Rust

Version 1.80 · verified 2024-05-20

A compact guide to Rust's ownership, types, and error handling.

Contents

> **Warning:** Last verified 2 years ago — may not reflect current behavior.

Mental model

Rust prevents data races and memory bugs at compile time through ownership. Every value has exactly one owner, and you either borrow it immutably to many readers, or mutably to exactly one writer. The compiler enforces these rules via the borrow checker, replacing runtime garbage collection with predictable memory management.

Variables & mutability

Syntax	Meaning
`let x = 5;`	Immutable variable
`let mut x = 5;`	Mutable variable
`const Y: i32 = 1;`	Compile-time constant

Variables are immutable by default. You opt into mutability with `mut`. Constants require an explicit type annotation and can only be set to constant expressions, not runtime values.

```
let x = 5;
// x = 6; // Error: cannot assign twice

let mut y = 10;
```

```
y = 11; // OK


const MAX_POINTS: u32 = 100_000;
```

> **Gotcha:** Shadowing lets you redeclare a variable with `let`, changing its type or value. This is different from `mut`, which modifies the existing value in place without changing its type.

Ownership & borrowing

Syntax	Meaning
`let y = x;`	Move ownership to `y`
`let y = &x;`	Immutable borrow
`let y = &mut x;`	Mutable borrow

When a value is moved, the previous owner becomes invalid. Borrowing gives access without taking ownership. You can have any number of immutable references, or exactly one mutable reference, but never both at the same time.

```
let s1 = String::from("hello");
let s2 = s1; // s1 is moved, invalid now


let mut s3 = String::from("world");
let r1 = &s3;     // Immutable borrow
let r2 = &mut s3; // Mutable borrow
```

> **Gotcha:** Using a value after it is moved triggers `E0382`. Types that implement the `Copy` trait (like integers) are copied instead of moved, so they remain valid after assignment.

Structs & enums

Type	Use case
`struct`	Named fields together
`enum`	One of several variants
Tuple struct	Unnamed fields

Structs group related data, while enums represent a value that can be one of several possibilities. Enum variants can carry data, making them powerful for modeling state.

```
struct User {
  name: String,
  active: bool,
}

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

Pattern matching

Syntax	Meaning
`match x { ... }`	Exhaustive match
`if let ... = x`	Match a single pattern
`_`	Catch-all wildcard

Match expressions force you to handle every possible case. When you only care about one specific variant, `if let` provides a concise alternative to a full match block.

```
let some_num = Some(5);

match some_num {
  Some(x) => println!("Got {x}"),
  None => println!("Nothing"),
}

if let Some(x) = some_num {
  println!("Got {x} again");
}
```

> **Gotcha:** `match` is exhaustive. If you match on an `enum`, the compiler rejects your code if you miss a variant and don't provide a `_` fallback.

Error handling

Type	Meaning
`Option<T>`	Value or nothing (`None`)
`Result<T, E>`	Success (`Ok`) or fail (`Err`)
`?` operator	Return early on error

Rust uses explicit types for errors instead of exceptions. Use `Result` for recoverable errors and the `?` operator to propagate them up the call stack automatically.

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

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

> **Warning:** Calling `.unwrap()` or `.expect()` on a `None` or `Err` crashes the program (panics). Use them only in tests or when you are absolutely certain failure is impossible.

Traits & generics

Syntax	Meaning
`impl T for U`	Implement trait `T` on `U`
`<T>`	Generic type parameter
`T: Trait`	Trait bound

Traits define shared behavior, similar to interfaces. Generics let you write code for multiple types, and trait bounds restrict those generics to types that implement specific behavior.

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

struct Article {
  title: String,
}

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

Lifetimes

Syntax	Meaning
`&'a T`	Reference with lifetime `'a`
`<'a>`	Generic lifetime parameter
`'static`	Lives for program duration

Lifetimes ensure that references are always valid. The compiler infers them in simple cases. When returning a reference, you must annotate lifetimes to tie the output's validity to the input's.

```
fn longest<'a>(
  x: &'a str,
  y: &'a str,
) -> &'a str {
  if x.len() > y.len() { x } else { y }
}
```

Further reading

- The Rust Programming Language
- Rust By Example