for number in 1..4 {
    println!("The value is: {number}");
}
```

🔑 **Ownership** - Overview

Ownership is a key feature of Rust regarding the management of stack (*static, compile-time known, LIFO*) and heap memory (*allocated at runtime, dynamic, FIFO*).

It ensures memory safety without garbage collection.

The 3 rules of ownership:

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

🔑 **Ownership** - String Type vs. literals

```rust
let s1: &str = "hello"; // string literal, immutable

{
    // s1 is still valid
    let mut s2 = String::from("hello"); // allocated on the heap
    s2.push_str(", world!"); // Mutable

} // calls drop(), s2 goes out of scope its memory is freed
```

- String literals hardcoded into binary. Immutable and fast.
- `String` type is allocated on the heap and is mutable. Memory freed when out of scope. Similar to smart pointers in C++.

🔑 **Ownership** - Move

```rust
// MOVE
let s1 = String::from("hello");
let s2 = s1; // s1 is moved to s2
println!("{s1}"); // ERROR! s1 is no longer valid

// DEEP COPY
let s3 = s2.clone(); // deep copy
println!("{s2}"); // s2 is still valid
```

No *double free* or *dangling pointers* with the **move** operation (first 3 lines of code).

🔑 Ownership - Copy

Types that implement the `Copy` trait are copied instead of moved. Stack-only data types (i.e. integers, booleans, char etc.) for speed and efficiency.

```
let x = 5;
let y = x; // x is copied to y

println!("{x}"); // x is still valid. Same as x.clone() but no needed
```

🔑 **Taking ownership** - Functions

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

    takes_ownership(s);             // s's value moves into the function...

    let x = 5;                      // x comes into scope
                                    // but i32 is Copy, so x available afterward

    println!("{s} world!");         // ERROR! s is no longer valid
}

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

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{some_integer}");
} // `some_integer` goes out of scope, nothing happens.
```

🔑 **Transfer Ownership** - Function return and scope

A bit tedious, but ownership can be transferred back to the calling function with the return value.

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

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

    let s3 = takes_and_gives_back(s2);  // s2 is moved into `takes_and_gives_back` becomes invalid
                                        // `takes_and_gives_back` returns a new String that into s3
}

fn gives_ownership() -> String {                 // `gives_ownership` move return val into the
                                                 // function that calls it

    let some_string = String::from("yours"); // some_string comes into scope

    some_string                                  // some_string is returned moves out of calling func
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into scope

    a_string  // a_string is returned and moves out to the calling function
}
```

🤝 **References and Borrowing** - Overview

Kind of like passing by reference in C/C++ but with some key differences:

- **References** are immutable by default
- **Borrowing** allows multiple references to the same data
- **Mutable references** are exclusive and have strict rules

References are created with the `&` symbol, and borrowing is done with `&mut` for mutable references (see next slide).

🤝 References and Borrowing - Simple borrowing example

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{s1}' is {len}."); // s1 still valid because it is borrowed
}

fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
} // s goes out of scope, but does not have ownership of what it refers to
```

🤝 **Mutable references** - General case

Borrowed references are not mutable by default. To allow mutation, use `&mut`

```rust
// let s = String::from("hello"); // WOULD NOT COMPILE!
let mut s = String::from("hello");
change(&mut s);

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

🤝 **Mutable references** - Data races safety

Compile time checks for mutable refs

⚠️ **NO** multiple mutable references to the same data

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s;    // ERROR! r1 is still active
println!("{}, {}", r1, r2);
```

⚠️ **NO** mutable references while immutable references are active

```
let mut s = String::from("hello");
let r1 = &s;         // OK
let r2 = &s;         // OK
let r3 = &mut s;     // ERROR! r1 and r2 are still active
println!("{}, {}, {}", r1, r2, r3);
```

🤝 **Mutable references** - Data races safety (2/2)

🔎 Use of scopes to limit mutable references

```rust
let mut s = String::from("hello");
{
    let r1 = &mut s;
} // r1 goes out of scope, allowing a new mutable reference
let r2 = &mut s; // OK!
```

🔎 Reference's scope ends after the last usage of the reference.

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

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{r1} and {r2}");
    // variables r1 and r2 will not be used after this point

    let r3 = &mut s; // no problem because r1/r2 are no longer valid
    println!("{r3}");
```

⚠️ **Reference caution** - Fixing a state management problem

❗ Tedious or even problematic when working on a reference

```
let mut s = String::from("hello world");
let word_index = first_word(&s); // word_index will get the value 5

s.clear(); // empties the String, making it equal to ""
// `word_index` still has the value 5 here, but no more string tied because s is invalid
println!("the first word is: {s[..word_index]}"); // ERROR! s is empty
```

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }
    s.len()
}
// Imagine implementing second_word() and managing state...
```

🔪 **String Slice Type** - A kind of reference

Slices are references to a contiguous sequence of elements in a collection. They are a reference to a part of a string or array.

```rust
let s = String::from("hello world");
let hello = &s[0..5];   // same as &s[..5]. Excludes the last index
let world = &s[6..11];  // same as &s[6..]. Includes the first index
```

🔪 String Slice - Refactoring `first_word()`

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i]; // return a slice(ref) of the string
        }
    }
    &s[..]  // or return the slice of whole string
}
```

```
// Compiler assures that the slice is valid as long as the string is valid
fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s); // immutable borrow (return type is &str)

    s.clear(); // error! mutable borrow while immutable borrow is active
    println!("the first word is: {word}");
}
```

🔪 **Other Slice types** - Array example `first_word()`

Similar to strings, slices can be used with arrays

```rust
fn main() {
    let a = [1, 2, 3, 4, 5];
    let slice = &a[1..3]; // slice is of type &[i32]
    assert_eq!([2, 3], slice);
}
```

Useful for passing parts of arrays to functions without copying the data.

🏗️ **Structs** - Overview

Structs data structure encapsulate fields of specific types and methods (just like in C++/OO language).

- If declared mutable, the whole struct is mutable.
- dot notation for named field access
- Methods are defined within the `impl` block

```rust
let mut user1 = User {
    username: String::from("user1"),
    phone: 1234567890
    active: true,
};

user1.active = false;
```

🏗️ **Structs** - Shorthands

```rust
fn build_user(username: String, phone: u32) -> User {  // Returns a User struct
    User {
        username, // shorthand for username: username
        phone,    // shorthand for phone: phone
        active: true
    }
}

// Struct update syntax
let user2 = User {
    phone: 9876543210
    ..user1 // copy the rest of the fields from user1
}
```

🏗️ Tuple Structs

Tuple structs are similar, but don't have named fields. Useful for naming tuples and creating new types.

```rust
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    // black and origin are different types
    let red = Color(255, 0, 0);
    let origin = Point(0, 0, 0);
}
// Cannot pass a Point even though they have the same fields' types
fn make_paler(color: Color) -> Color {
    Color(color.0 / 2, color.1 / 2, color.2 / 2)
}
```

⚙️ **Methods** - Basic implementation

Methods are defined within the `impl` block.

```rust
struct SumArgs {
  n1: i32,
  n2: i32,
}
impl SumArgs {
  fn add_numbers(&self) -> i32 { // self is alias for Self (instead of args: &SumArgs)
    self.n1 + self.n2
  }
}
fn main() {
  let args = SumArgs { n1: 2, n2: 3 };
  let sum = args.add_numbers(); // Or SumArgs::add_numbers(&args)
  println!("{} + {} = {}", args.n1, args.n2, sum);
}
```

Gian Lorenzetto. Rust - Structs, Functions and Methods. 2021. Medium Post

⚙️ **Methods** - Mutability

Use `&self` for read-only and `&mut self` for methods that modify the struct.

```rust
struct Rectangle {
    width: u32,
    height: u32
}
impl Rectangle {
    fn area(&self) -> u32 {  // takes ownership of self (read-only)
        self.width * self.height
    }
    fn half_rect(&mut self) {  // borrows mutably
        self.width /= 2;
        self.height /= 2;
    }
    fn width(&self) -> bool {  // Getters in Rust
        self.width > 0
    }
}
let mut rect = Rectangle { width: 10, height: 20 };
println!("rect's width is valid: {} because width={}", rect.width(), rect.width);
```

⚙️ **Methods** - Automatic referencing/dereferencing

Unlike in C/C++, Rust automatically references and dereferences when calling methods (No `->` operator or `(*object).something()`)

```
p1.distance(&p2); // Both are the same, version1 is more readable
(&p1).distance(&p2);
```

With `object.something()`, Rust automatically adds in `&`, `&mut`, or `*` to match signature of the method.

🔎 It depends wether method is **reading (`&self`), writing (`&mut self`), or consuming (`self`)**

⚙️ **Methods** - Associated function

When a function is associated with a struct, it doesn't take `self` as a parameter.

- Often used for constructor
- Called with the `::` syntax

```rust
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
let square = Rectangle::square(10);
```

📋 **Enums** - Overview

- `Enums` are a way to define a type by enumerating its possible variants
- Each variant can have different data associated with it (*i.e.* *`struct`*, *`String`* ...)
- Namespaced under identifier, accessed with `Enum::variant` syntax
- Default constructor is `Enum::variant(data)`

```rust
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
// Construct instances of each variant
let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

📋 **Enums** - Advantages over `struct`

Use of `impl` blocks for **common methods** that applies to **all variants**

```rust
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
} // Could be same as 4 different structs `struct Quit`, `struct Move{...}`

impl Message {
    fn send(&self) {
        // self ref to the variant instance
        println!("Sending message {:?}...", self);
    }
}
let m = Message::Write(String::from("Hello, world!"));
m.send();
```

❓ Option Enum - NULL free!

Rust has no `null` value, but uses the `Option` enum to represent the presence or absence of a value from standard library.

```rust
enum Option<T> { // Generic type T
    Some(T),     // Some value of type T
    None,
}

let x: i8 = 5;
let y: Option<i8> = Some(5); // Some value
let z: Option<i8> = None;  // No value

let sum = x + y;  // Won't compile because i8 + Option<i8> are different types
                  // and sum not implemented
```

With `Option`, the compiler forces you to handle the case where the value is `None`.

⚔️ Match Expression

☠️ Dangling Pointers -

⚙️ TODO

🛠️ **Practical project #1** - Write an I/O CLI program

Halfway project for a `grep` clone CLI app covers:

1. Code organization (crates, modules)
2. Use of containers and strings
3. Error handling
4. Using traits and lifetimes
5. Testing and documentation

Klabnik, Steve, and Carol Nichols. The Rust Programming Language. 2nd ed., No Starch Press.

🛠️ **Practical project #2** - Building a Multithreaded Web Server

Final Project from the book includes :

1. Learn TCP/IP networking and HTTP
2. Listen to TCP connections on a socket
3. Parse HTTP requests
4. Generate HTTP responses
5. Handle multiple requests concurrently with a thread pool

Klabnik, Steve, and Carol Nichols. The Rust Programming Language. 2nd ed., No Starch Press.

⚙️ THEORY STUFF

TODO