Rust Language Study — User Manual

# Overview

This folder contains a short Rust language study with slides and runnable demo source code. The demos include: TSimple.rs, TVar.rs, TSel.rs, TLoop.rs, TSub.rs

# Install Rust Toolchain

1. Install Rust via rustup ([https://rustup.rs](https://rustup.rs/)).
2. Verify installation:
   * rustc –version
   * cargo --version
3. Optional: install IDE support (VS Code + rust-analyzer).

# Build & Run Demos

1. Open a terminal and navigate to the project folder.
2. To compile a rust program (file with extension “.rs”) run:

* rustc <filename.rs>

1. To run the executable run:

* ./<filename>

# 4. Demo Descriptions

TSimple: main function, rust program structure.

TVar: primitive data types, compound data types, mutability, casting.

TSel: if & if-else statements, match statements, relational & logical operators.

TLoop: loop/while/for, break, ranges, array iterator functions.

TSub: pass-by-reference, borrowing, return values, fallibilism, recursion.

# 5. Language References

• The Rust Programming Language (The Book): https://doc.rust-lang.org/book/

• Rust by Example: https://doc.rust-lang.org/rust-by-example/

• Standard Library Docs: https://doc.rust-lang.org/std/

• Rustonomicon (advanced): https://doc.rust-lang.org/nomicon/

# 6. Source Code Copies

Source files are under src/. For convenience, the code is included verbatim below.

## TSimple.rs

/\*

This is a simple Rust program that prints "Hello world from Rust 🦀!" to the console.

It demonstrates the basic structure of a Rust program, including the main function.

To compile and run this Rust file:

$ rustc TSimple.rs

$ ./TSimple

\*/

fn main() {

println!("Hello world from Rust 🦀!");

}

## TVar.rs

## /\*

## This Rust program demonstrates various variable types, mutability, type inference, type casting, and basic operations.

## It covers signed and unsigned integers, floating-point numbers, booleans, characters, strings, tuples, and arrays.

## It also illustrates mutable vs immutable variables, type casting, and basic arithmetic operations.

## Primitive Types:

## - Signed Integers: i8, i16, i32, i64, i128

## - Unsigned Integers: u8, u16, u32, u64, u128

## - Floating Point Numbers: f32, f64

## - Boolean: bool

## - Character: char

## Compound Types:

## - Tuple: (i32, f64, char)

## - Array: [i32; 5]

## - Slice: &[i32]

## - String: String

## - Note: Rust does not have a built-in string type; it uses the String type from the standard library.

## Mutability:

## - All variables are immutable by default. Use 'mut' keyword to make any variable mutable.

## To compile and run this Rust file:

## $ rustc TVar.rs

## $ ./TVar

## \*/

## fn main() {

## // Signed integers of different sizes

## // i8, i16, i32, i64, i128

## let a: i8 = -10;

## let b: i16 = -200;

## let c: i32 = -30000;

## let d: i64 = -4000000;

## let e: i128 = -5000000000;

## println!("Signed integers: {}, {}, {}, {}, {}", a, b, c, d, e);

## // Unsigned integers of different sizes

## // u8, u16, u32, u64, u128

## let f: u8 = 10;

## let g: u16 = 200;

## let h: u32 = 30000;

## let i: u64 = 4000000;

## let j: u128 = 5000000000;

## println!("Unsigned integers: {}, {}, {}, {}, {}", f, g, h, i, j);

## // Floating point numbers

## // f32, f64

## let k: f32 = 3.14;

## let l: f64 = 2.718281828459045;

## println!("Floating point numbers: {}, {}", k, l);

## // Boolean

## let m: bool = true;

## let n: bool = false;

## println!("Booleans: {}, {}", m, n);

## // Character

## let o: char = 'R';

## let p: char = '🦀';

## println!("Characters: {}, {}", o, p);

## // Tuple

## let q: (i32, f64, char) = (42, 3.14, 'R');

## println!("Tuple: {:?}", q); // Debug print

## let (x, y, z) = q; // Destructuring

## println!("Destructured Tuple: {}, {}, {}", x, y, z);

## println!("Accessed Tuple element: {}", q.0); // Accessing elements

## // Array

## let r: [i32; 5] = [1, 2, 3, 4, 5];

## println!("Array: {:?}", r); // Debug print

## let first\_element = r[0]; // Accessing elements

## let n: u128 = r.len(); // Array length

## r[2] = 10; // Modifying an element

## let s: [i32; 5] = [0; 5]; // Array of 5 elements initialized to 0

## println!("Initialized Array: {:?}", s);

## // Slice

## let t: &[i32] = &r; // Reference to an array (slice)

## println!("Array Slice: {:?}", t);

## // String

## let mut u: String = String::from("hello");

## u.push\_str(", Rust!");

## println!("String: {}", u);

## // mutable vs immutable

## let v: i32 = 10; // Immutable by default

## //v += 20; // This would cause a compile-time error

## let mut w: i32 = 10; // Mutable variable

## println!("Before update: {}", w);

## w += 20; // This is allowed

## println!("Immutable: {}, Mutable: {}", v, w);

## // Type inference

## let b = 10; // Rust infers this as i32

## let c = 3.14; // Rust infers this as f64

## println!("Inferred types: {}, {}", b, c);

## // Type casting

## let d: i32 = 10;

## let e: f64 = d as f64; // Casting i32 to f64

## println!("Type casting: {}", e);

## // Basic arithmetic operations

## let sum = 5 + 10; // Addition

## let difference = 95.5 - 4.3; // Subtraction

## let product = 4 \* 30; // Multiplication

## let quotient = 56.7 / 32.2; // Division

## let remainder = 43 % 5; // Modulus

## println!("Arithmetic operations: {}, {}, {}, {}, {}", sum, difference, product, quotient, remainder);

## // Compound assignment operators

## let mut f = 5;

## f += 10; // f = f + 10

## //f++; // Rust does not support ++ operator

## //f--; // Rust does not support -- operator

## f -= 2; // f = f - 2

## f \*= 3; // f = f \* 3

## f /= 4; // f = f / 4

## f %= 3; // f = f % 3

## println!("Compound assignment result: {}", f);

## }

## TSel.rs

## /\*

## This Rust program demonstrates conditional statements including if, if-else, and match.

## It shows relational operations, logical operations, and how to use these constructs.

## Relational Operators:

## - Equal to: ==

## - Not equal to: !=

## - Greater than: >

## - Less than: <

## - Greater than or equal to: >=

## - Less than or equal to: <=

## Logical Operators:

## - AND: &&

## - OR: ||

## - NOT: !

## To compile and run this Rust file:

## $ rustc TSel.rs

## $ ./TSel

## \*/

## use std::io;

## fn main() {

## // simple if

## let num1: i128 = 5;

## let num2: i128 = 9;

## if num1 < num2 {

## println!("{} is less than {}", num1, num2);

## }

## // if-else

## println!("How old are you?");

## let mut age: String = String::new();

## io::stdin()

## .read\_line(&mut age)

## .expect("Failed to read line");

## let age: i32 = age.trim().parse::<i32>().expect("Please enter a number") as i32;

## if age < 13 {

## println!("You are a child.");

## }

## else if age < 18 {

## println!("You are a teenager.");

## }

## else {

## println!("You are an adult.");

## }

## // if as expression

## let is\_even: bool = if age % 2 == 0 { true } else { false };

## println!("Is your age even? {}", is\_even);

## // match statement

## println!("Enter a number between 1 and 5:");

## let mut number: String = String::new();

## io::stdin()

## .read\_line(&mut number)

## .expect("Failed to read line");

## let number: i32 = number.trim().parse::<i32>().expect("Please enter a number") as i32;

## match number {

## 1 => println!("You entered one."),

## 2 => println!("You entered two."),

## 3 => println!("You entered three."),

## 4 => println!("You entered four."),

## 5 => println!("You entered five."),

## \_ => println!("Number out of range!"),

## }

## }

## TLoop.rs

/\*!  
TLoop: "Loops and iteration"  
- `loop`, `while`, and `for`  
- Iterator adapters (`map`, `filter`, `sum`)  
\*/  
fn main() {  
 // loop + break with value  
 let mut x = 0;  
 let until\_10 = loop {  
 x += 1;  
 if x == 10 { break x; }  
 };  
 println!("loop result: {until\_10}");  
  
 // while  
 let mut n = 3;  
 while n > 0 {  
 println!("while: {n}");  
 n -= 1;  
 }  
  
 // for over ranges and collections  
 let v = vec![1, 2, 3, 4, 5];  
 let evens\_squared\_sum: i32 = v.iter()  
 .filter(|&&k| k % 2 == 0)  
 .map(|k| k \* k)  
 .sum();  
 println!("sum of squares of evens: {evens\_squared\_sum}");  
  
 for i in 0..3 {  
 println!("for i in 0..3: {i}");  
 }  
}

## TSub.rs

/\*

This program demonstrates subprograms (functions) in Rust:

- Simple function with no params/returns

- Functions with parameters and return values

- Borrowing & mutation using references (& and &mut)

- Working with slices

- Recursion

Pass by Reference:

- Read-only reference: &myVar

- Mutable reference: &mut myVar

To compile and run this Rust file:

$ rustc TSub.rs

$ ./TSub

\*/

fn main() {

println!("=== Rust Subprograms Demo 🦀 ===");

// 1) Simple function

greet();

// 2) "Borrowing" a string slice (&string) (Pass by reference)

// Notice - this is an immutable (read only) reference and not a pointer like C/C++

let string = String::from("Alice");

greet\_name(&string);

// 3) Return values

let s: i32 = add(3, 4);

println!("add(3, 4) = {}", s);

// 4) "Borrowing" with a mutable reference (&mut)

let mut value = 10;

println!("Before increment: {}", value);

increment(&mut value);

println!("After increment: {}", value);

// 5) Slices

let nums = [1, 2, 3, 4, 5];

let total = sum\_slice(&nums[1..4]); // Pass a slice (sub array) of the array - indices 1, 2, 3

println!("sum([2,3,4]) = {}", total);

// 6) Recursion

let n: u64 = 5;

println!("factorial({}) = {}", n, factorial(n));

// 7) Option return for fallible operations

match divide(10, 2) {

Some(q) => println!("10 / 2 = {}", q),

None => println!("10 / 2 = undefined (division by zero)"),

}

match divide(10, 0) {

Some(q) => println!("10 / 0 = {}", q),

None => println!("10 / 0 = undefined (division by zero)"),

}

}

// --- Subprogram definitions ---

// 1) No params, no return

fn greet() {

println!("Greetings from greet() subprogram!");

}

// 2) Borrow a string slice parameter

fn greet\_name(name: &str) {

println!("Hello, {}!", name);

}

// 3) Parameters with a return value

fn add(a: i32, b: i32) -> i32 {

a + b // expression (no semicolon) is the return value

}

// 4) Mutably borrow an i32 and modify it in place

fn increment(n: &mut i32) {

\*n += 1;

}

// 5) Take a slice and compute a sum

fn sum\_slice(nums: &[i32]) -> i32 {

nums.iter().sum()

}

// 6) Recursive factorial

fn factorial(n: u64) -> u64 {

if n <= 1 { 1 } else { n \* factorial(n - 1) }

}

// 7) Fallible division using Option (None for division by zero)

fn divide(a: i32, b: i32) -> Option<i32> {

if b == 0 { None } else { Some(a / b) }

}