



# RUST

## ESSENTIALS

This is the most comprehensive guide to the Rust online. I'll programmatically introduce you to Rust in this Rust Essential.

**Masteringbackend.com**



### **What is Rust?**

Rust is a modern systems programming language created in 2006 and published in 2015 by Graydon Hoare, a software developer at Mozilla Research, as a personal project. It is a systems programming language focusing on performance, memory safety, and safe concurrency. Developers largely see it as a replacement for the mature C and C++, languages as old as they are powerful.

### **Key features of Rust**

Rust is a modern systems programming language focusing on safety, performance, and concurrency. It was designed to address common programming pitfalls and challenges, particularly in systems-level programming. Here are some key features that make Rust stand out:

1. **Memory Safety without Garbage Collection:** Rust introduces a unique ownership system that enforces strict rules about accessing and managing memory. This prevents common bugs like null pointer dereferences, dangling pointers, and data races without needing a garbage collector.
2. **Ownership, Borrowing, and Lifetimes:** Rust's ownership system ensures that memory is managed efficiently and safely. Variables have ownership of their data, and the concept of borrowing allows functions to temporarily use data without taking ownership. Lifetimes ensure that references remain valid and prevent dangling references.
3. **Zero-Cost Abstractions:** Rust allows high-level programming constructs and abstractions without sacrificing performance. Its "zero-cost abstractions" philosophy means that the abstractions used at the high level compile into efficient machine code with minimal overhead.
4. **Concurrency and Parallelism:** Rust's ownership and borrowing model also enables safe concurrency and parallelism. The `Send` and `Sync` traits ensure that data can be safely transferred between threads and shared between threads. The `async/await` syntax simplifies asynchronous programming.
5. **Trait System and Pattern Matching:** Rust's trait system provides a powerful mechanism for defining shared behavior across different types. Pattern matching allows for concise and readable code when working with complex data structures.

6. **Pattern Matching:** Rust's pattern matching is expressive and powerful, enabling concise code for complex data manipulation and control flow. It helps in handling different cases and scenarios efficiently.
7. **Functional and Imperative Paradigms:** Rust supports both functional and imperative programming styles. It allows you to choose the paradigm that best fits your problem while enforcing safety and performance guarantees.
8. **Crates and Package Management:** Rust has a robust package management system with Cargo, simplifying project setup, dependency management, and building. Crates are Rust's version of libraries, which can be easily shared and distributed.
9. **No Null or Undefined Behavior:** Rust eliminates null pointer dereferences and undefined behavior by enforcing strict rules for memory access. The `Option` and `Result` types are used to safely handle the absence of values and errors.
10. **Cross-Platform Compatibility:** Rust's focus on portability and well-defined behavior makes it suitable for building cross-platform applications. It supports various operating systems and architectures.
11. **Ecosystem and Community:** Rust has a rapidly growing and supportive community. The Rust ecosystem includes various libraries and tools covering various domains, from web development to embedded systems.

12. **Compiled Language:** Rust is a compiled language, resulting in efficient and performant binaries. Its compile-time checks catch many errors before runtime, reducing the likelihood of bugs in production code.

These features collectively make Rust a strong choice for systems programming, embedded development, web services, and other applications where safety, performance, and concurrency are crucial.

## Setting up Rust development environment

The Rust development environment is pretty easy to set up. The best way to install Rust is via Rust's toolchain manager called `rustup`, available at <https://rustup.rs>. They laid down instructions on installing Rust there for your operating system.

**Note:** The Rust community has put up a set of [teams](#) and standards for managing Rust. One such standard is that all Rust-related websites use the `.rs` domain extension.

The installation you just did came with the following tools:

- `cargo` : The Rust package manager.
- `rustc`: The Rust compiler is sometimes invoked by cargo since your code is also treated as a package.
- `rustdoc` : The Rust documentation tool for documenting comments in Rust code.

To check the version of these tools installed, run each of them with the `--version` flag:

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

```
> rustc --version && cargo --version && rustdoc --version
rustc 1.71.1 (eb26296b5 2023-08-03)
cargo 1.71.1 (7f1d04c00 2023-07-29)
rustdoc 1.71.1 (eb26296b5 2023-08-03)
```

Let's get familiar with each of these, starting with `rustc`.

## Project Setup

Create a new project directory and create a file in it called `hello.rs`. In it, input this simple Hello World script:

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

Now, run this command from the project directory:

```
rustc hello.rs
```

If you use `ls` command to check the content of the project directory, you should see a new file named `hello`. Run it like a bash script:

```
./hello
# Output is:
Hello World!
```

Great! You just used the Rust compiler to compile a Rust program into a binary file.

```
~/workspace/rustacean
> mkdir mb

~/workspace/rustacean
> z mb

workspace/rustacean/mb
> touch hello.rs

workspace/rustacean/mb via v1.71.1
> vi hello.rs

workspace/rustacean/mb via v1.71.1 took 9s
> bat hello.rs
```

	File: hello.rs
1	fn main() {
2	println!("Hello World!");
3	}

```
workspace/rustacean/mb via v1.71.1
> rustc hello.rs

workspace/rustacean/mb via v1.71.1
> ls
hello          hello.rs

workspace/rustacean/mb via v1.71.1
> ./hello
Hello World!
```

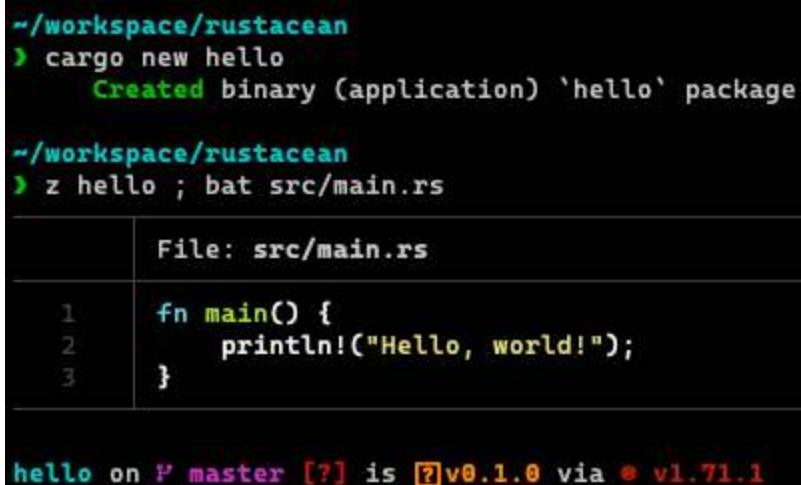
**Note:** The Rust compiler is allegedly one of the smartest and most helpful compilers in the history of programming. The Rust community has reason to believe that if your code compiles without any errors, it is memory safe and almost bug-free.

Next, let's see what `cargo` is. Delete this project folder you just created and run the following command:

```
cargo new hello
```

Next, change the directory into the `src/` subdirectory and open the `main.rs` file. You should see the content:

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



```
~/workspace/rustacean  
> cargo new hello  
    Created binary (application) `hello` package  
  
~/workspace/rustacean  
> z hello ; bat src/main.rs
```

	File: src/main.rs
1	fn main() {
2	println!("Hello, world!");
3	}

```
hello on P master [?] is [?]v0.1.0 via ● v1.71.1
```

Great. You just used the Rust compilation manager to initialize a new binary project. Cargo is a very helpful tool that allows you to initialize, build, run, test, and release Rust projects and leverage existing Rust libraries in your project. There is [an entire book](#) to learn about Cargo, and its possibilities for your perusal.



**Note:** One of the greatest strengths promoting the adoption and evolution of the Rust programming language is the immersive documentation they provide for everything Rust-related. The libraries (called crates) available to Rust are available at [Crates.io](https://crates.io), the documentation for each crate is available at [Docs.rs](https://docs.rs), jobs in the blockchain space are available at [Rib.rs](https://rib.rs), and more.

Since we have not learned Rust, let's pass on the `rustdoc` tool.

Next, run this command to compile and run the auto-generated program with Cargo:

```
cargo run
```

Great. You now know how to use Cargo to run a Rust program.

## IDEs and tools

When writing Rust code, you will need all the help you can get. Install the Rust-Analyzer plugin from the extensions marketplace if you use Visual Studio Code. If you are a fan of the JetBrains ecosystem of IDEs, you should install the Rust plugin for your code and turn on the Cargo Check feature at the bottom of the window.

## Hello, World!

We have already seen how to write "Hello, World" in Rust. In this section, we will demystify all the parts of the Rust syntax relevant to the "Hello, World" code in the previous section.

## The structure of Rust projects

The important parts of a Rust project are seen when we use Cargo to initialize a new project. Our `hello` the project has the following files:

```
.
├── Cargo.toml
├── .gitignore
├── src
│   └── main.rs
```

Let's understand what they stand for.

- `Cargo.toml`: stores metadata for the package. The contents are grouped into two sections `package` and `dependencies`. The `dependencies` section contains records of the names and versions of external crates used in the project.
- `.gitignore`: stores files that Git should not track. Git is Rust's official version control software, and the `cargo new` command initializes Git in the directory. When Rust code is built or compiled, a `target/` sub-directory is generated containing the build files. Adding it to the `.gitignore` file is good practice to avoid pushing large unnecessary files to GitHub.
- `src/main.rs`: The `src/` subdirectory is where all Rust code is normally written. The `main.rs` file is the entry point for all binary crates.

**Note:** With Cargo, you can create two types of projects (preferably called packages or crates): binary packages/crates and library packages/crates. Let's see how these are done. Run:

```
cargo new libpkg --lib
```

You should see a new `libpkg` project containing a `lib.rs` file inside the `src/` directory instead of a `main.rs` file. You'll also notice an auto-generated test code inside the `lib.rs` file. Library crates do not have/need a `main` function and do not allow the `cargo run` command. You can only use the tests to check your code by running `cargo test`. You can trick Cargo by creating a `main.rs` file inside the `src/` directory, but if you publish the project as a package, remember to delete it.

## Understanding basic syntax and structure

We have been introduced to the `main` function from the "Hello World" code above. Most programming languages like Go and C/C++ also have the `main` function as the entry-point function. But the difference is that Rust's `main` function can behave like every other function by accepting and returning values. Weird, right?

Let's go further with understanding Rust's syntax for now.

Comments for documentation are made with the double forward slash: `//`. Try this out in the `main.rs` file:

```
fn main() {  
    // greet me on the terminal  
    println!("Hello World!");  
}
```

The `fn` keyword is the Rust way of specifying that a code block is a function. All expressions and separate statements in Rust code must end with a semicolon, just like in C/C++. If you remove or skip it, the compiler panics.

Remove the semicolon after the closing brackets to print out the greeting and running `cargo run`.

Surprise! It still works. Why? The Rust compiler can infer the meaning of your code even if you omit the semicolon at the end of a statement. Smart, right?

Let's update the `main.rs` file to this:

```
fn main() {  
    let x = 5 // there is a missing semicolon here.  
    println!("{}", x) // there is also a missing semicolon here that can be omitted.  
}
```

The output should be:

```
> cargo run
Compiling hello v0.1.0 (/.../workspace/hello)
error: expected `;`, found `println`
--> src/main.rs:2:14
|
2 | let x = 5
|      ^ help: add `;` here
3 | println!("The value of x is: {}", x)
| ----- unexpected token

error: could not compile `hello` (bin "hello") due to previous error
```

The compiler tells you exactly what went wrong: `expected `;``. It further gives you a helping hand: `help: add `;` here`.

Add the semicolon at the end of line 2, and run the program. This works. But it isn't ideal. Add another semicolon at the end of line 3 too. There you go; a properly written Rust code.

The `let` keyword is a statement and the `println!` keyword (called a macro in Rust) is an expression. We will learn more about the use of the `let` keyword in the next section.

If you come from a C/C++ background, you can confirm that the `main` function in Rust receives arguments and return statements by updating the `main.rs` file and running it:

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

The `->` symbol is how you specify return values. This is a more robust way to write the `main` function in Rust. [Rust Essentials - Mastering Backend](#)

We have seen how variables are created in Rust using the `let` keyword. But what kind of variables can we create? Let's explore them. Remember to run `cargo run` every time you update the code.

## Variables

Rust supports the mutability and immutability of variables. If this sounds alien to you, it simply means that with Rust, you can make a variable have the ability to change its value or not change it during compile time. Let's see this more practically.

The variable `x` above is generally a number. This variable cannot be changed throughout this code, so this won't work:

```
fn main() -> () {  
    let x = 5;  
    println!("The value of x is: {}", x);  
    x = 10;  
    println!("The value of x is: {}", x);  
}
```

The only way it will work is if we reassign `x` or copy the value into another variable:

```
fn main() -> () {  
    let x = 5;  
    println!("The value of x is: {}", x);  
    let x = 10;  
    println!("The value of x is: {}", x);  
}
```

Like this, the second instance of the variable `x` is different from the first. But then, you had to write two more lines of code to achieve this change. To avoid this, we can simply use the `mut` keyword, which stands for mutable when creating `x`:

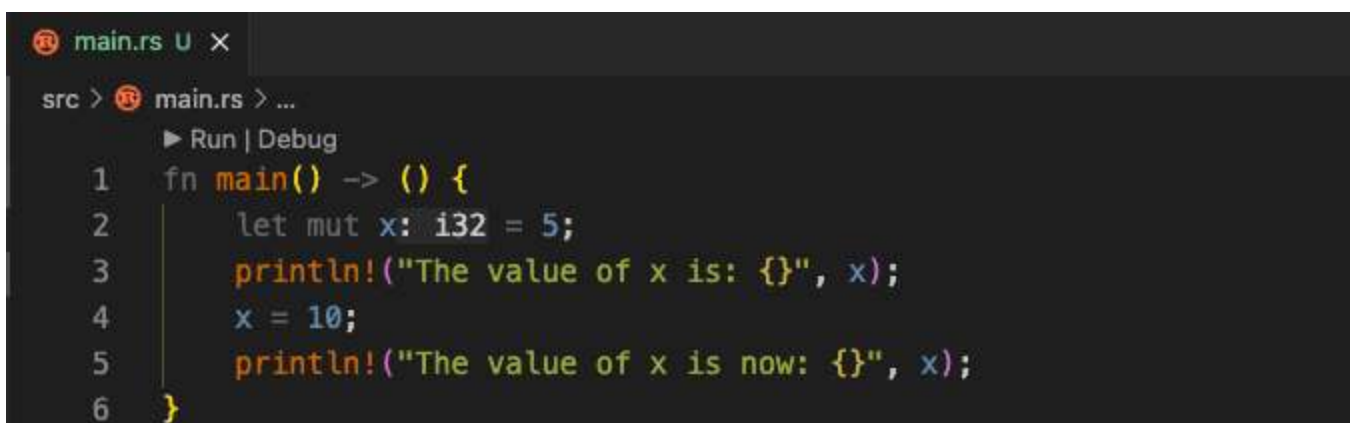
```
fn main() -> () {  
    let mut x = 5;  
    println!("The value of x is: {}", x);  
    x = 10;  
    println!("The value of x is now: {}", x);  
}
```

This works, and the compiler doesn't panic like it did before.

Now, let's move on to data types.

## Data types

**Note:** If you have configured your IDE or code editor to have the Rust plugin, you must have seen that a type inference was appended to `x`:

A screenshot of an IDE window titled 'main.rs'. The editor shows the following Rust code:

```
1 fn main() -> () {  
2     let mut x: i32 = 5;  
3     println!("The value of x is: {}", x);  
4     x = 10;  
5     println!("The value of x is now: {}", x);  
6 }
```

The code is color-coded, and the variable `x` is explicitly typed as `i32` in line 2. The IDE interface includes a 'Run | Debug' button above the code.

This was done by the rust-analyzer and means that you can write the code as:

```
fn main() -> () {  
    let mut x: i32 = 5;  
    println!("The value of x is: {}", x);  
    x = 10;  
    println!("The value of x is now: {}", x);  
}
```

Sometimes, we will need to specify the exact type of a variable to avoid anomaly behavior in our code.

**Note:** Rust has a reference that is being released every six weeks. It is the primary reference you should run to when you need to remember something, and it is available [here](#). For example, you can find all the types available in Rust [here](#). You can click on each link to find the subtypes under them.

### Numeric types

Under **Numeric types** in the Primitive types collection, you should find **Integer types**, **Floating-point types**, and **Machine-dependent integer types**. You can also learn how types are specified. The `i32` type we used above is for integers between  $2^{32} - 1$ . You have seen how to create numeric typed variables with and without type inference. Let's learn how to do so in other types.

### Textual types

Under **Textual types**, you should find characters `char` and strings `str`. They are created like this:



```
fn main() -> () {  
    let single_alphabet = 'a';  
    println!("The single character is: {}", single_alphabet);  
  
    let second_alphabet: char = 'b';  
    println!("The single character with type inference is: {}", second_alphabet);  
}
```

When we explore strings in Rust, you will be introduced to memory management, an intermediate aspect of Rust. Nevertheless, let's explore how to create string variables in Rust:

```
fn main() -> () {  
    let my_string = "Hello World";  
    println!("The string content is: {}", my_string);  
  
    let second_string: &str = "Hello, other world";  
    println!("The string content with type inference is: {}", second_string);  
}
```

Notice the ampersand (&) we used during the type inferred variable `second_string`. This is how strings in Rust work. If you create a string without it, the compiler will panic, and this is because Rust understands that strings can have a dynamic length during compile time, so it gets stored in a heap.

The non-complicated way to create strings in Rust is to use the automatic dynamically sized `String` keyword:

```
fn main() -> () {  
    let another_string = String::from("Hello, world");  
    println!("The string content is: {}", another_string);  
  
    let new_string: String = String::from("Hey, world!");  
    println!("The type inferred string content is: {}", new_string);  
}
```

From the Rust reference, you should take your time playing with the other Rust types. Let's move on to the next section.

### Sequence types

In Rust, sequential types refer to data structures that store a sequence of values in a specific order. These types allow you to store and manipulate collections of items. Rust provides several built-in sequential types with characteristics and uses cases. Here are some examples:

1. **Arrays:** Arrays in Rust have a fixed size determined at compile time and contain elements of the same type. They are stored on the stack and are useful when you need a fixed number of elements with a known size.

```
// Declaration and initialization of an array
let numbers: [i32; 5] = [1, 2, 3, 4, 5];
```

2. **Slices:** Slices are references to a contiguous sequence of elements within another sequential type (like an array or a vector). They allow you to work with a portion of the data without copying it.

```
let numbers = [1, 2, 3, 4, 5];
let slice: &[i32] = &numbers[1..4]; // Slice containing elements 2, 3, and 4
```

3. **Vectors:** Vectors are dynamic arrays that can grow or shrink in size. They are stored on the heap and allow you to store various elements.

```
// Creating and modifying a vector
let mut vec = Vec::new();
vec.push(1);
vec.push(2);
vec.push(3)
```

4. **Strings:** Rust's `String` type is a dynamically sized, UTF-8 encoded string. It is implemented as a vector of bytes and provides a convenient way to work with text.

```
// Creating and manipulating strings
let mut text = String::from("Hello, ");
text.push_str("world!");
```

5. **Ranges:** Ranges are sequential types representing a sequence of values from a start value to an end value. They are often used in loops to iterate over a range of numbers.

```
for number in 1..=5 { // Inclusive range (1 to 5 inclusive)
    println!("Current number: {}", number);
}
```

### 6. **Tuples:**

Tuples are collections of values of different types, and they have a fixed size that's determined at compile time. Each element of a tuple can have its type.

```
// Creating tuples
let person: (String, i32, bool) = ("Alice".to_string(), 30, true);
// Accessing tuple elements
let name = person.0; // Access the first element (name)
let age = person.1; // Access the second element (age)
let is_adult = person.2; // Access the third element (is_adult)
```

These sequential types in Rust provide various options for storing and manipulating data collections. Depending on your use case and requirements, you can choose the appropriate type to effectively manage your data and perform operations efficiently.

## Functions

You were first introduced to the function, arguably the most important part of a program. We have seen the `main` function, so let's learn how to create other functions.

Here are examples of different types of functions in Rust.

## 1. Function with No Return Statement or Argument:

```
fn greet() {  
    println!("Hello, world!");  
}
```

In this example, the `greet` function doesn't take any arguments or return any value. It simply prints out, "Hello, world!".

## Function with Argument and No Return Statement:

```
fn say_hello(name: &str) {  
    println!("Hello, {}!", name);  
}
```

Here, the `say_hello` function takes a single argument of type `&str` (a string slice) and prints a personalized greeting using that argument.

## Function with Argument and Return Statement:

```
fn square(n: i32) -> i32 {  
    n * n  
}
```

In this example, the `square` function takes an `i32` argument and returns the square of that argument as an `i32`.

## Function with Multiple Arguments and Return Statements:

```
fn calculate_power(base: f64, exponent: i32) -> f64 {
    if exponent == 0 {
        1.0
    } else {
        let mut result = base;
        for _ in 1..exponent.abs() {
            result *= base;
        }
        if exponent < 0 {
            1.0 / result
        } else {
            result
        }
    }
}
```

In this example, the `calculate_power` function takes a `base` of type `f64` and an `exponent` of type `i32`. It calculates the base's power raised to the exponent, considering both positive and negative exponents.

You can call each of these in the main function and run the program with cargo:

```
fn main() {
fn main() {
    greet();
    say_hello("John");
    println!("{}", square(5));
    println!("{}", calculate_power(10.0, 10));
}
/* Output is:
» cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.00s
   Running `target/debug/hello`
Hello, world
Hello, John
25
10000000000
*/
```

**Note:** The `/* ... */` is how to make longer, multi-line comments in Rust code. Also, the placeholders ( `{ }` ) must be used to print out a message to the console. In Rust, the practice of printing out messages is often called debugging. Sometimes, the Rust compiler will not allow you to use the `println` macro for some operations because they do not support debugging.

## Modules

You will be familiar with modules if you know software engineering principles like hexagonal architecture. Generally, breaking down functions into different modules in a growing code base is best. We can do this in Rust by leveraging the `mod.rs` file, a specially recognized and reserved file for Rust projects. Inside the `src/` directory, you can also have other subdirectories called modules. Using them in the project must be linked to the `main.rs` or `lib.rs` file.

In Rust, modules organize code into separate, named units, allowing you to group related functions, types, and other items. This helps improve code organization, maintainability, and readability by breaking your program into smaller, manageable pieces.

Here's how you can work with modules in Rust. Inside the `src/` directory, create a new directory called `database`. In it, create a `mod.rs` file and a `model.rs` file. The `mod.rs` file gives visibility to other files in any subdirectory under the `src/` directory. To practically understand this, add the following code to the specified files:

**mod.rs** :

```
pub mod model;
```

**model.rs** :

```
pub fn create_user(name: &str, age: i32) {  
    println!("New user created: {} of age {}", name, age);  
}
```

**main.rs** :

```
pub mod database; // make the database module available in this file.  
pub use database::*; // give this file access to all public modules and their functions (*) inside of the database module.  
  
fn main() {  
    let name = "John";  
    let age = 35;  
    database::model::create_user(name, age);  
}
```

Note that we are using a clean **main.rs** file. Run the code with cargo, and you should see:

```
> cargo run  
Compiling hello v0.1.0 (/.../workspace/hello)  
Finished dev [unoptimized + debuginfo] target(s) in 0.56s  
Running `target/debug/hello`  
New user created: John of age 35
```



What did we do? We created a subdirectory, added a `model.rs` file, and exported it with `mod.rs`. We then called the subdirectory inside the main function (as we would do in any Rust source file that needs the contents of `model.rs`), and then we used the `create_user` function in the `main` function.

**Note:** In Rust, the double semicolon notation is used to call methods of a module/package, not the dot notation like in Go. Since we already specified to use all modules public modules and their respective functions using this line `pub use database::*`, we can shorten the calling technique to this: `model::create_user(name, age)`.

We can also do this:

```
pub mod database;
pub use database::model::*;

fn main() {
    let name = "John";
    let age = 35;
    create_user(name, age);
}
```

Or this:

```
pub mod database;
pub use database::model::create_user;
// use only the create_user function from the model module in the database package.

fn main() {
    let name = "John";
    let age = 35;
    create_user(name, age);
}
```

Ensure you get the logic before proceeding to the next section.

**Note:** Rust is feature-rich. For example, we can create separate modules inside one single Rust file using the `mod` keyword. We can also write modules inside the `mod.rs` file instead of having a separate `model.rs` file.

But this way shown above is a more maintainable and readable way to do it, as is the initial goal of modular programming. If your application is small and lightweight, you can do either, but use the approach taught here if it is bound to scale/grow bigger.

When creating custom data structures (precisely called models in software engineering), structs are pretty useful. They are used to store variables that are related. For example, say we want to create a model of a User. We can do so in Rust like this:

```
struct User {  
    name: String,  
    age: i32,  
    account_balance: f64,  
}
```

This is usually done outside of a function (in global scope) if we intend to use it in more than one place. To use the struct, we must have a method that implements it. A method is a function that is tied to a struct type. Let's see how to implement methods for a type.

## Creating Method Implementations for Structs

Implementing methods for a struct type in Rust allows you to define functions that operate specifically on instances of that struct. This is a powerful feature that promotes encapsulation and enhances code organization. To illustrate, let's build upon the `User` struct example and implement some methods.

```
struct User {
    name: String,
    age: i32,
    account_balance: f64,
}

impl User {
    // A method to greet the user
    fn greet(&self) {
        println!("Hello, my name is {} and I'm {} years old.", self.name, self.age);
    }

    // A method to deposit money into the user's account
    fn deposit(&mut self, amount: f64) {
        self.account_balance += amount;
        println!("Deposited {:.2} units. New balance: {:.2}", amount, self.account_balance);
    }

    // A method to check if the user is eligible for a discount
    fn is_eligible_for_discount(&self) -> bool {
        self.age >= 60 || self.account_balance > 1000.0
    }
}
```

In the example above, we've defined three methods associated with the `User` struct. Let's break down how they work:

1. The `greet` method takes a shared reference ( `&self` ) to a `User` instance and prints a personalized greeting using the user's name and age.
2. The `deposit` method takes a mutable reference ( `&mut self` ) to a `User` instance and an `amount` parameter. It adds the specified amount to the user's `account_balance` and prints out the new balance.
3. The `is_eligible_for_discount` method checks whether the user is eligible for a discount based on age and account balance. It returns `true` if the user is 60 years or older or their account balance exceeds 1000 units.

Using these methods, you can interact with `User` instances more intuitively and structured. Here's an example of how you might use these methods:

```
fn main() {  
    let mut user1 = User {  
        name: String::from("Alice"),  
        age: 28,  
        account_balance: 750.0,  
    };  
  
    user1.greet();  
    user1.deposit(250.0);  
  
    if user1.is_eligible_for_discount() {  
        println!("Congratulations! You are eligible for a discount.");  
    } else {  
        println!("Sorry, you are not eligible for a discount.");  
    }  
}
```

In this way, methods enable you to encapsulate behavior related to a struct and promote cleaner, more readable code. Rust's strict ownership and borrowing rules help ensure that your code remains safe and free from common programming errors.

**Note:** You can split method implementations into separate modules to make code more readable and accessible.

In the world of Rust programming, one of its most distinctive features revolves around the concepts of ownership, borrowing, and lifetimes.

These concepts are fundamental in ensuring memory safety and preventing many bugs, including data races and null pointer dereferences.

This guide will delve deep into these concepts to understand how Rust achieves these goals.

## Introduction to Ownership, Borrowing, and Lifetimes

Before diving into the intricacies of ownership, let's grasp the basic concepts:

- **Ownership:** In Rust, every value has a single owner. This owner is responsible for deallocating memory when the value is no longer needed. This eliminates manual memory management, as the ownership system handles memory deallocation automatically.

```
fn main() {  
    let original = String::from("Hello");  
    let wrong_copy = original;  
  
    println!("Original: {}", original);  
    println!("Wrong copied: {}", wrong_copy);  
}
```

In the code above, we created a String variable and passed it as a value to another variable. When we try to debug (print out what is stored in) the `original` variable, we see that it has nothing in it, and its content has been transferred to the other variable `wrong_copy`. A better way to share values between two variables in a way that original still owns the value is using `.clone()`:

```
let original = String::from("Hello");  
let right_copy = original.clone();  
  
println!("Original: {}", original);  
println!("Copied: {}", right_copy);
```

This works. Try to understand this clearer before proceeding.

- **Borrowing:** When you want to use a value without taking ownership of it, you can borrow it. Borrowing allows multiple parts of your code to access data without risking memory leaks or data races.

```
fn main() {
    let message = String::from("Hello, borrowing!");

    // Borrowing with references
    let reference = &message;
    println!("Reference: {}", reference);

    // Mutable borrowing
    let mut mutable_reference = message;
    mutable_reference.push_str(" And mutation!");
    println!("Mutable Reference: {}", mutable_reference);
}
```

- **Lifetimes:** Lifetimes define how long references are valid. They ensure that borrowed references don't outlive the data they point to, preventing the dreaded "dangling references."

```
fn main() {
    loop {
        let x = 5;
    }
    println!("{}", x); // unreachable code.
}
```

In the above, `x` was given the value 5, but it was created in a loop and dropped afterward.

## Ownership Rules and the Borrowing System

Rust enforces a set of strict rules to manage ownership effectively:

1. **Ownership Transfer:** When a value is assigned to another variable or passed to a function, ownership is transferred. This ensures that each value has a single owner.
2. **Borrowing:** Multiple immutable borrows are allowed, but only one mutable borrow can exist simultaneously. This prevents data races by disallowing concurrent modification.
3. **No Null or Dangling Pointers:** Rust's type system guarantees that references are always valid, eliminating null pointer dereference and dangling reference issues.
4. **Ownership Scope:** Ownership scopes are well-defined. When a variable goes out of scope, its value is automatically deallocated. This enables Rust to prevent memory leaks.

## Avoiding Common Ownership-Related Errors

Understanding ownership and borrowing is essential to writing safe Rust code. Here are some common scenarios that lead to errors and how Rust's system helps mitigate them:



1. **Double Free:** Rust prevents double freeing of memory by ensuring only one owner can exist for a value, and ownership is automatically transferred when needed.
2. **Use After Free:** Since values are automatically deallocated when they go out of scope, Rust prevents using values after they have been deallocated.
3. **Data Races:** The borrowing system ensures that data races, which occur when multiple threads access data concurrently, are virtually eliminated.
4. **Null Pointer Dereference:** Rust's type system doesn't allow null pointers, removing the possibility of null pointer dereference bugs.

By understanding the principles of ownership, borrowing, and lifetimes, Rust developers can write code that is not only safe but also highly performant. Embracing these concepts might require a shift in mindset for programmers coming from languages without similar systems, but the benefits in terms of code quality and reliability are well worth the effort.

**Note:** Read extensively on [memory management in computer programming](#) and the heap and stack. These concepts are crucial to writing performant Rust code, as Rust pushes you closer to bare-metal computer programming than most languages. Our little example with the `&str` data type has shown us how memory management is done in Rust, but that is just the tip of the iceberg.

Control flow constructs empower programmers to steer the execution of their code. This guide delves into the core tools that Rust provides for making decisions and iterating through tasks.

## Conditional Statements: If, Else If, Else

Conditional statements are the bedrock of decision-making in any programming language. In Rust, they are both powerful and flexible:

### If Statement

The `if` statement evaluates a condition and executes a code block if the condition is true.

```
let num = 10;

if num > 0 {
    println!("Number is positive.");
}
```

### Else If Clause

The `else if` clause allows you to test multiple conditions sequentially.

```
let temperature = 25;

if temperature > 30 {
    println!("It's hot outside!");
} else if temperature > 20 {
    println!("It's warm.");
} else {
    println!("It's chilly.");
}
```

## Else Clause

The `else` clause provides a default action when none of the preceding conditions are met.

```
let is_raining = true;

if is_raining {
    println!("Don't forget an umbrella!");
} else {
    println!("Enjoy the weather!");
}
```

## Matching and Pattern Matching

Rust's `match` statement takes control flow to the next level by enabling pattern matching:

```
let grade = "B";

match grade {
    "A" => println!("Excellent!"),
    "B" | "C" => println!("Good."),
    "D" => println!("Passing."),
    _ => println!("Not a valid grade."),
}
```

In this example, the `match` statement compares the value of `grade` against various patterns and executes the corresponding code block for the first matching pattern. The `_` serves as a catch-all for patterns not explicitly listed.

## Looping with While and For

Loops allow repetitive tasks to be performed efficiently. Rust offers two fundamental loop constructs:

### While Loop

The `while` loop continues to execute a block of code as long as a given condition remains true.

```
let mut count = 0;

while count < 5 {
    println!("Count: {}", count);
    count += 1;
}
```

### For Loop

The `for` loop iterates over a range, collection, or data that implements the `Iterator` trait.

```
for number in 1..=5 {
    println!("Number: {}", number);
}
```

In this example, the `for` loop iterates through the range from 1 to 5 (inclusive) and prints each number.

In the upcoming sections, we'll delve into these control flow constructs with more detailed examples and insights, allowing you to harness their capabilities to craft efficient and expressive Rust programs.

This milestone project will provide an excellent opportunity to consolidate your learning and showcase your mastery of Rust programming. It combines concepts from each guide into a real-world application, allowing you to see how they interact and complement each other in a practical setting.

Happy coding!

### Step 1: Initialize the project

Run:

```
cargo new task_manager
```

When this is created, open the project in your code editor.

### Step 2: Create a module named `task`

Inside of the `src/` directory, create a directory named `task`, then create a `mod.rs` file and a `task.rs` file:

```
.
├── Cargo.toml
└── src/
    ├── main.rs
    └── task/
        ├── mod.rs
        └── task.rs
```

### Step 3: Add the `chrono` crate for working with time

Run:

```
cargo add chrono
```

When it is done installing the crate, check that `chrono` and its version are recorded in the `[dependencies]` section of the `Cargo.toml` file.

### Step 4: Create the Task struct

Create the `Task` struct inside of `task.rs` to handle task descriptions.

```
pub struct Task {  
    pub title: String,  
    pub description: String,  
    pub due_date: chrono::NaiveDate,  
    pub completed: bool,  
}  
  
impl Task {  
    pub fn new(title: String, description: String, due_date: chrono::NaiveDate) -> Self {  
        Task {  
            title,  
            description,  
            due_date,  
            completed: false,  
        }  
    }  
}
```

**Note:** You can one of the ways an imported crate is used. But we will also have to call it properly in the `main.rs` file, as you will soon see. Also, we are yet to explore advanced method implementations, but this is what it looks like. Remember to export the `task.rs` file inside `mod.rs`:

```
pub mod task;
```

## Step 5: Create the main program

Use the task module inside the main program:

```
pub mod task;
pub use task::task::Task;

use std::io;
use chrono::prelude::Local;

fn main() {
    println!("Welcome to Rusty Task Manager!\n");

    let tasks: Vec<Task> = Vec::new();

    loop {
        println!("Commands:");
        println!("- add <title> <description> <due_date>");
        println!("- view");
        println!("- complete <task_index>");
        println!("- filter <completed | upcoming>\n");

        let mut input = String::new();
        io::stdin().read_line(&mut input).expect("Failed to read input");

        let parts: Vec<&str> = input.trim().split_whitespace().collect();
        match parts[0] {
            // ... rest of the code remains the same
        }
    }
}
```

```
"filter" => {
    if parts.len() == 2 {
        match parts[1] {
            "completed" => {
                for (index, task) in tasks.iter().enumerate().filter(|t| t.1.completed) {
                    println!("{}", [Complete], index + 1, task.title);
                }
            }
            "upcoming" => {
                let today = Local::today().naive_local();
                for (index, task) in tasks.iter().enumerate().filter(|t| t.1.due_date >= today) {
                    let status = if task.completed { "[Complete]" } else { "[Incomplete]" };
                    println!("{}", [{}], (Due: {}), index + 1, task.title, status, task.due_date);
                }
            }
            _ => println!("Invalid filter option."),
        }
    } else {
        println!("Invalid input. Use: filter <completed | upcoming>");
    }
}
_ => println!("Invalid command."),
}
}
```



## Step 6: Run the project

Run:

```
cargo run
```

The project should run as expected. There you go! You have built your first real-world application in Rust.

**Note:** You can run `cargo build` instead to make Cargo build a binary file for the project with the same name as the project name. Then you can change the directory into `target/debug/`, where there is a `task_manager` binary executable that you can run like this:

```
./task_manager
```

```
task_manager/target/debug on P master [?]  
> ./task_manager  
Welcome to Rusty Task Manager!  
  
Commands:  
- add <title> <description> <due_date>  
- view  
- complete <task_index>  
- filter <completed | upcoming>
```

You can tweak this project to find ways to improve it. Otherwise, see you in the Intermediate Rust series!

**Want more?**

## Get Rust Intermediate

**Here's what you will learn:**

**Advanced Data Structures**

**Advanced Methods, Enums and Pattern Matching**

**Traits and Generics**

**Advanced Functions and Closures**

**Concurrency and Multithreading**

**Error Handling**

**File I/O and Serialization**

**Grab Your Copy Now**

# Want to become a great Backend Engineer?

## Join the Academy

Mastering Backend is a great resource for backend engineers.  
Next-level Backend Engineering training and Exclusive  
resources.