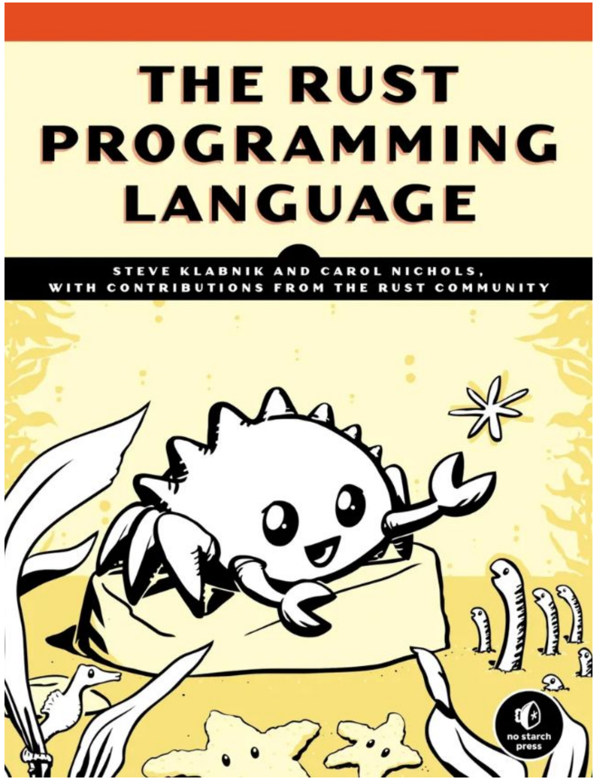
The Rust Programming Language

Prof. Dr. Jefferson de Carvalho

This material was based on https://doc.rust-lang.org/book/title-page.html

Reference


THE RUST PROGRAMMING LANGUAGE
STEVE KLABNIK AND CAROL NICHOLS, WITH CONTRIBUTIONS FROM THE RUST COMMUNITY
no starch press

By Steve Klabnik and Carol Nichols, with contributions from the Rust Community

The HTML format is available online at https://doc.rust-lang.org/stable/book/

Introduction

- Rust is a **general-purpose** programming language emphasizing **performance**, **type safety**, and **concurrency**.
- Rust enforces memory safety, meaning that all references point to **valid memory**.
- Rust **does not enforce a programming paradigm**, but was influenced by ideas from **functional programming**, including **immutability**, **higher-order functions**, **algebraic data types**, and **pattern matching**. It also supports **object-oriented programming via structs**, **enums**, **traits**, and **methods**. It is popular for systems programming.

https://en.wikipedia.org/wiki/Rust_(programming_language)

Introduction

- Rust was adopted by companies including **Amazon**, **Discord**, **Dropbox**, **Google** (Alphabet), **Meta**, and **Microsoft**. In December 2022, it became the first language other than **C and assembly** to be supported in the development of the **Linux kernel**.

https://en.wikipedia.org/wiki/Rust_(programming_language)

Introduction

- The **Rust** programming language helps you write faster, more reliable software.
- Rust gives you the option to **control low-level details** (such as memory usage) without all the hassle traditionally associated with such control.
- **Cargo**, the included dependency manager and build tool, makes **adding**, **compiling**, and **managing** dependencies painless and consistent across the Rust ecosystem.

Installation

- **Linux**
  - $ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
- **Windows**
  - On Windows, go to https://www.rust-lang.org/tools/install and follow the instructions for installing Rust./
- **Rust Playground**
  - https://play.rust-lang.org/

Hello, World!

For Linux, macOS, and PowerShell on Windows, enter this:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

Next, make a new source file and call it *hello_world.rs*

```rust
fn main() {
    println!("Hello, world!"); //calling a macro!
}
```

Hello, World!

Compile and run your code (Linux):

```
$ rustc hello_world.rs
$ ./hello
Hello, world!
```

On Windows:

```
> rustc main.rs
> .\main.exe
Hello, world!
```

Hello, World!

- Rust is an ***ahead-of-time compiled*** language, meaning you can compile a program and give the executable to someone else, and they can run it even without having Rust installed.
- Just compiling with `rustc` is fine for simple programs, but as your project grows, you'll want to **manage all the options** and make it **easy to share your code**.

Hello, Cargo!

- **Cargo** is Rust’s build system and package manage.r
- Cargo handles a lot of tasks for you, such as **building** your code, **downloading** the libraries your code depends on, and building those libraries (dependencies).
- As you write more complex Rust programs, you’ll add **dependencies**, and if you start a project using Cargo, adding dependencies will be much easier to do.

Hello, Cargo!

- Test you Cargo installation

```
$ cargo --version
```

- Create a Cargo project:

```
$ cargo new hello_cargo --vcs none
$ cd hello_cargo
```

- Build your project:

```
$ cargo build
```

- Run

```
$ cargo run
```

Hello, Cargo!

- Another commands:
  - `$ cargo check`
  - `$ cargo build --release`

Implementing a Guessing Game (Exercise)

This program will generate a **random integer between 1 and 100**. It will then prompt the player to enter a guess. After a guess is entered, the program will indicate whether the guess is too low or too high. If the guess is correct, the game will print a congratulatory message and exit.

Setting Up a New Project

```
$ cargo new guessing_game
$ cd guessing_game
$ cargo run
```

First Version - Input and Output Test

```rust
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

Second Version - Generate Random Number

- **Crate** is a collection of Rust **source code files**.
- The project we've been building is a binary crate, which is an **executable;**
- The **rand crate** is a **library crate**, which contains code that is intended to be used in other programs and can't be executed on its own.
- Filename: **Cargo.toml** *(Tom's Obvious, Minimal Language)***.**

```
[dependencies]
rand = "0.8.5"
```

https://crates.io/

Second Version - Generate Random Number

```
use std::io;
use rand::Rng; //*trait

fn main() {
    println!("Guess the number!");

    let secret_number = rand::thread_rng().gen_range(1..=100);
    println!("The secret number is: {secret_number}");

    println!("Please input your guess.");
    let mut guess = String::new();
    io::stdin()
        .read_line(&mut guess)
        .expect("Failed to read line");
    println!("You guessed: {guess}");
}
```

**A trait defines the functionality a particular type has and can share with other types*

Third Version - Comparing Guess and Secret Number (!)

```rust
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    // --snip--

    println!("You guessed: {guess}");

    guess.cmp(&secret_number) {
        Ordering::Less => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal => println!("You win!"),
    }
}
```

The `Ordering` type is another enum and has the variants `Less`, `Greater`, and `Equal`.

Third Version - Comparing Guess and Secret Number

```rust
// --snip--

let mut guess = String::new();

io::stdin()
    .read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = guess.trim().parse().expect("Please type a number!");

println!("You guessed: {guess}");

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
```

use a `match` expression to decide what to do next based on which variant of `Ordering`

Fourth Version - Loops

```
    // --snip--

    println!("The secret number is: {secret_number}");

    loop {
        println!("Please input your guess.");

        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => println!("You win!"),
        }
    }
}
```

Fifth Version - Loops can Quit

```
        // --snip--

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Sixth Version - Handling Invalid Input

```rust
// --snip--

io::stdin()
    .read_line(&mut guess)
    .expect("Failed to read line");

let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};

println!("You guessed: {guess}");

// --snip--
```

`parse` returns a `Result` type and `Result` is an enum that has the variants `Ok` and `Err.`

The underscore, `_`, is a catchall value; in this example, we're saying we want to match all `Err` values, no matter what information they have inside them.

Complete Version

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Guess the number!");

    let secret_number =
rand::thread_rng().gen_range(1..=100);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Failed to read line");
```

```
        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Too
small!"),
            Ordering::Greater => println!("Too
big!"),
            Ordering::Equal => {
                println!("You win!");
                break;
            }
        }
    }
}
```

Variables and Mutability

- As mentioned earlier, by default, variables are **immutable**;
- This is one of many nudges Rust gives you to write your code in a way that takes advantage of the **safety** and **easy concurrency** that Rust offers;
- However, you still have the option to make your variables **mutable**!

Variables and Mutability - mut (!)

```rust
fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

When a variable is immutable, once a value is bound to a name, you can’t change that value.

Variables and Mutability - mut

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

Mutability can be very useful, and can make code more convenient to write. Although variables are immutable by default, you can make them mutable by adding `mut` in front of the variable name.

Adding `mut` also conveys intent to future readers of the code by indicating that other parts of the code will be changing this variable’s value.

Variables and Mutability - constants

```rust
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

Constants VS Immutable

You aren’t allowed to use `mut` with constants. Constants aren’t just immutable by default—they’re always immutable. You declare constants using the `const` keyword instead of the `let` keyword, and the type of the value *must* be annotated.

Constants can be declared in any scope, including the global scope, which makes them useful for values that many parts of code need to know about.

Constants may be set only to a constant expression, not the result of a value that could only be computed at runtime

Variables and Mutability - shadowing

```
fn main() {
    let x = 5;
    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {x}");
    }

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

We can shadow a variable by using the same variable's name and repeating the use of the `let` keyword

Data Types

- Every value in Rust is of a certain **data type**, which tells Rust what kind of data is being specified so it knows how to work with that data;
- Rust is a **statically typed language**, which means that it **must know** the types of all variables at **compile time**;
- In cases when many types are possible, such as when we converted a String to a numeric type using parse, we must add a **type annotation**, like this:

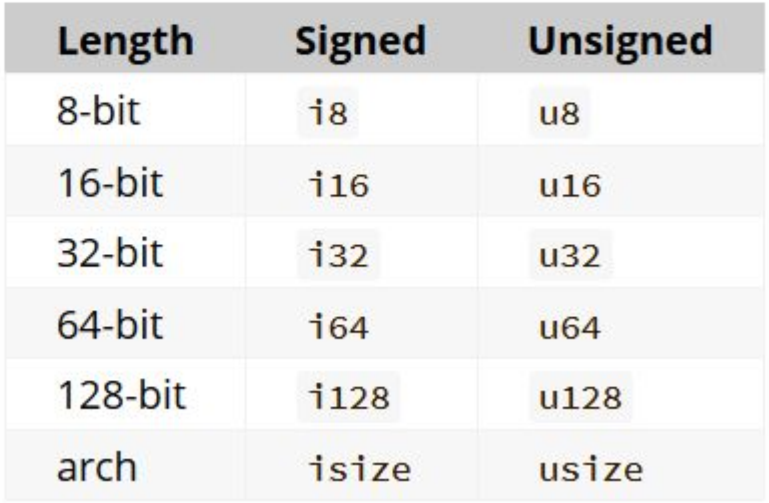
```rust
let guess: u32 = "42".parse().expect("Not a number!");
```

Scalar Types

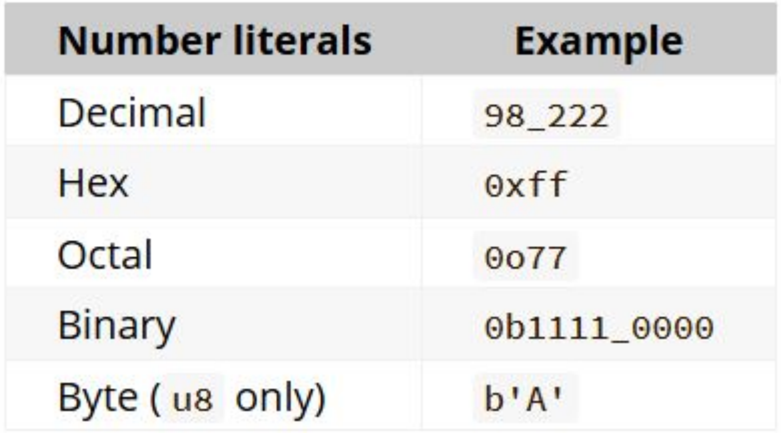
- A scalar type represents a **single value**;
- Rust has four primary scalar types: **integers, floating-point numbers, booleans,** and **characters**;

Integer Types

An integer is a number without a fractional component;



Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`



Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

Floating-Point Types

Rust also has two primitive types for floating-point numbers, which are numbers with decimal points.

```rust
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

Floating-Point Types - Printing

Default formatting:

```
let x: f64 = 3.14159;
println!("{}", x); // Output: 3.14159
```

This will print the float number with the default formatting, which includes a decimal point and up to 6 significant digits.

Specifying precision:

```
let x: f64 = 3.14159;
println!("{:.2}", x); // Output: 3.14
```

This will print the float number with 2 decimal places.

Floating-Point Types - Printing

Using a custom format string:

```
let x: f64 = 3.14159;
println!("{:10.4}", x); // Output:  3.1416
```

This will print the float number with the default formatting, which includes a decimal point and up to 6 significant digits.This will print the float number with a minimum width of 10 characters, and 4 decimal places.

Printing with scientific notation:

```
let x: f64 = 1e-40;
println!("{:e}", x); // Output: 1.00e-40
```

This will print the float number in scientific notation.

Numeric Operations

Rust supports the basic mathematical operations you'd expect for all the number types: **addition, subtraction, multiplication, division, and remainder**. Integer division truncates toward zero to the nearest integer.

```rust
fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;
    let truncated = -5 / 3; // Results in -1

    // remainder
    let remainder = 43 % 5;
}
```

The Boolean Type

As in most other programming languages, a Boolean type in Rust has two possible values: **true** and **false**. Booleans are one byte in size. The Boolean type in Rust is specified using `bool`.

```rust
fn main() {
    let t = true;
    let f: bool = false; // with explicit type annotation
}
```

The Character Type

Rust’s char type is the language’s most primitive alphabetic type. Here are some examples of declaring char values:

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

Compound Types

Compound types can group multiple values into one type. Rust has two primitive compound types: **tuples** and **arrays**.

Compound Types - Tuples

A tuple is a general way of **grouping together a number of values** with a **variety of types** into one compound type. Tuples have a fixed length: once declared, they **cannot grow** or **shrink** in size.

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}

fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

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

Compound Types - Tuples

We can also access a tuple element directly by using a period (.) followed by the index of the value we want to access. For example:

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Compound Types - Tuples - Exercise!

Create two tuples representing a 3-dimensional point (x,y,z). Then, calculate de Euclidean distance between them. Below, an example using parsing “as” and square root function.

```
use std::f32; //you’ll need f32 or f64 to use sqrt()

fn main() {
    let x : i32 = 16;
    let res : f32 = x as f32;
    println!("{}", res.sqrt());
}
```

Compound Types - Arrays

Another way to have a collection of multiple values is with an **array**. Unlike a tuple, every element of an array must have the **same type**. Unlike arrays in some other languages, arrays in Rust have a fixed length.

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let months = ["January", "February", "March", "April", "May", "June", "July",
                  "August", "September", "October", "November", "December"];
}
```

Compound Types - Arrays

```
let a: [i32; 5] = [1, 2, 3, 4, 5];

let a = [3; 5]; // the same as “let a = [3, 3, 3, 3, 3];”
```

Compound Types - Arrays

Accessing array elements:

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let first = a[0];
    let second = a[1];
}
```

Compound Types - Arrays (Invalid Access)

```
use std::io;

fn main() {
    let a = [1, 2, 3, 4, 5];

    println!("Please enter an array index.");

    let mut index = String::new();

    io::stdin()
        .read_line(&mut index)
        .expect("Failed to read line");

    let index: usize = index
        .trim()
        .parse()
        .expect("Index entered was not a number");

    let element = a[index];

    println!("The value of the element at index {index} is: {element}");
}
```

`usize` - The size of this primitive is how many bytes it takes to reference any location in memory. For example, on a 32 bit target, this is 4 bytes and on a 64 bit target, this is 8 bytes.

Compound Types - Arrays and Loops

```
let arr = [1, 2, 3, 4, 5];
for elem in arr.iter() {
    println!("{}", elem);
}
```

```
let arr = [1, 2, 3, 4, 5];
let mut i = 0;
while i < arr.len() {
    println!("{}", arr[i]);
    i += 1;
}
```

```
let arr = [1, 2, 3, 4, 5];
let mut i = 0;
loop {
    if i >= arr.len() {
        break;
    }
    println!("{}", arr[i]);
    i += 1;
}
```

```
let arr = [1, 2, 3, 4, 5];
for elem in &arr[1..=4] {
     println!("{}", elem);
}
```

Compound Types - Arrays - Exercise!

- Find the biggest element in a array;
- Find the smallest element in a array;

Control Flow

The ability to run some code **depending on whether a condition is true and to run some code repeatedly while a condition is true** are basic building blocks in most programming languages.

The most common constructs that let you control the flow of execution of Rust code are **if expressions** and **loops**.

Control Flow - if Expressions

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

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

Control Flow - Multiple elses

```rust
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 - if in a let statement

```rust
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

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

Control Flow - if - Exercise!

Implement your UFC’s system of grades.

Control Flow - loops

```
fn main() {
    loop {
        println!("again!");
    }
}
```

Control Flow - Returning Values from loops

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

    let result = loop {
        counter += 1;

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

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

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 - while and collections

```rust
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);

        index += 1;
    }
}
```

Control Flow - for and collections

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

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

Control Flow - for reverse

```rust
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
```

Control Flow - Exercise!

Implement a program that returns if a integer is a prime number or not.

Functions

Functions are **prevalent** in Rust code. You’ve already seen one of the most important functions in the language: the `main` function, which is the entry point of many programs.

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

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

Functions - Parameters

We can define functions to have **parameters**, which are special variables that are part of a function's **signature**.

When a function has parameters, you can provide it with **concrete values** for those parameters. Technically, the concrete values are called **arguments**.

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

Functions - Parameters

In function signatures, you **must** declare the type of each parameter.

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

Functions - Statements and Expressions

- **Statements** are instructions that perform some action and do not return a value.
- **Expressions** evaluate to a resultant value. Let’s look at some examples.

Functions - Statements and Expressions

Creating a variable and assigning a value to it with the let keyword is a **statement**.

```
fn main() {
    let y = 6;
}
```

```
fn main() {
    let x = (let y = 6); //results in an error
}
```

Functions - Statements and Expressions

Expressions evaluate to a value and make up most of the rest of the code that you'll write in Rust.

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

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

Functions - Return Values

Functions can return values to the code that calls them. We don't name return values, but we **must declare their type** after an arrow (`->`).

In Rust, the **return value** of the function is **synonymous** with the value of the **final expression** in the block of the body of a function.

```rust
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

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

Functions - Return Values

Running this code will print The value of x is: 6. But if we place a **semicolon** at the end of the line containing x + 1, changing it from an **expression** to a **statement**, we'll get an **error**.

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

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

fn plus_one(x: i32) -> i32 {
    x + 1 // do not put a semicolon “;” here!
}
```

Functions - Exercise!

Implement a simple calculator.

Structs

A **struct**, or **structure**, is a custom data type that lets you **package together** and name **multiple related values** that make up a **meaningful group**.

If you’re familiar with an **object-oriented language**, a struct is like an object’s data **attributes**.

Structs - Defining and Instantiating

To define a struct, we enter the keyword `struct` and name the entire struct.

A struct’s name should describe the significance of the pieces of data being grouped together. Then, **inside curly brackets**, we define the names and types of the pieces of data, which we call **fields**.

```
struct User {

    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

Structs - Defining and Instantiating

To use a struct after we've defined it, we create an instance of that struct by specifying concrete values for each of the fields.

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

`String::from` is a method in Rust's standard library that creates a new `String` instance from a given string literal or a `&str` reference. It is a convenience function that wraps the `String` type's constructor, making it easier to create a `String` from a string-like value.

Structs - Defining and Instantiating

To get a specific value from a struct, we use dot notation. For example, to access this user’s email address, we use **user1.email**.

Note that the entire instance must be mutable; Rust doesn’t allow us to mark only certain fields as mutable.

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

    user1.email = String::from("anotheremail@example.com");
}
```

Structs - Defining and Instantiating

Using functions:

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```

Structs - Shorthand

Because the function parameter names and the struct field names are exactly the same, we can use the field init shorthand syntax to rewrite build_user so it behaves exactly the same but doesn’t have the repetition of username and email.

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

Structs - Update

It’s often useful to create a new instance of a struct that includes most of the values from another instance, but changes some. You can do this using *struct update syntax*.

```rust
fn main() {
    // --snip--

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
```

Structs - Update

The syntax .. specifies that the remaining fields not explicitly set should have the same value as the fields in the given instance.

```rust
fn main() {
    // --snip--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

Structs - Tuple Structs

Rust also supports structs that look similar to tuples, called tuple structs. Tuple structs have the added meaning the struct name provides but don’t have names associated with their fields; rather, they just have the types of the fields

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

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Structs - Unit-Like

You can also define structs that don’t have any fields!

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Struct - Example

To understand when we might want to use structs, let’s write a program that calculates the **area of a rectangle**.

We’ll start by using single variables, and then refactor the program until we’re using structs instead.

Let’s make a new binary project with Cargo called **rectangles** that will take the **width** and **height** of a rectangle specified in pixels and calculate the area of the rectangle.

Struct - Example

```rust
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Struct - Example

```rust
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

Struct - Example

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

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

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

Struct - Example

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

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

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

Struct - Example

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

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

Obrigado!

Perguntas?