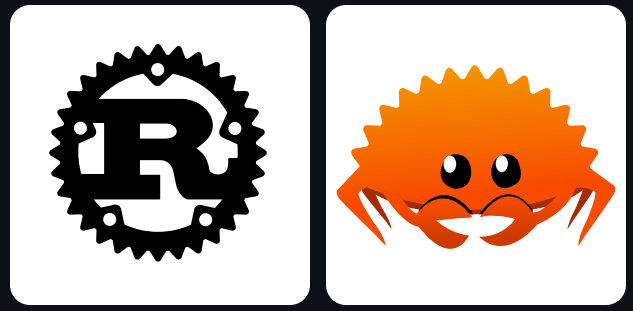
Introduction to the Rust programming Language



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

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

📚 **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. Functional and OO features
8. Smart pointers and Concurrency
9. Patterns and matching + Advanced features

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

Rust Overview

- **Systems programming language** focused on **safety** and **performance**
- TODO

Currently known projects

TODO

Predicted use cases

TODO

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

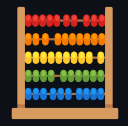
Variables are immutable by default

```
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

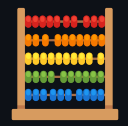
`rust-analyzer` provides type hints and suggestions

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

Explicit type annotations can or must be used

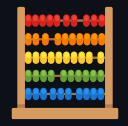
```
let mut num: String = String::new(); // Can be annotated or inferred
num = "42".to_string();

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

```
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

```
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

```
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)

```rust
// 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

```
// 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*)

```
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}");
};
```

Ownership - TODO

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