DOMINANDO RUST EM 1 ANO

Caderno de Respostas

Gabarito comentado: 53 aulas de prática, do #01 ao #53

Rust 1.98 · edição 2024 · Cargo · 11 módulos · 67 questões · 2 desafios integradores

Autor Desconhecido · theknowledgeisle.com

Consulte este gabarito só depois de tentar.

As respostas são comentadas para ensinar, não só para conferir. Nos exercícios de código existem outras soluções igualmente válidas: o que importa é entender por que a sua funciona e o que ela troca em relação à mostrada aqui.

Os programas com um quadro verde de **Saída** foram compilados com rustc 1.98 (edição 2024, -O) e executados no Linux, e a saída mostrada é a real.

Sumário

Módulo	Tema	Aulas	Questões
MÓDULO 1	Fundamentos da linguagem	#01–#04	Q1–Q6
MÓDULO 2	Ownership e borrowing	#05–#06	Q7–Q11
MÓDULO 3	Tipos próprios e erros	#07–#09	Q12–Q17
MÓDULO 4	Coleções, iteradores e closures	#10–#11	Q18–Q23
MÓDULO 5	Traits, generics e lifetimes	#12–#14	Q24–Q29
MÓDULO 6	Módulos e testes	#15–#16	Q30–Q32
MÓDULO 7	Recursos avançados	#17–#21	Q33–Q42
MÓDULO 8	Ecossistema e web	#22–#26	Q43–Q48
MÓDULO 9	Sistemas e baixo nível	#27–#36	Q49–Q56
MÓDULO 10	Engenharia de projetos	#37–#41	Q57–Q61
MÓDULO 11	Rust em domínios específicos e o futuro	#42–#53	Q62–Q67
DESAFIOS	Desafios integradores	todas	D1–D2

MÓDULO 1

Fundamentos da linguagem

Por que Rust existe, variáveis e imutabilidade, funções e expressões, e o controle de fluxo.

Artigos 01 a 04

#01 Por que Rust existe e o que ele quer de você
#02 Variáveis, Tipos e a Arte da Imutabilidade
#03 Funções, Expressões e Como Rust Pensa Diferente sobre Retorno de Valores
#04 Controle de Fluxo — if, loop, while e for como você nunca viu antes

Perguntas conceituais — respostas

Q1 O que Rust quer de você

> *O artigo #01 diz que Rust oferece segurança de memória sem coletor de lixo. Que duas classes de bugs comuns em C e C++ o compilador do Rust impede, e qual é o preço que o programador paga por isso?*

- **Uso de memória inválida**: ponteiro solto (*use after free*), liberação dupla e leitura fora dos limites. O sistema de posse e empréstimo garante, em tempo de compilação, que nenhuma referência sobrevive ao valor que ela aponta; o acesso a vetores é verificado.
- **Corridas de dados**: duas threads acessando o mesmo dado, ao menos uma escrevendo, sem sincronização. As regras de empréstimo e os *traits* `Send` e `Sync` tornam isso erro de compilação.

O preço é uma curva de aprendizado mais íngreme: o compilador recusa programas que **talvez** fossem corretos, e é preciso aprender a estruturar o código do jeito que ele consegue provar seguro. Em troca, uma classe inteira de falhas, que em C só aparece em produção, nem chega a compilar.

Q2 Imutável por padrão

> *Por que `let x = 5; x = 6;` não compila em Rust? Qual a diferença entre `let mut`, shadowing (`let x = x + 1;`) e `const`?*

Variáveis são **imutáveis por padrão**: a mutação precisa ser pedida com `mut`, o que deixa visível, na declaração, quais valores mudam. O `let mut x` cria uma variável que pode receber novos valores **do mesmo tipo**. O *shadowing* cria uma variável **nova** com o mesmo nome, que esconde a anterior; ela pode até ter outro tipo (`let espacos = " "; let espacos = espacos.len();`), e a anterior continua imutável. O `const` é um valor conhecido em tempo de compilação, com tipo obrigatório, que pode ser global e é copiado onde é usado (`const MAX_JOGADORES: u32 = 4;`).

Q3 **Expressão ou instrução**

Por que a função abaixo não compila, e como corrigi-la com uma mudança de um caractere?

```
fn dobro(x: i32) -> i32 {
    x * 2;
}
```

Em Rust, quase tudo é **expressão** (produz um valor), e um bloco `{ ... }` vale a sua última expressão, **se ela não terminar em ponto e vírgula**. O `;` transforma `x * 2` numa **instrução**, que vale `()` (a unidade); a função promete `i32` e devolve `()`, e o compilador acusa "*mismatched types*". A correção é tirar o ponto e vírgula: `x * 2`. O mesmo princípio permite `let nota = if pontos > 90 { 'A' } else { 'B' };` e `let r = loop { ... break valor; };`.

Q4 **Estouro de inteiro**

O que acontece em `let x: u8 = 255; let y = x + 1;` no modo debug e no modo release? Que métodos tornam a intenção explícita?

Em ***debug***, o estouro aritmético faz o programa entrar em **pânico** ("*attempt to add with overflow*"); em ***release*** (`-O`), por padrão, o valor **dá a volta** (*wrap*) e `y` vale 0, sem aviso. (Quando o valor é uma constante conhecida, como aqui, o compilador já recusa na compilação.) Para não depender do modo, os inteiros têm métodos explícitos: `checked_add` devolve `Option` (`None` no estouro), `wrapping_add` dá a volta de propósito, `saturating_add` para no limite (255) e `overflowing_add` devolve o resultado e um `bool`.

Exercícios de código — respostas

Q5 **Tipos e conversões**

Escreva um programa que mostre: o maior `u8` e o menor `i16`; a conversão com `as` de `300_i32` para `u8` e de `-1_i32` para `u32`; a conversão segura com `u8::try_from`; e as quatro formas de somar `250_u8 + 10` (`checked`, `wrapping`, `saturating` e `overflowing`).

```
fn main() {
    println!("u8::MAX = {}, i16::MIN = {}", u8::MAX, i16::MIN);
    println!("300 as u8 = {} | -1 as u32 = {}", 300_i32 as u8, -1_i32 as u32);

    for v in [42_i32, 300, -5] {
        match u8::try_from(v) {
            Ok(b) => println!("try_from({v}) = Ok({b})"),
            Err(e) => println!("try_from({v}) = Err({e})"),
        }
    }

    let x: u8 = 250;
    println!("checked:     {:?}", x.checked_add(10));
    println!("wrapping:    {}", x.wrapping_add(10));
    println!("saturating:  {}", x.saturating_add(10));
    println!("overflowing: {:?}", x.overflowing_add(10));
}
```

Saída

```
u8::MAX = 255, i16::MIN = -32768
300 as u8 = 44 | -1 as u32 = 4294967295
try_from(42) = Ok(42)
try_from(300) = Err(out of range integral type conversion attempted)
try_from(-5) = Err(out of range integral type conversion attempted)
checked:     None
wrapping:    4
saturating:  255
overflowing: (4, true)
```

O `as` entre inteiros **nunca falha**: trunca os bits que não cabem (300 é 0x12C, e sobra 0x2C = 44) e reinterpreta o sinal (-1 vira o maior `u32`). Por isso é a escolha errada para dados vindos de fora; o `try_from` devolve um `Result` e obriga a tratar o caso que não cabe.

Q6 Expressões que devolvem valores

Escreva `fn classificar(nota: u32) -> &'static str` usando `match` com faixas (`90..=100`), uma função `fn primeiro_divisor(n: u32) -> u32` que use `loop` com `break valor`, e um laço com rótulo (`'externo`) que pare os dois laços quando achar o primeiro par `(a, b)` com `a * b == 42` e `a < b`.

```
fn classificar(nota: u32) -> &'static str {
    match nota {
        90..=100 => "A",
        70..=89 => "B",
        50..=69 => "C",
        0..=49 => "D",
        _ => "nota inválida",
    }
}

fn primeiro_divisor(n: u32) -> u32 {
    let mut d = 2;
    loop {
        if n % d == 0 {
            break d; // o loop "vale" d
        }
        d += 1;
    }
}

fn main() {
    for n in [95, 72, 40, 120] {
        println!("{n}: {}", classificar(n));
    }
    println!("primeiro divisor de 91: {}", primeiro_divisor(91));

    let mut achado = None;
    'externo: for a in 1..=42 {
        for b in a + 1..=42 {
            if a * b == 42 {
                achado = Some((a, b));
                break 'externo;
            }
        }
    }
    println!("primeiro par: {achado:?}");
}
```

Saída

```
95: A
72: B
40: D
120: nota inválida
primeiro divisor de 91: 7
primeiro par: Some((1, 42))
```

O `match` sobre um `u32` precisa ser **exaustivo**: sem o braço `_`, o compilador recusa, porque valores acima de 100 não estariam cobertos. O `loop` é o único laço que pode devolver valor com `break`, porque é o único que o compilador sabe que só termina pelo `break`. O rótulo evita a variável "achei" e o segundo `break` que o C exigiria.

MÓDULO 2

Ownership e borrowing

Posse, movimento e cópia; referências, empréstimos mutáveis e imutáveis, e slices.

Artigos 05 a 06

#05 Ownership — A Ideia que Muda Tudo
#06 Borrowing e Referências — Usando sem Possuir

Perguntas conceituais — respostas

Q7 As três regras da posse

Enuncie as regras de ownership e explique por que `let s1 = String::from("oi"); let s2 = s1; println!("{s1}");` não compila, enquanto o mesmo código com `i32` compila.

1. Cada valor tem **um único dono** (uma variável).
2. Só pode haver um dono por vez.
3. Quando o dono sai de escopo, o valor é **liberado** (`drop`).

A `String` guarda os caracteres no *heap*. `let s2 = s1;` **move** a posse: copia o ponteiro, o tamanho e a capacidade, e invalida `s1`, para que o *heap* não seja liberado duas vezes quando as duas saírem de escopo. Usar `s1` depois é "*borrow of moved value*". O `i32` vive todo na pilha e implementa o *trait* `Copy`: a atribuição copia os bits, e as duas variáveis continuam válidas. Para duplicar uma `String`, pede-se explicitamente: `s1.clone()`, e a cópia cara fica visível no código.

Q8 Uma mutável ou várias imutáveis

Qual é a regra dos empréstimos, e que bug clássico de C++ ela impede no código abaixo?

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

Em qualquer momento, pode haver **várias referências imutáveis** (`&T`) **ou uma única mutável** (`&mut T`), nunca as duas ao mesmo tempo. Aqui, `primeiro` é um empréstimo imutável de `v`, ainda usado no `println!`, e o `push` precisa de um empréstimo mutável: o compilador recusa. O motivo é concreto: o `push` pode **realocar** o vetor, e `primeiro` apontaria para memória liberada, exatamente a invalidação de iteradores do `std::vector` em C++, que lá compila e falha em execução. Graças às *non-lexical lifetimes*, se `primeiro` não fosse usado depois do `push`, o código compilaria: o empréstimo acaba no último uso.

Q9 String e &str

Qual a diferença entre `String` e `&str`? Por que uma função que só lê um texto deve receber `&str`?

A `String` é **dona** de um texto UTF-8 no *heap*, que pode crescer e mudar. O `&str` é uma **fatia emprestada** (um ponteiro e um tamanho) de um texto que pertence a outro: uma `String`, um literal (que vive no binário, `&'static str`) ou parte de outro texto. Uma função que só lê deve receber `&str` porque aceita os três casos: uma `&String` converte sozinha para `&str` (*deref coercion*), e um literal já é `&str`. Receber `String` obrigaria quem chama a entregar a posse ou a clonar; receber `&String` recusaria literais sem uma conversão.

Exercícios de código — respostas

Q10 Movimento, cópia e empréstimo

Escreva um programa que mostre: (a) uma `String` movida para uma função e devolvida no retorno; (b) a mesma função reescrita para receber `&str`; (c) uma função que recebe `&mut String` e acrescenta texto; (d) um `struct` com `#[derive(Clone, Copy)]` sendo copiado.

```
fn consome(s: String) -> String {
    format!("{s}!") // recebe a posse e devolve uma String nova
}

fn tamanho(s: &str) -> usize {
    s.chars().count()
}

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

#[derive(Debug, Clone, Copy)]
struct Ponto {
    x: i32,
    y: i32,
}

fn main() {
    let s = String::from("ação");
    let s = consome(s); // s foi movida e recebemos outra de volta
    println!("(a) {s}");

    println!("(b) {} caracteres, {} bytes", tamanho(&s), s.len());

    let mut grito = String::from("socorro");
    exclamar(&mut grito);
    println!("(c) {grito}");

    let p = Ponto { x: 1, y: 2 };
    let mut q = p; // cópia: Ponto é Copy
    q.x = 10;
    println!("(d) p = ({}, {}), q = ({}, {})", p.x, p.y, q.x, q.y);
}
```

Saída

```
(a) ação!
(b) 5 caracteres, 7 bytes
(c) socorro!!
(d) p = (1, 2), q = (10, 2)
```

O `len()` conta **bytes** (UTF-8), e o `chars().count()`, caracteres: "ação!" tem 5 caracteres e 7 bytes. Um `struct` só pode ser `Copy` se todos os campos forem `Copy`; com um campo `String`, o `derive(Copy)` não compila, porque copiar os bits duplicaria o dono do *heap*.

Q11 Slices e a primeira palavra

Escreva `fn primeira_palavra(s: &str) -> &str` que devolva a primeira palavra (até o primeiro espaço) como uma fatia do texto original, sem alocar. Mostre que tentar esvaziar a `String` (`clear`) enquanto a fatia ainda está em uso não compila (em comentário), e use a função com um literal e com uma `String`.

```
fn primeira_palavra(s: &str) -> &str {
    match s.find(' ') {
        Some(i) => &s[..i],
        None => s,
    }
}

fn main() {
    let mut frase = String::from("Ferris é o caranguejo");
    let p = primeira_palavra(&frase);
    // frase.clear(); // erro: frase está emprestada (imutável) para p, que é usada abaixo
    println!("primeira palavra: {p}");

    frase.clear(); // aqui pode: o empréstimo de p já terminou
    println!("frase vazia? {}", frase.is_empty());

    println!("de um literal: {}", primeira_palavra("Rust"));

    let numeros = [10, 20, 30, 40, 50];
    let meio: &[i32] = &numeros[1..4];
    println!("fatia de array: {meio:?}, soma {}", meio.iter().sum::<i32>());
}
```

Saída

```
primeira palavra: Ferris
frase vazia? true
de um literal: Rust
fatia de array: [20, 30, 40], soma 90
```

A fatia devolvida carrega um vínculo com a `String` original (um *lifetime*, que o compilador deduziu sozinho aqui): enquanto ela for usada, a `String` não pode ser alterada nem liberada. Em C, a função devolveria um `char*` para dentro do texto, e um `clear` antes do uso produziria lixo sem nenhum aviso. Cuidado com fatias

de texto: `&s[..i]` com `i` no meio de um caractere de dois bytes (como o "é") **entra em pânico**; o `find` devolve sempre um índice de início de caractere, por isso aqui é seguro.

MÓDULO 3

Tipos próprios e erros

Structs e métodos, enums e pattern matching, e o tratamento de erros com Result.

Artigos 07 a 09

#07 Structs — Criando seus próprios tipos de dados
#08 Enums e Pattern Matching — O Sistema Mais Expressivo que Você Já Viu
#09 Tratamento de Erros com Result — Erros como Valores, não Exceções

Perguntas conceituais — respostas

Q12 Option no lugar de null

Por que Rust não tem `null`? O que o `Option<T>` obriga o programador a fazer que um ponteiro nulo não obriga?

Tony Hoare, o criador da referência nula, a chamou de "erro de um bilhão de dólares": qualquer referência pode ser nula, e o tipo não diz quais. Em Rust, uma referência **sempre** aponta para algo válido, e a ausência é um tipo próprio: `Option<T>`, com os casos `Some(valor)` e `None`. Um `Option<String>` **não é** uma `String`: para usar o valor, é preciso tratá-lo (com `match`, `if let`, `?`, `unwrap_or` e afins), e o compilador não deixa esquecer o caso `None`. A ausência deixa de ser uma surpresa em execução e passa a ser uma informação no tipo. E não custa memória a mais: um `Option<&T>` ou `Option<Box<T>>` ocupa o mesmo que o ponteiro, porque o `None` é representado pelo endereço zero, que uma referência nunca tem.

Q13 O operador ?

O que o operador `?` faz numa função que devolve `Result<T, E>`? Por que `unwrap()` é aceitável num exemplo e perigoso numa biblioteca?

Aplicado a um `Result`, o `?` devolve o valor se for `Ok`, ou **retorna imediatamente** da função com o `Err`, convertendo o tipo do erro com `From` se for preciso. Substitui o `match` repetido a cada chamada que pode falhar, mantendo a propagação explícita (cada ponto de falha tem o seu `?` visível). O `unwrap()` extrai o valor ou **entra em pânico**: num exemplo ou num teste, ele encurta o código e a falha é aceitável; numa biblioteca, derruba o programa de quem a usa por um erro que ele poderia ter tratado. Fora de testes, prefira `?`, ou `expect("motivo")` quando a falha realmente for um *bug* impossível, com a explicação de por quê.

Q14 Enum com dados

Como um `enum` do Rust difere de um `enum` do C? Mostre um `enum Mensagem` com quatro variantes de formatos diferentes.

No C, um `enum` é só um conjunto de inteiros com nome. No Rust, cada variante pode **carregar dados próprios**, com formatos diferentes, e o compilador sabe sempre qual variante está guardada (é a união etiquetada, segura):

```
enum Mensagem {
    Sair,                       // sem dados
    Mover { x: i32, y: i32 },   // campos nomeados
    Escrever(String),           // um valor
    MudarCor(u8, u8, u8),       // uma tupla
}
```

Um `match` sobre uma `Mensagem` precisa tratar as quatro variantes, e cada braço extrai os dados da sua (`Mensagem::Mover { x, y } => ...`). Acrescentar uma variante faz o compilador apontar cada `match` que ficou incompleto. `Option` e `Result` são *enums* comuns da biblioteca padrão, definidos assim.

Exercícios de código — respostas

Q15 **Uma struct com métodos**

Escreva `struct Retangulo { largura: f64, altura: f64 }` com um construtor associado `quadrado(lado)`, os métodos `area(&self)`, `escalar(&mut self, fator)` e `cabe_em(&self, outro: &Retangulo) -> bool`, e implemente o trait `Display` para imprimi-lo como `3x4`.

```rust
use std::fmt;

#[derive(Debug, Clone, PartialEq)]
struct Retangulo {
    largura: f64,
    altura: f64,
}

impl Retangulo {
    fn quadrado(lado: f64) -> Self {
        Self { largura: lado, altura: lado }
    }
    fn area(&self) -> f64 {
        self.largura * self.altura
    }
    fn escalar(&mut self, fator: f64) {
        self.largura *= fator;
        self.altura *= fator;
    }
    fn cabe_em(&self, outro: &Retangulo) -> bool {
        self.largura <= outro.largura && self.altura <= outro.altura
    }
}

impl fmt::Display for Retangulo {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{}x{}", self.largura, self.altura)
    }
}

fn main() {
    let mut r = Retangulo { largura: 3.0, altura: 4.0 };
    let q = Retangulo::quadrado(5.0);
    println!("{r} tem área {}; cabe em {q}? {}", r.area(), r.cabe_em(&q));
    r.escalar(2.0);
    println!("escalado: {r} (Debug: {r:?}); cabe agora? {}", r.cabe_em(&q));
    println!("igual a 6x8? {}", r == Retangulo { largura: 6.0, altura: 8.0 });
}
```

Saída

```
3x4 tem área 12; cabe em 5x5? true
escalado: 6x8 (Debug: Retangulo { largura: 6.0, altura: 8.0 }); cabe agora? false
igual a 6x8? true
```

O primeiro parâmetro diz o que o método faz com o objeto: `&self` só lê, `&mut self` altera, e `self` consumiria o valor. A chamada `r.escalar(2.0)` empresta `r` como mutável sozinha (*auto-referencing*), o que exige `let mut r`. O `Display` é para pessoas; o `Debug` (derivado) é para o programador. O `PartialEq` derivado compara campo a campo.

Q16 Uma máquina de estados com enum

Modele um pedido com `enum Estado { Novo, Pago { valor: u32 }, Enviado { codigo: String }, Cancelado { motivo: String } }` e escreva `fn avancar(estado: Estado, evento: Evento) -> Result<Estado, String>` com `match` sobre a tupla `(estado, evento)`. Rode uma sequência válida e duas inválidas.

```
#[derive(Debug)]
enum Estado {
    Novo,
    Pago { valor: u32 },
    Enviado { codigo: String },
    Cancelado { motivo: String },
}

#[derive(Debug)]
enum Evento {
    Pagar(u32),
    Enviar(String),
    Cancelar(String),
}

fn avancar(estado: Estado, evento: Evento) -> Result<Estado, String> {
    match (estado, evento) {
        (Estado::Novo, Evento::Pagar(valor)) if valor > 0 => Ok(Estado::Pago { valor }),
        (Estado::Pago { .. }, Evento::Enviar(codigo)) => Ok(Estado::Enviado { codigo }),
        (Estado::Novo | Estado::Pago { .. }, Evento::Cancelar(motivo)) => Ok(Estado::Cancelado {
motivo }),
        (e, ev) => Err(format!("não dá para {ev:?} no estado {e:?}")),
    }
}

fn descrever(e: &Estado) -> String {
    match e {
        Estado::Novo => "novo".to_string(),
        Estado::Pago { valor } => format!("pago (R$ {},{:02})", valor / 100, valor % 100),
        Estado::Enviado { codigo } => format!("enviado, rastreio {codigo}"),
        Estado::Cancelado { motivo } => format!("cancelado: {motivo}"),
    }
}

fn main() {
    let mut estado = Estado::Novo;
    for ev in [Evento::Pagar(4990), Evento::Enviar("BR123".into())] {
        estado = avancar(estado, ev).expect("sequência válida");
        println!("-> {}", descrever(&estado));
    }
    println!("{}", avancar(estado, Evento::Cancelar("desisti".into())).unwrap_err());
    println!("{}", avancar(Estado::Novo, Evento::Pagar(0)).unwrap_err());
    let pago = avancar(Estado::Novo, Evento::Pagar(2500)).unwrap();
    println!("-> {}", descrever(&avancar(pago, Evento::Cancelar("endereço
errado".into())).unwrap()));
}
```

Saída

```
-> pago (R$ 49,90)
-> enviado, rastreio BR123
não dá para Cancelar("desisti") no estado Enviado { codigo: "BR123" }
não dá para Pagar(0) no estado Novo
-> cancelado: endereço errado
```

Cada estado carrega só os dados que fazem sentido nele: um pedido `Novo` não tem código de rastreio, e não existe um campo `codigo` vazio esperando ser preenchido. A guarda `if valor > 0` refina o padrão. Como `avancar` recebe o estado **por valor**, o estado antigo é consumido e não pode ser usado por engano depois da transição.

Q17 Erros próprios e o operador ?

Escreva `fn ler_config(texto: &str) -> Result<Config, ErroConfig>` que leia linhas `chave=valor` com as chaves `porta` (número de 1 a 65535) e `host` (obrigatório). Defina `enum ErroConfig` com as variantes `LinhaInvalida(usize)`, `Faltando(&'static str)` e `Porta(std::num::ParseIntError)`, implemente `Display` e `From<ParseIntError>`, e use o `?`.

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

#[derive(Debug)]
struct Config {
    host: String,
    porta: u16,
}

#[derive(Debug)]
enum ErroConfig {
    LinhaInvalida(usize),
    Faltando(&'static str),
    Porta(ParseIntError),
}

impl fmt::Display for ErroConfig {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            ErroConfig::LinhaInvalida(n) => write!(f, "linha {n} não está no formato
chave=valor"),
            ErroConfig::Faltando(chave) => write!(f, "falta a chave '{chave}'"),
            ErroConfig::Porta(e) => write!(f, "porta inválida: {e}"),
        }
    }
}

impl From<ParseIntError> for ErroConfig {
    fn from(e: ParseIntError) -> Self {
        ErroConfig::Porta(e)
    }
}

fn ler_config(texto: &str) -> Result<Config, ErroConfig> {
    let (mut host, mut porta) = (None, 8080);
    for (i, linha) in texto.lines().enumerate().filter(|(_, l)| !l.trim().is_empty()) {
        let (chave, valor) = linha.split_once('=').ok_or(ErroConfig::LinhaInvalida(i + 1))?;
        match chave.trim() {
            "host" => host = Some(valor.trim().to_string()),
            "porta" => porta = valor.trim().parse::<u16>()?, // ParseIntError vira ErroConfig
pelo From
            _ => {} // chaves desconhecidas são ignoradas
        }
    }
    let host = host.ok_or(ErroConfig::Faltando("host"))?;
    if porta == 0 {
        return Err(ErroConfig::LinhaInvalida(0));
    }
    Ok(Config { host, porta })
}

fn main() {
    let casos = [
```

```
        "host = livraria.local\nporta = 3000",
        "porta = 3000",
        "host = x\nporta = 70000",
        "host = x\nisto não é config",
    ];
    for c in casos {
        match ler_config(c) {
            Ok(cfg) => println!("ok: {}:{}", cfg.host, cfg.porta),
            Err(e) => println!("erro: {e}"),
        }
    }
}
```

Saída

```
ok: livraria.local:3000
erro: falta a chave 'host'
erro: porta inválida: number too large to fit in target type
erro: linha 2 não está no formato chave=valor
```

O `parse::<u16>()` já recusa 70000 ("*number too large to fit in target type*"), por isso o tipo certo dispensa a verificação manual do limite superior. A linha `if porta == 0` é a única regra que o tipo não cobre; reaproveitar `LinhaInvalida(0)` para ela é um atalho preguiçoso, e uma variante `PortaZero` seria mais honesta. Em projetos reais, a *crate* `thiserror` gera o `Display` e o `From` a partir de atributos, e a `anyhow` simplifica os erros em aplicações.

MÓDULO 4

Coleções, iteradores e closures

Vec, HashMap e HashSet, iteradores, adaptadores e closures.

Artigos 10 a 11

#10 Coleções — Vec, HashMap e HashSet na Prática
#11 Iteradores e Closures — O Estilo Funcional de Rust

Perguntas conceituais — respostas

Q18 iter, iter_mut e into_iter

> *Qual a diferença entre percorrer um `Vec<String>` com `v.iter()`, `v.iter_mut()` e `v.into_iter()` (ou `for x in v`)? O que acontece com `v` depois do último?*

- `iter()` empresta os elementos: cada item é `&String`. O vetor continua intacto.
- `iter_mut()` empresta mutavelmente: cada item é `&mut String`, e pode ser alterado no lugar.
- `into_iter()` **consome** o vetor: cada item é a `String` em si, com a posse. Depois dele, `v` foi movido e não pode mais ser usado.

O `for x in &v` equivale ao `iter()`, o `for x in &mut v` ao `iter_mut()`, e o `for x in v` ao `into_iter()`. Escolher o certo é dizer, no código, se você só olha, se altera ou se fica com os elementos.

Q19 Iteradores preguiçosos e de custo zero

> *Por que `v.iter().map(|x| x * 2);` sozinho não faz nada (e o compilador avisa)? O que significa dizer que os iteradores do Rust são uma "abstração de custo zero"?*

Os adaptadores (`map`, `filter`, `take`...) são **preguiçosos**: só montam a cadeia. Nada é calculado até que um consumidor (`collect`, `sum`, `for`, `count`) peça os itens, e o compilador avisa "*unused `Map` that must be used*". "Custo zero" quer dizer que a cadeia `v.iter().filter(...).map(...).sum()` compila, depois das otimizações, para um código de máquina tão rápido quanto o laço escrito à mão: o compilador funde os passos, elimina as verificações de limite que sabe desnecessárias e, muitas vezes, vetoriza. A abstração é para quem lê; o processador não paga por ela.

Q20 Fn, FnMut e FnOnce

> *Qual a diferença entre os traits `Fn`, `FnMut` e `FnOnce`? Dê uma closure de cada tipo.*

Eles dizem o que a closure faz com o que capturou:

- ** `Fn` : só** lê** as capturas; pode ser chamada várias vezes, inclusive em paralelo. `let soma = |x| x + base;`
- ** `FnMut` : altera** as capturas; pode ser chamada várias vezes, mas exige acesso exclusivo. `let mut conta = || { contador += 1; };`

- ** `FnOnce` : consome** algo capturado; só pode ser chamada uma vez. `let entrega = move || nome;` (devolve a `String`, que deixa de existir na closure).

Toda `Fn` é também `FnMut` e `FnOnce`. Uma função que recebe closures pede o *trait* menos exigente que lhe basta: `thread::spawn` pede `FnOnce`, porque a closure roda uma vez; `sort_by` pede `FnMut`.

Exercícios de código — respostas

Q21 Contando palavras com HashMap

> *Conte a frequência das palavras de um texto (minúsculas, sem pontuação) com um `HashMap<String, usize>` e a API `entry`, mostre as três mais frequentes desempatando em ordem alfabética, e use um `HashSet` para listar as palavras que aparecem só uma vez, em ordem.*

```
use std::collections::{BTreeSet, HashMap};

fn main() {
    let texto = "O rato roeu a roupa do rei. O rei riu, o rato fugiu e a roupa ficou.";
    let mut freq: HashMap<String, usize> = HashMap::new();
    for palavra in texto.split_whitespace() {
        let limpa: String = palavra.chars().filter(|c|
c.is_alphabetic()).flat_map(char::to_lowercase).collect();
        if !limpa.is_empty() {
            *freq.entry(limpa).or_insert(0) += 1;
        }
    }

    let mut ranking: Vec<(&String, &usize)> = freq.iter().collect();
    ranking.sort_by(|a, b| b.1.cmp(a.1).then(a.0.cmp(b.0)));
    for (p, n) in ranking.iter().take(3) {
        println!("{p}: {n}");
    }

    let unicas: BTreeSet<&str> = freq.iter().filter(|(_, n)| **n == 1).map(|(p, _)|
p.as_str()).collect();
    println!("aparecem uma vez: {unicas:?}");
}
```

Saída

```
o: 3
a: 2
rato: 2
aparecem uma vez: {"do", "e", "ficou", "fugiu", "riu", "roeu"}
```

O `entry(chave).or_insert(0)` busca a chave uma vez só e devolve uma referência mutável ao valor, existente ou recém-inserido. O `then` encadeia o critério de desempate. Usei `BTreeSet` em vez de `HashSet` para as palavras únicas porque ele mantém a ordem: a de um `HashSet` muda a cada execução, já que o `HashMap` do Rust usa uma semente aleatória para se proteger de ataques de colisão.

Q22 Uma cadeia de iteradores

Dado um vetor de pedidos (`cliente`, `valor` em centavos, `pago`), use só iteradores, sem laços explícitos, para obter: o total pago; o maior pedido pago (com `max_by_key`); os clientes com pedido pago, sem repetição e em ordem; os pedidos em lotes de 2 (`chunks`); e se algum pedido passa de R$ 150 (`any`).

```
use std::collections::BTreeSet;

struct Pedido {
    cliente: &'static str,
    valor: u32,
    pago: bool,
}

fn main() {
    let pedidos = vec![
        Pedido { cliente: "Carla", valor: 12_000, pago: true },
        Pedido { cliente: "Ana", valor: 8_000, pago: false },
        Pedido { cliente: "Bruno", valor: 20_000, pago: true },
        Pedido { cliente: "Ana", valor: 4_550, pago: true },
    ];

    let pagos = || pedidos.iter().filter(|p| p.pago);
    let total: u32 = pagos().map(|p| p.valor).sum();
    println!("total pago: R$ {},{:02}", total / 100, total % 100);

    if let Some(maior) = pagos().max_by_key(|p| p.valor) {
        println!("maior pago: {} ({})", maior.cliente, maior.valor);
    }

    let clientes: BTreeSet<_> = pagos().map(|p| p.cliente).collect();
    println!("clientes: {clientes:?}");

    for (i, lote) in pedidos.chunks(2).enumerate() {
        let nomes: Vec<_> = lote.iter().map(|p| p.cliente).collect();
        println!("lote {}: {nomes:?}", i + 1);
    }
    println!("algum acima de R$ 150? {}", pedidos.iter().any(|p| p.valor > 15_000));
}
```

Saída

```
total pago: R$ 365,50
maior pago: Bruno (20000)
clientes: {"Ana", "Bruno", "Carla"}
lote 1: ["Carla", "Ana"]
lote 2: ["Bruno", "Ana"]
algum acima de R$ 150? true
```

A closure `pagos` devolve um iterador novo a cada chamada: como os iteradores são preguiçosos e de uso único, guardar um só e consumi-lo três vezes não funcionaria (o primeiro `sum` o esgotaria). Dinheiro em

centavos inteiros (`u32`), formatado só na saída, evita os erros de `f64`. O `collect` decide o tipo do resultado pela anotação: o mesmo iterador vira `Vec`, `BTreeSet`, `HashMap` ou `String`.

Q23 Closures que capturam

Escreva `fn criar_contador(inicio: i32, passo: i32) -> impl FnMut() -> i32`, que devolve uma closure com estado próprio; uma função genérica `fn aplicar_duas_vezes<F: Fn(i32) -> i32>(f: F, x: i32) -> i32`; e mostre o `move` necessário para devolver uma closure que captura uma variável local.

```
fn criar_contador(inicio: i32, passo: i32) -> impl FnMut() -> i32 {
    let mut atual = inicio;
    move || {
        let r = atual;
        atual += passo;
        r
    }
}

fn aplicar_duas_vezes<F: Fn(i32) -> i32>(f: F, x: i32) -> i32 {
    f(f(x))
}

fn somador(n: i32) -> impl Fn(i32) -> i32 {
    move |x| x + n // sem move, a closure tentaria emprestar n, que morre no fim da função
}

fn main() {
    let mut de2 = criar_contador(0, 2);
    let mut de10 = criar_contador(100, -10);
    let seq: Vec<_> = (0..4).map(|_| (de2(), de10())).collect();
    println!("{seq:?}");

    let mais5 = somador(5);
    println!("aplicar_duas_vezes(+5, 1) = {}", aplicar_duas_vezes(&mais5, 1));
    println!("aplicar_duas_vezes(x*3, 2) = {}", aplicar_duas_vezes(|x| x * 3, 2));
}
```

Saída

```
[(0, 100), (2, 90), (4, 80), (6, 70)]
aplicar_duas_vezes(+5, 1) = 11
aplicar_duas_vezes(x*3, 2) = 18
```

Cada closure é um tipo anônimo próprio: `de2` e `de10` têm estados independentes, como dois objetos. O `impl FnMut() -> i32` no retorno diz "algum tipo que implementa isso", sem alocação; para guardar closures de tipos diferentes no mesmo vetor, seria preciso `Box<dyn Fn(i32) -> i32>`. O `&mais5` funciona como argumento porque uma referência a um `Fn` também implementa `Fn`.

MÓDULO 5

Traits, generics e lifetimes

Comportamento compartilhado com traits, código genérico e a anotação de lifetimes.

Artigos 12 a 14

#12 Traits — Definindo Comportamento Compartilhado
#13 Generics — Código que Funciona para Qualquer Tipo
#14 Lifetimes — Quando o Compilador Precisa de Mais Informação

Perguntas conceituais — respostas

Q24 Genérico ou dyn Trait

Compare `fn desenhar<T: Forma>(f: &T)` e `fn desenhar(f: &dyn Forma)`. Quando usar cada um?

- **Genérico** (`T: Forma`, ou `impl Forma`): **despacho estático**. O compilador gera uma cópia da função para cada tipo usado (monomorfização), e cada chamada é direta e pode ser embutida (*inlining*). Mais rápido; executável maior; e um `Vec<T>` só guarda **um** tipo.
- ** `dyn Forma` : despacho dinâmico**, por uma tabela de métodos (*vtable*), como as funções virtuais do C++. Uma cópia só da função, e um `Vec<Box<dyn Forma>>` guarda círculos e quadrados juntos; cada chamada passa por um ponteiro.

Use genéricos por padrão, em bibliotecas e caminhos críticos; use `dyn` quando a coleção é heterogênea, quando o tipo é escolhido em execução (um *plugin*, uma configuração) ou para reduzir o tamanho do binário e o tempo de compilação.

Q25 A regra do órfão

Por que você pode implementar o seu trait `Resumo` para `Vec<i32>`, e o trait `Display` para o seu `struct Livro`, mas não pode implementar `Display` para `Vec<i32>`? Como contornar?

É a **regra do órfão** (*coherence*): uma implementação de *trait* para um tipo só é permitida se o *trait* **ou** o tipo for definido na sua *crate*. `Display` e `Vec` são da biblioteca padrão; se qualquer *crate* pudesse implementar `Display for Vec<i32>`, duas dependências poderiam fazê-lo de formas diferentes, e o compilador não teria como escolher. O contorno é o padrão ***newtype***: `struct Lista(Vec<i32>);`, um tipo seu que envolve o outro, sem custo em execução, para o qual você implementa o que quiser.

Q26 Por que lifetimes

Por que esta função não compila sem a anotação de lifetime? O que `'a` promete?

```
fn maior(a: &str, b: &str) -> &str {
    if a.len() >= b.len() { a } else { b }
}
```

O retorno é uma referência, e o compilador precisa saber **a qual dos parâmetros** ela está ligada, para garantir que ela não sobreviva ao valor emprestado. Como ela pode vir de `a` ou de `b`, ele não consegue deduzir sozinho e pede a anotação: `fn maior<'a>(a: &'a str, b: &'a str) -> &'a str`. O `'a` promete que o resultado vive **no máximo tanto quanto o menor** dos dois empréstimos. Ele não muda quanto tempo nada vive: só descreve a relação, para que o compilador recuse um uso do resultado depois que uma das origens morreu. Nas funções com uma só referência de entrada, as regras de **elisão** fazem a dedução, e a anotação não é necessária.

Exercícios de código — respostas

Q27 **Um trait com método padrão**

*Defina `trait Descrevivel` com `fn nome(&self) -> String` e um método **padrão** `fn descricao(&self) -> String` que use o `nome`. Implemente para `Livro` (sobrescrevendo a descrição) e para `Revista` (usando o padrão). Escreva uma função com `impl Descrevivel` e outra que percorra um `Vec<Box<dyn Descrevivel>>`.*

```
trait Descrevivel {
    fn nome(&self) -> String;
    fn descricao(&self) -> String {
        format!("[{}]", self.nome())
    }
}

struct Livro {
    titulo: String,
    autor: String,
}

struct Revista {
    titulo: String,
    edicao: u32,
}

impl Descrevivel for Livro {
    fn nome(&self) -> String {
        self.titulo.clone()
    }
    fn descricao(&self) -> String {
        format!("{}, de {}", self.titulo, self.autor)
    }
}

impl Descrevivel for Revista {
    fn nome(&self) -> String {
        format!("{} nº {}", self.titulo, self.edicao)
    }
}

fn anunciar(item: &impl Descrevivel) {
    println!("Novidade: {}", item.descricao());
}

fn main() {
    let l = Livro { titulo: "Vidas Secas".into(), autor: "Graciliano Ramos".into() };
    anunciar(&l);

    let estante: Vec<Box<dyn Descrevivel>> = vec![
        Box::new(l),
        Box::new(Revista { titulo: "Ciência Hoje".into(), edicao: 412 }),
    ];
    for item in &estante {
        println!("- {}", item.descricao());
    }
}
```

Saída

```
Novidade: Vidas Secas, de Graciliano Ramos
- Vidas Secas, de Graciliano Ramos
- [Ciência Hoje nº 412]
```

A `Revista` não escreveu `descricao` e herdou a versão padrão, que chama o `nome` dela. O `anunciar` é genérico (despacho estático), e a estante usa `dyn` (despacho dinâmico) porque mistura tipos. O `Box` é necessário porque `Livro` e `Revista` têm tamanhos diferentes, e um vetor precisa de elementos de tamanho fixo: cada `Box` é um ponteiro.

Q28 Uma função genérica com restrições

> *Escreva `fn maior_e_menor<T: PartialOrd + Copy>(itens: &[T]) -> Option<(T, T)>` e uma `struct Par<T>` com um método `maior(&self) -> &T` disponível só quando `T: PartialOrd`. Teste com inteiros, `f64` e caracteres, e com um slice vazio.*

```
fn maior_e_menor<T: PartialOrd + Copy>(itens: &[T]) -> Option<(T, T)> {
    let (&primeiro, resto) = itens.split_first()?;
    let mut menor = primeiro;
    let mut maior = primeiro;
    for &x in resto {
        if x < menor {
            menor = x;
        }
        if x > maior {
            maior = x;
        }
    }
    Some((maior, menor))
}

struct Par<T> {
    a: T,
    b: T,
}

impl<T: PartialOrd> Par<T> {
    fn maior(&self) -> &T {
        if self.a >= self.b { &self.a } else { &self.b }
    }
}

fn main() {
    println!("{:?}", maior_e_menor(&[3, 9, -2, 7]));
    println!("{:?}", maior_e_menor(&[2.5, -0.5, 1.0]));
    println!("{:?}", maior_e_menor(&['r', 'u', 's', 't']));
    println!("{:?}", maior_e_menor::<i32>(&[]));
    println!("maior do par: {}", Par { a: "pera", b: "uva" }.maior());
}
```

Saída

```
Some((9, -2))
Some((2.5, -0.5))
Some(('u', 'r'))
None
maior do par: uva
```

O `?` também funciona com `Option`: o `split_first()` devolve `None` para o slice vazio, e a função devolve `None` sem mais código. O `PartialOrd` (e não o `Ord`) admite `f64`, que tem NaN e por isso não é totalmente ordenado; o `Copy` permite devolver os valores sem clonar. O método `maior` só existe para `Par<T>` cujo `T` seja comparável: um `Par` de um tipo sem ordem continua podendo ser criado, só não tem esse método.

Q29 Uma struct que empresta

Escreva `struct Analisador<'a> { texto: &'a str }` com um método `palavras_longas(&self, minimo: usize) -> Vec<&'a str>` que devolva fatias do texto original, e uma função `fn maior<'a>(a: &'a str, b: &'a str) -> &'a str`. Mostre (em comentário) o uso que o compilador recusa.

```
struct Analisador<'a> {
    texto: &'a str,
}

impl<'a> Analisador<'a> {
    fn palavras_longas(&self, minimo: usize) -> Vec<&'a str> {
        self.texto.split_whitespace().filter(|p| p.chars().count() >= minimo).collect()
    }
}

fn maior<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() >= b.len() { a } else { b }
}

fn main() {
    let texto = String::from("o compilador garante que nenhuma referência sobrevive ao dono");
    let longas;
    {
        let an = Analisador { texto: &texto };
        longas = an.palavras_longas(9);
    } // o Analisador morre aqui, mas as fatias pertencem ao texto, que continua vivo
    println!("{longas:?}");

    let a = String::from("curta");
    let resultado;
    {
        let b = String::from("bem mais longa");
        resultado = maior(&a, &b);
        println!("maior: {resultado}");
    }
    // println!("{resultado}"); // erro: b não vive o bastante (resultado pode ser b)
}
```

Saída

```
["compilador", "referência", "sobrevive"]
maior: bem mais longa
```

O método devolve `Vec<&'a str>`, ligado ao *lifetime* do **texto**, e não ao do `&self`: por isso as fatias sobrevivem ao `Analisador`. Se o retorno fosse `Vec<&str>` sem anotação, a elisão o ligaria ao `&self`, e o código acima não compilaria. No segundo bloco, o compilador não sabe qual referência o `maior` devolveu (isso depende dos dados, em execução) e assume o pior caso: o resultado vive tanto quanto a menor das duas, `b`.

MÓDULO 6

Módulos e testes

Módulos, visibilidade e organização de projetos; testes unitários, de integração e de documentação.

Artigos 15 a 16

#15 Módulos e Organização de Código — Estruturando Projetos Reais
#16 Testes — Escrevendo Código que Prova que Seu Código Funciona

Perguntas conceituais — respostas

Q30 Crate, módulo e visibilidade

Qual a diferença entre crate, package e módulo? O que `pub`, `pub(crate)` e a ausência de modificador significam para uma função dentro de `mod banco`?

Uma ***crate*** é a unidade de compilação: uma biblioteca (`lib.rs`) ou um executável (`main.rs`). Um ***package*** é o que o `Cargo.toml` descreve: uma ou mais *crates* (no máximo uma biblioteca). Os **módulos** organizam o código dentro de uma *crate*, como pastas de *namespaces*, e cada arquivo é um módulo. Sem modificador, um item é **privado** ao módulo em que foi declarado (e visível nos filhos dele); `pub(crate)` o torna visível em toda a *crate*, mas não para quem a usa como dependência; `pub` o torna público, inclusive fora da *crate*, desde que o caminho até ele também seja público. O padrão privado faz da API pública uma decisão explícita.

Q31 Onde ficam os testes

No Rust, por que os testes unitários ficam no mesmo arquivo do código (`#[cfg(test)] mod tests`), e os de integração na pasta `tests/`? O que um teste de integração não consegue acessar?

Os **testes unitários** ficam junto do código para testarem também funções **privadas**: como `mod tests` é um módulo filho, enxerga tudo o que o pai tem. O `#[cfg(test)]` faz o módulo ser compilado só no `cargo test`, sem entrar no binário final. Os **testes de integração**, na pasta `tests/`, são compilados como *crates* separadas que usam a sua biblioteca **como um usuário externo**: só enxergam a API pública. Eles verificam que as partes funcionam juntas pela porta da frente, e não conseguem acessar nada privado. Há ainda os **testes de documentação**: os exemplos dos comentários `///` são compilados e executados pelo `cargo test`, o que garante que a documentação não mente.

Exercícios de código — respostas

Q32 Módulos e testes num arquivo só

Num único `main.rs`, crie o módulo `carrinho` com a `struct Carrinho` pública, um campo privado com os itens, os métodos públicos `novo`, `adicionar` e `total`, e uma função privada `aplicar_desconto` (10% a partir de 3 itens). Escreva, dentro do módulo, um `mod tests` com três testes, inclusive um da função privada e um com `#[should_panic]`. Rode o programa e, separadamente, os testes com `rustc --test`.

```
mod carrinho {
    pub struct Carrinho {
        itens: Vec<(String, u32)>, // privado: só o módulo mexe
    }

    impl Carrinho {
        pub fn novo() -> Self {
            Carrinho { itens: Vec::new() }
        }

        pub fn adicionar(&mut self, titulo: &str, centavos: u32) {
            assert!(centavos > 0, "preço deve ser positivo");
            self.itens.push((titulo.to_string(), centavos));
        }

        pub fn total(&self) -> u32 {
            let bruto = self.itens.iter().map(|(_, c)| c).sum();
            aplicar_desconto(bruto, self.itens.len())
        }
    }

    fn aplicar_desconto(bruto: u32, qtd: usize) -> u32 {
        if qtd >= 3 { bruto - bruto / 10 } else { bruto }
    }

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

        #[test]
        fn sem_desconto_com_dois_itens() {
            let mut c = Carrinho::novo();
            c.adicionar("A", 1000);
            c.adicionar("B", 2000);
            assert_eq!(c.total(), 3000);
        }

        #[test]
        fn desconto_privado_a_partir_de_tres() {
            assert_eq!(aplicar_desconto(10_000, 3), 9_000);
            assert_eq!(aplicar_desconto(10_000, 2), 10_000);
        }

        #[test]
        #[should_panic(expected = "preço deve ser positivo")]
        fn preco_zero_e_recusado() {
            Carrinho::novo().adicionar("grátis", 0);
        }
    }
}

use carrinho::Carrinho;
```

```
fn main() {
    let mut c = Carrinho::novo();
    for (t, p) in [("Vidas Secas", 4990), ("Iracema", 2500), ("Dom Casmurro", 3500)] {
        c.adicionar(t, p);
    }
    println!("total com desconto: {}", c.total());
    // c.itens.clear();              // erro: campo privado
    // carrinho::aplicar_desconto(1, 1); // erro: função privada
}
```

Saída

```
total com desconto: 9891
```

`testar.sh`:

```
rustc --edition 2024 --test main.rs -o testes 2>&1 && ./testes 2>&1 | grep -E '^(test |test result)' | sed -E 's/ finished in [0-9.]+s//' | sort
```

Saída

```
test carrinho::tests::desconto_privado_a_partir_de_tres ... ok
test carrinho::tests::preco_zero_e_recusado - should panic ... ok
test carrinho::tests::sem_desconto_com_dois_itens ... ok
test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
```

O `use super::*` traz para o módulo de testes tudo o que o pai tem, inclusive o privado. O `rustc --test` (ou o `cargo test`, num projeto) compila um executável que roda cada `#[test]` em paralelo e informa os que passaram; o `sort` do exemplo só deixa a ordem previsível, porque a dos testes paralelos varia. O `assert!` com mensagem no `adicionar` é uma **pré-condição**: um preço zero é erro de programação de quem chama, e entrar em pânico é a resposta certa; para dados vindos do usuário, o certo seria devolver um `Result`.

MÓDULO 7

Recursos avançados

Closures avançadas, smart pointers (Box, Rc e RefCell), concorrência com threads e canais, async/await com Tokio e macros.

Artigos 17 a 21

#17 Closures Avançadas e Programação Funcional — Indo Além do map e filter
#18 Smart Pointers — Box, Rc e RefCell
#19 Concorrência — Threads, Canais e a Promessa de Rust de Concorrência Sem Medo
#20 Async/Await — Concorrência sem Threads com Tokio
#21 Macros — Escrevendo Código que Escreve Código

Perguntas conceituais — respostas

Q33 Box, Rc e RefCell

> *Para que serve cada um: `Box<T>`, `Rc<T>` e `RefCell<T>`? Por que `Rc<RefCell<T>>` é uma combinação comum, e qual é o risco dela?*

- ** `Box<T>` **: um valor no *heap* com** um** dono. Serve para tipos recursivos (uma lista ou uma árvore precisa de tamanho conhecido), para objetos grandes que se movem barato e para `dyn Trait`.
- ** `Rc<T>` **: posse** compartilhada** com contagem de referências, numa thread só. O valor vive enquanto houver um `Rc` apontando para ele. Só dá acesso de leitura.
- ** `RefCell<T>` **:** mutabilidade interior**. Permite alterar o valor por uma referência imutável, verificando as regras de empréstimo** em execução**: dois `borrow_mut()` ativos ao mesmo tempo causam pânico.

O `Rc<RefCell<T>>` dá "vários donos que podem alterar", como num grafo de objetos. O risco é duplo: as violações de empréstimo, que antes eram erro de compilação, viram pânico em execução; e dois `Rc` que se apontam formam um **ciclo** que nunca é liberado. O `Weak<T>` quebra o ciclo, como o `weak_ptr` do C++.

Q34 Send e Sync

> *O que significam os traits `Send` e `Sync`? Por que `thread::spawn` recusa uma closure que captura um `Rc<T>`, e aceita uma com `Arc<T>`?*

Um tipo é ** `Send` **se pode ser** transferido **para outra thread, e** `Sync` **se pode ser** compartilhado** entre threads por referência (`&T` é `Send`). O compilador os implementa sozinho para os tipos cujas partes também os implementam. O `Rc<T>` atualiza o contador com operações comuns, não atômicas: duas threads clonando o mesmo `Rc` corromperiam a contagem. Por isso ele **não** é `Send`, e o `spawn`, que exige `Send` na closure, recusa. O `Arc<T>` (*atomic* `Rc`) usa operações atômicas no contador e é `Send` e `Sync` quando `T` é. Para alterar o dado compartilhado, combina-se com `Mutex` ou `RwLock`: `Arc<Mutex<T>>`. É assim que a corrida de dados vira erro de compilação.

Q35 Thread ou async

Quando usar threads e quando usar `async` com Tokio? Por que chamar uma função bloqueante (`std::thread::sleep`, leitura síncrona de arquivo grande) dentro de uma tarefa `async` é um erro?

Threads servem para trabalho de **CPU** em paralelo e para poucas tarefas concorrentes: cada uma tem a sua pilha e é agendada pelo sistema operacional. ***Async*** serve para **muitas** tarefas que passam a maior parte do tempo **esperando E/S** (milhares de conexões de rede): cada tarefa é uma máquina de estados pequena, e um punhado de threads do *runtime* alterna entre elas a cada `.await`. A troca é **cooperativa**: uma tarefa só cede a thread num `.await`. Uma chamada bloqueante prende a thread do *runtime*, e todas as outras tarefas agendadas nela param junto. O certo é usar as versões assíncronas (`tokio::time::sleep`, `tokio::fs`) ou mandar o trabalho bloqueante para `tokio::task::spawn_blocking`.

Q36 macro_rules ou função

Quando uma macro declarativa (`macro_rules!`) é justificada, e quando uma função resolve? Por que `println!` e `vec!` precisam ser macros?

Uma macro opera sobre **código**, na compilação: pode receber um número **variável** de argumentos, de tipos diferentes, gerar itens (funções, `impl`) e receber trechos de sintaxe. O `println!` precisa ser macro porque recebe uma quantidade variável de argumentos e **verifica a string de formato contra eles na compilação**; o `vec![1, 2, 3]` também aceita quantos elementos houver. Quando uma função genérica resolve, ela é melhor: tem tipos claros, mensagens de erro legíveis, aparece na documentação e não surpreende quem lê. As macros procedurais (`#[derive(...)]`) vão além e geram implementações a partir da definição de um tipo.

Exercícios de código — respostas

Q37 Uma expressão recursiva com Box

Defina `enum Expr { Num(f64), Soma(Box<Expr>, Box<Expr>), Mult(Box<Expr>, Box<Expr>), Neg(Box<Expr>) }`, uma função `avaliar(&Expr) -> f64` e uma `mostrar(&Expr) -> String` que imprima com parênteses. Monte e avalie `-(2 + 3) * 4`.

```
enum Expr {
    Num(f64),
    Soma(Box<Expr>, Box<Expr>),
    Mult(Box<Expr>, Box<Expr>),
    Neg(Box<Expr>),
}

use Expr::*;

fn avaliar(e: &Expr) -> f64 {
    match e {
        Num(n) => *n,
        Soma(a, b) => avaliar(a) + avaliar(b),
        Mult(a, b) => avaliar(a) * avaliar(b),
        Neg(a) => -avaliar(a),
    }
}

fn mostrar(e: &Expr) -> String {
    match e {
        Num(n) => n.to_string(),
        Soma(a, b) => format!("({} + {})", mostrar(a), mostrar(b)),
        Mult(a, b) => format!("{} * {}", mostrar(a), mostrar(b)),
        Neg(a) => format!("-{}", mostrar(a)),
    }
}

fn main() {
    let e = Mult(Box::new(Neg(Box::new(Soma(Box::new(Num(2.0)), Box::new(Num(3.0)))))),
Box::new(Num(4.0)));
    println!("{} = {}", mostrar(&e), avaliar(&e));
    println!("tamanho de Expr: {} bytes", std::mem::size_of::<Expr>());
}
```

Saída

```
-(2 + 3) * 4 = -20
tamanho de Expr: 24 bytes
```

Sem o `Box`, o tipo `Expr` conteria a si mesmo diretamente, e o compilador não conseguiria calcular o tamanho ("*recursive type has infinite size*"). Com o `Box`, cada filho é um ponteiro: o `enum` inteiro ocupa o tamanho da maior variante (dois ponteiros) mais a etiqueta. As árvores do módulo 7 do caderno de C, que exigiam `malloc` e `free` à mão, aqui são liberadas sozinhas quando a raiz sai de escopo.

Q38 Uma árvore com Rc, RefCell e Weak

> *Monte uma árvore de categorias (`Livros → Ficção → Romance`) em que cada nó tem `filhos: RefCell<Vec<Rc<No>>>` e `pai: RefCell<Weak<No>>`. Mostre o caminho de um nó até a raiz, as contagens forte e fraca, e que todos os nós são liberados no fim (com um `Drop` que imprime).*

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

struct No {
    nome: String,
    filhos: RefCell<Vec<Rc<No>>>,
    pai: RefCell<Weak<No>>,
}

impl Drop for No {
    fn drop(&mut self) {
        println!("liberado: {}", self.nome);
    }
}

fn novo(nome: &str) -> Rc<No> {
    Rc::new(No { nome: nome.into(), filhos: RefCell::new(vec![]), pai: RefCell::new(Weak::new())
})
}

fn ligar(pai: &Rc<No>, filho: &Rc<No>) {
    *filho.pai.borrow_mut() = Rc::downgrade(pai);
    pai.filhos.borrow_mut().push(Rc::clone(filho));
}

fn main() {
    let raiz = novo("Livros");
    {
        let ficcao = novo("Ficção");
        let romance = novo("Romance");
        ligar(&raiz, &ficcao);
        ligar(&ficcao, &romance);

        let mut caminho = vec![romance.nome.clone()];
        let mut atual = romance.pai.borrow().upgrade();
        while let Some(no) = atual {
            caminho.push(no.nome.clone());
            atual = no.pai.borrow().upgrade();
        }
        println!("caminho: {}", caminho.join(" <- "));
        println!("Ficção: {} fortes, {} fraca(s)", Rc::strong_count(&ficcao),
Rc::weak_count(&ficcao));
    }
    println!("fim do bloco: os nós continuam vivos pela raiz ({} filho)",
raiz.filhos.borrow().len());
}
```

Saída

```
caminho: Romance <- Ficção <- Livros
Ficção: 2 fortes, 1 fraca(s)
fim do bloco: os nós continuam vivos pela raiz (1 filho)
liberado: Livros
liberado: Ficção
liberado: Romance
```

"Ficção" tem dois donos fortes (a variável local e o vetor de filhos da raiz) e uma referência fraca (a do "Romance" para o pai). As referências **fracas** para o pai são o que permite a liberação: se o pai fosse `Rc`, pai e filho se manteriam vivos para sempre, e nenhuma linha "liberado" apareceria. O `upgrade()` devolve `Option<Rc<No>>`: `None` se o pai já tiver sido liberado. A ordem da liberação é de cima para baixo: quando a raiz morre, o vetor dela solta o último dono da "Ficção", e assim por diante.

Q39 Soma em paralelo com scoped threads

Some os números de 1 a 10 milhões dividindo o intervalo entre 4 threads com `std::thread::scope`, que permite às threads emprestar dados da pilha do `main` sem `Arc`. Depois faça um contador compartilhado com `Arc<Mutex<u64>>` e outro com `AtomicU64`, e confira os três resultados.

```rust
use std::sync::atomic::{AtomicU64, Ordering};
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let numeros: Vec<u64> = (1..=10_000_000).collect();
    let partes = 4;
    let tamanho = numeros.len().div_ceil(partes);

    let soma: u64 = thread::scope(|s| {
        let tarefas: Vec<_> = numeros.chunks(tamanho).map(|c| s.spawn(move || c.iter().sum::<u64>
())).collect();
        tarefas.into_iter().map(|t| t.join().unwrap()).sum()
    });
    println!("scope:  {soma}");

    let protegido = Arc::new(Mutex::new(0u64));
    let atomico = Arc::new(AtomicU64::new(0));
    let mut threads = vec![];
    for _ in 0..4 {
        let (p, a) = (Arc::clone(&protegido), Arc::clone(&atomico));
        threads.push(thread::spawn(move || {
            for _ in 0..100_000 {
                *p.lock().unwrap() += 1;
                a.fetch_add(1, Ordering::Relaxed);
            }
        }));
    }
    for t in threads {
        t.join().unwrap();
    }
    println!("mutex:  {}", protegido.lock().unwrap());
    println!("atomic: {}", atomico.load(Ordering::Relaxed));
    println!("fórmula: {}", 10_000_000u64 * 10_000_001 / 2);
}
```

Saída

```
scope:  50000005000000
mutex:  400000
atomic: 400000
fórmula: 50000005000000
```

O `thread::scope` garante que todas as threads terminam antes de ele retornar, e por isso elas podem receber fatias (`&[u64]`) do vetor do `main`: o compilador sabe que o empréstimo não sobrevive aos dados. Com `thread::spawn` comum, a closure precisa ser `'static`, e o vetor teria de ir num `Arc`. O `lock()` devolve um *guard* que destrava o mutex ao sair de escopo, o RAII do Rust; e não há como acessar o dado sem travá-lo, porque ele fica **dentro** do `Mutex`. O `Ordering::Relaxed` basta para um contador; para publicar dados entre threads, é preciso `Release` e `Acquire`.

Q40 Produtor e consumidor com canais

Use `std::sync::mpsc` para que três produtores enviem pedidos (`(produtor, número)`) a um consumidor, que os processa até todos os produtores terminarem. Mostre como o consumidor sabe que acabou e por que o `drop` do transmissor original é necessário.

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel::<(u32, u32)>();

    for produtor in 1..=3 {
        let tx = tx.clone();
        thread::spawn(move || {
            for n in 1..=2 {
                tx.send((produtor, n)).unwrap();
            }
        }); // o clone de tx é liberado aqui
    }
    drop(tx); // sem isso, o rx esperaria para sempre pelo tx original

    let mut recebidos: Vec<_> = rx.iter().collect(); // termina quando todos os tx morrem
    recebidos.sort();
    println!("{} pedidos: {recebidos:?}", recebidos.len());
}
```

Saída

```
6 pedidos: [(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2)]
```

O `rx.iter()` bloqueia esperando mensagens e só termina quando **todos** os transmissores foram liberados: é assim que o consumidor sabe que acabou, sem sentinela. Cada produtor tem o seu clone, mas o `tx` original, criado no `main`, ainda existe até o `drop` explícito. O `sort` é só para a saída ficar previsível, porque a ordem de chegada depende do agendamento das threads. O `send` **move** o valor para o canal: depois de enviado, o produtor não tem mais acesso a ele, e a corrida de dados é impossível por construção. "Não se comunique compartilhando memória; compartilhe memória comunicando-se."

Q41 Tarefas assíncronas com Tokio

Num projeto Cargo com `tokio = { version = "1", features = ["full"] }`, escreva um programa que dispare três "consultas" a fornecedores com tempos diferentes (`tokio::time::sleep`), espere todas com `join!`, e outra rodada em que só vale a primeira que responder (`select!`), com um tempo limite (`timeout`) para a mais lenta.

```rust
use std::time::Duration;
use tokio::time::{sleep, timeout};

async fn consultar(fornecedor: &str, ms: u64, preco: u32) -> (String, u32) {
    sleep(Duration::from_millis(ms)).await;
    (fornecedor.to_string(), preco)
}

#[tokio::main]
async fn main() {
    let (a, b, c) = tokio::join!(
        consultar("A", 300, 4750),
        consultar("B", 100, 4990),
        consultar("C", 200, 4500),
    );
    let melhor = [a, b, c].into_iter().min_by_key(|(_, p)| *p).unwrap();
    println!("todas responderam; menor preço: {} ({})", melhor.0, melhor.1);

    tokio::select! {
        r = consultar("rápido", 50, 5000) => println!("primeiro a responder: {}", r.0),
        r = consultar("lento", 500, 4000) => println!("primeiro a responder: {}", r.0),
    }

    match timeout(Duration::from_millis(150), consultar("muito lento", 1000, 3000)).await {
        Ok((f, _)) => println!("{f} respondeu"),
        Err(_) => println!("tempo esgotado: consulta cancelada"),
    }
}
```

Saída (conferida com o Tokio 1.54):

```
todas responderam; menor preço: C (4500)
primeiro a responder: rápido
tempo esgotado: consulta cancelada
```

O `join!` roda as três consultas **ao mesmo tempo** na mesma tarefa: o tempo total é o da mais lenta (300 ms), e não a soma. O `select!` espera a primeira que terminar e **cancela** as outras, simplesmente parando de fazer o *poll* delas: um *future* do Rust não faz nada sozinho, só avança quando alguém o aguarda. Por isso o cancelamento é gratuito, e por isso um `async fn` chamado sem `.await` não executa nada (o compilador avisa).

Q42 Uma macro declarativa

> *Escreva `macro_rules! mapa` que crie um `HashMap` a partir de `mapa!{ "a" => 1, "b" => 2 }`, aceitando vírgula final, e uma macro `contar!` que conte quantos argumentos recebeu, em tempo de compilação. Use as duas.*

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

macro_rules! mapa {
    ($($chave:expr => $valor:expr),* $(,)?) => {{
        let mut m = HashMap::new();
        $( m.insert($chave, $valor); )*
        m
    }};
}

macro_rules! contar {
    () => { 0usize };
    ($primeiro:tt $($resto:tt)*) => { 1usize + contar!($($resto)*) };
}

fn main() {
    let precos = mapa! {
        "Vidas Secas" => 4990,
        "Iracema" => 2500,
    };
    let mut titulos: Vec<_> = precos.keys().collect();
    titulos.sort();
    println!("{} livros: {titulos:?}", precos.len());

    const N: usize = contar!(a b c d e);
    println!("contar!(a b c d e) = {N}");
}
```

Saída

```
2 livros: ["Iracema", "Vidas Secas"]
contar!(a b c d e) = 5
```

O `$(...),*` repete o padrão para cada par separado por vírgula, e o `$(,)?` aceita uma vírgula opcional no fim, como o `vec!`. As chaves duplas `{{ }}` fazem a macro expandir para **um bloco**, que vale o `m` final. O `contar!` é recursivo: cada nível consome um *token* e soma 1, e como o resultado é uma expressão constante, ele pode inicializar um `const`. O `cargo expand` (uma ferramenta externa) mostra o código que uma macro gerou.

MÓDULO 8

Ecossistema e web

Cargo e as crates essenciais, uma API REST com Axum, banco de dados com SQLx, autenticação JWT e testes avançados.

Artigos 22 a 26

#22 O Ecossistema Rust — Cargo, crates.io e as Bibliotecas Essenciais
#23 Construindo uma API REST com Axum
#24 Banco de Dados com SQLx — Persistência Real para sua API
#25 Autenticação JWT — Protegendo sua API com Tokens
#26 Testes Avançados — Propriedades, Mocks e Benchmarks

Perguntas conceituais — respostas

Q43 Cargo.toml e Cargo.lock

> *O que significa `serde = "1.0"` no `Cargo.toml`? Por que o `Cargo.lock` é versionado num executável e, tradicionalmente, não era numa biblioteca?*

Uma versão sem operador no Cargo é um **requisito compatível** (como o `^` do npm): `"1.0"` aceita qualquer versão `>= 1.0.0` e `< 2.0.0`. O `Cargo.lock` registra as versões **exatas** resolvidas, com os *checksums*, e garante que todos compilam o mesmo código. Num **executável**, ele é versionado sempre: o que foi testado é o que vai para produção. Numa **biblioteca**, o *lockfile* dela é ignorado por quem a usa (vale o do projeto final), e por isso a recomendação antiga era não versioná-lo; desde 2023, a orientação do Cargo é versionar também nas bibliotecas, para que o CI delas seja reprodutível.

Q44 Extractors do Axum

> *No Axum, um handler como `async fn criar(State(db): State<Pool>, Json(novo): Json<NovoLivro>) -> impl IntoResponse` não lê a requisição manualmente. Como os parâmetros são preenchidos, e o que acontece quando o JSON do corpo é inválido?*

Cada parâmetro é um ***extractor***: um tipo que implementa `FromRequestParts` ou `FromRequest`, e o Axum chama a extração de cada um antes de chamar o *handler*. `State` entrega o estado compartilhado registrado no roteador; `Json<T>` lê o corpo, confere o `Content-Type` e desserializa com o `serde`; `Path`, `Query` e `HeaderMap` funcionam igual. Se um *extractor* falha (JSON malformado, campo faltando, tipo errado), o *handler* **nem é chamado**, e a resposta de erro do próprio *extractor* é devolvida (400, 415 ou 422, conforme o caso). O *handler* recebe dados já validados quanto ao formato e só cuida das regras de negócio.

Q45 Consultas verificadas na compilação

> *O que a macro `sqlx::query_as!` faz de diferente de uma consulta montada em texto? Qual a exigência dela para compilar?*

O `query_as!(Livro, "SELECT id, titulo FROM livros WHERE id = $1", id)` **conecta ao banco na compilação** (ou lê um cache gerado com `cargo sqlx prepare`), pede ao banco que analise o SQL e confere:

que a consulta é válida, que as colunas existem, que os tipos delas casam com os campos de `Livro` e com os parâmetros, e se cada coluna pode ser nula (o que exige `Option` no campo). Um nome de coluna errado vira erro de **compilação**, não de execução. E os parâmetros são sempre separados do texto, sem injeção possível. A exigência é ter um banco acessível pela `DATABASE_URL` no momento da compilação, ou o cache `.sqlx/` versionado, que é o que se usa no CI.

Q46 Testes de propriedade

> *O que é um teste baseado em propriedades (com `proptest` ou `quickcheck`), e que propriedade testaria uma função que codifica e decodifica Base64?*

Em vez de escrever exemplos (`codificar("oi") == "b2k="`), descreve-se uma **propriedade** que deve valer para **qualquer** entrada, e a ferramenta gera centenas de entradas aleatórias, inclusive casos extremos (vazio, bytes zero, tamanhos que não são múltiplos de 3), procurando um contraexemplo. Quando acha um, ela o **reduz** (*shrinking*) ao menor caso que ainda falha. Para Base64, a propriedade central é a ida e volta: `decodificar(codificar(bytes)) == bytes` para todo `bytes`; e também que o tamanho da saída é `4 * teto(n / 3)` e que ela só tem caracteres do alfabeto. É uma ótima forma de achar os casos de borda que ninguém pensou em escrever.

Exercícios de código — respostas

Q47 Uma API com Axum

> *Escreva uma API Axum com `GET /livros`, `GET /livros/{id}` (404 quando não existe) e `POST /livros` (201 com o livro criado, ou 422 se o título estiver vazio), guardando os livros num `Arc<Mutex<Vec<Livro>>>` como estado. Use `serde` para o JSON.*

```
use axum::{
    Json, Router,
    extract::{Path, State},
    http::StatusCode,
    routing::get,
};
use serde::{Deserialize, Serialize};
use std::sync::{Arc, Mutex};

#[derive(Clone, Serialize)]
struct Livro {
    id: u32,
    titulo: String,
    preco_centavos: u32,
}

#[derive(Deserialize)]
struct NovoLivro {
    titulo: String,
    preco_centavos: u32,
}

type Estado = Arc<Mutex<Vec<Livro>>>;

async fn listar(State(livros): State<Estado>) -> Json<Vec<Livro>> {
    Json(livros.lock().unwrap().clone())
}

async fn buscar(State(livros): State<Estado>, Path(id): Path<u32>) -> Result<Json<Livro>,
StatusCode> {
    livros.lock().unwrap().iter().find(|l| l.id ==
id).cloned().map(Json).ok_or(StatusCode::NOT_FOUND)
}

async fn criar(
    State(livros): State<Estado>,
    Json(novo): Json<NovoLivro>,
) -> Result<(StatusCode, Json<Livro>), (StatusCode, &'static str)> {
    if novo.titulo.trim().is_empty() {
        return Err((StatusCode::UNPROCESSABLE_ENTITY, "título obrigatório"));
    }
    let mut lista = livros.lock().unwrap();
    let livro = Livro { id: lista.len() as u32 + 1, titulo: novo.titulo, preco_centavos:
novo.preco_centavos };
    lista.push(livro.clone());
    Ok((StatusCode::CREATED, Json(livro)))
}

#[tokio::main]
async fn main() {
    let estado: Estado = Arc::new(Mutex::new(Vec::new()));
    let app = Router::new()
        .route("/livros", get(listar).post(criar))
```

```
        .route("/livros/{id}", get(buscar))
        .with_state(estado);
    let ouvinte = tokio::net::TcpListener::bind("127.0.0.1:3000").await.unwrap();
    axum::serve(ouvinte, app).await.unwrap();
}
```

Dependências: `axum = "0.8"`, `tokio` com `full` e `serde` com `derive`. O Axum 0.8 usa `{id}` nas rotas (antes era `:id`). O `std::sync::Mutex` serve aqui porque nenhuma trava é mantida durante um `.await`; se fosse, seria preciso o `tokio::sync::Mutex`. Um JSON sem o campo `titulo` nem chega ao *handler*: o *extractor* `Json` devolve 422 com a explicação. Num serviço real, o vetor vira um *pool* do SQLx no estado, e o `id` vem do banco. Para este caderno, o código foi compilado com o Axum 0.8.9, sem avisos, mas o servidor não foi executado.

Q48 Um teste de propriedade feito à mão

> *Sem bibliotecas, escreva um gerador pseudoaleatório simples (xorshift), uma função `codificar_rle(&str) -> String` e `decodificar_rle(&str) -> String` (codificação por repetição: "aaab" vira "3a1b"), e teste a propriedade de ida e volta em 1000 textos aleatórios feitos das letras `a`, `b` e `c`. Mostre o que acontece com um texto que contém dígitos, e por quê.*

```
struct Xorshift(u64);

impl Xorshift {
    fn proximo(&mut self) -> u64 {
        self.0 ^= self.0 << 13;
        self.0 ^= self.0 >> 7;
        self.0 ^= self.0 << 17;
        self.0
    }
}

fn codificar_rle(s: &str) -> String {
    let mut saida = String::new();
    let mut chars = s.chars().peekable();
    while let Some(c) = chars.next() {
        let mut n = 1;
        while chars.peek() == Some(&c) {
            chars.next();
            n += 1;
        }
        saida.push_str(&format!("{n}{c}"));
    }
    saida
}

fn decodificar_rle(s: &str) -> String {
    let mut saida = String::new();
    let mut n = 0;
    for c in s.chars() {
        if let Some(d) = c.to_digit(10) {
            n = n * 10 + d as usize;
        } else {
            saida.push_str(&c.to_string().repeat(n));
            n = 0;
        }
    }
    saida
}

fn main() {
    let mut rng = Xorshift(2026);
    let mut falhas = 0;
    for _ in 0..1000 {
        let tamanho = (rng.proximo() % 30) as usize;
        let texto: String = (0..tamanho).map(|_| ['a', 'b', 'c'][(rng.proximo() % 3) as
usize]).collect();
        if decodificar_rle(&codificar_rle(&texto)) != texto {
            falhas += 1;
        }
    }
    println!("1000 textos de a, b e c: {falhas} falha(s)");
    println!("exemplo: {:?} -> {:?}", "aaabccccccccccccd", codificar_rle("aaabccccccccccccd"));
```

```
    let com_digito = "a11";
    let ida_e_volta = decodificar_rle(&codificar_rle(com_digito));
    println!("com dígitos: {com_digito:?} -> {:?} -> {ida_e_volta:?}",
codificar_rle(com_digito));
}
```

Saída

```
1000 textos de a, b e c: 0 falha(s)
exemplo: "aaabccccccccccccd" -> "3a1b12c1d"
com dígitos: "a11" -> "1a21" -> "a"
```

A propriedade vale para o alfabeto testado, inclusive para repetições de dois dígitos (os 12 "c" do exemplo). Com um texto que **contém dígitos**, ela quebra: "a11" vira "1a21", que o decodificador lê como "1 a" e "21 vezes nada". É exatamente o tipo de falha que um teste de propriedade com um gerador mais amplo encontraria sozinho, e que exemplos escritos à mão costumam não cobrir. O gerador com semente fixa torna o teste reprodutível; o `proptest` faz o mesmo e ainda reduz a entrada que falhou ao menor exemplo possível.

MÓDULO 9

Sistemas e baixo nível

WebAssembly, programação de sistemas e de redes, parsing, criptografia, embarcados, unsafe, desempenho, FFI e macros procedurais.

Artigos 27 a 36

#27 WebAssembly com Rust — Executando Rust no Navegador
#28 Programação de Sistemas — Arquivos, Processos e Sinais
#29 Programação de Redes — TCP, UDP e Sockets
#30 Parsing e Processamento de Texto — nom, pest e Expressões Regulares
#31 Criptografia e Segurança — Hashing, Criptografia Simétrica e TLS
#32 Embarcados e Bare Metal — Rust sem Sistema Operacional
#33 Unsafe Rust — Quando e Como Usar Código Inseguro com Responsabilidade
#34 Performance e Otimização — Profiling, SIMD e Técnicas Avançadas
#35 FFI — Integrando Rust com C e Outras Linguagens
#36 Macros Procedurais — Criando Derive Macros e Atributos Customizados

Perguntas conceituais — respostas

Q49 Quando unsafe

> *O que um bloco `unsafe` permite que o Rust seguro não permite? Ele desliga o borrow checker? Qual a responsabilidade de quem escreve uma função segura que usa `unsafe` por dentro?*

O `unsafe` libera **cinco** operações: desreferenciar ponteiros crus, chamar funções `unsafe` (inclusive as de FFI), acessar ou alterar variáveis `static mut`, implementar *traits* `unsafe` (como `Send` e `Sync` à mão) e acessar campos de `union`. Ele **não** desliga o *borrow checker* nem as outras verificações: referências comuns continuam verificadas dentro do bloco. Quem escreve uma função **segura** que usa `unsafe` por dentro garante que **nenhuma** forma de chamá-la, com quaisquer argumentos válidos, causa comportamento indefinido: a função é uma fronteira, e a responsabilidade é de quem a escreve, não de quem chama. Por isso o costume é um comentário `// SAFETY:` explicando por que cada bloco está correto, e manter os blocos pequenos e encapsulados.

Q50 no_std

> *O que muda num programa `#![no_std]` para um microcontrolador? O que deixa de existir, e o que continua disponível?*

Sem o sistema operacional, deixa de existir a biblioteca `std`, que depende dele: alocação no *heap* (`Vec`, `String`, `Box`), arquivos, threads, rede, `println!`. Continua disponível a `core`: tipos primitivos, `Option`, `Result`, iteradores, *traits*, `slice`, formatação (sem destino de saída) e as operações atômicas. Com um alocador fornecido, volta também a `alloc` (`Vec`, `String`, `Box`). O programa precisa definir o próprio `#[panic_handler]` (o que fazer num pânico: parar, reiniciar, piscar um LED) e o ponto de entrada (com o `cortex-m-rt`, `#[entry]`). O *borrow checker* e o sistema de tipos continuam valendo, e é isso que torna o Rust atraente para embarcados: os registradores de periféricos viram tipos que só podem ser usados do jeito certo.

Q51 Um protocolo sobre TCP

> *Por que um servidor TCP não pode assumir que um `read` traz uma mensagem completa? Que duas formas de delimitar mensagens são comuns?*

O TCP entrega um **fluxo de bytes**, sem fronteiras: uma mensagem pode chegar partida em vários `read`, e duas mensagens podem chegar juntas num só. O protocolo da aplicação é que define onde cada mensagem termina. As duas formas comuns são: um **delimitador** (uma quebra de linha, como em HTTP/1, SMTP e Redis em parte), lido com um `BufReader` e `read_line`; ou um **prefixo de tamanho** (os primeiros 4 bytes dizem quantos vêm a seguir, como no gRPC e em muitos protocolos binários), lido com `read_exact`. O delimitador é simples e legível; o prefixo de tamanho suporta dados binários arbitrários e permite alocar o *buffer* certo de uma vez (com um limite máximo, para não aceitar um "tamanho" de 4 GB enviado por um atacante).

Q52 Medir antes de otimizar

> *Que ferramentas o curso usa para achar o gargalo de um programa Rust lento, e por que olhar o código e "adivinhar" costuma falhar?*

Primeiro, compilar em **release** (`cargo build --release`): a diferença para o *debug* chega a dezenas de vezes, e muitos "problemas de desempenho" são só isso. Depois, **medir**: o `criterion` para comparar versões de uma função com estatística (aquecimento, várias amostras, detecção de regressão), e um *profiler* de amostragem (`perf` com o `cargo flamegraph`) para ver **onde** o tempo realmente vai, num gráfico de chamas. Adivinhar falha porque a intuição aponta para o código "complicado", enquanto o tempo costuma estar em lugares inesperados: alocações repetidas num laço, uma cópia de `String` escondida num `clone()`, um `HashMap` com o *hasher* padrão (seguro contra ataques, mas não o mais rápido) ou E/S sem *buffer*.

Exercícios de código — respostas

Q53 Arquivos com buffer e processos

> *Escreva um programa que grave 10 mil linhas num arquivo com `BufWriter`, releia contando as linhas e as que contêm "7" com `BufReader`, e execute um processo externo (`wc -l`) com `std::process::Command`, conferindo que a contagem bate. Trate os erros com `?` e `Box<dyn Error>`.*

```rust
use std::error::Error;
use std::fs::File;
use std::io::{BufRead, BufReader, BufWriter, Write};
use std::process::Command;

fn main() -> Result<(), Box<dyn Error>> {
    {
        let mut saida = BufWriter::new(File::create("dados.txt")?);
        for i in 1..=10_000 {
            writeln!(saida, "linha {i}")?;
        }
    } // o BufWriter descarrega o buffer ao ser liberado

    let mut total = 0;
    let mut com_sete = 0;
    for linha in BufReader::new(File::open("dados.txt")?).lines() {
        let linha = linha?;
        total += 1;
        if linha.contains('7') {
            com_sete += 1;
        }
    }
    println!("lidas: {total}, com o dígito 7: {com_sete}");

    let saida = Command::new("wc").args(["-l", "dados.txt"]).output()?;
    let contagem: usize = String::from_utf8(saida.stdout)?.split_whitespace().next().ok_or("saída
vazia")?.parse()?;
    println!("wc -l: {contagem} -> {}", if contagem == total { "confere" } else { "diferente!"
});

    match File::open("nao-existe.txt") {
        Ok(_) => println!("abriu?"),
        Err(e) => println!("erro esperado: {} ({:?})", e, e.kind()),
    }
    Ok(())
}
```

Saída

```
lidas: 10000, com o dígito 7: 3439
wc -l: 10000 -> confere
erro esperado: No such file or directory (os error 2) (NotFound)
```

Sem o `BufWriter`, cada `writeln!` seria uma chamada ao sistema operacional: 10 mil chamadas em vez de algumas dezenas. O `Box<dyn Error>` aceita qualquer tipo de erro (E/S, UTF-8, conversão de número), e o `?` converte cada um sozinho: é a forma prática para um `main` ou um *script*. O `main` que devolve `Result` imprime o erro e sai com código 1 se algo falhar. A mensagem do erro de arquivo vem do sistema operacional ("*No such file or directory*"); o `kind()` dá a categoria portátil, para o código decidir o que fazer.

Q54 Um servidor TCP de eco por linhas

Com `std::net`, suba um servidor TCP numa porta escolhida pelo sistema (`127.0.0.1:0`), que atenda cada conexão numa thread e responda cada linha recebida em maiúsculas, encerrando a conexão com a linha `sair`. No mesmo programa, conecte dois clientes e mostre as respostas.

```
use std::io::{BufRead, BufReader, Write};
use std::net::{TcpListener, TcpStream};
use std::thread;

fn atender(conexao: TcpStream) -> std::io::Result<()> {
    let mut escrita = conexao.try_clone()?;
    for linha in BufReader::new(conexao).lines() {
        let linha = linha?;
        if linha == "sair" {
            writeln!(escrita, "até mais")?;
            break;
        }
        writeln!(escrita, "{}", linha.to_uppercase())?;
    }
    Ok(())
}

fn cliente(endereco: std::net::SocketAddr, nome: &str, mensagens: &[&str]) ->
std::io::Result<Vec<String>> {
    let mut conexao = TcpStream::connect(endereco)?;
    for m in mensagens {
        writeln!(conexao, "{m}")?;
    }
    writeln!(conexao, "sair")?;
    let respostas = BufReader::new(conexao).lines().collect::<Result<Vec<_>, _>>()?;
    Ok(respostas.into_iter().map(|r| format!("[{nome}] {r}")).collect())
}

fn main() -> std::io::Result<()> {
    let ouvinte = TcpListener::bind("127.0.0.1:0")?;
    let endereco = ouvinte.local_addr()?;
    thread::spawn(move || {
        for conexao in ouvinte.incoming().flatten() {
            thread::spawn(move || atender(conexao));
        }
    });

    for r in cliente(endereco, "ana", &["olá, servidor", "ação"])? {
        println!("{r}");
    }
    for r in cliente(endereco, "bruno", &["rust"])? {
        println!("{r}");
    }
    Ok(())
}
```

Saída

```
[ana] OLÁ, SERVIDOR
[ana] AÇÃO
[ana] até mais
[bruno] RUST
[bruno] até mais
```

A porta 0 pede ao sistema uma porta livre, e o `local_addr()` diz qual foi: o exemplo nunca colide com outro programa. O `to_uppercase` do Rust trata Unicode corretamente ("ação" vira "AÇÃO"), ao contrário do `strtoupper` do PHP visto no outro caderno. O protocolo é delimitado por linha, e o `BufReader` cuida de juntar os pedaços que o TCP entregar. Uma thread por conexão é simples e funciona bem para dezenas de clientes; para milhares, o caminho é o `tokio`, com uma tarefa por conexão.

Q55 split_at_mut com unsafe

> *A biblioteca padrão implementa `split_at_mut`, que divide um slice mutável em duas partes mutáveis. Escreva a sua versão, `dividir_mut(v: &mut [i32], meio: usize) -> (&mut [i32], &mut [i32])`, mostrando por que a versão segura não compila (em comentário) e usando `unsafe` com ponteiros crus e um comentário `SAFETY`. Use-a para alterar as duas metades.*

```
fn dividir_mut(v: &mut [i32], meio: usize) -> (&mut [i32], &mut [i32]) {
    let len = v.len();
    assert!(meio <= len, "meio fora do slice");
    // Versão segura, recusada: dois empréstimos mutáveis do mesmo v ao mesmo tempo.
    // (&mut v[..meio], &mut v[meio..])
    let p = v.as_mut_ptr();
    // SAFETY: as faixas [0, meio) e [meio, len) não se sobrepõem, ficam dentro do slice
    // (garantido pelo assert) e herdam o lifetime de v, então ninguém mais acessa esses
    // elementos enquanto as duas fatias existirem.
    unsafe { (std::slice::from_raw_parts_mut(p, meio),
std::slice::from_raw_parts_mut(p.add(meio), len - meio)) }
}

fn main() {
    let mut v = [1, 2, 3, 4, 5, 6];
    let (esq, dir) = dividir_mut(&mut v, 2);
    esq[0] = 100;
    dir.iter_mut().for_each(|x| *x *= 10);
    println!("{v:?}");
}
```

Saída

```
[100, 2, 30, 40, 50, 60]
```

O compilador recusa a versão segura porque não sabe que as duas faixas são **disjuntas**: para ele, são dois empréstimos mutáveis do mesmo `v`. O programador sabe, e o `unsafe` é a forma de afirmar isso, ficando responsável pela prova. O `assert!` é parte essencial da segurança: sem ele, um `meio` maior que o tamanho criaria fatias fora do *array*, e a função "segura" causaria comportamento indefinido. A ferramenta `cargo miri` executa o código num interpretador que detecta essas violações e é o teste certo para todo código `unsafe`.

Q56 Chamando C com FFI

Declare as funções `strlen` e `abs` da biblioteca C num bloco `unsafe extern "C"` (a sintaxe da edição 2024) e chame-as a partir do Rust, convertendo uma `&str` com `CString`. Mostre o que acontece ao tentar criar uma `CString` com um `\0` no meio.

```
use std::ffi::{CString, c_char, c_int};

unsafe extern "C" {
    fn strlen(s: *const c_char) -> usize;
    safe fn abs(x: c_int) -> c_int; // declarada segura: não recebe ponteiro, não há como usá-la
errado
}

fn main() {
    let texto = "ação";
    let c = CString::new(texto).expect("sem \\0 no meio");
    // SAFETY: c é uma string C válida, terminada em \0, e vive até o fim da chamada.
    let n = unsafe { strlen(c.as_ptr()) };
    println!("strlen(\"{texto}\") = {n} bytes; chars = {}", texto.chars().count());
    println!("abs(-42) = {}", abs(-42));

    match CString::new("meio\0quebrado") {
        Ok(_) => println!("aceitou"),
        Err(e) => println!("recusado: \\0 na posição {}", e.nul_position()),
    }
}
```

Saída

```
strlen("ação") = 6 bytes; chars = 4
abs(-42) = 42
recusado: \0 na posição 4
```

Na edição 2024, os blocos `extern` precisam ser marcados `unsafe`, porque o compilador não tem como conferir se a assinatura declarada bate com a função real: declarar o tipo errado é comportamento indefinido. Cada item pode então ser marcado `safe` quando chamá-lo nunca é perigoso, como o `abs`. A `CString` garante o terminador `\0` e recusa um `\0` no meio, que faria o C enxergar uma string mais curta. O `strlen` conta bytes: os mesmos 6 de "ação" que o caderno de C# mostrou.

MÓDULO 10

Engenharia de projetos

Padrões de projeto em Rust, compilação condicional e features, documentação e doc tests, publicação de crates e ferramentas de linha de comando.

Artigos 37 a 41

#37 Design Patterns em Rust — Adaptando Padrões Clássicos à Linguagem
#38 Compilação Condicional e Feature Flags — Gerenciando Variações do Código
#39 Documentação e Doc Tests — Escrevendo Documentação que Não Fica Desatualizada
#40 Gerenciamento de Dependências e Publicação de Crates — Do Projeto Local ao crates.io
#41 Rust para CLI — Construindo Ferramentas de Linha de Comando Profissionais com clap

Perguntas conceituais — respostas

Q57 Padrões que mudam em Rust

Por que alguns padrões clássicos da orientação a objetos mudam de forma em Rust? Dê o exemplo do Builder e do State (typestate).

O Rust não tem herança de implementação, nem `null`, e tem um sistema de tipos forte com posse: muitos padrões que existem para contornar limitações de outras linguagens ficam diferentes ou desnecessários. O **Builder** é muito usado, porque o Rust não tem argumentos nomeados nem opcionais: `Requisicao::builder().url("...").timeout(5).build()`, com métodos que consomem e devolvem `self`. O **State** ganha uma versão em tempo de compilação, o ***typestate***: cada estado é um **tipo** (`Pedido<Novo>`, `Pedido<Pago>`), e os métodos só existem nos estados em que fazem sentido (`enviar()` só em `Pedido<Pago>`). Uma transição inválida deixa de ser um erro em execução e passa a ser um programa que não compila. O **Visitor** costuma dar lugar a um `match` sobre um `enum`, e o **Strategy**, a uma closure.

Q58 Features no Cargo

*O que são features no `Cargo.toml`, e por que elas devem ser **aditivas**? Dê um exemplo de feature bem e mal projetada.*

São opções de compilação nomeadas: `[features] json = ["dep:serde_json"]` liga uma dependência opcional e o código marcado com `#[cfg(feature = "json")]`. O Cargo **unifica** as *features*: se duas dependências do seu projeto usam a mesma *crate*, uma com `json` e outra sem, ela é compilada **uma vez, com a união** das duas. Por isso uma *feature* só pode **acrescentar** funcionalidade, nunca remover nem mudar o comportamento de algo que já existe. Bem projetada: `json`, que acrescenta funções de serialização. Mal projetada: `sem-std` (que remove coisas) ou `algoritmo-rapido` que troca o resultado de uma função existente; ligada por uma dependência distante, ela quebraria outra parte do programa que contava com o comportamento padrão.

Q59 Documentação que não mente

O que são os doc tests, e por que eles evitam que a documentação fique desatualizada?

Os exemplos em blocos de código dentro dos comentários de documentação (`///`) são **compilados e executados** pelo `cargo test`, como testes. Se a API muda e o exemplo deixa de compilar ou de produzir o resultado afirmado num `assert_eq!`, o teste falha. A documentação passa a ter a mesma garantia que o código: um exemplo errado é um teste vermelho. Linhas que começam com `#` ficam ocultas na documentação renderizada, mas são compiladas, o que permite esconder a preparação (os `use`, a criação de objetos) e mostrar só o essencial. O `cargo doc --open` gera o site da documentação, o mesmo formato do docs.rs.

Exercícios de código — respostas

Q60 **Typestate: transições que não compilam**

Implemente um pedido com typestate: `struct Pedido<E> { id: u32, estado: E }` com os estados `Novo`, `Pago { valor: u32 }` e `Enviado { codigo: String }` como tipos, e os métodos `pagar` (só em `Pedido<Novo>`), `enviar` (só em `Pedido<Pago>`) e `rastreio` (só em `Pedido<Enviado>`). Mostre em comentários as chamadas que o compilador recusa.

```
struct Novo;
struct Pago {
    valor: u32,
}
struct Enviado {
    codigo: String,
}

struct Pedido<E> {
    id: u32,
    estado: E,
}

impl Pedido<Novo> {
    fn novo(id: u32) -> Self {
        Pedido { id, estado: Novo }
    }
    fn pagar(self, valor: u32) -> Pedido<Pago> {
        Pedido { id: self.id, estado: Pago { valor } }
    }
}

impl Pedido<Pago> {
    fn enviar(self, codigo: &str) -> Pedido<Enviado> {
        println!("pedido {} de {} centavos enviado", self.id, self.estado.valor);
        Pedido { id: self.id, estado: Enviado { codigo: codigo.into() } }
    }
}

impl Pedido<Enviado> {
    fn rastreio(&self) -> &str {
        &self.estado.codigo
    }
}

fn main() {
    let p = Pedido::novo(42);
    // p.enviar("BR1");       // erro: Pedido<Novo> não tem o método enviar
    let p = p.pagar(4990);
    // p.pagar(10);           // erro: Pedido<Pago> não tem o método pagar
    let p = p.enviar("BR123");
    println!("rastreio: {}", p.rastreio());
    println!("tamanho de Pedido<Novo>: {} bytes", std::mem::size_of::<Pedido<Novo>>());
}
```

Saída

```
pedido 42 de 4990 centavos enviado
rastreio: BR123
tamanho de Pedido<Novo>: 4 bytes
```

Cada transição **consome** o pedido anterior (`self` por valor) e devolve um tipo novo: não há como guardar um "pedido pago" e enviá-lo duas vezes. O estado `Novo`, sem campos, não ocupa espaço nenhum: um

`Pedido<Novo>` tem só os 4 bytes do `id`. Compare com a máquina de estados com `enum` do módulo 3, que verificava as transições em execução e devolvia `Result`: o *typestate* é mais forte quando o estado é conhecido no código; o `enum` é necessário quando o estado vem de fora, como de um banco de dados.

Q61 Uma CLI sem dependências

Escreva o núcleo de uma ferramenta de linha de comando que conte linhas, palavras e bytes (como o `wc`), com as opções `-l`, `-w`, `-c` e `--help`, analisando os argumentos com um `match` sobre slices e devolvendo `Result` para opções inválidas. Para o exemplo ser reprodutível, passe os argumentos como um vetor e o texto como uma string.

```rust
#[derive(Debug, Default)]
struct Opcoes {
    linhas: bool,
    palavras: bool,
    bytes: bool,
}

fn analisar(args: &[&str]) -> Result<Option<Opcoes>, String> {
    let mut o = Opcoes::default();
    for a in args {
        match *a {
            "-l" => o.linhas = true,
            "-w" => o.palavras = true,
            "-c" => o.bytes = true,
            "-h" | "--help" => return Ok(None),
            outro => return Err(format!("opção desconhecida: {outro} (use --help)")),
        }
    }
    if !(o.linhas || o.palavras || o.bytes) {
        o = Opcoes { linhas: true, palavras: true, bytes: true }; // sem opções: tudo, como o wc
    }
    Ok(Some(o))
}

fn contar(texto: &str, o: &Opcoes) -> String {
    let mut partes = vec![];
    if o.linhas {
        partes.push(texto.lines().count().to_string());
    }
    if o.palavras {
        partes.push(texto.split_whitespace().count().to_string());
    }
    if o.bytes {
        partes.push(texto.len().to_string());
    }
    partes.join(" ")
}

fn main() {
    let texto = "o rato roeu\na roupa do rei\nde Roma\n";
    for args in [vec![], vec!["-l"], vec!["-w", "-c"], vec!["-x"], vec!["--help"]] {
        let rotulo = format!("wc {}", args.join(" "));
        match analisar(&args) {
            Ok(Some(o)) => println!("{rotulo:<10} -> {}", contar(texto, &o)),
            Ok(None) => println!("{rotulo:<10} -> uso: wc [-l] [-w] [-c]"),
            Err(e) => println!("{rotulo:<10} -> erro: {e}"),
        }
    }
}
```

Saída

```
wc          -> 3 9 35
wc -l       -> 3
wc -w -c    -> 9 35
wc -x       -> erro: opção desconhecida: -x (use --help)
wc --help   -> uso: wc [-l] [-w] [-c]
```

Num programa de verdade, os argumentos vêm de `std::env::args().skip(1)`, o texto da entrada padrão ou de arquivos, e o erro sai pela saída de erro com `std::process::exit(2)`. Para ferramentas maiores, o `clap` (com `#[derive(Parser)]`) gera tudo isso a partir de um `struct`, incluindo a ajuda, a validação e o autocompletar do terminal. Separar `analisar` e `contar` da E/S é o que permite testar o núcleo sem processos nem arquivos, como a calculadora dos desafios.

MÓDULO 11

Rust em domínios específicos e o futuro

Ciência de dados, jogos, sistemas distribuídos, compiladores, blockchain, machine learning, segurança, produção, comunidade, o futuro da linguagem e os projetos integradores.

Artigos 42 a 53

#42 Rust para Data Science — Processamento de Dados com Polars e ndarray
#43 Rust para GameDev — Motores de Jogo, ECS e Bevy
#44 Sistemas Distribuídos — Consenso, Mensageria e Tolerância a Falhas
#45 Compiladores e Interpretadores — Construindo uma Linguagem com Rust
#46 Rust e Blockchain — Smart Contracts, Criptografia Aplicada e Web3
#47 Rust e Machine Learning — Inferência, ONNX e Aceleração de Modelos
#48 Segurança e Auditoria — Fuzzing, Análise Estática e Hardening de Código Rust
#49 Rust em Produção — Observabilidade, Métricas e Operação de Sistemas Rust
#50 Contribuindo com o Ecossistema Rust — Open Source, RFCs e a Comunidade
#51 O Futuro do Rust — Async Traits, GATs, Especialização e o Roadmap da Linguagem
#52 Projetos Integradores — Construindo Sistemas Completos do Zero
#53 Epílogo — O que Vem Depois, Como Continuar Crescendo, e a Filosofia do Rust

Perguntas conceituais — respostas

Q62 ECS no Bevy

> *O Bevy organiza os jogos com ECS. Como um sistema do Bevy declara os dados de que precisa, e por que isso permite ao motor rodar sistemas em paralelo com segurança?*

Um sistema é uma função comum cujos parâmetros dizem o que ela acessa: `fn mover(mut q: Query<(&mut Transform, &Velocidade)>, tempo: Res<Time>)` pede, para cada entidade com `Transform` e `Velocidade`, acesso **mutável** ao primeiro e de **leitura** ao segundo, além do recurso global `Time`. Como esse acesso está na **assinatura**, o agendador do Bevy sabe, antes de rodar, quais sistemas leem e escrevem quais componentes. Dois sistemas que só leem `Velocidade`, ou que mexem em componentes diferentes, rodam em paralelo; dois que escrevem no mesmo componente rodam em sequência. São as regras de empréstimo do Rust aplicadas ao agendamento: a corrida de dados entre sistemas fica impossível sem nenhuma trava manual.

Q63 Consenso

> *Num sistema distribuído com três réplicas de um banco, por que é preciso um algoritmo de consenso como o Raft? O que acontece quando a rede separa uma réplica das outras duas?*

Sem consenso, cada réplica poderia aceitar escritas diferentes, e o sistema teria versões conflitantes do mesmo dado. O Raft elege um **líder**, que recebe todas as escritas e as replica; uma escrita só é **confirmada** quando a **maioria** (2 de 3) a gravou. Quando a rede separa uma réplica, o lado com duas réplicas continua tendo maioria: elege ou mantém o líder e segue funcionando. A réplica isolada não consegue maioria, não confirma escritas (e, se era o líder, deixa de ser) e fica parada até a rede voltar, quando recebe o que perdeu. É a troca do teorema CAP: diante de uma partição, o Raft escolhe **consistência** em vez de disponibilidade no

lado minoritário. É também por isso que os grupos têm tamanho ímpar: com 4 réplicas, uma partição de 2 e 2 deixa os dois lados sem maioria.

Q64 **Fuzzing**

> *O que é fuzzing, e por que ele encontra bugs em código Rust "seguro"? Que tipo de função é a melhor candidata?*

Fuzzing é executar uma função milhões de vezes com entradas geradas automaticamente, guiado pela **cobertura**: o *fuzzer* (como o `cargo fuzz`, sobre o libFuzzer) muta as entradas que exercitam caminhos novos do código, procurando uma que cause pânico, estouro, laço infinito ou consumo excessivo de memória. O Rust seguro impede o comportamento indefinido, mas não impede o **pânico** (um índice fora dos limites, um `unwrap` num `None`, um estouro em *debug*) nem os erros de lógica, e um pânico num servidor é uma negação de serviço. As melhores candidatas são as funções que recebem **dados não confiáveis**: *parsers* de formatos de arquivo, de protocolos de rede, de imagens, de configuração. São exatamente as que um atacante alimenta.

Q65 **Como o Rust evolui**

> *O que são as **edições** do Rust (2018, 2021, 2024), e como elas permitem mudar a linguagem sem quebrar o código existente? Qual o papel das RFCs?*

Uma **edição** é um conjunto de mudanças **incompatíveis** que cada *crate* adota quando quiser, declarando `edition = "2024"` no `Cargo.toml`. *Crates* de edições diferentes compilam juntas no mesmo projeto, porque a edição afeta só a análise do código de cada uma; o compilador entende todas. Assim o Rust pôde, por exemplo, transformar `async` em palavra reservada e exigir `unsafe extern` sem quebrar nenhuma biblioteca antiga. O `cargo fix --edition` migra a maior parte do código sozinho. As **RFCs** (*Requests for Comments*) são o processo público de proposta: qualquer mudança relevante na linguagem, na biblioteca ou no Cargo é escrita, discutida abertamente e aprovada pela equipe responsável antes de ser implementada, e geralmente passa um tempo como recurso instável, disponível só no compilador *nightly*.

Exercícios de código — respostas

Q66 **Uma cadeia de blocos com hash**

> *Para entender a ideia por trás de um blockchain, monte uma cadeia de blocos em que cada bloco guarda os dados e o hash do anterior, calculado com o `DefaultHasher` da biblioteca padrão (que **não** é criptográfico: serve só para o exemplo). Escreva `valida(&cadeia) -> Result<(), usize>` e mostre que alterar um bloco do meio invalida a cadeia a partir dele.*

```rust
use std::collections::hash_map::DefaultHasher;
use std::hash::{Hash, Hasher};

#[derive(Hash, Clone)]
struct Bloco {
    indice: u32,
    dados: String,
    hash_anterior: u64,
}

fn hash_de(b: &Bloco) -> u64 {
    let mut h = DefaultHasher::new();
    b.hash(&mut h);
    h.finish()
}

fn encadear(dados: &[&str]) -> Vec<Bloco> {
    let mut cadeia: Vec<Bloco> = vec![];
    for (i, d) in dados.iter().enumerate() {
        let anterior = cadeia.last().map(hash_de).unwrap_or(0);
        cadeia.push(Bloco { indice: i as u32, dados: d.to_string(), hash_anterior: anterior });
    }
    cadeia
}

fn valida(cadeia: &[Bloco]) -> Result<(), usize> {
    for par in cadeia.windows(2) {
        if par[1].hash_anterior != hash_de(&par[0]) {
            return Err(par[1].indice as usize);
        }
    }
    Ok(())
}

fn main() {
    let mut cadeia = encadear(&["Ana paga 10 a Bruno", "Bruno paga 3 a Carla", "Carla paga 1 a
Ana", "Ana paga 2 a Bruno"]);
    println!("cadeia original: {:?}", valida(&cadeia));

    cadeia[1].dados = "Bruno paga 300 a Carla".into(); // adulteração
    match valida(&cadeia) {
        Ok(()) => println!("adulterada: passou?!"),
        Err(i) => println!("adulterada: quebra no bloco {i}, que não reconhece o anterior"),
    }

    let refeita = encadear(&cadeia.iter().map(|b| b.dados.as_str()).collect::<Vec<_>>());
    println!("recalculando todos os hashes: {:?}", valida(&refeita));
}
```

Saída

```
cadeia original: Ok(())
adulterada: quebra no bloco 2, que não reconhece o anterior
recalculando todos os hashes: Ok(())
```

Alterar o bloco 1 muda o *hash* dele, e o bloco 2, que guardava o *hash* antigo, deixa de reconhecê-lo. O atacante pode **recalcular** todos os *hashes* seguintes, como a última linha mostra; o que impede isso num *blockchain* real é a **prova de trabalho** (cada bloco exige um cálculo caro) ou de participação, somada a uma rede de participantes que mantém cópias e só aceita a cadeia com mais trabalho acumulado. O `DefaultHasher` também muda entre versões do Rust e não resiste a ataques: o caso real usa SHA-256 (`sha2`) ou BLAKE3.

Q67 Um interpretador de expressões

*Escreva um analisador de expressões aritméticas com precedência e parênteses, por **descida recursiva**: um tokenizer e as funções `expressao` (soma e subtração), `termo` (multiplicação e divisão) e `fator` (número, negativo ou parênteses), que devolvem `Result<f64, String>`. Avalie quatro expressões, uma com erro.*

```
#[derive(Debug, Clone, Copy, PartialEq)]
enum Token {
    Num(f64),
    Op(char),
}

fn tokenizar(s: &str) -> Result<Vec<Token>, String> {
    let mut tokens = vec![];
    let cs: Vec<char> = s.chars().collect();
    let mut i = 0;
    while i < cs.len() {
        let c = cs[i];
        if c.is_whitespace() {
            i += 1;
        } else if c.is_ascii_digit() || c == '.' {
            let ini = i;
            while i < cs.len() && (cs[i].is_ascii_digit() || cs[i] == '.') {
                i += 1;
            }
            let texto: String = cs[ini..i].iter().collect();
            tokens.push(Token::Num(texto.parse().map_err(|_| format!("número inválido:
{texto}"))?));
        } else if "+-*/()".contains(c) {
            tokens.push(Token::Op(c));
            i += 1;
        } else {
            return Err(format!("caractere inesperado: {c}"));
        }
    }
    Ok(tokens)
}

struct Analisador {
    tokens: Vec<Token>,
    pos: usize,
}

impl Analisador {
    fn espiar(&self) -> Option<Token> {
        self.tokens.get(self.pos).copied()
    }
    fn expressao(&mut self) -> Result<f64, String> {
        let mut v = self.termo()?;
        while let Some(Token::Op(op @ ('+' | '-'))) = self.espiar() {
            self.pos += 1;
            let d = self.termo()?;
            v = if op == '+' { v + d } else { v - d };
        }
        Ok(v)
    }
    fn termo(&mut self) -> Result<f64, String> {
        let mut v = self.fator()?;
        while let Some(Token::Op(op @ ('*' | '/'))) = self.espiar() {
```

```
            self.pos += 1;
            let d = self.fator()?;
            if op == '/' && d == 0.0 {
                return Err("divisão por zero".into());
            }
            v = if op == '*' { v * d } else { v / d };
        }
        Ok(v)
    }
    fn fator(&mut self) -> Result<f64, String> {
        self.pos += 1;
        match self.tokens.get(self.pos - 1).copied() {
            Some(Token::Num(n)) => Ok(n),
            Some(Token::Op('-')) => Ok(-self.fator()?),
            Some(Token::Op('(')) => {
                let v = self.expressao()?;
                if self.espiar() != Some(Token::Op(')')) {
                    return Err("faltou fechar o parêntese".into());
                }
                self.pos += 1;
                Ok(v)
            }
            outro => Err(format!("esperava um número, achei {outro:?}")),
        }
    }
}

fn avaliar(s: &str) -> Result<f64, String> {
    let mut a = Analisador { tokens: tokenizar(s)?, pos: 0 };
    let v = a.expressao()?;
    match a.espiar() {
        None => Ok(v),
        Some(t) => Err(format!("sobrou {t:?} no fim")),
    }
}

fn main() {
    for e in ["2 + 3 * 4", "(2 + 3) * 4", "-(1.5 + 2) * -2", "2 * (3 + ", "7 / (2 - 2)"] {
        match avaliar(e) {
            Ok(v) => println!("{e:<18} = {v}"),
            Err(m) => println!("{e:<18} -> erro: {m}"),
        }
    }
}
```

Saída

```
2 + 3 * 4          = 14
(2 + 3) * 4        = 20
-(1.5 + 2) * -2    = 7
2 * (3 +           -> erro: esperava um número, achei None
7 / (2 - 2)        -> erro: divisão por zero
```

A precedência sai da **estrutura** das funções: `expressao` chama `termo`, que chama `fator`, e por isso a multiplicação é resolvida antes da soma sem nenhuma tabela. Cada regra da gramática vira uma função, o que torna a descida recursiva o jeito mais simples de escrever um *parser* à mão; é a base do interpretador construído no artigo #45. O padrão `op @ ('+' | '-')` casa com um dos dois caracteres e dá nome a ele. Para gramáticas maiores, as bibliotecas `nom` e `pest` do módulo 9 geram a mesma estrutura com menos código.

PARA IR ALÉM

Desafios integradores

Projetos maiores, que juntam vários módulos. Não há uma única resposta certa: o gabarito mostra um caminho.

D1

Uma calculadora de terminal

Escreva uma calculadora de terminal com sete operações: soma, subtração, multiplicação, divisão, potência (expoente inteiro), porcentagem (`15 % 200` é 15% de 200) e raiz quadrada (`raiz 2`). Ela deve: (a) aceitar vírgula ou ponto como separador decimal; (b) guardar o último resultado em `ans`, utilizável na conta seguinte; (c) recusar divisão por zero, raiz de negativo, expoente fracionário, resultado infinito e texto que não é número com uma mensagem clara, ***sem encerrar o programa****; (d) ter os comandos `hist` (histórico) e `sair`. Separe o núcleo (um módulo `calc`, sem E/S, com um `enum` de erros e `Result`) da interface (`main`), e escreva testes para o núcleo.*

Caminho da solução

- **Núcleo**: o módulo `calc` tem a `struct Calculadora` com o `ans` e o método `avaliar(&mut self, linha: &str) -> Result<f64, ErroCalc>`. O `ErroCalc` é um `enum` com uma variante por tipo de erro e implementa `Display`; nenhuma `String` de erro solta, e o `match` da interface não pode esquecer um caso.
- **Números**: `str::parse::<f64>()` depois de trocar a vírgula por ponto, recusando `inf` e `NaN` escritos por extenso (o `parse` os aceita). O resultado é conferido com `is_finite`.
- **Interface**: `stdin().lines()`, um `Vec<String>` de histórico e a vírgula decimal na exibição.
- **Testes**: um `#[cfg(test)] mod tests` dentro do módulo, rodado com `cargo test` (ou `rustc --test`).

```
mod calc {
    use std::fmt;

    #[derive(Debug, PartialEq)]
    pub enum ErroCalc {
        Formato,
        Numero(String),
        Operacao(String),
        DivisaoPorZero,
        RaizNegativa,
        Expoente,
        Estouro,
    }

    impl fmt::Display for ErroCalc {
        fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
            match self {
                ErroCalc::Formato => write!(f, "use \"a op b\" ou \"raiz a\""),
                ErroCalc::Numero(t) => write!(f, "\"{t}\" não é um número"),
                ErroCalc::Operacao(op) => write!(f, "operação desconhecida \"{op}\" (use + - * /
^ % ou raiz)"),
                ErroCalc::DivisaoPorZero => write!(f, "divisão por zero"),
                ErroCalc::RaizNegativa => write!(f, "raiz de número negativo"),
                ErroCalc::Expoente => write!(f, "o expoente precisa ser inteiro"),
                ErroCalc::Estouro => write!(f, "resultado grande demais"),
            }
        }
    }

    #[derive(Default)]
    pub struct Calculadora {
        ans: f64,
    }

    impl Calculadora {
        fn numero(&self, t: &str) -> Result<f64, ErroCalc> {
            if t == "ans" {
                return Ok(self.ans);
            }
            let ok = t.chars().all(|c| c.is_ascii_digit() || matches!(c, ',' | '.' | '-'));
            match t.replace(',', ".").parse::<f64>() {
                Ok(v) if ok => Ok(v),
                _ => Err(ErroCalc::Numero(t.to_string())), // recusa "inf", "NaN" e texto
            }
        }

        pub fn avaliar(&mut self, linha: &str) -> Result<f64, ErroCalc> {
            let p: Vec<&str> = linha.split_whitespace().collect();
            let r = match p.as_slice() {
                ["raiz", x] => {
                    let x = self.numero(x)?;
                    if x < 0.0 { return Err(ErroCalc::RaizNegativa) }
                    x.sqrt()
```

```
                }
                [a, op, b] => {
                    let (a, b) = (self.numero(a)?, self.numero(b)?);
                    match *op {
                        "+" => a + b,
                        "-" => a - b,
                        "*" => a * b,
                        "/" if b == 0.0 => return Err(ErroCalc::DivisaoPorZero),
                        "/" => a / b,
                        "%" => a * b / 100.0,
                        "^" if b.fract() != 0.0 => return Err(ErroCalc::Expoente),
                        "^" => a.powi(b as i32),
                        outro => return Err(ErroCalc::Operacao(outro.to_string())),
                    }
                }
                _ => return Err(ErroCalc::Formato),
            };
            if !r.is_finite() {
                return Err(ErroCalc::Estouro);
            }
            self.ans = r; // só um resultado válido substitui o ans
            Ok(r)
        }
    }

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

        #[test]
        fn usa_o_ans() {
            let mut c = Calculadora::default();
            c.avaliar("2 + 3").unwrap();
            assert_eq!(c.avaliar("ans * 2"), Ok(10.0));
        }

        #[test]
        fn erro_nao_altera_o_ans() {
            let mut c = Calculadora::default();
            c.avaliar("4 * 2").unwrap();
            assert_eq!(c.avaliar("1 / 0"), Err(ErroCalc::DivisaoPorZero));
            assert_eq!(c.avaliar("ans + 0"), Ok(8.0));
        }
    }
}

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

fn com_virgula(v: f64) -> String {
    let s = format!("{v:.15}");
    let s = s.trim_end_matches('0').trim_end_matches('.');
    s.replace('.', ",")
}
```

```
fn main() {
    let mut c = calc::Calculadora::default();
    let mut historico: Vec<String> = vec![];
    println!("Calculadora: + - * / ^ % e raiz. Comandos: hist, sair.");
    let mut linhas = io::stdin().lock().lines();
    loop {
        print!("> ");
        io::stdout().flush().unwrap();
        let Some(Ok(linha)) = linhas.next() else { break };
        match linha.trim() {
            "sair" => break,
            "hist" => historico.iter().enumerate().for_each(|(i, h)| println!("  {}. {h}", i +
1)),
            "" => {}
            conta => match c.avaliar(conta) {
                Ok(r) => {
                    historico.push(format!("{conta} = {}", com_virgula(r)));
                    println!("{}", historico.last().unwrap());
                }
                Err(e) => println!("erro: {e}"),
            },
        }
    }
}
```

Entrada digitada:

```
2 + 3
0,1 + 0,2
ans * 10
10 / 0
2 ^ 0,5
raiz -4
raiz 2
ans * ans
10 / 3
15 % 200
inf + 1
2 ^ 2000
2 $ 3
hist
sair
```

Saída

```
Calculadora: + - * / ^ % e raiz. Comandos: hist, sair.
> 2 + 3 = 5
> 0,1 + 0,2 = 0,3
> ans * 10 = 3
> erro: divisão por zero
> erro: o expoente precisa ser inteiro
> erro: raiz de número negativo
> raiz 2 = 1,414213562373095
> ans * ans = 2
> 10 / 3 = 3,333333333333333
> 15 % 200 = 30
> erro: "inf" não é um número
> erro: resultado grande demais
> erro: operação desconhecida "$" (use + - * / ^ % ou raiz)
>   1. 2 + 3 = 5
  2. 0,1 + 0,2 = 0,3
  3. ans * 10 = 3
  4. raiz 2 = 1,414213562373095
  5. ans * ans = 2
  6. 10 / 3 = 3,333333333333333
  7. 15 % 200 = 30
>
```

Com a entrada redirecionada, o terminal não ecoa o que foi digitado, e cada resposta aparece logo depois do `>`.

`testar.sh`:

```
rustc --edition 2024 --test main.rs -o testes 2>&1 && ./testes 2>&1 | grep -E '^(test |test result)' | sed -E 's/ finished in [0-9.]+s//' | sort
```

Saída

```
test calc::tests::erro_nao_altera_o_ans ... ok
test calc::tests::usa_o_ans ... ok
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out;
```

O `{v:.15}` com os zeros finais removidos esconde o ruído do `f64`, como o `%.15g` dos cadernos de C e de C++: `0,1 + 0,2` guarda 0,30000000000000004. O `match` sobre `p.as_slice()` com padrões de slice (`["raiz", x]`, `[a, op, b]`) analisa o formato da linha numa só expressão. Sem a verificação de caracteres em `numero`, o `parse` aceitaria `inf` como número, e a calculadora faria contas com infinito. O `let ... else` encerra o laço no fim da entrada (ou num erro de leitura) sem um `match` aninhado.

Uma versão gráfica fica como extensão do módulo 9: compilada para **WebAssembly** com o `wasm-bindgen`, a mesma `Calculadora` roda no navegador, ligada a uma página com um campo de texto e um botão; ou, no desktop, com o `egui`, numa janela com o visor e os botões. Nos dois casos, o módulo `calc` não muda uma linha, e os testes dele continuam valendo.

D2

Um servidor chave-valor concorrente

Escreva um servidor TCP que guarde pares chave-valor em memória, com os comandos de texto `SET chave valor`, `GET chave`, `DEL chave` e `KEYS`, uma linha por comando, atendendo vários clientes ao mesmo tempo. Acrescente expiração opcional (`SET chave valor EX 10`), persistência num arquivo de log e testes de integração que conectem clientes reais.

Caminho da solução

- **Protocolo**: uma linha por comando e uma por resposta (`OK`, o valor, `(nil)` ou `ERR mensagem`), como uma versão mínima do Redis. A análise do comando é uma função pura `fn analisar(linha: &str) -> Result<Comando, String>`, com um `enum Comando`, testável sem rede.
- **Concorrência**: com `std`, um `Arc<RwLock<HashMap<String, Entrada>>>` e uma thread por conexão (o servidor de eco do módulo 9); com `tokio`, uma tarefa por conexão. O `RwLock` permite muitos `GET` em paralelo e um `SET` por vez. Para escalar, o mapa pode ser dividido em vários pedaços (*shards*), cada um com a sua trava.
- **Expiração**: cada `Entrada` guarda o valor e um `Option<Instant>`. O `GET` verifica e apaga o que expirou (expiração preguiçosa), e uma thread de fundo varre o mapa de tempos em tempos para não acumular chaves esquecidas.
- **Persistência**: cada `SET` e `DEL` é acrescentado a um arquivo (com `BufWriter` e `flush` a cada comando, ou a cada segundo, uma troca entre segurança e velocidade); ao iniciar, o servidor relê o log. Uma compactação reescreve o log só com o estado atual, num arquivo temporário trocado com `rename`.
- **Testes**: testes unitários do `analisar` e da expiração (com o tempo injetado, não com `sleep`), e testes de integração que sobem o servidor na porta 0, conectam dois clientes e verificam que o `SET` de um aparece no `GET` do outro.

É a mesma ideia do banco chave-valor do caderno de C++, agora com concorrência: o compilador garante que nenhuma thread lê o mapa enquanto outra o altera sem passar pela trava.

Fim do gabarito.
Dominando Rust em 1 Ano · theknowledgeisle.com