Learning Rust: A Beginner's Guide

From Zero to Rustacean

Rust Tutorial Series

December 14, 2025

Contents

Chapter 1

Getting Started with Rust

1.1 What is Rust?

Rust is a systems programming language that focuses on three key pillars: safety, speed, and concurrency. Unlike many other programming languages, Rust achieves memory safety without requiring a garbage collector, making it an excellent choice for systems programming where performance is critical.

1.1.1 Core Features

Safety: Rust provides memory safety without garbage collection through its innovative ownership system. This eliminates entire classes of bugs at compile time, including null pointer dereferences, buffer overflows, and data races.

Speed: Rust's performance is comparable to C and C++. It provides zero-cost abstractions, meaning you can use high-level features without sacrificing runtime performance.

Concurrency: Rust's type system and ownership model enable fearless concurrent programming. The compiler prevents data races at compile time, making it safe to write multi-threaded code.

Created by Mozilla Research, Rust has been voted the "most loved programming language" in Stack Overflow's developer surveys for several consecutive years, reflecting the strong appreciation developers have for its innovative approach to systems programming.

1.2 Why Learn Rust?

There are several compelling reasons to learn Rust:

1. **Memory Safety:** Rust eliminates null pointers and prevents data races through its ownership and borrowing system. These guarantees are enforced at compile time, meaning many bugs are caught before your code ever runs.

2. **Zero-Cost Abstractions:** You can use high-level programming features without runtime overhead. The abstractions compile down to code as efficient as if you had written it by hand in a lower-level language.

3. **Growing Ecosystem:** Rust comes with Cargo, an excellent package manager and build system. The Rust ecosystem includes a rich collection of libraries (called "crates") for everything from web development to embedded systems.

4. **Career Opportunities:** There is high demand for Rust developers in systems programming, web services, blockchain, embedded systems, and more. Companies like Mozilla, Dropbox, Amazon, Microsoft, and Google use Rust in production.

5. **Great Tooling:** Rust provides excellent compiler error messages that help you understand and fix problems. The language includes built-in testing support, documentation tools, and a formatter.

1.3 Your First Rust Program

Let's start with the traditional "Hello, World!" program:

```
fn main() {
    println!("Hello, World!");
}
```

This simple program demonstrates several key elements of Rust:

- `fn main()` is the entry point of every Rust program. When you run a Rust program, execution begins with the `main` function.
- `println!` is a macro (note the exclamation mark `!`). Macros are a powerful feature in Rust that allow code generation at compile time.
- Statements in Rust typically end with semicolons (`;`).

1.4 Installing Rust

Installing Rust is straightforward using `rustup`, the official Rust installer and version management tool. Visit `rustup.rs` and run the following command:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

This command downloads and runs the installer script. Follow the on-screen instructions to complete the installation.

After installation, verify that Rust is correctly installed by checking the versions of the Rust compiler and Cargo:

```
rustc --version
cargo --version
```

If both commands display version information, you're ready to start programming in Rust!

Chapter 2

Basic Syntax and Concepts

2.1 Variables and Mutability

One of Rust's distinctive features is that variables are immutable by default. This design choice encourages writing safer, more predictable code.

2.1.1 Immutable Variables

When you declare a variable with `let`, it is immutable:

```
let x = 5;
// x = 6;  // ERROR! Cannot assign twice to immutable variable
```

Attempting to change `x` after its initial assignment will result in a compilation error.

2.1.2 Mutable Variables

If you need to change a variable's value, you must explicitly declare it as mutable using the `mut` keyword:

```
let mut y = 5;
y = 6;  // OK!
println!("y = {}", y);
```

This explicit declaration makes it clear which variables can change throughout your program, improving code readability and making bugs easier to catch.

2.2 Data Types

Rust is a statically typed language, meaning it must know the types of all variables at compile time. However, the compiler can often infer types from the context.

2.2.1 Scalar Types

Scalar types represent a single value. Rust has four primary scalar types:

```
// Integer types (signed and unsigned)
let a: i32 = 42;       // 32-bit signed integer
let b: u64 = 100;      // 64-bit unsigned integer

// Floating-point types
let c: f64 = 3.14;     // 64-bit floating-point

```

```rust
// Boolean type
let d: bool = true;    // true or false

// Character type
let e: char = 'R';     // Unicode scalar value
```

2.2.2 Compound Types

Compound types can group multiple values. Rust has two primitive compound types:

Tuples: Group together values of different types:

```rust
let tup = (500, 6.4, 'x');
let (x, y, z) = tup;  // Destructuring
```

Arrays: Fixed-length collections of values of the same type:

```rust
let arr = [1, 2, 3, 4, 5];
let first = arr[0];     // Indexing starts at 0
```

2.3 Functions

Functions are fundamental building blocks in Rust. They are declared using the `fn` keyword.

```rust
fn add(a: i32, b: i32) -> i32 {
    a + b  // Expression (no semicolon) - this is the return value
}

fn subtract(a: i32, b: i32) -> i32 {
    return a - b;  // Explicit return statement
}

fn main() {
    println!("5 + 3 = {}", add(5, 3));
    println!("10 - 4 = {}", subtract(10, 4));
}
```

Note that in Rust, the last expression in a function is automatically returned if there's no semicolon. You can also use the `return` keyword for early returns or when you prefer explicit return statements.

2.4 Control Flow

2.4.1 if/else Expressions

Rust's `if` expressions allow branching based on conditions:

```rust
fn main() {
    let number = 7;

    if number < 5 {
        println!("Less than 5");
    } else if number < 10 {
        println!("Between 5 and 10");
    } else {
        println!("10 or greater");
    }
```

```
11
12      // if is an expression, so it returns a value
13      let x = if number < 5 { 1 } else { 2 };
14  }
```

Because `if` is an expression, you can use it on the right side of a `let` statement.

2.4.2 Loops

Rust provides three kinds of loops: `loop`, `while`, and `for`.

Infinite Loop: The `loop` keyword creates an infinite loop:

```
let mut count = 0;
loop {
    count += 1;
    if count == 10 {
        break;
    }
}
```

Conditional Loop: The `while` loop continues while a condition is true:

```
let mut n = 3;
while n != 0 {
    println!("{}!", n);
    n -= 1;
}
```

For Loop: The `for` loop iterates over collections:

```
let arr = [10, 20, 30];
for element in arr.iter() {
    println!("{}", element);
}

// Ranges are also common with for loops
for n in 1..4 {
    println!("{}!", n);
}
```

Chapter 3

Core Concepts: Ownership

Ownership is Rust's most unique feature. It enables memory safety without requiring a garbage collector. Understanding ownership is crucial to mastering Rust.

3.1 The Rules of Ownership

Rust's ownership system is governed by three rules:

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** (freed from memory).

These simple rules enable Rust to manage memory automatically without a garbage collector, preventing memory leaks and ensuring memory safety.

3.2 Move Semantics

When you assign a value from one variable to another, Rust doesn't create a copy by default for heap-allocated data. Instead, it *moves* the value:

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;  // s1 is moved to s2

    // println!("{}", s1);  // ERROR! s1 is no longer valid
    println!("{}", s2);     // OK
}
```

After the move, `s1` is no longer valid. This prevents double-free errors—a common bug in languages like C and C++.

3.3 Clone and Copy

3.3.1 Deep Copying with Clone

If you want to create a deep copy of heap data, use the `clone` method:

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

```

```
// Both s1 and s2 are valid
println!("{}", s1);
println!("{}", s2);
```

3.3.2 Stack-Only Copy

Simple types that are stored entirely on the stack implement the `Copy` trait. These types are copied rather than moved:

```
let x = 5;
let y = x;  // x is copied to y, not moved

// Both x and y are valid
println!("{}", x);
println!("{}", y);
```

Types that implement `Copy` include all integer types, floating-point types, booleans, and characters.

Chapter 4

References and Borrowing

While ownership is powerful, it can be restrictive. Rust provides *references* to allow you to refer to a value without taking ownership of it.

4.1 Using References

A reference is created using the `&` operator:

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

    println!("Length of '{}' is {}", s1, len);
    // s1 is still valid here!
}

fn calculate_length(s: &String) -> usize {
    s.len()
}  // s goes out of scope, but it doesn't own the String, so nothing is
    dropped
```

Creating a reference is called **borrowing**. When you borrow a value, you don't take ownership, so the value isn't dropped when the reference goes out of scope.

4.2 The Borrowing Rules

Rust enforces strict borrowing rules at compile time to prevent data races:

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

These rules prevent data races at compile time, ensuring thread safety.

4.3 Mutable References

You can create mutable references using `&mut`:

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

```
    println!("{}", s);
}

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

The restriction is that you can only have one mutable reference to a value in a particular scope. This prevents simultaneous mutations that could cause data races.

4.4 Slices

Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection:

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

    let hello = &s[0..5];    // or &s[..5]
    let world = &s[6..11];   // or &s[6..]

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

String slices have the type `&str`. Slices are references, so they don't take ownership.

Chapter 5

Structs and Enums

Structs and enums allow you to create custom data types that are meaningful for your domain.

5.1 Defining and Using Structs

Structs let you group related data together:

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

fn main() {
    let user1 = User {
        email: String::from("user@example.com"),
        username: String::from("user123"),
        active: true,
        sign_in_count: 1,
    };

    println!("User: {}", user1.username);
}
```

5.2 Methods

Methods are functions defined within the context of a struct (or enum or trait). They use the `impl` block:

```
struct Rectangle {
    width: u32,
    height: u32,
}

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

    // Associated function (doesn't take self)
```

```
    fn square(size: u32) -> Rectangle {
        Rectangle { width: size, height: size }
    }
}
```

5.3 Enums

Enums allow you to define a type by enumerating its possible variants:

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

let home = IpAddr::V4(127, 0, 0, 1);
let msg = Message::Write(String::from("hello"));
```

5.4 Pattern Matching

Pattern matching with `match` is a powerful control flow construct:

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

5.5 The Option Enum

Rust doesn't have null values. Instead, it uses the `Option<T>` enum to represent optional values:

```
fn divide(a: i32, b: i32) -> Option<i32> {
    if b == 0 {
        None
    } else {
        Some(a / b)
    }
```

```
}

fn main() {
    match divide(10, 2) {
        Some(result) => println!("Result: {}", result),
        None => println!("Cannot divide by zero"),
    }
}
```

The `Option<T>` enum forces you to handle the case where a value might be absent, preventing null pointer errors.

Chapter 6

Collections

Rust's standard library includes several useful data structures called collections. Unlike arrays and tuples, collections store data on the heap, meaning their size can change at runtime.

6.1 Vectors

Vectors (`Vec<T>`) are resizable arrays:

```rust
fn main() {
    // Creating vectors
    let v1: Vec<i32> = Vec::new();
    let v2 = vec![1, 2, 3];

    // Adding elements
    let mut v3 = Vec::new();
    v3.push(5);
    v3.push(6);

    // Accessing elements
    let third = &v3[2];
    match v3.get(2) {
        Some(third) => println!("Third: {}", third),
        None => println!("No third element"),
    }
}
```

6.2 Hash Maps

Hash maps store key-value pairs:

```rust
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    // Getting values
    let team = String::from("Blue");
    let score = scores.get(&team);

```

```
13      // Iterating
14      for (key, value) in &scores {
15          println!("{}: {}", key, value);
16      }
17  }
```

Chapter 7

Error Handling

Rust groups errors into two major categories: recoverable and unrecoverable errors. For recoverable errors, Rust uses the `Result<T, E>` type.

7.1 The Result Type

The `Result` enum has two variants: `Ok` and `Err`:

```
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
            println!("Error: {:?}", error);
            return;
        }
    };
}
```

7.2 The ? Operator

The `?` operator makes error handling more concise:

```
use std::fs::File;
use std::io::{self, Read};

fn read_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}

// Even more concise
fn read_file_short() -> Result<String, io::Error> {
    std::fs::read_to_string("hello.txt")
}
```

The `?` operator propagates errors to the calling function, making error handling clean and readable.

Chapter 8

Generics and Traits

Generics allow you to write flexible, reusable code. Traits define shared behavior.

8.1 Generic Functions

You can write functions that work with multiple types:

```
fn largest<T: PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];
    println!("Largest: {}", largest(&numbers));
}
```

8.2 Traits

Traits define shared behavior:

```
trait Summary {
    fn summarize(&self) -> String;
}

struct Article {
    headline: String,
    content: String,
}

impl Summary for Article {
    fn summarize(&self) -> String {
        format!("{}", self.headline)
    }
}
```

Traits are similar to interfaces in other languages but with some unique features that make them more powerful.

Chapter 9

Fearless Concurrency

Rust's ownership and type systems enable safe concurrent programming. The compiler prevents data races at compile time.

9.1 Threads

Creating threads in Rust is straightforward:

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Thread: {}", i);
        }
    });

    for i in 1..5 {
        println!("Main: {}", i);
    }

    handle.join().unwrap();
}
```

9.2 Message Passing with Channels

Channels allow threads to communicate by sending messages:

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        tx.send(String::from("hi")).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

9.3 Shared State with Mutex

For shared state, Rust provides `Mutex` and `Arc`:

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            *counter.lock().unwrap() += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Chapter 10

Summary and Next Steps

10.1 What You've Learned

Congratulations! You've covered the fundamental concepts of Rust programming:

- **Basics**: Variables, types, functions, and control flow
- **Ownership**: Rust's unique memory management system
- **Borrowing**: References and the borrow checker
- **Structs and Enums**: Creating custom data types
- **Collections**: Vectors, strings, and hash maps
- **Error Handling**: Using Result and Option types
- **Generics and Traits**: Writing reusable, polymorphic code
- **Concurrency**: Safe concurrent programming with threads and channels

10.2 Next Steps

To continue your Rust journey:

1. **Practice Daily**: Write Rust code every day to build muscle memory and deepen your understanding.
2. **Build Projects**: Apply what you've learned by building real projects. Start small and gradually increase complexity.
3. **Read Code**: Study well-written Rust projects on GitHub. Reading others' code is one of the best ways to learn.
4. **Join the Community**: Participate in the Rust forum, Discord, or Reddit. The Rust community is welcoming and helpful.
5. **Advanced Topics**: Explore more advanced topics like macros, async/await, and unsafe Rust.
6. **Contribute**: Consider contributing to open source Rust projects. It's a great way to learn and give back.

10.3 Resources

10.3.1 Official Resources

- **The Rust Book**: `doc.rust-lang.org/book/` - The comprehensive official guide
- **Rust by Example**: `doc.rust-lang.org/rust-by-example/` - Learn through annotated examples
- **Rustlings**: Interactive exercises to practice Rust concepts
- **Rust Playground**: `play.rust-lang.org` - Try code in your browser

10.3.2 Community Resources

- **Forum**: `users.rust-lang.org` - Ask questions and share knowledge
- **Reddit**: `r/rust` - News, discussions, and community
- **Discord**: Rust Community Server - Real-time chat

10.3.3 This Tutorial

Find the source code and more resources at:

- **GitHub**: `github.com/adiel2012/rust-beginner`

10.4 Conclusion

Rust is a powerful language that brings together performance, reliability, and productivity. While it has a steeper learning curve than some languages, the investment pays off in safer, faster, and more maintainable code.

Keep practicing, stay curious, and enjoy your journey as a Rustacean!

Happy Coding!