Zero to Beginner Rust

A practical first course in Rust programming

Muhammed Doğukan Cengiz

Released under the MIT License | 2026

Contents

1. Introduction to Rust 3

2. Installation and Your First Program 4

3. Variables and Data Types 6

4. Functions and Control Flow 9

5. Ownership: The Heart of Rust 12

6. Strings and Collections 17

7. Structs and Enums 19

8. Error Handling Basics 22

9. Modules and Crates 24

10. Mini Project: A Simple Calculator 26

11. Next Steps 29

License

MIT License

Copyright (c) 2026 Muhammed Doğukan Cengiz

Permission is hereby granted, free of charge, to any person obtaining a copy of this book and associated files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

This is an independent educational work. It is not affiliated with or endorsed by the Rust Project or the Rust Foundation. The code examples in this book are covered by the same license.

CHAPTER 1

Introduction to Rust

What Is Rust and Why Is It Popular?

Rust is a general purpose programming language designed for performance, reliability and productivity. It began in 2006 as a personal project of Graydon Hoare, Mozilla started sponsoring it in 2009, and version 1.0 was released on 15 May 2015. Since 2021 the independent Rust Foundation supports the project, while the language itself is shaped in the open by volunteer and company teams.

Two habits of the project make Rust pleasant to live with. A new stable release appears every six weeks, and code that compiled on an older stable release keeps compiling. Larger changes arrive through *editions* (2015, 2018, 2021 and 2024), which you opt into per project, so the language can improve without breaking old code. Developers have rated Rust as the language they most want to keep using in the Stack Overflow Developer Survey for many years in a row.

Rust is popular because it removes a trade-off that programmers used to accept. Languages like C and C++ are fast but let you make dangerous memory mistakes. Languages with a garbage collector are safer but pay for it in runtime cost and unpredictability. Rust aims for both safety and speed.

The Advantages of Rust

Safety

Rust checks memory usage at compile time with a system called *ownership* (the subject of Chapter 5). In safe Rust there are no null pointers, no dangling references, no use after free and no unchecked array access. This matters in practice: Microsoft and the Chromium project have each reported that roughly 70% of their serious security bugs were memory safety problems, which is exactly the class of bug Rust is built to prevent.

Speed

Rust compiles to native machine code through the LLVM toolchain and has no garbage collector and no heavy runtime. Its abstractions, such as iterators and generics, are designed to cost nothing at run time compared with writing the same logic by hand. Performance is in the same range as C and C++.

Concurrency

The same ownership rules that protect memory also prevent *data races*, where two threads touch the same data at once and at least one changes it. In safe Rust a program with a data race does not compile. The community calls this fearless concurrency.

Rust is not free of costs. The learning curve is steeper than in many languages, and the compiler is strict. You will argue with it in the beginning. The good news is that its error messages are unusually helpful, and most beginners find that after a few weeks the arguments become conversations.

Who Uses Rust?

Rust has moved well beyond hobby projects. Mozilla built parts of Firefox's rendering engine with it. Amazon Web Services created Firecracker, the micro virtual machine technology behind Lambda and Fargate, in Rust. Cloudflare wrote its Pingora proxy framework in Rust, and Discord and Dropbox have rewritten performance critical services in it. The Linux kernel gained support for Rust code in version 6.1, Google uses it for new low level code in Android, and Microsoft has been introducing it into parts of Windows. Everyday command line tools such as ripgrep, fd and bat are written in Rust, as are the Deno JavaScript runtime and many WebAssembly projects.

How to Read This Book

The chapters build on each other, so read them in order. Chapter 5 on ownership is the heart of the book, and it is fine to read it twice. Type every example yourself instead of pasting it, because your fingers will notice details your eyes skip. When the compiler complains, read the message slowly. It usually tells you what is wrong and often suggests the fix.

Each chapter ends with a **Summary** box and a few exercises. Boxes marked **Warning** point out places where beginners usually get stuck. Every program in this book was compiled and run before it was included, and the code needs nothing newer than Rust 1.75. Examples that are meant to fail are clearly labelled, together with the error the compiler reports.

Before You Install

You need a computer running Windows, macOS or Linux, an internet connection, and a plain text editor. Visual Studio Code with the rust-analyzer extension is a popular choice because it shows errors and types as you type, but any editor works. You should be comfortable opening a terminal and using three commands: one to change directory (`cd`), one to list files (`ls` on macOS and Linux, `dir` on Windows), and one to create a folder (`mkdir`). Prior programming experience helps, though the book explains every concept it uses. No knowledge of C or C++ is needed.

> **Summary**
>
> - Rust combines the speed of C and C++ with memory safety checked at compile time and no garbage collector.
> - Ownership is the mechanism behind both memory safety and safe concurrency.
> - Rust releases every six weeks, and editions let the language evolve without breaking old code.
> - The compiler is strict but helpful. Read its messages carefully.

Exercises

1. Install a code editor and, if you choose VS Code, find the rust-analyzer extension in its marketplace.
2. Open a terminal and practice `cd`, listing files and creating a folder called `rust_projects`.
3. Write down one program you would like to build someday. Keep it for Chapter 11.

CHAPTER 2

Installation and Your First Program

Installing Rust with rustup

The official way to install Rust is `rustup`, a tool that installs the compiler, manages versions and keeps everything up to date. On Linux and macOS, open a terminal and run the command shown on rustup.rs:

Linux and macOS

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

On Windows, download and run `rustup-init.exe` from the same site. Rust on Windows needs the Microsoft C++ build tools, and the installer will guide you if they are missing. Rust also needs a linker: macOS users can get one with `xcode-select --install`, and Linux users should install a C compiler package such as `build-essential` on Ubuntu or Debian. Accept the default options, then open a new terminal and check the installation:

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

Both commands should print a version number. Later, `rustup update` installs the newest stable release, and `rustup doc --open` opens a full copy of the documentation in your browser, which works offline.

What Is cargo and Why Use It?

`rustc` is the compiler, but you will rarely call it directly. **Cargo** is Rust's build system and package manager, and it comes with rustup. It creates projects, compiles them, downloads and builds dependencies, runs tests and generates documentation. Because every Rust project follows the same layout, you can open any project and immediately know where things are.

Creating a Project

Move to your projects folder and let Cargo create a new binary project:

```
cargo new hello_rust
cd hello_rust
```

Cargo creates a folder with a `Cargo.toml` file, a `src` folder containing `main.rs`, and a Git repository with a `.gitignore` file. The manifest `Cargo.toml` describes the project:

`Cargo.toml`

```
[package]
name = "hello_rust"
version = "0.1.0"
edition = "2024"

[dependencies]
```

The `edition` line selects the language edition. Cargo versions before 1.85 write 2021 here, and both work for everything in this book. The empty `[dependencies]` section is where external crates will be listed in Chapter 9.

Understanding main.rs

Open `src/main.rs`. Cargo has already written a complete program for you:

`src/main.rs`

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

`fn` declares a function, and `main` is the special function where every executable program starts. The empty parentheses mean it takes no parameters, and the curly braces hold its body. Inside, `println!` prints a line of text. The exclamation mark tells you it is a *macro* and not an ordinary function, something you will learn more about later. Statements end with a semicolon, and Rust style uses four spaces for indentation.

cargo run and cargo build

From inside the project folder, `cargo run` compiles the program and starts it in one step:

```
$ cargo run
   Compiling hello_rust v0.1.0 (/home/you/hello_rust)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/hello_rust`
Hello, world!
```

The exact wording of Cargo's messages differs a little between versions. Two related commands are worth knowing. `cargo build` compiles without running and places the program in `target/debug`. Adding the `--release` flag produces an optimized build in `target/release`, which takes longer to compile but runs much faster, so use it when you measure performance or ship a program. `cargo check` only verifies that the code compiles, without producing a program, and is the quickest way to get feedback while writing.

Your Own Hello World

Replace the contents of `main.rs` with the following version. Curly braces inside the text can hold a variable name, and Rust inserts its value:

src/main.rs

```
fn main() {
    let name = "Ferris";
    println!("Hello, {name}! Welcome to Rust.");
}
```

```
Hello, Ferris! Welcome to Rust.
```

> **Summary**
>
> - `rustup` installs and updates Rust, and `cargo` is the tool you will use every day.
> - `cargo new` creates a project, `cargo run` builds and runs it, `cargo check` is the fast syntax and type check.
> - Debug builds compile quickly, and release builds (`--release`) run quickly.
> - `fn main()` is where a program starts, and `println!` prints a line.

Exercises

1. Create a new project called `greeter` and make it print your name and your favourite programming language on two lines.
2. Delete one closing brace in `main.rs` and run `cargo check`. Read the error message, then repair the code.
3. Run `cargo build --release` and find the executable inside `target/release`.

CHAPTER 3

Variables and Data Types

Variables, Immutability and mut

You declare a variable with `let`. By default, variables in Rust are **immutable**: once a value is bound to a name, you cannot change it. This default is deliberate. When you read code and see a variable without `mut`, you know its value never changes, which makes programs easier to reason about. When you do need change, you add `mut`.

```
fn main() {
    let x = 5;
    println!("x is {x}");

    let mut score = 10;
    score += 5;
    println!("score is {score}");

    // Shadowing: a brand new variable with the same name
    let spaces = "   ";
    let spaces = spaces.len();
    println!("number of spaces: {spaces}");
}
```

```
x is 5
score is 15
number of spaces: 3
```

The last part of the program shows *shadowing*. Declaring a new variable with `let` and the same name creates a fresh variable that hides the old one. Unlike `mut`, shadowing can change the type, which is why the text `"   "` could become the number 3. Use `mut` when a value genuinely changes over time, and shadowing when you transform a value into something new.

> **Warning**
> Assigning to a variable that is not marked `mut` is a compile error (E0384, "cannot assign twice to immutable variable"). The fix is almost always to write `let mut`.

Constants and Statics

A `const` is a value that is inlined wherever it is used. It must have a type annotation, must be computable at compile time, and by convention is written in capital letters with underscores. A `static` is similar but represents a single fixed location in memory for the whole run of the program. Choose `const` by default, and reach for `static` when you need a real memory address, for example for a large table. Modifying a mutable `static` requires `unsafe` code, so beginners should avoid it.

```
const SECONDS_IN_HOUR: u32 = 60 * 60;
static GREETING: &str = "Hello from a static";

fn main() {
    println!("{SECONDS_IN_HOUR} seconds in an hour");
    println!("{GREETING}");
}
```

Scalar Types

A *scalar* type holds a single value. Rust has four kinds: integers, floating point numbers, booleans and characters. Integer types are named by sign and size, and the `isize` and `usize` types match the pointer size of your machine and are used for indexing and lengths.

Size	Signed	Unsigned	Range of the signed type
8 bits	`i8`	`u8`	-128 to 127
16 bits	`i16`	`u16`	-32,768 to 32,767
32 bits	`i32`	`u32`	about -2.1 billion to 2.1 billion
64 bits	`i64`	`u64`	about -9.2 quintillion to 9.2 quintillion
128 bits	`i128`	`u128`	very large
pointer size	`isize`	`usize`	depends on the platform

The default integer type is `i32` and the default float type is `f64`. There are two floating point types, `f32` and `f64`. A `bool` is `true` or `false`. A `char` is written with single quotes, takes four bytes and holds any Unicode scalar value, so letters, accented characters and emoji all fit.

```
fn main() {
    let a: i32 = -42;
    let b: u32 = 42;
    let pi: f64 = 3.14159;
    let ready: bool = true;
    let letter: char = 'R';
    let heart = '❤';

    println!("{a} {b} {pi} {ready} {letter} {heart}");

    // Integer literals can be written in several ways
    let big = 1_000_000;
    let hex = 0xff;
    let bin = 0b1010;
    println!("{big} {hex} {bin}");

    // Inferred types
    let guess = "42".parse::<i32>().unwrap();
    let total = guess + 8;
    println!("total: {total}");
}
```

```
-42 42 3.14159 true R ❤
1000000 255 10
total: 50
```

> **Warning**
> Integers can overflow. In a debug build, `255u8 + 1` makes the program panic. In a release build the result silently wraps around to 0. When overflow is possible, use methods like `checked_add` or `wrapping_add` that state your intention.

Compound Types: Tuples and Arrays

A **tuple** groups a fixed number of values that may have different types. You read its fields by position (`person.0`) or take it apart with destructuring. An **array** holds a fixed number of values of the same type, and its length is part of its type: `[i32; 5]` is five integers. The syntax `[0; 5]` creates five zeros. Arrays are stored on the stack and cannot grow. For a collection that can change size, Chapter 6 introduces `Vec<T>`.

```
fn main() {
    // Tuple: fixed length, mixed types
    let person: (&str, u32, f64) = ("Ada", 36, 1.68);
    let (name, age, height) = person;
    println!("{name} is {age} years old and {height} m tall");
    println!("first field: {}", person.0);

    // Array: fixed length, one type
    let days = ["Mon", "Tue", "Wed", "Thu", "Fri"];
    let zeros = [0; 5];
    println!("first day: {}, length: {}", days[0], days.len());
    println!("zeros: {:?}", zeros);
}
```

Type Inference

You rarely have to write types, because the compiler infers them from how a value is used. In the program above, `let guess = "42".parse::<i32>().unwrap();` needed a hint, because text can be parsed into many different number types. When the compiler cannot decide, it asks for an annotation, either on the variable (`let n: u8 = "7".parse().unwrap();`) or with the *turbofish* syntax `::<i32>` as shown.

Converting Between Types

Rust never converts numbers silently. Use the `as` operator for explicit conversions, and remember that it never fails: it truncates, wraps or saturates. Converting a float to an integer drops the fractional part, and converting to a smaller integer keeps only the low bits.

```
fn main() {
    let a = 3.99_f64;
    println!("{}", a as i32);        // 3, the fraction is cut off
    let b = 300_i32;
    println!("{}", b as u8);         // 44, the value wraps around
    let c = -1_i32;
    println!("{}", c as u32);        // 4294967295
    let d = 'A' as u32;
    println!("{d}");                 // 65
    // A safer alternative that reports failure
    let result = u8::try_from(300_i32);
    println!("{:?}", result.is_err());
}
```

```
3
44
4294967295
65
true
```

When a conversion might lose information, prefer `try_from`, which returns a `Result` that you can inspect. The last lines of the program use it, and Chapter 8 explains `Result` fully.

> **Summary**
>
> - `let` creates immutable variables, `let mut` creates mutable ones, and shadowing creates a new variable of the same name.
> - `const` needs a type and a compile time value. `static` has one fixed memory location.
> - Scalar types are integers (`i32` by default), floats (`f64` by default), `bool` and `char`.
> - Tuples mix types, arrays hold one type, and both have a fixed length.
> - Use `as` for explicit casts and `try_from` when a conversion may fail.

Exercises

1. Declare a `const` for the number of minutes in a day, then print how many hours that is as an `f64`.
2. Create a tuple holding a name, an age and a boolean, destructure it, and print each part.
3. Predict what `-1i8 as u8` prints, then run the code to check your answer.

CHAPTER 4

Functions and Control Flow

Defining Functions

Functions are declared with `fn`, and Rust style names them in `snake_case`. Parameters must have explicit types, and a return type follows an arrow. The body of a function is made of statements and ends with an optional expression. The difference matters: a *statement* performs an action and produces no value, while an *expression* evaluates to a value. If the last line of a function is an expression without a semicolon, that value is returned. You can also leave early with `return`.

```
fn main() {
    greet("Rustacean");
    let result = add(7, 5);
```

```
    println!("7 + 5 = {result}");
    println!("square of 9 = {}", square(9));
    println!("abs diff = {}", abs_diff(3, 10));
}

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

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

fn square(n: i32) -> i32 {
    return n * n; // explicit return also works
}

fn abs_diff(a: i32, b: i32) -> i32 {
    if a > b {
        return a - b;
    }
    b - a
}
```

> **Warning**
>
> Adding a semicolon to the last line, as in `a + b;`, turns the expression into a statement, and the function now returns `()` instead of `i32`. The compiler reports a "mismatched types" error (E0308). If you see it on a function that looks right, check for a stray semicolon.

Conditions with if

The condition of an `if` must be a `bool`. Rust does not treat numbers or strings as true or false. Because `if` is an expression, it can produce a value, as long as each branch produces the same type.

```
fn main() {
    let temperature = 22;

    if temperature < 0 {
        println!("freezing");
    } else if temperature < 20 {
        println!("cool");
    } else {
        println!("warm");
    }

    // if is an expression
    let label = if temperature % 2 == 0 { "even" } else { "odd" };
    println!("temperature is {label}");
}
```

A First Look at match

`match` compares a value against a list of patterns and runs the code of the first one that fits. Patterns can be literals, ranges (`2..=5`), alternatives (`6 | 7 | 8`), conditions called guards (`x if x > 50`) and the wildcard `_`. The compiler insists that every possible value is covered, so you can never forget a case. Chapter 7 shows how powerful this becomes with enums.

```
fn main() {
    for n in [1, 3, 7, 12, 100] {
        let kind = match n {
            1 => "one",
            2..=5 => "a few",
            6 | 7 | 8 => "six to eight",
            x if x > 50 => "huge",
            _ => "something else",
        };
        println!("{n} is {kind}");
    }
}
```

```
1 is one
3 is a few
7 is six to eight
12 is something else
100 is huge
```

Loops

Rust has three loops. `loop` repeats forever until you `break`, and it can hand a value back through `break value`. `while` repeats as long as a condition holds. `for` walks through anything iterable, such as a range or an array, and is the loop you will use most because it cannot run past the end of a collection. The range `1..=10` includes 10, while `1..10` stops at 9. `break` leaves a loop and `continue` jumps to its next round.

```
fn main() {
    // loop with a value returned through break
    let mut counter = 0;
    let found = loop {
        counter += 1;
        if counter * counter > 50 {
            break counter;
        }
    };
    println!("first n with n*n > 50: {found}");

    // while
    let mut countdown = 3;
    while countdown > 0 {
        println!("{countdown}...");
        countdown -= 1;
    }
    println!("liftoff!");

    // for with a range and continue
    for i in 1..=10 {
        if i % 2 == 0 {
            continue;
        }
        if i > 7 {
            break;
        }
        println!("odd number: {i}");
    }

    // for over an array, in reverse
    for day in ["Mon", "Tue", "Wed"].iter().rev() {
        println!("{day}");
    }
}
```

```
first n with n*n > 50: 8
3...
2...
1...
liftoff!
odd number: 1
odd number: 3
odd number: 5
odd number: 7
Wed
Tue
Mon
```

> **Summary**
> - Functions need typed parameters. The last expression without a semicolon is the return value.
> - `if` conditions must be `bool`, and `if` itself produces a value.
> - `match` handles every case, with literals, ranges, alternatives, guards and `_`.
> - `loop`, `while` and `for` cover repetition. `break` and `continue` control them.

Exercises

1. Write a function `is_leap_year(year: i32) -> bool` and test it with 1900, 2000 and 2024.
2. Print the numbers 1 to 30, writing "Fizz" for multiples of 3, "Buzz" for multiples of 5, and "FizzBuzz" for both.
3. Use `loop` and `break value` to find the first power of 2 greater than 1000.

CHAPTER 5

Ownership: The Heart of Rust

What Is Ownership and Why Does It Exist?

Every program must decide when memory is given back. C and C++ leave this to the programmer, which leads to leaks, double frees and dangling pointers. Languages such as Java, Go and Python use a garbage collector that frees memory at unpredictable moments. Rust takes a third path: a set of ownership rules, checked at compile time, that determine exactly when each value is freed. The rules cost nothing at run time. If a program breaks them, it does not compile.

The Stack and the Heap

A program uses two regions of memory. The **stack** stores values of a size known at compile time and works last in, first out: pushing and popping are extremely fast. Integers, floats, booleans and fixed size arrays live here. The **heap** is for data whose size is unknown or can change. Storing something on the heap means asking the allocator for space and receiving a pointer, which is slower to allocate and to access. A `String` is the classic example. Its text is kept on the heap, while the stack holds a small handle of three values: a pointer, a length and a capacity.

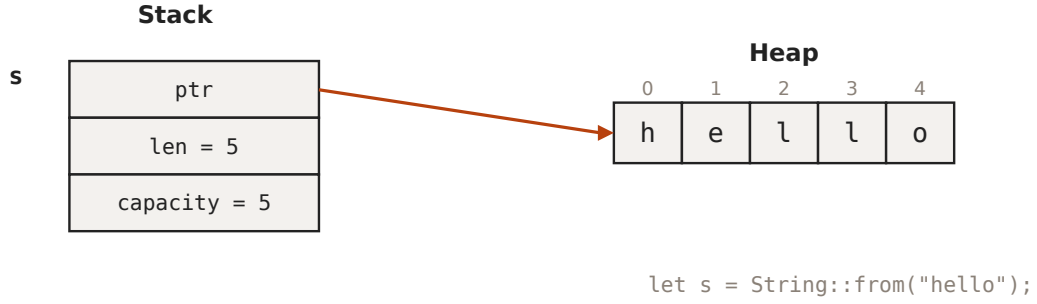


Figure 1: a String on the stack pointing to its text on the heap

The Three Ownership Rules

Everything in this chapter follows from three rules:

- Each value in Rust has an **owner**, which is a variable.
- There can be only one owner at a time.
- When the owner goes out of scope, the value is **dropped** and its memory is freed.

```
fn main() {
    let literal = "hello";             // &str, stored in the program binary
    let mut owned = String::from(literal); // String, data lives on the heap
    owned.push_str(", world");
    println!("{owned}");
}
```

When `owned` goes out of scope at the closing brace of `main`, Rust calls `drop` automatically and the heap memory is released. You never write a `free`, and you can never forget one.

Move and Copy

Assigning a heap owning value to another variable does not duplicate the heap data. It **moves** ownership. After `let s2 = s1;` the variable `s1` is no longer valid, because otherwise both variables would try to free the same memory when they leave scope.

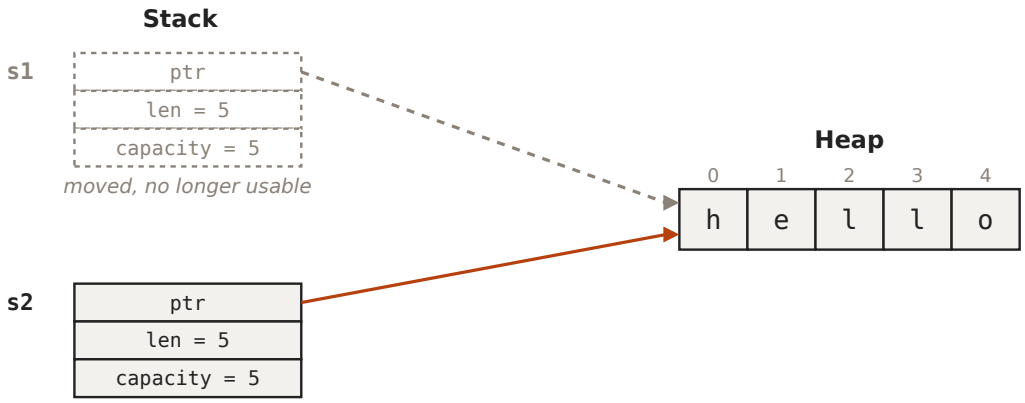


Figure 2: after a move, only s2 may use the heap data

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // ownership moves from s1 to s2
    println!("{s2}");
    // println!("{s1}"); // would not compile: s1 no longer owns the data
}
```

If you really want two independent copies, ask for one explicitly with `.clone()`. Simple types that live entirely on the stack, such as integers, floats, `bool`, `char` and tuples of these, implement the `Copy` trait and are duplicated on assignment, so the original stays usable.

```
fn main() {
    // Clone makes an explicit deep copy of heap data
    let a = String::from("hello");
    let b = a.clone();
    println!("{a} and {b}");

    // Simple stack-only types implement Copy
    let x = 5;
    let y = x;
    println!("{x} and {y}");
}
```

Passing a value to a function behaves exactly like assignment: it moves or copies. A function can give ownership back by returning the value.

```
fn take_ownership(s: String) {
    println!("got: {s}");
} // s is dropped here and the heap memory is freed

fn make_copy(n: i32) {
    println!("got: {n}");
}

fn give_back(s: String) -> String {
    s
}

fn main() {
    let text = String::from("rust");
    take_ownership(text);
    // text is no longer valid here

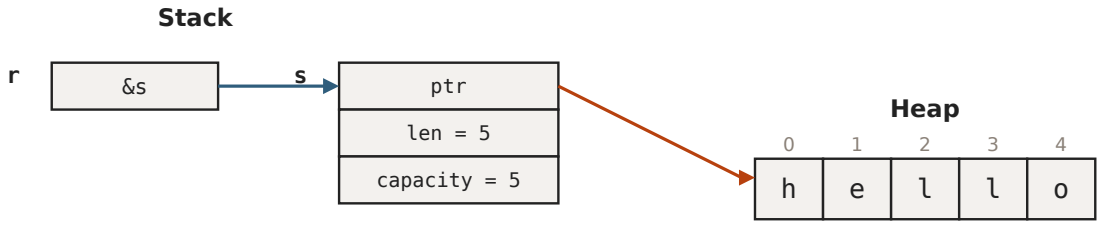
    let number = 7;
    make_copy(number);
    println!("number is still usable: {number}");

    let again = give_back(String::from("returned"));
    println!("{again}");
}
```

> **Warning**
> A value that has been moved cannot be used again. The error is E0382, "borrow of moved value". Giving ownership away and receiving it back through return values is clumsy, which is why Rust has references.

Borrowing and References

A **reference** lets you use a value without owning it. Writing `&value` creates a shared reference, and `&mut value` creates a mutable one. Creating a reference is called **borrowing**, and the owner keeps responsibility for freeing the data. Functions that only need to look at data should take a reference, so that the caller keeps its value.



r borrows s and owns nothing

Figure 3: a reference points to the owner, not directly to the heap

```
fn length(s: &String) -> usize {
    s.len()
}

fn shout(s: &mut String) {
    s.push_str("!!!");
}

fn main() {
    let mut word = String::from("rust");

    let n = length(&word); // shared borrow
    println!("{word} has {n} bytes");

    shout(&mut word); // exclusive (mutable) borrow
    println!("{word}");
}
```

The Borrowing Rules

At any moment you may have **either** one mutable reference **or** any number of shared references to the same data, and references must always point to valid data. These two rules are what prevent data races and dangling pointers. A reference stays "alive" only until its last use, a feature called non-lexical lifetimes, so borrows often end sooner than the braces suggest.

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

    let r1 = &s;
    let r2 = &s;
    println!("{r1} and {r2}"); // r1 and r2 are not used after this line

    let r3 = &mut s; // allowed: the shared borrows have ended
    r3.push_str(" world");
    println!("{r3}");
}
```

Lifetimes: A Simple Introduction

A *lifetime* describes how long a reference is valid. Most of the time the compiler works lifetimes out itself. When a function returns a reference and there is more than one input reference, the compiler cannot tell which input the result is tied to, and you describe the relationship with a lifetime parameter such as `'a`. An annotation never changes how long anything lives. It only tells the compiler how the lifetimes of inputs and output relate. Below, the result of `longest` is guaranteed to be valid for as long as both inputs are.

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

fn main() {
    let first = String::from("ownership");
    let second = String::from("borrowing!");
    let result = longest(&first, &second);
    println!("the longer word is {result}");
}
```

String literals have the special lifetime `'static`, which means they are valid for the entire run of the program. Lifetimes in structs and more advanced cases are best learned after you are comfortable with everything in this chapter.

Examples of Ownership Errors

You will meet the following errors often. Each one is shown with the message the compiler reports.

1. Using a value after it moved (E0382)

```
fn main() {
    let a = String::from("hello");
    let b = a;
    println!("{a} {b}");
}
```

Compiler: `error[E0382]: borrow of moved value: `a``. The line `let b = a;` moved the string. Fix it with `a.clone()`, or borrow with `&a`.

2. Changing a collection while holding a reference into it (E0502)

```
fn main() {
    let mut v = vec![1, 2, 3];
    let first = &v[0];
    v.push(4);
    println!("{first}");
}
```

Compiler: `error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable`. `push` may reallocate the vector, which would leave `first` pointing at freed memory. Use `first` before the push, or copy the value out with `let first = v[0];`.

3. Two mutable references at once (E0499)

```
fn main() {
    let mut s = String::from("hi");
    let a = &mut s;
    let b = &mut s;
    a.push('!');
    b.push('?');
}
```

Compiler: `error[E0499]: cannot borrow `s` as mutable more than once at a time`. Finish using the first reference before creating the second one.

4. Returning a reference to a local value (E0106)

```
fn dangle() -> &String {
    let s = String::from("temporary");
    &s
}
fn main() {
    let r = dangle();
    println!("{r}");
}
```

Compiler: `error[E0106]: missing lifetime specifier`. The string is destroyed when `dangle` ends, so a reference to it would dangle. Return the `String` itself and let the caller own it.

> **Summary**
> - Each value has one owner, and the value is dropped when the owner leaves scope.
> - Moving transfers ownership. `Copy` types are duplicated, and `.clone()` makes explicit deep copies.
> - `&T` allows many readers, `&mut T` allows one writer, and never both at once.
> - References must never outlive the data they point to. Lifetime annotations describe those relationships.

Exercises

1. Write `fn first_word(s: &str) -> &str` that returns the part of the text before the first space.
2. Write a function that takes `&mut Vec<i32>` and doubles every element. Call it and print the vector.
3. Cause errors E0382 and E0499 on purpose in a scratch project, then fix each one.

CHAPTER 6

Strings and Collections

String and &str

Rust has two main string types, and the difference is ownership. A `String` is an owned, growable, heap allocated piece of text. A `&str` (a "string slice") is a borrowed view into text stored somewhere else, for example in the program binary, which is where string literals live. Both hold UTF-8, so a character can take between one and four bytes.

	`String`	`&str`
Ownership	owns its data	borrows its data
Size can change	yes	no
Typical source	`String::from("a")`, `format!`	literals such as `"a"`, slices of a `String`
Best use	storing and building text	function parameters that only read text

Because a `&String` converts automatically to a `&str`, a function that takes `&str` accepts both kinds of text. That makes `&str` the right parameter type for most functions.

String Methods

```
fn main() {
    let mut s = String::new();
    s.push_str("  Hello");
    s.push(',');
    s.push_str(" Rust!  ");

    println!("length in bytes: {}", s.len());
    let trimmed = s.trim();
    println!("trimmed: '{trimmed}'");
    println!("replaced: {}", trimmed.replace("Rust", "World"));
    println!("upper: {}", trimmed.to_uppercase());
    println!("contains Rust? {}", trimmed.contains("Rust"));

    // Joining strings
    let first = String::from("Hello");
    let second = String::from("World");
    let joined = format!("{first}, {second}!");
    println!("{joined}");

    // Slicing uses byte positions
    let slice = &joined[0..5];
    println!("slice: {slice}");

    // Characters, not bytes
    let word = "naïve";
    println!("bytes: {}, chars: {}", word.len(), word.chars().count());
}
```

```
length in bytes: 16
trimmed: 'Hello, Rust!'
replaced: Hello, World!
upper: HELLO, RUST!
contains Rust? true
Hello, World!
slice: Hello
bytes: 6, chars: 5
```

`len` counts bytes, not characters, which the last lines demonstrate. For the same reason Rust does not allow `s[0]` on a string, and slicing with `&s[0..5]` panics if it would cut a character in half. Use `.chars()` to walk over characters.

> **Warning**
> Slicing strings by byte positions only works safely for ASCII text, or when you know the boundaries are valid. For general text, iterate with `.chars()` instead of indexing.

Vectors

`Vec<T>` is a growable list of values of one type, stored on the heap. Create one with `Vec::new()` or the `vec![]` macro. `push` adds to the end, and `pop` removes the last element and returns an `Option`, because the vector might be empty. Indexing with `v[i]` panics if `i` is out of range, while `v.get(i)` returns an `Option` and lets you handle the problem.

```
fn main() {
    let mut numbers = vec![10, 20, 30];
    numbers.push(40);
    println!("length: {}", numbers.len());

    let last = numbers.pop(); // returns an Option
    println!("popped: {:?}", last);

    // Safe access with get, panicking access with []
    match numbers.get(10) {
        Some(n) => println!("found {n}"),
        None => println!("no element at index 10"),
    }
    println!("first: {}", numbers[0]);

    // Reading
    for n in numbers.iter() {
        println!("value: {n}");
    }

    // Modifying
    for n in numbers.iter_mut() {
        *n += 1;
    }
    println!("{:?}", numbers);

    let total: i32 = numbers.iter().sum();
    println!("sum: {total}");
}
```

```
length: 4
popped: Some(40)
no element at index 10
first: 10
value: 10
value: 20
value: 30
[11, 21, 31]
sum: 63
```

HashMap

A `HashMap<K, V>` stores key and value pairs and finds a value quickly by its key. It is not imported automatically, so add `use std::collections::HashMap;`. The `entry` API is the idiomatic way to insert only if a key is missing or to update an existing value, and it makes counting very short. Iteration order is unspecified, so sort the keys when you need stable output.

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

fn main() {
    let mut scores: HashMap<String, u32> = HashMap::new();
    scores.insert(String::from("Ada"), 90);
    scores.insert(String::from("Linus"), 75);

    // get returns Option<&V>
    match scores.get("Ada") {
        Some(score) => println!("Ada scored {score}"),
        None => println!("Ada not found"),
```

```
    }

    // Overwrite or add only if missing
    scores.insert(String::from("Linus"), 80);
    scores.entry(String::from("Grace")).or_insert(95);

    // Count words with entry
    let text = "the quick brown fox jumps over the lazy dog the end";
    let mut counts: HashMap<&str, u32> = HashMap::new();
    for word in text.split_whitespace() {
        *counts.entry(word).or_insert(0) += 1;
    }
    println!("'the' appears {} times", counts["the"]);

    // Iteration order is not guaranteed, so sort for stable output
    let mut names: Vec<_> = scores.iter().collect();
    names.sort();
    for (name, score) in names {
        println!("{name}: {score}");
    }
}
```

```
Ada scored 90
'the' appears 3 times
Ada: 90
Grace: 95
Linus: 80
```

Looping over collections

A `for` loop over `&collection` borrows each element, over `&mut collection` lets you change them (through `iter_mut` as above), and over the collection itself consumes it. Choose the form that matches what you need, and the borrow rules from Chapter 5 will tell you if you chose wrongly.

> **Summary**
> - `String` owns and can grow. `&str` borrows. Prefer `&str` for read only parameters.
> - Strings are UTF-8, so `len` counts bytes and indexing by position is not allowed.
> - `Vec<T>` is the everyday growable list. `get` and `pop` return `Option` instead of panicking.
> - `HashMap` maps keys to values, and `entry(...).or_insert(...)` handles insert or update.

Exercises

1. Write a function that counts the vowels in a `&str` and returns the count.
2. Read a sentence, store each word in a `Vec<String>`, and print the words in reverse order.
3. Use a `HashMap<char, u32>` to count how often each letter appears in a word.

CHAPTER 7

Structs and Enums

Defining and Using Structs

A **struct** bundles related values under one name, and each piece has its own name called a *field*. You create an instance by giving a value for every field, access a field with a dot, and change it if the whole instance is declared `mut` (Rust has no per field mutability). If a variable has the same name as a field you can write it once, and `..other` fills the remaining fields from another instance. Note that `..user` in the example below moves the `email` string out of `user`, so `user` cannot be used as a whole afterwards. Adding `#[derive(Debug)]` lets you print a struct with `{:?}` or, in a nicer multi line layout, `{:#?}`. A *tuple struct* such as `Point(f64, f64)` has unnamed fields.

```
#[derive(Debug)]
struct User {
    username: String,
```

```
    email: String,
    active: bool,
    login_count: u32,
}

struct Point(f64, f64); // tuple struct

fn build_user(username: String, email: String) -> User {
    User {
        username, // field init shorthand
        email,
        active: true,
        login_count: 0,
    }
}

fn main() {
    let mut user = build_user(String::from("ferris"), String::from("ferris@example.com"));
    user.login_count += 1;
    println!("{} has logged in {} time(s)", user.username, user.login_count);

    // Struct update syntax
    let second = User {
        username: String::from("gopher"),
        ..user
    };
    println!("{:#?}", second);

    let origin = Point(0.0, 0.0);
    println!("origin: ({}, {})", origin.0, origin.1);
}
```

Methods with impl

An `impl` block attaches functions to a type. A function whose first parameter is `self`, `&self` or `&mut self` is a **method** and is called with dot syntax. `&self` borrows the instance to read it, `&mut self` borrows it to change it, and plain `self` consumes it. Functions without `self` are called *associated functions* and are used with `::`, as in `Rectangle::new`. `Self` is an alias for the type the block is written for.

```
#[derive(Debug)]
struct Rectangle {
    width: f64,
    height: f64,
}

impl Rectangle {
    // Associated function (no self), called with Rectangle::new
    fn new(width: f64, height: f64) -> Self {
        Self { width, height }
    }

    fn square(size: f64) -> Self {
        Self::new(size, size)
    }

    // Method that only reads
    fn area(&self) -> f64 {
        self.width * self.height
    }

    // Method that changes the value
    fn scale(&mut self, factor: f64) {
        self.width *= factor;
        self.height *= factor;
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}

fn main() {
    let mut r = Rectangle::new(4.0, 3.0);
    println!("area: {}", r.area());
    r.scale(2.0);
    println!("{:?}, area: {}", r, r.area());
```

```
    let small = Rectangle::square(2.0);
    println!("can hold small square? {}", r.can_hold(&small));
}
```

```
area: 12
Rectangle { width: 8.0, height: 6.0 }, area: 48
can hold small square? true
```

Enums

An **enum** defines a type that can be exactly one of several *variants*. Unlike enums in many languages, each variant can carry its own data, with unnamed fields (`Circle(f64)`), named fields (`Rectangle { width, height }`) or none at all. Combined with `match`, enums let you model a situation so that impossible states cannot be written down.

```
#[derive(Debug)]
enum Shape {
    Circle(f64),
    Rectangle { width: f64, height: f64 },
    Point,
}

fn area(shape: &Shape) -> f64 {
    match shape {
        Shape::Circle(r) => std::f64::consts::PI * r * r,
        Shape::Rectangle { width, height } => width * height,
        Shape::Point => 0.0,
    }
}

fn main() {
    let shapes = vec![
        Shape::Circle(1.5),
        Shape::Rectangle { width: 2.0, height: 4.0 },
        Shape::Point,
    ];
    for s in &shapes {
        println!("{:?} has area {:.2}", s, area(s));
    }
}
```

```
Circle(1.5) has area 7.07
Rectangle { width: 2.0, height: 4.0 } has area 8.00
Point has area 0.00
```

Option and Result

Rust has no `null`. A value that might be missing is an `Option<T>`, which is either `Some(value)` or `None`. An operation that might fail returns `Result<T, E>`, which is either `Ok(value)` or `Err(error)`. Both are ordinary enums from the standard library, and `match` forces you to handle both cases before you can reach the value. `if let` is a shorter form when you care about one pattern only.

```
fn find_first_even(numbers: &[i32]) -> Option<i32> {
    for &n in numbers {
        if n % 2 == 0 {
            return Some(n);
        }
    }
    None
}

fn parse_age(text: &str) -> Result<u8, std::num::ParseIntError> {
    text.trim().parse::<u8>()
}

fn main() {
    match find_first_even(&[1, 3, 8, 5]) {
        Some(n) => println!("first even: {n}"),
```

```
        None => println!("no even numbers"),
    }

    // if let is shorthand for matching a single pattern
    if let Some(n) = find_first_even(&[1, 3, 5]) {
        println!("found {n}");
    } else {
        println!("nothing found");
    }

    for input in ["42", "-5", "abc"] {
        match parse_age(input) {
            Ok(age) => println!("{input:?} is a valid age: {age}"),
            Err(e) => println!("{input:?} is not valid: {e}"),
        }
    }
}
```

```
first even: 8
nothing found
"42" is a valid age: 42
"-5" is not valid: invalid digit found in string
"abc" is not valid: invalid digit found in string
```

Summary

- Structs group named fields. `impl` blocks add methods and associated functions.
- `&self` reads, `&mut self` changes, `self` consumes, and `Self` is the type itself.
- Enum variants can carry data, and `match` must cover every variant.
- `Option<T>` replaces null and `Result<T, E>` represents operations that can fail.

Exercises

1. Create a `Circle` struct with a `radius` field and methods `area` and `circumference`.
2. Define an enum `Command` with the variants `Quit`, `Echo(String)` and `Move { x: i32, y: i32 }`, and write a function that describes each.
3. Write `fn safe_divide(a: i32, b: i32) -> Option<i32>` that returns `None` when `b` is zero.

CHAPTER 8

Error Handling Basics

Two Kinds of Errors

Rust separates errors into two groups. **Unrecoverable** errors are bugs or broken assumptions, such as indexing past the end of a vector, and the right response is to stop the program. **Recoverable** errors are expected problems, such as a missing file or text that is not a number, and the program should decide what to do. Rust has no exceptions. Instead, unrecoverable problems use `panic!` and recoverable ones use `Result`.

The panic! Macro

`panic!("message")` prints a message, unwinds the stack and ends the program with a failure exit code. Many operations panic by themselves, for instance an out of range index. Running the program below ends with a message like "index out of bounds: the len is 3 but the index is 99" and the exit code 101. Set the environment variable `RUST_BACKTRACE=1` to see the chain of calls that led to the panic.

```
fn main() {
    let numbers = vec![1, 2, 3];
    println!("about to panic...");
    let value = numbers[99]; // index out of bounds: the program panics
    println!("{value}");
}
```

Returning Errors with Result

A function that can fail returns `Result<T, E>`, and the caller chooses how to react. The first function below reports failures with a message, and the second reads a file with the `?` operator.

```
use std::fs;
use std::io;

fn read_username(path: &str) -> Result<String, io::Error> {
    let text = fs::read_to_string(path)?; // ? returns early on Err
    Ok(text.trim().to_string())
}

fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("cannot divide by zero"))
    } else {
        Ok(a / b)
    }
}

fn main() {
    match divide(10.0, 4.0) {
        Ok(v) => println!("10 / 4 = {v}"),
        Err(e) => println!("error: {e}"),
    }
    match divide(1.0, 0.0) {
        Ok(v) => println!("{v}"),
        Err(e) => println!("error: {e}"),
    }

    match read_username("does_not_exist.txt") {
        Ok(name) => println!("user: {name}"),
        Err(e) => println!("could not read file: {e}"),
    }
}
```

```
10 / 4 = 2.5
error: cannot divide by zero
could not read file: No such file or directory (os error 2)
```

The ? Operator

Writing `?` after an expression that returns a `Result` means: if it is `Ok`, give me the value inside; if it is `Err`, return that error from the current function right away. It replaces a long `match` with one character, and it can convert the error type automatically. `?` only works inside a function whose own return type is compatible, a `Result` or an `Option`. `main` may also return a `Result`, which makes `?` available there too. If `main` returns an error, the program prints it and exits with a failure code.

```
use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {
    let n: i32 = "42".parse()?;
    println!("parsed {n}");

    let m: i32 = "oops".parse()?; // fails here, main returns the error
    println!("never printed: {m}");
    Ok(())
}
```

This program prints `parsed 42`, then reports `Error: ParseIntError { kind: InvalidDigit }` on the error stream and exits with code 1.

unwrap and expect: When Are They Fine?

`unwrap()` returns the value inside an `Ok` or `Some` and panics otherwise. `expect("message")` does the same but with a message of your choice, which makes a failure far easier to track down. Both are reasonable in tests, in quick experiments, and when you can prove the failure is impossible, as with parsing a literal you wrote yourself. In real programs, handle the error, or use `unwrap_or` and related methods to supply a fallback.

```
fn main() {
    // unwrap_or gives a fallback instead of panicking
    let port: u16 = "8080".parse().unwrap_or(3000);
    let bad: u16 = "not a number".parse().unwrap_or(3000);
    println!("{port} {bad}");

    // expect: panics with your message if the value is missing
    let value: i32 = "17".parse().expect("the literal 17 must be a valid number");
    println!("{value}");

    // The ? operator also works with Option inside a function returning Option
    println!("{:?}", first_char_upper("rust"));
    println!("{:?}", first_char_upper(""));
}

fn first_char_upper(s: &str) -> Option<char> {
    let c = s.chars().next()?;
    Some(c.to_ascii_uppercase())
}
```

Summary

- `panic!` is for bugs. `Result` is for failures a caller can reasonably handle.
- `?` returns early on `Err` (or `None`) and unwraps on success.
- Prefer `expect` with a clear message over `unwrap`, and prefer real handling over both in production code.

Exercises

1. Write `fn parse_positive(text: &str) -> Result<u32, String>` that returns an error for negative or invalid input.
2. Change the file reading example so that a missing file produces the fallback text "guest" instead of an error.
3. Make `main` return `Result<(), Box<dyn std::error::Error>>` and use `?` to parse two command line values.

CHAPTER 9

Modules and Crates

Organizing Code with mod

A **module** groups related items and controls what is visible to the rest of the program. A **crate** is a compilation unit, either a binary or a library, and a **package** is what Cargo builds, holding one or more crates. Modules can be written inline with braces, as below, or moved into their own file: writing `mod geometry;` in `main.rs` tells Rust to load `src/geometry.rs` (or `src/geometry/mod.rs`).

Visibility with pub and Imports with use

Everything is private by default: code outside a module cannot see its contents unless they are marked `pub`. Struct fields have their own privacy, so a public struct still needs `pub` on each field you want to expose. To avoid writing long paths, bring items into scope with `use`. You can rename with `as`, group imports as `use std::io::{self, Write};`, and refer to the parent module with `super::`.

```
mod geometry {
    pub const PI: f64 = 3.14159;

    pub fn circle_area(radius: f64) -> f64 {
        PI * square(radius)
    }

    // Private: only usable inside this module
    fn square(x: f64) -> f64 {
        x * x
    }

    pub mod shapes {
        pub struct Square {
            pub side: f64,
        }

        impl Square {
            pub fn area(&self) -> f64 {
                self.side * self.side
            }
        }
    }
}

use geometry::shapes::Square;

fn main() {
    println!("circle: {:.2}", geometry::circle_area(2.0));
    let sq = Square { side: 3.0 };
    println!("square: {}", sq.area());
}
```

```
circle: 12.57
square: 9
```

Using External Crates

The Rust community shares libraries on crates.io, and their documentation is hosted on docs.rs. To use one, add it to the `[dependencies]` section of `Cargo.toml`, either by editing the file or by running `cargo add rand` in your project folder. The next `cargo build` downloads and compiles it.

`Cargo.toml`

```
[package]
name = "dice"
version = "0.1.0"
edition = "2024"

[dependencies]
rand = "0.9"
```

Version `"0.9"` means any compatible 0.9.x release. Cargo records the exact versions it resolved in `Cargo.lock`, which keeps your builds reproducible, and you should commit that file for programs. Here is a program that uses `rand` for random numbers:

`src/main.rs`

```
fn main() {
    let secret: u32 = rand::random_range(1..=100);
    println!("The secret number is {secret}");

    let heads = rand::random_bool(0.5);
    println!("Heads? {heads}");
}
```

Run it a few times with `cargo run` and you will see different numbers. Run `cargo doc --open` to build documentation for your project and all of its dependencies.

Understanding Cargo.toml

The manifest has a few sections you will use again and again. `[package]` holds the name, version and edition. `[dependencies]` lists the crates your code needs. `[dev-dependencies]` lists crates used only for tests and examples. A section like `[profile.release]` tunes how the optimized build is made. The format is TOML, which is plain `key = value` text arranged in sections.

> **Summary**
> - Modules organize code, and everything inside is private until marked `pub`.
> - `use` brings names into scope. `mod name;` loads a module from its own file.
> - `cargo add` or the `[dependencies]` section brings in crates from crates.io, and `Cargo.lock` pins exact versions.

Exercises

1. Create a module `temperature` with public functions `celsius_to_fahrenheit` and `fahrenheit_to_celsius`, and call them from `main`.
2. Move that module into `src/temperature.rs` and load it with `mod temperature;`.
3. Add the `rand` crate and write a program that simulates rolling two six sided dice.

CHAPTER 10

Mini Project: A Simple Calculator

Project Description and Requirements

You will build a command line calculator that reads one expression per line, such as `12 * 3.5`, prints the result, and keeps going until you type `quit`. The program should support addition, subtraction, multiplication, division and remainder, and it must never crash on bad input. Instead, it should explain what went wrong and ask again. This small program practices nearly everything from the earlier chapters: input and output, parsing, `Result`, `enum`, `match`, functions and loops.

Design

We split the work into small functions, each with one job. `read_line` handles the keyboard, `parse_number` and `parse_op` turn text into typed values, `calculate` performs the arithmetic, and `evaluate` ties these together for one line. Every step that can fail returns a `Result<_, String>`, so errors flow upward with `?` to a single place in `main`, where they are printed.

Reading Input with std::io

Reading a line from the keyboard uses `io::stdin().read_line(&mut buffer)`. It needs a mutable `String` to append to, and it returns a `Result` holding the number of bytes read. A result of `Ok(0)` means the input has ended. The buffer keeps the trailing newline, so call `trim()`. Output written with `print!` stays in a buffer until a newline appears, so call `io::stdout().flush()` after printing a prompt that does not end in one.

Parsing and Error Checking

Splitting the line with `split_whitespace()` and collecting into a `Vec<&str>` gives us the three parts. We check that there are exactly three, then convert them. `str::parse::<f64>()` returns a `Result` whose error type is not very descriptive, so `map_err` replaces it with a friendly message. Dividing by zero must be handled explicitly: a float division by zero would silently yield infinity.

The Operation Logic with match

Operators are modelled as an enum, which gives the compiler a chance to verify that `calculate` handles all of them. A match guard (`if b == 0.0`) lets one arm cover both division and remainder when the divisor is zero. Because arms are tried in order, that arm must come before the plain `Divide` and `Remainder` arms.

The Complete Program

Create a project with `cargo new calculator`, put the following code in `src/main.rs` and start it with `cargo run`. Notice how `main` itself contains no arithmetic or parsing, and how each function can be understood on its own.

src/main.rs

```
use std::io::{self, Write};

#[derive(Debug, Clone, Copy)]
enum Op {
    Add,
    Subtract,
    Multiply,
    Divide,
    Remainder,
}

fn read_line(prompt: &str) -> Option<String> {
    print!("{prompt}");
    io::stdout().flush().expect("failed to flush stdout");

    let mut input = String::new();
    match io::stdin().read_line(&mut input) {
        Ok(0) => None, // end of input (Ctrl+D / Ctrl+Z)
        Ok(_) => Some(input.trim().to_string()),
        Err(_) => None,
    }
}

fn parse_number(text: &str) -> Result<f64, String> {
    text.parse::<f64>()
        .map_err(|_| format!("'{text}' is not a valid number"))
}

fn parse_op(text: &str) -> Result<Op, String> {
    match text {
        "+" => Ok(Op::Add),
        "-" => Ok(Op::Subtract),
        "*" => Ok(Op::Multiply),
        "/" => Ok(Op::Divide),
        "%" => Ok(Op::Remainder),
        other => Err(format!("unknown operator '{other}', use + - * / %")),
    }
}

fn calculate(a: f64, op: Op, b: f64) -> Result<f64, String> {
    match op {
        Op::Add => Ok(a + b),
        Op::Subtract => Ok(a - b),
        Op::Multiply => Ok(a * b),
        Op::Divide | Op::Remainder if b == 0.0 => Err(String::from("division by zero")),
        Op::Divide => Ok(a / b),
        Op::Remainder => Ok(a % b),
    }
}

fn evaluate(line: &str) -> Result<f64, String> {
    let parts: Vec<&str> = line.split_whitespace().collect();
    if parts.len() != 3 {
        return Err(String::from("expected the form: number operator number"));
    }
    let a = parse_number(parts[0])?;
    let op = parse_op(parts[1])?;
    let b = parse_number(parts[2])?;
    calculate(a, op, b)
}
```

```
fn main() {
    println!("Simple Calculator. Type an expression like 12 * 3.5, or 'quit' to exit.");

    while let Some(line) = read_line("> ") {
        if line.is_empty() {
            continue;
        }
        if line == "quit" {
            break;
        }
        match evaluate(&line) {
            Ok(result) => println!("= {result}"),
            Err(message) => println!("Error: {message}"),
        }
    }
    println!("Goodbye!");
}
```

A sample session

```
Simple Calculator. Type an expression like 12 * 3.5, or 'quit' to exit.
> 12 + 3.5
= 15.5
> 10 / 4
= 2.5
> 5 / 0
Error: division by zero
> 7 % 3
= 1
> abc + 1
Error: 'abc' is not a valid number
> 2 ^ 3
Error: unknown operator '^', use + - * / %
> 1 2
Error: expected the form: number operator number
> quit
Goodbye!
```

How the pieces fit

- `read_line` returns `Option<String>`, so `while let Some(line) = ...` ends the loop automatically when input ends.
- `evaluate` uses `?` three times. The first failure among parsing the two numbers and the operator becomes its return value, and `calculate` provides the final `Result`.
- `Op` derives `Clone` and `Copy`, so it can be passed around like a number.
- In `main`, a single `match` on `evaluate`'s `Result` decides between printing the answer and printing the error.

> **Summary**
> - A larger program is a set of small functions, each with a clear job and a clear return type.
> - Failures travel upward as `Result` values and are reported in one place.
> - Enums plus `match` make it impossible to forget an operation.

Exercises

1. Add a `^` operator for powers using `f64::powf`, and support `quit` in uppercase.
2. Remember the last result, and let the user type `ans + 5` to use it.
3. Change `calculate` to return a custom `enum CalcError` instead of `String` and implement `std::fmt::Display` for it.

CHAPTER 11

Next Steps

Resources for Continued Learning

Resource	What it offers
The Rust Programming Language ("The Book"), doc.rust-lang.org/book	The official, free, in depth guide by Steve Klabnik and Carol Nichols. The natural next read.
Rustlings, github.com/rust-lang/rustlings	Small exercises that you fix until they compile. Install with `cargo install rustlings`, then run `rustlings init`.
Rust by Example, doc.rust-lang.org/rust-by-example	Short runnable examples for each feature.
Standard library docs, doc.rust-lang.org/std	The reference you will use daily. Also available offline through `rustup doc`.
Rust Playground, play.rust-lang.org	Run Rust in the browser with nothing installed.
crates.io and docs.rs	The package registry and its documentation host.
Exercism Rust track, exercism.org	Practice problems with optional mentor feedback.
users.rust-lang.org	The friendly official forum for questions.

A First Look at Testing

Rust has testing built in. A function marked `#[test]` is a test, and `assert_eq!` compares two values. Tests usually live next to the code in a module marked `#[cfg(test)]`, so they are compiled only when you test. Run them with `cargo test`.

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

fn main() {
    println!("2 + 3 = {}", add(2, 3));
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn adds_two_numbers() {
        assert_eq!(add(2, 3), 5);
    }

    #[test]
    fn adds_negative_numbers() {
        assert_eq!(add(-2, -3), -5);
    }
}
```

```
running 2 tests
test tests::adds_negative_numbers ... ok
test tests::adds_two_numbers ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Writing a test for `calculate` from the previous chapter is an excellent first exercise.

Topics to Explore Next

- **Traits** define shared behaviour, like interfaces in other languages, and power much of the standard library.
- **Generics** let one function or type work with many types, which is how `Vec<T>` and `Option<T>` are built.
- **Iterators and closures** give you `map`, `filter` and friends for expressive data processing.
- **Smart pointers** such as `Box`, `Rc` and `RefCell` extend the ownership model for more flexible designs.
- **Concurrency** with threads, channels and `Arc<Mutex<T>>`, where the ownership rules from Chapter 5 pay off.
- **Async Rust** for network programs that handle many connections at once.

Closing Advice

Build things. A small project that you care about, such as a to do list for the terminal, a word counter, a file renamer or a text adventure, will teach you more than another chapter of reading. Run `cargo fmt` to format your code in the standard style and `cargo clippy` to receive suggestions about idiomatic Rust. Read other people's code on GitHub, and do not be discouraged by the borrow checker. Every Rust programmer has fought with it, and nearly all of them end up thankful for it. Take one step at a time, and enjoy the feeling of a program that compiles.

> **Summary**
> - Continue with The Book, Rustlings and your own small projects.
> - `cargo test` runs the tests marked `#[test]`.
> - Traits, generics, iterators and concurrency are the natural next topics.

Exercises

1. Install Rustlings and complete its first three groups of exercises.
2. Write two tests for `evaluate` in the calculator: one for a valid expression and one for an error.
3. Choose a project idea from your list in Chapter 1 and build its smallest working version this week.

Thank You for Reading

Thank you for reading this book. I hope it made your first steps in Rust clear and enjoyable. If anything is unclear, if you found a mistake, or if you simply have a question, you can always write to me. I will be glad to hear from you.

Muhammed Doğukan Cengiz

E-mail: contact@dogukancengiz.com

Website: dogukancengiz.com