The Compact Rust Handbook

From Basics to Advanced — in one sitting

Basic · Intermediate · Advanced

Contents

1 Basic

Rust is a statically typed, compiled systems language built around *ownership* — a compile-time discipline that guarantees memory safety without a garbage collector. This part covers the everyday building blocks.

1.1 Variables and Constants

Rust bindings are **immutable by default**. You choose mutability explicitly.

- `let x = 5;` — immutable binding; cannot be reassigned.
- `let mut x = 5;` — mutable binding; can be reassigned (same type).
- `const MAX: u32 = 100;` — compile-time constant; type **required**, always immutable, `SCREAMING_CASE`, usable in any scope including module level.
- `static GREETING: &str = "hi";` — a single fixed memory location with `'static` lifetime (unlike `const`, which is inlined).
- **Shadowing**: re-declaring with `let` creates a new variable, optionally a new type.

```
fn main() {
    let x = 5;            // immutable
    // x = 6;             // ❌ compile error
    let mut y = 10;       // mutable
    y += 1;               // ✅ now 11
    const PI: f64 = 3.14; // typed constant

    let s = "42";             // &str
    let s = s.len();          // shadow: now usize, new type allowed
    println!("{x} {y} {PI} {s}");
}
```

Note. `const` vs `let`: `const` is evaluated at compile time and inlined everywhere it's used; `let` is a runtime binding. `mut` only affects `let`.

Exercises

1. Declare an immutable `let`, try to reassign it, and read the compiler error.
2. Rewrite it with `mut` so reassignment compiles.
3. Define a `const` for seconds-in-a-day and print it.
4. Shadow a `String` into its `.len()` and print both stages.

1.2 Scalar and Compound Types

Scalars: integers (`i8`..`i128`, `u8`..`u128`, `isize`/`usize`), floats (`f32`, `f64`), `bool`, `char` (4-byte Unicode). Compounds: **tuples** (fixed, mixed types) and **arrays** (fixed length, one type).

```
let int: i32 = -7;
let big: u64 = 9_000_000_000;
let pi: f64 = 3.14159;
let flag: bool = true;
let letter: char = 'λ';

let tup: (i32, f64, char) = (1, 2.0, 'z');
let (a, b, c) = tup;        // destructuring
let first = tup.0;          // index access

let arr: [i32; 3] = [1, 2, 3];
```

```rust
let zeros = [0u8; 4];       // [0, 0, 0, 0]
let len = arr.len();
```

Note. Integer overflow panics in debug builds and wraps in release. Use `wrapping_add`, `checked_add`, `saturating_add`, or `overflowing_add` to be explicit.

Exercises

1. Create a tuple of `(name, age, initial)` and destructure it.
2. Make a `[f64; 5]` array and sum it with a loop.
3. Trigger an overflow with `i8` `127 + 1` in debug mode.
4. Replace it with `checked_add` and handle the `None` case.

1.3 Creating Strings: `String::new` vs `&str`

Rust has two everyday string types, and the distinction is the same ownership story as everywhere else. A `&str` (a **string slice**) is a **borrowed**, fixed view into UTF-8 bytes that someone else owns — string literals are `&'static str`, baked into the binary, so they cannot grow. A `String` is an **owned**, heap-allocated, growable buffer that you control and can mutate.

There are several ways to make one:

- `String::new()` — a brand-new **empty** owned string (no allocation until you push).
- `String::from("hi")` — owned string built from a literal.
- `"hi".to_string()` — same result via the `ToString` trait.
- `"hi".to_owned()` — same result, emphasizing "borrow → owned".
- `String::with_capacity(n)` — empty, but pre-allocates room for `n` bytes (fewer reallocations).
- `format!("{a}-{b}")` — builds an owned string from parts.

```rust
let empty = String::new();              // owned, "", growable
let mut s = String::from("Hello");      // owned from literal
let t = "Hello".to_string();            // owned (ToString)
let u = "Hello".to_owned();             // owned (borrow -> owned)
let buf = String::with_capacity(16);    // owned, preallocated

s.push_str(", world");                  // ✅ String can grow
let view: &str = &s;                    // borrow a String as &str (deref coercion)
let literal: &str = "fixed";            // borrowed, cannot grow
// literal.push('!');                   // ❌ &str has no push

println!("{s} | {t} | {u} | {} | cap {}", empty.is_empty(), buf.capacity());
```

Note. Difference in one line: `&str` is a **borrowed, immutable view** (often a literal or a slice of a `String`); `String` is an **owned, mutable, heap** buffer. Take `&str` in function parameters (it accepts both `String` and `&str` via deref coercion); return `String` when the function must own/produce new text. `String::new()` differs from `&str` literals in that it is owned and empty rather than borrowed and fixed.

Exercises

1. Create an empty `String` with `String::new()`, push text, and print it.
2. Make the same string four ways: `String::from`, `to_string`, `to_owned`, `format!`.
3. Write `fn shout(s: &str) -> String` and call it with both a `String` and a literal.

4. Show that a `&str` literal cannot be pushed to, then convert it to `String` and push.

1.4 Functions and Expressions

Functions use `fn`. Parameters need explicit types; return type follows `->`. Rust is **expression-oriented**: a block's final expression (no semicolon) is its value.

```
fn add(a: i32, b: i32) -> i32 {
    a + b          // expression — no semicolon = return value
}

fn classify(n: i32) -> &'static str {
    if n > 0 { "positive" } else { "non-positive" }  // if is an expression
}

fn main() {
    let sum = add(2, 3);
    let block_val = { let t = sum * 2; t + 1 }; // block returns 11
    println!("{sum} {block_val} {}", classify(-4));
}
```

Exercises

1. Write `fn square(x: i32) -> i32` using an expression return.
2. Write a function returning the larger of two values.
3. Assign the result of a `{ ... }` block to a variable.
4. Make a function that returns `()` (unit) and explain why.

1.5 Control Flow: `if` / `else`

`if` is an expression, so its value can be assigned. All branches must yield the same type.

```
let n = 7;
if n % 2 == 0 {
    println!("even");
} else if n % 3 == 0 {
    println!("divisible by 3");
} else {
    println!("other");
}

// if as a value-producing expression
let parity = if n % 2 == 0 { "even" } else { "odd" };
let max = if n > 10 { n } else { 10 };
```

Note. The condition must be a `bool` — Rust does not coerce integers or `Option` to booleans.

Exercises

1. Assign `"adult"` / `"minor"` to a variable using an `if` expression.
2. Build a 3-branch `if/else if/else` grade classifier.
3. Cause a type-mismatch error by returning different types per branch.
4. Refactor a small `if/else` chain into a single value-returning expression.

1.6 Loops: `loop`, `while`, `for`

- `loop {}` — infinite; `break value` can return a value.
- `while cond {}` — runs while the condition holds.
- `for x in iter {}` — iterates anything implementing `IntoIterator`.
- Labels (`'name:`) break/continue outer loops.

```
// loop returning a value
let mut i = 0;
let doubled = loop {
    i += 1;
    if i == 5 { break i * 2; }   // returns 10
};

// while
let mut n = 3;
while n > 0 { print!("{n} "); n -= 1; }

// for over a range and a collection
for k in 0..3 { print!("{k} "); }      // 0 1 2
for x in [10, 20, 30] { print!("{x} "); }

// labeled break
'outer: for a in 0..3 {
    for b in 0..3 {
        if a + b == 3 { break 'outer; }
    }
}
println!("{doubled}");
```

Exercises

1. Use `loop` + `break value` to find the first square > 50.
2. Sum 1..=100 with a `while`.
3. Iterate a `Vec<&str>` with `for` and its `.iter().enumerate()`.
4. Use a labeled break to exit nested loops on a condition.

1.7 Ownership, Borrowing, and Slices

The core of Rust. Each value has one **owner**; when the owner drops, the value frees. You can **move** ownership, **borrow** immutably (`&T`, many at once), or **borrow** mutably (`&mut T`, exactly one, no shared reads at the same time).

```
fn takes(s: String) {}          // takes ownership (s moves in)
fn reads(s: &String) -> usize { s.len() }   // immutable borrow
fn appends(s: &mut String) { s.push('!'); } // mutable borrow

fn main() {
    let owned = String::from("hi");
    let n = reads(&owned);      // borrow, owned still usable
    let mut m = owned;          // move: owned no longer valid
    appends(&mut m);            // mutable borrow
    println!("{m} {n}");        // "hi!" 2

    let slice = &m[0..2];       // string slice (&str), a borrow
```

```
    println!("{slice}");        // "hi"
}
```

Note. Copy types (`i32`, `bool`, `char`, fixed-size arrays of `Copy`, …) are **copied** not moved. `String`, `Vec`, etc. are moved.

Exercises

1. Move a `String` into a function and observe the "use after move" error.
2. Fix it by borrowing with `&`.
3. Write a function taking `&mut Vec<i32>` that pushes an element.
4. Take a slice `&v[1..3]` of a vector and print it.

1.8 Strings

Two main types: `String` (owned, growable, heap) and `&str` (borrowed string slice). String literals are `&'static str`.

```
let lit: &str = "literal";
let mut owned = String::from("Hello");
owned.push_str(", world");
owned.push('!');
let combined = format!("{owned} ({} chars)", owned.len());
let upper = owned.to_uppercase();
for ch in "abc".chars() { print!("{ch} "); }
let slice: &str = &owned[0..5];     // "Hello"
```

Note. Indexing strings by integer (`s[0]`) is **not** allowed because UTF-8 chars vary in byte length. Use `.chars()`, `.bytes()`, or ranges on byte boundaries.

Exercises

1. Build a greeting with `format!`.
2. Count characters vs bytes for `"héllo"`.
3. Split `"a,b,c"` on `,` and collect into a `Vec<&str>`.
4. Convert a `&str` to `String` two different ways.

1.9 Pattern Matching: `match`

`match` compares a value against patterns, must be **exhaustive**, and is an **expression** (its arm value can be assigned). `_` is the catch-all; `|` combines patterns; `..=` matches ranges; guards add `if`.

```
let n = 4;
match n {
    0 => println!("zero"),
    1 | 2 => println!("one or two"),
    3..=9 => println!("three to nine"),
    x if x % 2 == 0 => println!("even {x}"),
    _ => println!("something else"),
}

// match as a value (assigning the result)
let label = match n {
```

```
    0 => "none",
    1..=9 => "small",
    _ => "large",
};

// destructuring in match
let point = (0, 5);
let where_ = match point {
    (0, 0) => "origin",
    (0, y) => "on y-axis",          // binds y
    (x, 0) => "on x-axis",
    (x, y) => "elsewhere",
};
println!("{label} {where_}");
```

Note. Because `match` is exhaustive, the compiler forces you to handle every case — a key source of Rust's reliability. Assigning `let v = match ... {}` is idiomatic.

Exercises

1. Match an `i32` into `"neg"`, `"zero"`, `"pos"` and assign to a variable.
2. Use `1 | 3 | 5` to detect small odd numbers.
3. Destructure a `(i32, i32)` tuple, binding one field with a guard.
4. Rewrite an `if/else if` chain as a `match`.

1.10 Modules, Paths, and Visibility

Code is organized into a **module tree**. `mod` declares a module; items are **private by default** and made public with `pub`. Paths address items (`crate::`, `super::`, `self::`); `use` brings a path into scope. A **crate** is a compilation unit (a binary or library); a **package** (with `Cargo.toml`) bundles crates.

```
mod geometry {
    pub mod shapes {                // nested, public module
        pub struct Circle { pub r: f64 }
        impl Circle {
            pub fn new(r: f64) -> Self { Circle { r } }   // pub constructor
            pub fn area(&self) -> f64 { 3.14159 * self.r * self.r }
        }
        fn helper() {}              // private to this module
    }
}

use geometry::shapes::Circle;       // bring into scope

fn main() {
    let c = Circle::new(2.0);       // no long path needed
    println!("{}", c.area());       // ~12.57
    // Fully-qualified alternative:
    let c2 = geometry::shapes::Circle::new(1.0);
    println!("{}", c2.area());
}
```

Note. Visibility levels: `pub` (public), `pub(crate)` (visible within the crate), `pub(super)` (visible to the parent module), and the default (private to the current module). Struct **fields** are private even when the struct is `pub` — mark fields `pub` individually. In real projects, modules usually live in separate files (`mod foo;` loads `foo.rs` or `foo/mod.rs`).

Exercises

1. Wrap a function in a `mod` and call it with a full path, then with `use`.
2. Make a module `pub` and one of its items private; observe the access error.
3. Define a `pub struct` with one `pub` and one private field; try constructing it outside.
4. Use `super::` to call a parent-module function from a child module.

1.11 Console I/O: Reading Input with `std::io`

Reading from the keyboard goes through `std::io::stdin()`. The key gotcha: `read_line` returns the text **including** the trailing newline, so you almost always `.trim()` it, and it appends rather than replaces, so reuse needs `buffer.clear()`. Parsing turns the `String` into a number via `.parse()`, which returns a `Result`.

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

fn main() {
    // 1) Simple line input
    let mut name = String::new();
    print!("Name: ");
    io::stdout().flush().unwrap();          // ensure prompt shows before input
    io::stdin().read_line(&mut name).unwrap();
    let name = name.trim();                 // strip the trailing '\n'
    println!("Hello, {name}!");

    // 2) Read and parse a number (handle bad input)
    let mut line = String::new();
    io::stdin().read_line(&mut line).unwrap();
    let age: i32 = match line.trim().parse() {
        Ok(n) => n,
        Err(_) => { eprintln!("not a number"); return; }
    };
    println!("Next year you are {}", age + 1);
}
```

Note. `print!` (no newline) is **line-buffered**, so a prompt may not appear until you `flush()` stdout. `println!` / `eprintln!` flush on the newline. Use `eprintln!` for errors so they go to **stderr**.

Looping until valid input. A common pattern is to re-prompt until parsing succeeds, clearing the buffer each round:

```
use std::io;

fn read_number(prompt: &str) -> i32 {
    loop {
        println!("{prompt}");
        let mut buf = String::new();
        io::stdin().read_line(&mut buf).unwrap();
```

```
        match buf.trim().parse() {
            Ok(n) => return n,
            Err(_) => println!("Please enter a valid integer."),
        }
    }
}
```

Multiple values on one line. Split on whitespace and collect or parse each token.

```
use std::io;

fn main() {
    let mut line = String::new();
    io::stdin().read_line(&mut line).unwrap();
    // "3 10 -2" -> vector of i32
    let nums: Vec<i32> = line
        .split_whitespace()
        .map(|t| t.parse().unwrap())
        .collect();
    let sum: i32 = nums.iter().sum();
    println!("{:?} sums to {sum}", nums);
}
```

Reading many lines / piped input. Iterate `stdin().lines()` to process input until end-of-file (Ctrl-D / a piped file). This is the idiomatic way to read an entire stream.

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

fn main() {
    let stdin = io::stdin();
    let mut count = 0;
    for line in stdin.lock().lines() {
        let line = line.unwrap();           // each line is a Result
        if line.trim().is_empty() { continue; }
        count += 1;
        println!("{count}: {line}");
    }
    println!("Read {count} non-empty lines");
}
```

Note. Read functions return `io::Result<_>`, so in real programs you propagate with `?` from a `fn main() -> io::Result<()>` rather than `.unwrap()`. For heavy input (competitive programming, large files) wrap stdin in a `BufReader` and lock it once (`stdin.lock()`) to avoid per-call locking overhead. Command-line **arguments** (not interactive input) come from `std::env::args()` instead.

Exercises

1. Prompt for a name, read a line, `trim` it, and greet the user.
2. Read a line and parse it to `f64`, handling the parse error with a message.
3. Write a `read_number` loop that re-prompts until the user enters a valid integer.
4. Read one line of space-separated integers and print their sum.
5. Read all lines from stdin with `lines()` and print only the non-empty ones, numbered.

2 Intermediate

This part covers data modeling, collections, error handling, and the trait/generic system — the tools you use to structure real programs.

2.1 Structs

Three forms: **named-field**, **tuple**, and **unit** structs. Methods live in `impl` blocks; `&self` borrows, `&mut self` mutably borrows, `self` consumes. Associated functions (no `self`, e.g. `new`) act as constructors.

```
struct Point { x: f64, y: f64 }      // named-field
struct Pair(i32, i32);               // tuple struct
struct Marker;                       // unit struct

impl Point {
    fn new(x: f64, y: f64) -> Self { Point { x, y } }   // associated fn
    fn dist(&self) -> f64 { (self.x.powi(2) + self.y.powi(2)).sqrt() }
    fn shift(&mut self, dx: f64) { self.x += dx; }       // mutates
}

fn main() {
    let mut p = Point::new(3.0, 4.0);
    println!("{}", p.dist());   // 5
    p.shift(1.0);
    let pair = Pair(1, 2);
    println!("{} {}", pair.0, pair.1);
    let _orig = Point { ..p };  // struct update syntax (move remaining)
}
```

Exercises

1. Define a `Rectangle` struct with `width` / `height` and an `area` method.
2. Add an associated `square(size)` constructor.
3. Write a `&mut self` method that doubles both fields.
4. Create a tuple struct `Rgb(u8,u8,u8)` and print its parts.

2.2 Enums and `Option`

Enums model a value that is **one of several variants**, each optionally carrying data. `Option<T>` (`Some` / `None`) replaces null; `Result<T, E>` models success/failure (next section).

```
enum Shape {
    Circle(f64),                 // tuple-like variant
    Rect { w: f64, h: f64 },     // struct-like variant
    Unit,                        // no data
}

fn area(s: &Shape) -> f64 {
    match s {
        Shape::Circle(r) => 3.14159 * r * r,
        Shape::Rect { w, h } => w * h,
        Shape::Unit => 0.0,
    }
}

fn main() {
```

```rust
    let maybe: Option<i32> = Some(10);
    if let Some(v) = maybe { println!("got {v}"); }  // if let
    let doubled = maybe.map(|v| v * 2).unwrap_or(0); // combinator chain
    println!("{doubled} {}", area(&Shape::Circle(2.0)));
}
```

Note. `if let` / `while let` are concise single-pattern matches. `Option` combinators (`map`, `and_then`, `unwrap_or`, `unwrap_or_else`) avoid verbose `match` blocks.

Exercises

1. Define an `enum Direction` with four variants and `match` over it.
2. Make an enum variant carry a `String` payload and extract it.
3. Use `if let Some(x)` to handle an `Option<i32>`.
4. Chain `.map().unwrap_or()` on an `Option`.

2.3 Collections: `Vec`, `HashMap`, `String`

- `Vec<T>` — growable array. `push`, `pop`, indexing, iteration, slicing.
- `HashMap<K, V>` — key/value store. `insert`, `get`, `entry` API.
- `String` — owned UTF-8 (covered earlier; it's a collection of bytes).

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

let mut v = vec![1, 2, 3];
v.push(4);
let total: i32 = v.iter().sum();
let evens: Vec<i32> = v.iter().filter(|&&x| x % 2 == 0).copied().collect();

let mut scores: HashMap<String, i32> = HashMap::new();
scores.insert("alice".into(), 10);
*scores.entry("alice".into()).or_insert(0) += 5;   // upsert pattern
if let Some(s) = scores.get("alice") { println!("{s}"); } // 15
println!("{total} {:?}", evens);
```

Exercises

1. Build a `Vec<i32>`, push 3 items, and print its sum.
2. Filter a vector to keep only values > 10 via iterators.
3. Count word frequencies into a `HashMap` using the `entry` API.
4. Iterate a `HashMap` and print each key/value pair.

2.4 Error Handling with `Result`

`Result<T, E>` has `Ok(T)` and `Err(E)`. Idiomatic Rust returns `Result` instead of throwing. Handle it via `match`, combinators, `unwrap` / `expect` (panics on error — use sparingly), or the `?` operator.

```rust
use std::num::ParseIntError;

fn parse(s: &str) -> Result<i32, ParseIntError> {
    let n: i32 = s.parse()?;   // ? : return Err early, else unwrap Ok
    Ok(n * 2)
}
```

```rust
fn main() {
    // Explicit match
    match parse("21") {
        Ok(v) => println!("ok {v}"),     // ok 42
        Err(e) => println!("err {e}"),
    }

    // Combinators
    let v = parse("x").unwrap_or(-1);    // -1 on error
    let mapped = parse("5").map(|n| n + 1);

    // unwrap / expect (panic on Err — for prototypes/tests only)
    let forced = parse("3").expect("must parse");
    println!("{v} {:?} {forced}", mapped);
}
```

The `?` **operator** unwraps `Ok`, or **returns the** `Err` **early** from the enclosing function (which must itself return a compatible `Result` or `Option`). It auto-converts the error via the `From` trait, so heterogeneous errors fit one return type (often `Box<dyn Error>`).

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

fn pipeline(a: &str, b: &str) -> Result<i32, Box<dyn Error>> {
    let x: i32 = a.parse()?;     // ParseIntError -> Box<dyn Error>
    let y: i32 = b.parse()?;
    Ok(x + y)
}
```

Note. `unwrap()` panics with a generic message; `expect("msg")` panics with your message. Prefer `?` or combinators in library code; reserve `unwrap` / `expect` for cases that are truly impossible to fail or for quick prototypes and tests.

Exercises

1. Write a function returning `Result<f64, String>` that errors on divide-by-zero.
2. Handle its result with an explicit `match`.
3. Rewrite the handling using `unwrap_or` and then `map`.
4. Chain two `.parse()?` calls in a function returning `Result<_, Box<dyn Error>>`.

2.5 Generics and Traits

Generics (`<T>`) write code once for many types. **Traits** define shared behavior (like interfaces). Trait **bounds** constrain generics; `impl Trait` is shorthand; `dyn Trait` enables runtime polymorphism (trait objects). Default methods and derivable traits (`Debug`, `Clone`, `PartialEq`, …) reduce boilerplate.

```rust
use std::fmt::Display;

// Generic function with a trait bound
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut max = list[0];
    for &x in list { if x > max { max = x; } }
    max
}
```

```rust
// Defining and implementing a trait
trait Greet {
    fn name(&self) -> String;
    fn hello(&self) -> String { format!("Hi, {}", self.name()) } // default method
}

struct Cat;
impl Greet for Cat {
    fn name(&self) -> String { "Cat".into() }
}

fn announce(g: &impl Greet) { println!("{}", g.hello()); }       // impl Trait
fn boxed() -> Box<dyn Greet> { Box::new(Cat) }                   // dyn Trait

fn main() {
    println!("{}", largest(&[3, 7, 2, 9]));   // 9
    announce(&Cat);                           // Hi, Cat
    println!("{}", boxed().hello());
}
```

Note. `impl Trait` resolves statically (monomorphized, fast). `dyn Trait` uses dynamic dispatch via a vtable (flexible, slight runtime cost). Derive common traits with `#[derive(Debug, Clone, PartialEq)]`.

Exercises

1. Write a generic `fn first<T: Clone>(v: &[T]) -> T`.
2. Define a `Describable` trait with one required and one default method.
3. Implement it for two different structs.
4. Write a function taking `&impl Describable`, then one returning `Box<dyn Describable>`.

2.6 Copy, Clone, and Drop

These three traits govern how values are **duplicated** and **destroyed**. `Copy` means a value is duplicated implicitly with a cheap bitwise copy (assignment/passing leaves the original valid) — only for simple stack types. `Clone` is an **explicit**, possibly expensive deep copy via `.clone()`. `Drop` defines custom cleanup that runs automatically when a value goes out of scope (RAII).

```rust
#[derive(Clone, Copy, Debug)]
struct PointC { x: i32, y: i32 }     // Copy: small, all fields Copy

#[derive(Clone, Debug)]
struct Buffer { data: Vec<u8> }      // Clone only: owns heap memory

struct Guard { name: String }
impl Drop for Guard {                // custom destructor
    fn drop(&mut self) { println!("dropping {}", self.name); }
}

fn main() {
    let a = PointC { x: 1, y: 2 };
    let b = a;                        // COPY — a still valid
    println!("{:?} {:?}", a, b);
```

```
    let buf = Buffer { data: vec![1, 2, 3] };
    let buf2 = buf.clone();           // explicit deep copy
    println!("{:?}", buf2);

    let _g = Guard { name: "g1".into() };   // "dropping g1" prints at scope end
}
```

Note. A type can be `Copy` only if all its fields are `Copy`, and `Copy` requires `Clone`. A type that implements `Drop` **cannot** be `Copy`. You rarely call `drop` manually, but `std::mem::drop(x)` forces an early drop. Prefer borrowing over `.clone()` when you only need to read.

Exercises

1. Derive `Copy, Clone` on a small struct and show the original is still usable after assignment.
2. Make a struct that owns a `Vec`; derive only `Clone` and duplicate it explicitly.
3. Implement `Drop` for a struct and observe drop order with two values.
4. Explain why adding a `String` field prevents deriving `Copy`.

2.7 Type Conversions: `From`, `Into`, and `as`

Idiomatic conversions use the `From` / `Into` traits — implement `From` and you get `Into` for free. `TryFrom` / `TryInto` are the fallible versions returning `Result`. The `as` keyword does **primitive** numeric/pointer casts (may truncate). This is also why `?` can unify error types: it calls `From` on the error.

```
struct Celsius(f64);
struct Fahrenheit(f64);

impl From<Celsius> for Fahrenheit {
    fn from(c: Celsius) -> Self { Fahrenheit(c.0 * 9.0 / 5.0 + 32.0) }
}

use std::convert::TryFrom;

fn main() {
    let f = Fahrenheit::from(Celsius(100.0));   // explicit From
    let f2: Fahrenheit = Celsius(0.0).into();   // Into (reverse direction)
    println!("{} {}", f.0, f2.0);               // 212 32

    let n: i64 = i64::from(42u8);               // lossless widening via From
    let truncated = 300i32 as u8;               // `as` cast -> 44 (wraps)
    let big = 5_000i32;
    let small = u8::try_from(big);              // TryFrom -> Err (out of range)
    println!("{n} {truncated} {:?}", small.is_err());
}
```

Note. Prefer `From` / `Into` for lossless/typed conversions and `TryFrom` / `TryInto` when conversion can fail. Reserve `as` for primitive numeric casts where you accept possible truncation/wrapping. Implementing `From<MyError>` for a wrapper error type is what lets `?` "just work" across error types.

Exercises

1. Implement `From<&str> for` a `Name(String)` newtype.
2. Convert with both `Name::from(...)` and `.into()`.
3. Use `as` to cast an `f64` to `i32` and note the truncation.
4. Use `u8::try_from` on an out-of-range `i32` and handle the `Err`.

3 Advanced

This part covers advanced borrowing, closures and iterators, lifetimes, smart pointers, concurrency, async/await, and macros — the features that distinguish idiomatic, high-performance Rust.

3.1 Borrowing in Depth

The Basic part introduced `&T` and `&mut T`. Here are the rules the **borrow checker** enforces and the subtleties that matter in real code. At any given time you may have **either** any number of immutable references (`&T`) **or** exactly one mutable reference (`&mut T`) to a value — never both. References must also always be valid (no dangling). These rules prevent data races and use-after-free **at compile time**.

```
fn main() {
    let mut v = vec![1, 2, 3];

    // Many shared (&) borrows are fine simultaneously
    let r1 = &v;
    let r2 = &v;
    println!("{:?} {:?}", r1, r2);   // r1, r2 last used here

    // After the shared borrows end, a single &mut is allowed (NLL)
    let m = &mut v;
    m.push(4);
    println!("{:?}", m);             // [1, 2, 3, 4]

    // ❌ Cannot mix: a &mut while a & is still live
    // let a = &v; let b = &mut v; println!("{:?} {:?}", a, b);
}
```

Non-Lexical Lifetimes (NLL). A borrow ends at its **last use**, not at the end of the enclosing block. That's why `r1` / `r2` above can be followed by a `&mut` in the same scope — the shared borrows are already "dead" by then.

Reborrowing. Passing a `&mut T` to a function doesn't move it; the compiler implicitly **reborrows** (`&mut *x`), so you can keep using the original afterward.

```
fn bump(n: &mut i32) { *n += 1; }   // *n dereferences to the value

fn main() {
    let mut x = 10;
    let r = &mut x;
    bump(r);          // implicit reborrow: r still usable after
    bump(r);
    println!("{}", *r);   // 12
}
```

The classic conflict: borrowing during iteration. You cannot mutate a collection while iterating a borrow of it — the iterator holds a shared borrow.

```rust
fn main() {
    let mut v = vec![1, 2, 3];
    // ❌ for x in &v { v.push(*x); }   // cannot borrow `v` as mutable
    // ✅ collect first, then mutate
    let extra: Vec<i32> = v.iter().map(|x| x * 10).collect();
    v.extend(extra);
    println!("{:?}", v);   // [1, 2, 3, 10, 20, 30]
}
```

Splitting borrows. The compiler understands disjoint **fields** of a struct can be borrowed independently, but not disjoint **indices** of a slice via plain indexing — use `split_at_mut` for that.

```rust
fn main() {
    let mut arr = [1, 2, 3, 4];
    let (left, right) = arr.split_at_mut(2);   // two non-overlapping &mut
    left[0] += 100;
    right[0] += 200;
    println!("{:?}", arr);   // [101, 2, 203, 4]
}
```

Note. Mental model: `&T` = shared/read-only (aliasing allowed, no mutation); `&mut T` = exclusive/read-write (mutation allowed, no aliasing). When a borrow seems "stuck," check whether an earlier borrow is still live, or restructure so borrows don't overlap. When you genuinely need shared **and** mutable access, reach for interior mutability (`RefCell`, `Mutex`) covered under Smart Pointers and Concurrency.

Exercises

1. Create two `&` borrows of a `Vec`, use them, then take a `&mut` in the same scope (NLL).
2. Trigger the "cannot borrow as mutable because also borrowed as immutable" error, then fix it.
3. Write a function taking `&mut i32`, call it twice on the same reference (reborrow).
4. Use `split_at_mut` to mutate two halves of an array without a borrow conflict.

3.2 Closures and Iterators

Closures are anonymous functions that capture their environment, classified by the `Fn` / `FnMut` / `FnOnce` traits (by reference / by mutable reference / by value). **Iterators** are lazy; adapters (`map`, `filter`, `take`, `zip`, `enumerate`) build a pipeline, and a **consumer** (`collect`, `sum`, `fold`, `for_each`) drives it.

```rust
let factor = 3;
let scale = |x: i32| x * factor;          // captures `factor` (Fn)
let mut count = 0;
let mut tick = || { count += 1; };        // captures mutably (FnMut)
tick(); tick();

let nums = vec![1, 2, 3, 4, 5];
let result: Vec<i32> = nums.iter()
    .map(|&x| x * 2)        // adapter (lazy)
    .filter(|&x| x > 4)     // adapter (lazy)
    .collect();             // consumer (runs the pipeline) -> [6, 8, 10]
```

```rust
let sum: i32 = (1..=5).fold(0, |acc, x| acc + x);   // 15
println!("{:?} {sum} {} {count}", result, scale(10));
```

Note. Use `move` (`move || ...`) to force a closure to take ownership of captured variables — essential when passing closures to threads.

Exercises

1. Write a closure that adds a captured constant to its argument.
2. Build an iterator chain: square, keep even, `collect` to `Vec`.
3. Sum a range with `fold`.
4. Use `enumerate` + `for_each` to print index/value pairs.

3.3 Lifetimes

Lifetimes are compile-time annotations ensuring references never outlive the data they point to. Most are inferred (**elision**); you annotate (`'a`) when the relationship between input and output references is ambiguous. `'static` lives for the whole program.

```rust
// Output reference's lifetime tied to both inputs
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}

// Struct holding a reference needs a lifetime parameter
struct Excerpt<'a> { part: &'a str }

fn main() {
    let s1 = String::from("longer string");
    let s2 = "short";
    println!("{}", longest(&s1, s2));

    let novel = String::from("Call me Ishmael. Some years ago...");
    let first = novel.split('.').next().unwrap();
    let e = Excerpt { part: first };
    println!("{}", e.part);
}
```

Note. Lifetimes do not change how long values live; they **describe** existing relationships so the borrow checker can verify safety. The three elision rules let you omit them in most function signatures.

Exercises

1. Annotate a `longest` function and call it.
2. Trigger a "borrowed value does not live long enough" error, then fix it.
3. Define a struct holding a `&str` with a lifetime parameter.
4. Return a reference from a method using an elided lifetime.

3.4 Smart Pointers: `Box`, `Rc`, `RefCell`

- `Box<T>` — heap allocation, single owner; enables recursive types and trait objects.
- `Rc<T>` — reference-counted shared ownership (single-threaded).

- `RefCell<T>` — interior mutability with **runtime**-checked borrows.
- `Rc<RefCell<T>>` — shared, mutable graph nodes (common combo).
- `Arc<T>` — atomic `Rc` for threads (see Concurrency).

```
use std::rc::Rc;
use std::cell::RefCell;

// Recursive type needs indirection
enum List { Cons(i32, Box<List>), Nil }
use List::*;

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Nil))));

    let shared = Rc::new(RefCell::new(vec![1, 2, 3]));
    let clone = Rc::clone(&shared);     // refcount = 2, same data
    clone.borrow_mut().push(4);         // mutate through shared ref
    println!("{:?}", shared.borrow());  // [1, 2, 3, 4]
    println!("count = {}", Rc::strong_count(&shared)); // 2

    if let Cons(v, _) = list { println!("head {v}"); }
}
```

Note. `RefCell` moves borrow checking to runtime: violating borrow rules **panics** instead of failing to compile. `Rc` / `RefCell` are single-threaded; reach for `Arc` / `Mutex` across threads.

Exercises

1. Box a recursive `enum` list and traverse it.
2. Share a value with `Rc` and print `strong_count` after cloning.
3. Mutate data through `RefCell::borrow_mut`.
4. Trigger a `RefCell` double-borrow panic, then fix it.

3.5 Concurrency

Rust's ownership rules make data races a **compile error**. Spawn threads with `thread::spawn`; share immutable data and message-pass with channels (`mpsc`); share mutable state with `Arc<Mutex<T>>`.

```
use std::sync::{Arc, Mutex, mpsc};
use std::thread;

fn main() {
    // Shared mutable state
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];
    for _ in 0..4 {
        let c = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            *c.lock().unwrap() += 1;     // lock guards the data
        }));
    }
    for h in handles { h.join().unwrap(); }
    println!("total = {}", *counter.lock().unwrap());  // 4
```

```
    // Message passing
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || { tx.send("hello").unwrap(); });
    println!("{}", rx.recv().unwrap());
}
```

Note. `Send` and `Sync` are marker traits the compiler uses to decide what may cross threads. `move` transfers ownership into the thread closure. `Arc` (atomic) replaces `Rc` across threads.

Exercises

1. Spawn 3 threads that each print their index; `join` them.
2. Increment a shared `Arc<Mutex<i32>>` from multiple threads.
3. Send 5 numbers over an `mpsc` channel and sum them on the receiver.
4. Add `move` to a thread closure and explain why it's required.

3.6 Async / Await

`async fn` returns a **Future** — a lazy computation that does nothing until **driven** by a runtime (e.g. Tokio). `.await` suspends the current task until the future completes, yielding the thread to other tasks meanwhile (non-blocking concurrency, not parallelism by itself).

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

async fn fetch(id: u32) -> Result<String, Box<dyn Error>> {
    // imagine real I/O here
    Ok(format!("item-{id}"))
}

async fn run() -> Result<(), Box<dyn Error>> {
    let f = fetch(1);          // nothing happens yet (lazy future)
    let value = f.await?;      // drive it; ? propagates any Err
    println!("{value}");

    // Concurrent awaiting
    let (a, b) = tokio::join!(fetch(2), fetch(3));
    println!("{} {}", a?, b?);
    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    run().await
}
```

`await` vs `await?` — both drive the future to completion. The difference is **what they evaluate to** and **what the future returns**:

- `future.await` yields the future's output value **as-is**. If the output is a `Result`, you still hold a `Result` and must handle it yourself (`match`, `unwrap`, etc.).
- `future.await?` first awaits, **then** applies the `?` operator to the resulting `Result` (or `Option`): on `Ok(v)` it unwraps to `v`; on `Err(e)` it returns that error early from the enclosing `async fn`. So `await?` = "await, then propagate the error."

```rust
// await alone — you get the Result back
let r: Result<String, _> = fetch(1).await;
match r { Ok(v) => println!("{v}"), Err(e) => eprintln!("{e}") }

// await? — unwrap on success, return Err early on failure
let v: String = fetch(1).await?;   // only valid in a fn returning Result
```

Note. `await?` is only valid inside an `async fn` (or block) whose return type is a compatible `Result` / `Option`, because `?` needs somewhere to return the error to. Use plain `.await` when the output isn't a `Result`, or when you want to handle the error inline rather than propagate it.

Exercises

1. Write an `async fn` returning `Result<i32, String>` and call it with `.await` + `match`.
2. Rewrite the caller to use `.await?` inside an `async fn` returning `Result`.
3. Use `tokio::join!` to await two futures concurrently.
4. Explain (in a comment) why a future does nothing until `.await`ed.

3.7 Macros

Macros generate code at compile time. **Declarative** macros (`macro_rules!`) match token patterns and expand them — useful for variadic, repetitive code (`vec!`, `println!` are macros). **Procedural** macros (custom derive, attribute, function-like) operate on token streams in a separate crate and power things like `#[derive(...)]`.

```rust
// Declarative macro: a mini "max of many"
macro_rules! max_of {
    ($x:expr) => ($x);
    ($x:expr, $($rest:expr),+) => {
        { let r = max_of!($($rest),+); if $x > r { $x } else { r } }
    };
}

fn main() {
    let m = max_of!(3, 7, 2, 9, 1);   // expands recursively -> 9
    println!("{m}");

    // Common built-in macros
    let v = vec![0; 3];               // [0, 0, 0]
    assert_eq!(v.len(), 3);
    println!("{:?}", v);
}
```

Note. Fragment specifiers (`$x:expr`, `:ident`, `:ty`, `:tt`, …) define what each metavariable matches. `$(...),+` means "one or more, comma-separated." Procedural macros require a `proc-macro` crate and crates like `syn` / `quote`.

Exercises

1. Write a `macro_rules!` that prints any number of arguments each on its own line.
2. Extend `max_of!` to also support a trailing comma.
3. Use `vec![expr; n]` and explain how it differs from `vec![a, b, c]`.

4. Add `#[derive(Debug, Clone)]` to a struct and print it with `{:?}`.

End of the Compact Rust Handbook.
Compile, hack, and keep the borrow checker close.