Dominando Rust: Do Básico ao Avançado

Prefácio

Rust é uma linguagem de programação de sistemas moderna, focada em segurança, velocidade e concorrência. Desenvolvida pela Mozilla Research e com uma comunidade vibrante e crescente, Rust visa capacitar todos a construir software confiável e eficiente. Este ebook foi concebido para guiar programadores, desde aqueles com alguma experiência em outras linguagens até desenvolvedores experientes, através das nuances e poder da linguagem Rust.

Este material explora os conceitos fundamentais de Rust, como seu sistema de propriedade e empréstimo, que garante segurança de memória sem a necessidade de um coletor de lixo. Serão abordados desde a sintaxe básica, tipos de dados, controle de fluxo, até tópicos mais avançados como genéricos, traits, lifetimes, concorrência, programação assíncrona e macros. Além disso, o ecossistema de Rust, incluindo o gerenciador de pacotes Cargo e crates populares, será explorado.

O objetivo é fornecer uma compreensão abrangente e profunda de Rust, permitindo que o leitor não apenas escreva código Rust, mas também entenda os princípios de design da linguagem e como eles contribuem para a criação de software robusto e de alto desempenho. As informações apresentadas são baseadas em fontes confiáveis, incluindo a documentação oficial de Rust e livros de referência escritos por membros da equipe principal de Rust e especialistas da comunidade.[3]

Capítulo 1: Introdução ao Rust

Este capítulo serve como ponto de partida para a jornada no mundo da programação Rust. Serão abordados os passos iniciais, desde a instalação da linguagem até a criação e execução do primeiro programa. Também será introduzido o Cargo, o sistema de build e gerenciador de pacotes de Rust, essencial para o desenvolvimento de projetos mais complexos.

1.1. Primeiros Passos com Rust

Antes de escrever qualquer código Rust, é necessário instalar as ferramentas da linguagem. O método recomendado para instalar Rust é através do rustup, um instalador de linha de comando para a toolchain Rust.[6]

1.1.1. Instalando Rust

O processo de instalação do Rust é projetado para ser simples e multiplataforma. As

instruções detalhadas para Linux, macOS e Windows podem ser encontradas na documentação oficial.[7] Geralmente, envolve a execução de um script que baixa e instala o compilador Rust (rustc), o gerenciador de pacotes Cargo e a biblioteca padrão.

Após a instalação, é possível verificar se tudo ocorreu corretamente abrindo um terminal e executando:

```
Bash
```

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

Esses comandos exibirão as versões instaladas do compilador Rust e do Cargo, respectivamente.

1.1.2. Escrevendo e Executando um Programa "Olá, Mundo!"

Com Rust instalado, o próximo passo é criar o tradicional programa "Olá, Mundo!". Crie um diretório para o projeto e, dentro dele, um arquivo chamado main.rs. Arquivos fonte Rust sempre terminam com a extensão .rs.[8]

No arquivo main.rs, insira o seguinte código [8]:

```
Rust
```

```
fn main() {
    println!("Olá, Mundo!");
}
```

Para compilar e executar este programa, navegue até o diretório do projeto no terminal e execute os seguintes comandos [8]:

```
Bash
```

```
rustc main.rs
./main # No Linux/macOS
.\main.exe # No Windows
```

Isso compilará o arquivo main.rs em um executável binário (chamado main ou main.exe) e, em seguida, o executará, imprimindo "Olá, Mundo!" no terminal.

1.1.3. Anatomia de um Programa Rust

Analisando o programa "Olá, Mundo!" [8]:

- fn main() {... }: Esta linha define uma função chamada main. A função main é especial; é sempre o primeiro código a ser executado em todo programa Rust executável. As chaves {} marcam o corpo da função.
- println!("Olá, Mundo!");: Esta linha realiza o trabalho de imprimir texto na tela.
  - println! é uma macro Rust. Se fosse uma função, seria chamada como println (sem o !). Macros são uma forma de metaprogramação em Rust que permite escrever código que escreve outro código.
  - "Olá, Mundo!" é uma string passada como argumento para println!.
  - A linha termina com um ponto e vírgula (;), indicando que esta expressão terminou e a próxima pode começar. A maioria das linhas de código Rust termina com um ponto e vírgula.

Rust é uma linguagem compilada *ahead-of-time*, o que significa que a compilação e a execução são etapas separadas. Primeiro, o código é compilado para um executável binário usando rustc, e depois esse executável é executado.[8]

1.2. Olá, Cargo!

Cargo é o sistema de build e gerenciador de pacotes de Rust. Ele lida com muitas tarefas, como construir o código, baixar as bibliotecas das quais o código depende (conhecidas como *crates*) e construir essas bibliotecas.[7]

1.2.1. Criando um Projeto com Cargo

Para criar um novo projeto usando Cargo, use o comando cargo new:

```
Bash
```

```
cargo new ola_cargo
cd ola_cargo
```

Este comando cria um novo diretório chamado ola_cargo com a seguinte estrutura [11]:

- Cargo.toml: O arquivo de configuração do Cargo, no formato TOML (Tom's Obvious, Minimal Language). Ele contém metadados do projeto como nome, versão, edição do Rust e dependências.
- src/: Um diretório contendo os arquivos fonte.
  - src/main.rs: Cargo gera um programa "Olá, Mundo!" automaticamente neste arquivo.

Cargo também inicializa um novo repositório Git por padrão.[11]

1.2.2. Construindo e Executando um Projeto Cargo

Para construir um projeto Cargo, navegue até o diretório raiz do projeto e execute:

```
Bash
```

```
cargo build
```

Este comando cria um arquivo executável em target/debug/ola_cargo (ou target\debug\ola_cargo.exe no Windows).[11]

Para compilar e executar o projeto em um único passo, use:

```
Bash
```

```
cargo run
```

Cargo verificará se algum arquivo foi alterado desde a última compilação e reconstruirá o projeto se necessário antes de executá-lo.[11]

Para verificar rapidamente se o código compila sem produzir um executável, use:

Bash

```
cargo check
```

Este comando é mais rápido que cargo build porque pula a etapa de produção do executável.[11]

1.2.3. Construindo para Release

Quando estiver pronto para lançar o projeto, você pode otimizar a compilação com o comando:

Bash

```
cargo build --release
```

Este comando compila o código com otimizações, o que pode levar mais tempo para compilar, mas resultará em um código que executa mais rapidamente. O executável de release será colocado em target/release/.[11] É crucial usar builds de release para benchmarking de desempenho.

1.2.4. O que é um Crate?

Um crate é a menor unidade de compilação em Rust. Pode ser um crate binário (que produz um executável) ou um crate de biblioteca (que define funcionalidades para serem usadas por outros projetos). O arquivo src/main.rs é a raiz de um crate binário, enquanto src/lib.rs é a raiz de um crate de biblioteca.[12] Um pacote (*package*) é um conjunto de um ou mais crates que fornecem um conjunto de funcionalidades, definido por um arquivo Cargo.toml.[12]

Capítulo 2: Programando um Jogo de Adivinhação

Este capítulo guiará o leitor através da construção de um programa Rust completo: um jogo de adivinhação de números. Este projeto prático demonstrará o uso de vários conceitos fundamentais da linguagem, como variáveis, tipos de dados, recebimento de entrada do usuário, e controle de fluxo.[13]

2.1. Configurando um Novo Projeto

Para iniciar o jogo de adivinhação, um novo projeto Cargo é criado:

```Bash
cargo new jogo_adivinhacao
cd jogo_adivinhacao
```

Cargo gera um arquivo src/main.rs com um programa "Olá, Mundo!" e um Cargo.toml.[13]

2.2. Processando um Palpite

O primeiro passo é permitir que o jogador insira um palpite.

```Rust
use std::io; // Traz a biblioteca de entrada/saída para o escopo

fn main() {
    println!("Adivinhe o número!");
    println!("Por favor, insira seu palpite.");

    let mut palpite = String::new(); // Cria uma string mutável para armazenar o palpite

    io::stdin()
        .read_line(&mut palpite) // Lê a linha da entrada padrão
        .expect("Falha ao ler a linha"); // Lida com possíveis erros

    println!("Você palpitou: {palpite}");
}
```

Neste trecho [13]:

- use std::io; importa o módulo io da biblioteca padrão, necessário para entrada e

saída.

- let mut palpite = String::new(); declara uma variável mutável chamada palpite e a vincula a uma nova instância vazia de String. String::new() é uma função associada que cria uma nova string.
- io::stdin() retorna uma instância de std::io::Stdin, que é um handle para a entrada padrão.
- .read_line(&mut palpite) chama o método read_line no handle de entrada padrão para obter a entrada do usuário. Ele armazena a entrada em palpite e retorna um Result.
- .expect("Falha ao ler a linha") é um método em valores Result. Se o Result for um Err, expect fará o programa travar e exibirá a mensagem fornecida. Se for Ok, expect retornará o valor que Ok está guardando.
- println!("Você palpitou: {palpite}"); imprime a string que o usuário inseriu. As chaves {} são um placeholder.

2.3. Gerando um Número Secreto

O jogo precisa de um número secreto para o jogador adivinhar. Rust ainda não inclui funcionalidade de geração de números aleatórios em sua biblioteca padrão, mas depende de *crates* externas para isso. O crate rand é usado aqui.

Primeiro, adicione o rand como uma dependência no arquivo Cargo.toml:

```
Ini, TOML
```

```ini
[dependencies]
rand = "0.8.5"
```

Cargo baixará e compilará o crate rand e suas dependências.

Agora, o código em src/main.rs é atualizado para gerar um número aleatório:

```
Rust
```

```rust
use std::io;
use rand::Rng; // Traz o trait Rng para o escopo
```

```
fn main() {
    println!("Adivinhe o número!");

    let numero_secreto = rand::thread_rng().gen_range(1..=100); // Gera um número entre 1 e 100 (inclusivo)

    // println!("O número secreto é: {numero_secreto}"); // Para depuração

    println!("Por favor, insira seu palpite.");
    //... (código de leitura do palpite)
}
```

Explicação [13]:

- use rand::Rng; importa o trait Rng, que define métodos que geradores de números aleatórios implementam.
- rand::thread_rng() retorna um gerador de números aleatórios particular que é local para a thread atual de execução e é semeado pelo sistema operacional.
- .gen_range(1..=100) chama o método gen_range no gerador. Este método recebe uma expressão de intervalo e gera um número aleatório dentro desse intervalo. O formato 1..=100 é um intervalo inclusivo.

2.4. Comparando o Palpite com o Número Secreto

Para comparar o palpite do usuário (uma String) com o número secreto (um u32), o palpite precisa ser convertido para um número.

```
Rust
```

```
//... (após ler o palpite)
    let palpite: u32 = palpite.trim().parse().expect("Por favor, digite um número!");
    //... (comparação)
```

Detalhes [13]:

- let palpite: u32 =...: Aqui, Rust permite *sombrear* (shadow) a variável palpite anterior com uma nova. Isso é frequentemente usado para converter um valor de um tipo para outro. A anotação de tipo : u32 é necessária para que parse() saiba

para qual tipo numérico converter.
- palpite.trim(): O método trim() em uma instância de String elimina quaisquer espaços em branco no início e no fim, incluindo o caractere de nova linha (\n ou \r\n) que é adicionado quando o usuário pressiona Enter.
- .parse(): O método parse() em strings converte uma string em algum tipo de número. Como parse() pode falhar (e.g., se a string contiver "abc"), ele retorna um tipo Result.
- .expect("Por favor, digite um número!"): Se parse() retornar um Err (porque não pôde converter a string em um número), expect() fará o programa travar.

Agora, a comparação pode ser feita usando o tipo Ordering e uma expressão match:

Rust

```
use std::cmp::Ordering; // Traz Ordering para o escopo

//... (após converter o palpite para u32)
    println!("Você palpitou: {palpite}");

    match palpite.cmp(&numero_secreto) {
        Ordering::Less => println!("Muito baixo!"),
        Ordering::Greater => println!("Muito alto!"),
        Ordering::Equal => println!("Você acertou!"),
    }
```

Explicação [13]:

- use std::cmp::Ordering; importa o enum Ordering.
- palpite.cmp(&numero_secreto) compara palpite com numero_secreto. O método cmp pode ser chamado em qualquer coisa que possa ser comparada e retorna uma variante do enum Ordering.
- A expressão match decide o que fazer com base na variante de Ordering retornada. Cada "braço" do match consiste em um padrão e o código a ser executado se o valor corresponder a esse padrão.

2.5. Permitindo Múltiplos Palpites com Loops

Para permitir que o jogador faça múltiplos palpites, o código de adivinhação é

envolvido em um loop infinito usando a palavra-chave loop. O loop é interrompido quando o jogador acerta o número.

Rust

```
//...
fn main() {
    //... (geração do número secreto)

    loop { // Adiciona um loop infinito
        println!("Por favor, insira seu palpite.");

        let mut palpite = String::new();
        io::stdin().read_line(&mut palpite).expect("Falha ao ler a linha");

        let palpite: u32 = palpite.trim().parse().expect("Por favor, digite um número!");
        println!("Você palpitou: {palpite}");

        match palpite.cmp(&numero_secreto) {
            Ordering::Less => println!("Muito baixo!"),
            Ordering::Greater => println!("Muito alto!"),
            Ordering::Equal => {
                println!("Você acertou!");
                break; // Sai do loop
            }
        }
    }
}
```

A instrução break; é adicionada ao braço Ordering::Equal para sair do loop quando o jogador ganha.[13]

2.6. Lidando com Entradas Inválidas

Atualmente, se o jogador inserir algo que não seja um número, o programa trava devido ao .expect() na chamada parse(). Para tornar o jogo mais robusto, a entrada inválida deve ser tratada de forma que o jogador possa continuar tentando.

A chamada .expect() é substituída por uma expressão match para lidar com o Result retornado por parse():

Rust

```
//...
    let palpite: u32 = match palpite.trim().parse() {
        Ok(num) => num, // Se parse() for bem-sucedido, retorna o número
        Err(_) => { // Se parse() falhar (entrada não numérica)
            println!("Isso não é um número! Tente novamente.");
            continue; // Pula para a próxima iteração do loop
        }
    };
//...
```

Explicação [13]:

- Se parse() retornar Ok(num), o valor num é usado.
- Se parse() retornar Err(_), uma mensagem de erro é impressa e a palavra-chave continue é usada para ir para a próxima iteração do loop, solicitando outro palpite. O _ é um padrão catchall; neste caso, significa que estamos ignorando todos os valores de erro.

Com essas modificações, o jogo de adivinhação está completo e mais robusto, demonstrando conceitos básicos de Rust em um projeto prático.[13]

Capítulo 3: Conceitos Comuns de Programação

Este capítulo explora conceitos fundamentais encontrados na maioria das linguagens de programação e como eles funcionam em Rust. Serão abordados variáveis, tipos de dados básicos, funções, comentários e fluxo de controle.[14]

3.1. Variáveis e Mutabilidade

Por padrão, as variáveis em Rust são imutáveis. Isso é uma das maneiras pelas quais Rust incentiva a escrita de código seguro e concorrente.[15]

- **Declaração de Variáveis**: Variáveis são declaradas com a palavra-chave let.
  Rust
  ```
  let x = 5;
  ```

- **Imutabilidade**: Uma vez que um valor é vinculado a um nome, esse valor não pode ser alterado.
  Rust
  ```
  let x = 5;
  // x = 6; // Isso causaria um erro de compilação
  ```

- **Mutabilidade**: Para permitir que uma variável seja alterada, a palavra-chave mut deve ser usada na declaração.
  Rust
  ```
  let mut x = 5;
  println!("O valor de x é: {x}");
  x = 6;
  println!("O valor de x é: {x}");
  ```

  O uso de mut sinaliza a intenção de que o valor da variável mudará.[15]
- **Constantes**: Constantes são valores vinculados a um nome que não podem ser alterados. Diferentemente das variáveis imutáveis, constantes:
  - Não podem usar mut.
  - São declaradas com a palavra-chave const.
  - O tipo do valor *deve* ser anotado.
  - Só podem ser definidas com uma expressão constante (valor computável em tempo de compilação).
  - A convenção de nomenclatura é usar letras maiúsculas com underscores (e.g., THREE_HOURS_IN_SECONDS).

  Rust
  ```
  const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
  ```
  Constantes são válidas durante todo o tempo de execução do programa dentro do escopo em que foram declaradas.[15]
- **Sombreamento (Shadowing)**: Rust permite declarar uma nova variável com o mesmo nome de uma variável anterior. A nova variável "sombreia" a anterior. Isso é diferente de marcar uma variável como mut, pois o sombreamento cria uma nova variável e permite alterar o tipo do valor, mantendo a imutabilidade após a transformação.
  Rust
  ```
  let x = 5;
  let x = x + 1; // x agora é 6
  {
      let x = x * 2; // x agora é 12 (dentro deste escopo)
      println!("O valor de x no escopo interno é: {x}");
  ```

```
}
println!("O valor de x é: {x}"); // x volta a ser 6

let espacos = "   ";
let espacos = espacos.len(); // Sombreamento permite mudar o tipo
```

O sombreamento é útil para transformações de valor sem a necessidade de criar nomes diferentes (e.g., espacos_str e espacos_num).[15]

Rust também possui um sistema de tipos estático. Embora anotações de tipo sejam opcionais em muitos casos devido à inferência de tipo do compilador, elas podem ser adicionadas para clareza ou quando a inferência não é possível.[16]

3.2. Tipos de Dados

Todo valor em Rust pertence a um determinado tipo de dados, que informa ao compilador que tipo de dados está sendo especificado para que ele saiba como trabalhar com esses dados. Rust é uma linguagem de tipagem estática, o que significa que deve conhecer os tipos de todas as variáveis em tempo de compilação.[17]

Os tipos de dados em Rust se dividem em duas categorias principais: escalares e compostos.

3.2.1. Tipos Escalares

Um tipo escalar representa um valor único. Rust tem quatro tipos escalares primários: inteiros, números de ponto flutuante, booleanos e caracteres.[17]

- **Tipos Inteiros**: Representam números sem componente fracionário.
  - **Comprimento**: 8-bit, 16-bit, 32-bit, 64-bit, 128-bit, arch (dependente da arquitetura).
  - **Variantes**: i para inteiros com sinal (podem ser negativos) e u para inteiros sem sinal (apenas positivos). Por exemplo, i32 é um inteiro de 32 bits com sinal, e u8 é um inteiro de 8 bits sem sinal. isize e usize dependem da arquitetura do computador (32 ou 64 bits) e são usados principalmente para indexar coleções.
  - **Literais**: Podem ser decimais (98_222), hexadecimais (0xff), octais (0o77), binários (0b1111_0000) e bytes (b'A', apenas para u8). O caractere _ pode ser usado como separador visual (1_000).
  - **Padrão**: O tipo inteiro padrão é i32.
  - **Overflow**: Em modo de depuração, o overflow de inteiros causa pânico. Em modo de release, ocorre *two's complement wrapping* (e.g., u8 com valor 256

se torna 0). A biblioteca padrão oferece métodos para lidar explicitamente com overflow (wrapping_*, checked_*, overflowing_*, saturating_*).[17]

- **Tipos de Ponto Flutuante**: Representam números com casas decimais.
  - **Variantes**: f32 (precisão simples) e f64 (precisão dupla).
  - **Padrão**: O tipo padrão é f64.
  - Todos os tipos de ponto flutuante são com sinal e seguem o padrão IEEE-754.[17]
- **Operações Numéricas**: Rust suporta as operações matemáticas básicas: adição (+), subtração (-), multiplicação (*), divisão (/) e resto (%). A divisão inteira trunca em direção a zero.[17]
- **Tipo Booleano**:
  - Representado pela palavra-chave bool.
  - Pode ter dois valores: true ou false.
  - Ocupa um byte de tamanho.[17]
- **Tipo Caractere**:
  - Representado pela palavra-chave char.
  - Literais char são especificados com aspas simples (e.g., 'z', 'ℤ', '😻').
  - Ocupa quatro bytes e representa um Valor Escalar Unicode, permitindo uma vasta gama de caracteres além do ASCII.[17]
- **Tipo Unidade ()**:
  - Possui apenas um valor, uma tupla vazia ().
  - Não é considerado um tipo composto, pois não contém múltiplos valores.[18]

Tabela: Tipos Escalares Comuns em Rust

Categoria	Tipo	Descrição	Exemplo
Inteiro (sinal)	i8	Inteiro de 8 bits com sinal	-128 a 127
Inteiro (sinal)	i32	Inteiro de 32 bits com sinal (padrão para inteiros)	1000
Inteiro (sinal)	isize	Inteiro com sinal de tamanho de ponteiro (dependente da arquitetura)	some_index
Inteiro (s/sinal)	u8	Inteiro de 8 bits sem	0 a 255

		sinal	
Inteiro (s/sinal)	u32	Inteiro de 32 bits sem sinal	4294967295
Inteiro (s/sinal)	usize	Inteiro sem sinal de tamanho de ponteiro (usado para indexação)	vec.len()
Ponto Flutuante	f32	Número de ponto flutuante de 32 bits (precisão simples)	3.14
Ponto Flutuante	f64	Número de ponto flutuante de 64 bits (precisão dupla, padrão para floats)	2.71828
Booleano	bool	Valor booleano	true, false
Caractere	char	Caractere Unicode de 4 bytes	'a', 'Rust'
Tipo Unidade	()	Representa um valor vazio ou tipo de retorno vazio	()

3.2.2. Tipos Compostos

Tipos compostos podem agrupar múltiplos valores em um único tipo. Rust tem dois tipos compostos primitivos: tuplas e arrays.[17]

- **Tipo Tupla**:
  - Uma forma geral de agrupar um número de valores com uma variedade de tipos em um tipo composto.
  - Têm um comprimento fixo; uma vez declaradas, não podem crescer ou encolher.
  - Criadas escrevendo uma lista de valores separados por vírgulas dentro de parênteses (e.g., (500, 6.4, 1)).
  - Os valores podem ser de tipos diferentes.

    - Podem ser desestruturadas para extrair valores individuais (e.g., let (x, y, z) = tup;).
    - Elementos podem ser acessados por índice usando um ponto (.) (e.g., x.0).
    - A tupla sem valores, (), é o tipo unidade, representando um valor ou retorno vazio.[17]
- **Tipo Array**:
    - Outra forma de coletar múltiplos valores.
    - Todos os elementos de um array devem ter o mesmo tipo.
    - Têm um comprimento fixo.
    - Valores são escritos como uma lista separada por vírgulas dentro de colchetes (e.g., ``).
    - Úteis para dados alocados na pilha ou quando um número fixo de elementos é necessário. São menos flexíveis que o tipo *vector* da biblioteca padrão, que pode crescer ou encolher.
    - O tipo de um array é anotado como `` (e.g., [i32; 5]).
    - Podem ser inicializados com o mesmo valor para todos os elementos (e.g., [3; 5]).
    - Elementos são acessados por índice (e.g., a).
    - O acesso a um índice inválido causa pânico em tempo de execução, garantindo segurança de memória.[17]

Tabela: Tipos Compostos Comuns em Rust

Tipo	Descrição	Exemplo de Declaração	Exemplo de Acesso
Tupla	Coleção de valores de tipos possivelmente diferentes, comprimento fixo.	let tup: (i32, f64, u8) = (500, 6.4, 1);	tup.0
Array	Coleção de valores do mesmo tipo, comprimento fixo.	let a: [i32; 5] = ;	a

3.2.3. Inferência de Tipo e Conversão (Casting)

Rust possui um motor de inferência de tipo inteligente que não apenas considera o valor inicial de uma variável, mas também como ela é usada subsequentemente para

determinar seu tipo.[19] Isso reduz a necessidade de anotações de tipo explícitas.

Rust não realiza conversão implícita de tipo (coerção) entre tipos primitivos. No entanto, a conversão explícita de tipo, conhecida como *casting*, pode ser feita usando a palavra-chave as.[19]

- As regras para conversão entre tipos integrais geralmente seguem as convenções C, mas em Rust, o comportamento de todos os casts entre tipos integrais é bem definido.
- Ao converter para um tipo sem sinal T, T::MAX + 1 é adicionado ou subtraído até que o valor se encaixe.
- Ao converter para um tipo com sinal, o resultado bit a bit é o mesmo que converter primeiro para o tipo sem sinal correspondente; se o bit mais significativo for 1, o valor é negativo.
- Desde Rust 1.45, o cast de float para inteiro é um *saturating cast*: valores que excedem os limites do tipo inteiro de destino são fixados nesses limites.
- Métodos unsafe como to_int_unchecked::<T>() podem ser usados para evitar o custo do saturating cast, mas podem resultar em overflow e valores incorretos.[19]

3.3. Funções

Funções são onipresentes em código Rust. A função main é o ponto de entrada de muitos programas. A palavra-chave fn permite declarar novas funções.[20]

- **Definição de Funções**:
  ```Rust
  fn minha_funcao() {
      println!("Outra função.");
  }
  ```

- **Parâmetros de Função**: Parâmetros são variáveis especiais que fazem parte da assinatura de uma função. É obrigatório declarar o tipo de cada parâmetro.
  ```Rust
  fn outra_funcao(x: i32) {
      println!("O valor de x é: {x}");
  }
  ```

  Múltiplos parâmetros são separados por vírgulas.[20]
- **Corpo da Função (Instruções e Expressões)**: O corpo de uma função é composto por uma série de instruções que opcionalmente terminam com uma expressão.
  - **Instruções (Statements)**: Realizam uma ação, mas não retornam um valor

(e.g., let y = 6;).
    - **Expressões (Expressions)**: Avaliam para um valor resultante (e.g., 5 + 6, uma chamada de função, um bloco {} cujo último item não tem ponto e vírgula). Expressões não incluem ponto e vírgula no final. Adicionar um ponto e vírgula transforma uma expressão em uma instrução.[20]
- **Valores de Retorno de Função**: Funções podem retornar valores. O tipo de retorno é declarado após uma seta (->). Em Rust, o valor de retorno de uma função é sinônimo do valor da expressão final no bloco do corpo de uma função. Pode-se retornar cedo de uma função usando a palavra-chave return.

```Rust
fn cinco() -> i32 {
    5 // Sem ponto e vírgula, então é uma expressão cujo valor é retornado
}
fn mais_um(x: i32) -> i32 {
    x + 1
    // x + 1; // Adicionar um ; aqui causaria um erro de tipo!
}
```

Funções que não retornam um valor explicitamente têm um tipo de retorno () (o tipo unidade).[20]

3.4. Comentários

Comentários são ignorados pelo compilador, mas são úteis para que outras pessoas (e você mesmo no futuro) entendam o código.[22]

- **Comentários de Linha**: Começam com // e vão até o final da linha.

```Rust
// Olá, mundo
let numero_da_sorte = 7; // Estou me sentindo sortudo hoje
```

- **Comentários de Documentação**: Usam /// para itens que seguem ou //! para o item que os contém. Suportam Markdown e são usados para gerar documentação HTML.

```Rust
/// Adiciona um ao número fornecido.
///
/// # Examples
///
/// ```
/// let arg = 5;
/// let answer = meu_crate::add_one(arg);
///
```

```
/// assert_eq!(6, answer);
/// ```
pub fn add_one(x: i32) -> i32 {
    x + 1
}
```

Seções comuns em comentários de documentação incluem # Examples, # Panics, # Errors e # Safety.[22]

3.5. Controle de Fluxo

A capacidade de executar código com base em condições ou repetir código em um loop é fundamental.

- **Expressões if**: Permitem ramificar o código com base em condições. A condição *deve* ser um bool.

  ```Rust
  let numero = 3;
  if numero < 5 {
      println!("condição era verdadeira");
  } else {
      println!("condição era falsa");
  }
  ```

  Pode-se usar else if para múltiplas condições. Como if é uma expressão, seu resultado pode ser atribuído a uma variável let, desde que todos os blocos if e else retornem o mesmo tipo.[24]
- **Repetição com Loops**: Rust tem três tipos de loops: loop, while e for.
  - **loop**: Executa um bloco de código repetidamente até que seja explicitamente instruído a parar com break. Pode retornar um valor do loop usando break valor;. Rótulos de loop ('nome_do_rotulo: loop) podem ser usados com break ou continue para especificar a qual loop essas palavras-chave se aplicam quando há loops aninhados.[24]
  - **while**: Um loop condicional que continua enquanto uma condição for true.

    ```Rust
    let mut numero = 3;
    while numero!= 0 {
        println!("{numero}!");
        numero -= 1;
    }
    println!("LIFTOFF!!!");
    ```

- **for**: Usado para iterar sobre os elementos de uma coleção (como um array ou um range). É mais conciso e seguro do que um loop while com um contador manual.

```Rust
let a = ;
for elemento in a {
    println!("o valor é: {elemento}");
}
// Para um loop com contagem regressiva:
for numero in (1..4).rev() { // (1..4) é um Range,.rev() o inverte
    println!("{numero}!");
}
```

  O loop for é o mais comumente usado em Rust devido à sua segurança e concisão.[24]
- **Operador de Controle de Fluxo match**: (Abordado em detalhes no Capítulo 6) match permite comparar um valor com uma série de padrões e executar código com base no padrão que corresponde. Deve ser exaustivo, cobrindo todas as possibilidades.[25]

Capítulo 4: Entendendo o Ownership

Ownership é o recurso mais exclusivo de Rust e tem implicações profundas na linguagem. Ele permite que Rust faça garantias de segurança de memória sem precisar de um coletor de lixo.[28]

4.1. O que é Ownership?

O sistema de ownership consiste em um conjunto de regras que o compilador verifica. Se alguma das regras for violada, o programa não compilará. Nenhuma das funcionalidades de ownership retarda o programa enquanto ele está em execução.[28]

4.1.1. Regras de Ownership

1. Cada valor em Rust tem um *dono* (owner).
2. Só pode haver um dono por vez.
3. Quando o dono sai do escopo, o valor será *descartado* (dropped).[28]

4.1.2. Escopo da Variável

Um escopo é o intervalo dentro de um programa para o qual um item é válido. Uma variável é válida desde o ponto em que é declarada até o final do escopo atual.[28]

4.1.3. O Tipo String

Para ilustrar as regras de ownership, o tipo String é frequentemente usado. String gerencia dados alocados no heap e pode armazenar uma quantidade de texto desconhecida em tempo de compilação. É diferente de literais de string, que são imutáveis e embutidos no binário.[28]

Rust

```
let s = String::from("olá"); // 's' é válido a partir deste ponto
// faz coisas com s
// este escopo agora acabou, e s não é mais válido
```

4.1.4. Memória e Alocação

- **Stack e Heap**: O stack armazena valores em ordem LIFO (último a entrar, primeiro a sair) e todos os dados no stack devem ter um tamanho conhecido e fixo. O heap é menos organizado e é usado para dados cujo tamanho é desconhecido em tempo de compilação ou que pode mudar. Alocar no heap é mais lento do que no stack.[28]
- Quando String::from é chamado, a memória para a string é alocada no heap.
- Rust retorna automaticamente a memória ao alocador quando a variável que a possui sai do escopo. Uma função especial chamada drop é chamada automaticamente pela Rust nesse ponto.[28]

4.1.5. Variáveis e Dados Interagindo com Move

Quando se atribui uma variável String a outra, os dados do ponteiro (ponteiro, comprimento, capacidade) no stack são copiados, mas os dados no heap aos quais o ponteiro se refere *não* são copiados. Em vez disso, Rust considera a primeira variável como não mais válida para evitar um erro de *double free* (quando ambas as variáveis tentam liberar a mesma memória ao saírem do escopo). Isso é chamado de *move*.

Rust

```
let s1 = String::from("olá");
let s2 = s1; // s1 é movido para s2
```

```
// println!("{}, mundo!", s1); // Isso causaria um erro, pois s1 não é mais válido
```

Rust nunca criará automaticamente cópias "profundas" de dados, então qualquer cópia automática é barata em termos de desempenho em tempo de execução.[28]

4.1.6. Variáveis e Dados Interagindo com Clone

Se uma cópia profunda dos dados do heap de uma String for desejada, o método clone pode ser usado.

Rust

```
let s1 = String::from("olá");
let s2 = s1.clone(); // s2 é uma cópia profunda de s1
println!("s1 = {}, s2 = {}", s1, s2);
```

O método clone pode ser uma operação cara.[28]

4.1.7. Dados Apenas no Stack: Copy

Tipos como inteiros, que têm um tamanho conhecido em tempo de compilação e são armazenados inteiramente no stack, implementam o trait Copy. Quando se atribui uma variável de um tipo Copy a outra, uma cópia do valor é feita, e ambas as variáveis permanecem válidas.
Tipos que implementam Copy:

- Todos os tipos inteiros (u32, i64, etc.).
- O tipo booleano (bool).
- Todos os tipos de ponto flutuante (f32, f64).
- O tipo caractere (char).
- Tuplas, se todos os seus elementos implementarem Copy. Um tipo não pode implementar Copy se ele ou qualquer uma de suas partes implementou o trait Drop.[28]

4.1.8. Ownership e Funções

Passar um valor para uma função funciona de forma semelhante à atribuição a uma variável. Passar uma String para uma função move a ownership, e a variável original não pode mais ser usada. Passar um tipo Copy copia o valor.

Rust

```
fn main() {
    let s = String::from("olá");  // s entra no escopo
    toma_ownership(s);            // O valor de s é movido para a função...
                                  //... e não é mais válido aqui
    // println!("{s}"); // Erro! s foi movido

    let x = 5;                    // x entra no escopo
    faz_copia(x);                 // x seria movido para a função,
                                  // mas i32 é Copy, então está tudo bem usar x depois
    println!("{x}"); // Funciona!
}
fn toma_ownership(alguma_string: String) { // alguma_string entra no escopo
    println!("{}", alguma_string);
} // Aqui, alguma_string sai do escopo e `drop` é chamado. A memória de apoio é liberada.

fn faz_copia(algum_inteiro: i32) { // algum_inteiro entra no escopo
    println!("{}", algum_inteiro);
} // Aqui, algum_inteiro sai do escopo. Nada de especial acontece.
```

.[28]

4.1.9. Valores de Retorno e Escopo

Retornar valores de funções também transfere ownership.

Rust

```
fn main() {
    let s1 = da_ownership();        // da_ownership move seu valor de retorno para s1
    let s2 = String::from("olá");   // s2 entra no escopo
    let s3 = pega_e_devolve(s2); // s2 é movido para pega_e_devolve,
                                 // que também move seu valor de retorno para s3
} // Aqui, s3 sai do escopo e é descartado. s2 saiu do escopo, mas foi movido, então nada
```

```
// acontece. s1 sai do escopo e é descartado.

fn da_ownership() -> String {            // da_ownership moverá seu
                                         // valor de retorno para a função
                                         // que a chama
    let alguma_string = String::from("sua"); // alguma_string entra no escopo
    alguma_string                        // alguma_string é retornada e
                                         // movida para fora para a função chamadora
}

// Esta função pegará uma String e a retornará
fn pega_e_devolve(uma_string: String) -> String { // uma_string entra no escopo
    uma_string  // uma_string é retornada e movida para fora para a função chamadora
}
```

A ownership de uma variável sempre segue o mesmo padrão: atribuir um valor a outra variável o move. Quando uma variável que inclui dados no heap sai do escopo, o valor será limpo por drop, a menos que a ownership dos dados tenha sido movida para outra variável.[28]

Embora isso funcione, ter que retornar a ownership junto com quaisquer outros dados que uma função retorna pode ser tedioso. Rust oferece uma alternativa para isso através de *referências*.

4.2. Referências e Empréstimo (Borrowing)

Uma referência é como um ponteiro, pois é um endereço que se pode seguir para acessar dados armazenados nesse endereço; esses dados são de propriedade de alguma outra variável. Ao contrário de um ponteiro, uma referência garante apontar para um valor válido de um tipo particular durante sua vida útil. O ato de criar uma referência é chamado de *empréstimo* (borrowing).[29]

- **Referências Imutáveis**: Criadas com &. Permitem ler os dados, mas não modificá-los.
  Rust
  ```
  fn main() {
      let s1 = String::from("olá");
      let len = calcula_tamanho(&s1); // &s1 cria uma referência para s1
      println!("O tamanho de '{}' é {}.", s1, len);
  }
  ```

```
fn calcula_tamanho(s: &String) -> usize { // s é uma referência para uma String
    s.len()
} // Aqui, s sai do escopo. Mas como não possui ownership do que referencia,
  // nada acontece.
```

O escopo no qual a variável s na função calcula_tamanho é válida é o mesmo que qualquer escopo de parâmetro de função, mas o valor apontado pela referência não é descartado quando a referência para de ser usada porque a função não tem ownership.[29]

- **Referências Mutáveis**: Criadas com &mut. Permitem modificar os dados emprestados.

```
Rust
fn main() {
    let mut s = String::from("olá"); // s deve ser mutável
    altera(&mut s);
}

fn altera(alguma_string: &mut String) {
    alguma_string.push_str(", mundo");
}
```

Há uma grande restrição com referências mutáveis: se você tem uma referência mutável para um valor, não pode ter outras referências (mutáveis ou imutáveis) para esse mesmo valor simultaneamente. Esta regra é imposta pelo compilador Rust para prevenir *data races* (corridas de dados) em tempo de compilação.[29]

  - **Regra N-Leitores 1-Escritor**: Em qualquer momento, você pode ter *ou* uma referência mutável *ou* qualquer número de referências imutáveis para um dado específico.[29]
  - O escopo de uma referência começa onde ela é introduzida e continua até o último uso dessa referência. Isso permite múltiplas referências mutáveis ou uma mistura de referências mutáveis e imutáveis, desde que seus escopos não se sobreponham.[29]
- **Referências Pendentes (Dangling References)**: Em linguagens com ponteiros, é fácil criar um ponteiro pendente — um ponteiro que referencia uma localização na memória que pode ter sido dada a outra pessoa — ao liberar alguma memória enquanto preserva um ponteiro para essa memória. Em Rust, o compilador garante que os dados não sairão do escopo antes que a referência aos dados saia.

```
Rust
// fn dangle() -> &String { // dangle retorna uma referência para uma String
```

```
//     let s = String::from("olá"); // s é uma nova String
//     &s // retornamos uma referência para a String, s
// } // Aqui, s sai do escopo e é descartada. Sua memória se foi.
  // Perigo!
```

Este código não compila porque s é criado dentro de dangle, e quando dangle termina, s é desalocado. A referência retornada apontaria para uma String inválida. A solução é retornar a String diretamente, transferindo ownership.[29]

As regras de empréstimo são uma parte fundamental do que torna Rust seguro e eficiente, prevenindo uma classe inteira de bugs em tempo de compilação.

4.3. Slices

Slices permitem referenciar uma sequência contígua de elementos em uma coleção, em vez da coleção inteira. Um slice é um tipo de referência, então não tem ownership.[32]

- **String Slices (&str)**: Uma referência a uma parte de uma String.
  Rust
  ```
  let s = String::from("olá mundo");
  let ola = &s[0..5]; // ola se refere a "olá"
  let mundo = &s[6..11]; // mundo se refere a "mundo"
  ```

  A sintaxe de intervalo .. é usada: [indice_inicial..indice_final]. indice_final é um a mais que a última posição no slice. Internamente, a estrutura de dados do slice armazena a posição inicial e o comprimento do slice.
  Omissões na sintaxe de range:
  - &s[..2] é o mesmo que &s[0..2].
  - &s[3..] é o mesmo que &s[3..len].
  - &s[..] é um slice da string inteira. Os índices de range de string slices *devem* ocorrer em limites de caracteres UTF-8 válidos. Tentar criar um slice no meio de um caractere multibyte causará pânico.[32] Literais de string ("Olá, mundo!") são inerentemente string slices (&str) e são imutáveis.[32] Usar &str como tipo de parâmetro em funções torna a API mais geral, aceitando tanto &String quanto &str devido a *deref coercions*.[32]
- **Outros Slices (Array Slices)**: Slices podem referenciar partes de outros tipos de coleções, como arrays.
  Rust
  ```
  let a = ;
  let slice = &a[1..3]; // slice se refere a [1, 2]
  assert_eq!(slice, &);
  ```

Este tipo de slice tem o tipo & (e.g., &[i32]). Funciona da mesma forma que string slices, armazenando uma referência ao primeiro elemento e um comprimento.[32]

4.4. O Trait Drop e RAII

Em Rust, o padrão RAII (Resource Acquisition Is Initialization) é fundamental. Variáveis não apenas contêm dados, mas também *possuem* recursos. Quando um objeto sai do escopo, seu destrutor é chamado automaticamente, e seus recursos possuídos são liberados.[33] Isso é implementado através do trait Drop.

Rust

```
struct ToDrop;

impl Drop for ToDrop {
    fn drop(&mut self) {
        println!("ToDrop está sendo descartado");
    }
}

fn main() {
    let x = ToDrop;
    println!("Criado um ToDrop!");
} // x sai do escopo aqui, e `drop` é chamado.
```

Este mecanismo previne vazamentos de recursos e elimina a necessidade de liberação manual de memória.[33]

4.5. Lifetimes (Introdução)

Lifetimes são um construto que o compilador (ou mais especificamente, seu borrow checker) usa para garantir que todos os empréstimos sejam válidos. Essencialmente, o lifetime de uma variável é o escopo para o qual ela é válida.
Quando uma variável é emprestada via &, o empréstimo tem um lifetime determinado por onde é declarado. O empréstimo é válido desde que termine antes que o proprietário seja destruído.34

Rust

```
fn main() {
    let i = 3; // Lifetime para `i` começa.
    {
        let borrow1 = &i; // Lifetime de `borrow1` começa.
        println!("borrow1: {}", borrow1);
    } // `borrow1` termina.
    {
        let borrow2 = &i; // Lifetime de `borrow2` começa.
        println!("borrow2: {}", borrow2);
    } // `borrow2` termina.
}   // Lifetime de `i` termina.
```

Neste exemplo, i tem o lifetime mais longo. Os lifetimes de borrow1 e borrow2 são contidos dentro do lifetime de i. Lifetimes são frequentemente inferidos, mas podem precisar de anotações explícitas em casos mais complexos, que serão discutidos posteriormente (Capítulo 10).[34]

Capítulo 5: Usando Structs para Estruturar Dados Relacionados

Structs são tipos de dados customizados que permitem nomear e agrupar múltiplos valores relacionados que compõem um todo significativo. São semelhantes às tuplas, mas cada parte de dados em uma struct é nomeada, tornando seu significado claro.[35]

5.1. Definindo e Instanciando Structs

Structs são definidas usando a palavra-chave struct seguida do nome da struct e, entre chaves, os nomes e tipos de seus campos.

Rust

```
struct Usuario {
    ativo: bool,
    username: String,
    email: String,
    contador_de_acessos: u64,
```

```
}

fn main() {
    let usuario1 = Usuario {
        ativo: true,
        username: String::from("algumusername123"),
        email: String::from("alguem@exemplo.com"),
        contador_de_acessos: 1,
    };
    // Acessando campos
    println!("Email do usuário: {}", usuario1.email);
}
```

Para criar uma instância, especifica-se o nome da struct e pares chave: valor para cada campo. A ordem não precisa corresponder à declaração. Se a instância for mutável (let mut), seus campos podem ser alterados usando notação de ponto.[35]

Funções podem retornar instâncias de structs. Se os nomes dos parâmetros da função e os campos da struct forem os mesmos, a *abreviação de inicialização de campo* pode ser usada:

```
    Rust


fn build_user(email: String, username: String) -> Usuario {
    Usuario {
        ativo: true,
        username, // Abreviação para username: username
        email,    // Abreviação para email: email
        contador_de_acessos: 1,
    }
}
```

.[35]

5.2. Sintaxe de Atualização de Struct

A sintaxe de atualização de struct permite criar uma nova instância usando a maioria dos valores de uma instância existente, alterando apenas alguns campos. A sintaxe ..

especifica que os campos restantes devem ter o mesmo valor que os campos da instância fornecida.

Rust

```
let usuario2 = Usuario {
    email: String::from("outro@exemplo.com"),
    ..usuario1 // Campos restantes de usuario1 (username, ativo, contador_de_acessos)
};
```

A sintaxe ..instancia deve vir por último. Esta operação move dados. Se campos como String são movidos, a instância original pode não ser mais utilizável para esses campos. Campos com tipos que implementam Copy são copiados.[35]

5.3. Usando Tuple Structs sem Campos Nomeados para Criar Tipos Diferentes

Tuple structs são similares a tuplas, mas têm um nome. Não possuem campos nomeados, apenas os tipos de seus campos. São úteis para dar um nome distinto a uma tupla.

Rust

```
struct Cor(i32, i32, i32);
struct Ponto(i32, i32, i32);

fn main() {
    let preto = Cor(0, 0, 0);
    let origem = Ponto(0, 0, 0);
    // preto.0 acessa o primeiro elemento
}
```

Mesmo que os campos tenham os mesmos tipos, Cor e Ponto são tipos diferentes.[35]

5.4. Unit-Like Structs sem Campos

Structs sem campos são chamadas de *unit-like structs* porque se comportam de

forma similar a (), o tipo unidade. São úteis quando se precisa implementar um trait em um tipo, mas não há dados para armazenar.

Rust

```
struct SempreIgual;

fn main() {
    let sujeito = SempreIgual;
}
```

.[35]

5.5. Ownership de Dados de Struct

É importante que instâncias de struct possuam todos os seus dados para que esses dados sejam válidos enquanto a struct for válida. Por isso, é comum usar tipos que possuem ownership, como String, em vez de referências (&str) nos campos, a menos que se use lifetimes (Capítulo 10).[35]

5.6. Um Programa de Exemplo Usando Structs

Este exemplo calcula a área de um retângulo, refatorando de variáveis separadas para tuplas e, finalmente, para structs, para maior clareza e estrutura.[37]

Inicialmente com variáveis separadas:

Rust

```
fn area(largura: u32, altura: u32) -> u32 {
    largura * altura
}
```

Refatorado com tuplas:

Rust

```
fn area_tupla(dimensoes: (u32, u32)) -> u32 {
    dimensoes.0 * dimensoes.1
}
```

Refatorado com structs para maior clareza:

Rust

```
# // Permite imprimir a struct para depuração
struct Retangulo {
    largura: u32,
    altura: u32,
}

fn area_struct(retangulo: &Retangulo) -> u32 {
    retangulo.largura * retangulo.altura
}

fn main() {
    let ret1 = Retangulo { largura: 30, altura: 50 };
    println!("A área do retângulo é {} pixels quadrados.", area_struct(&ret1));
    println!("ret1 é {:?}", ret1); // Usando o Debug trait
    // println!("ret1 é {:#?}", ret1); // Para impressão formatada
    // dbg!(&ret1); // Outra forma de imprimir para depuração
}
```

O atributo # permite que a struct seja impressa usando o formatador de depuração {:?} ou {:#?}. A macro dbg! também é útil para depuração, pois toma ownership da expressão, imprime informações de arquivo/linha e o valor, e então retorna ownership.[37]

5.7. Sintaxe de Método

Métodos são similares a funções, mas são definidos no contexto de uma struct (ou

enum, ou objeto de trait) e seu primeiro parâmetro é sempre self, que representa a instância da struct na qual o método está sendo chamado.[38]

5.7.1. Definindo Métodos

Métodos são definidos dentro de um bloco impl (implementação) para a struct.

Rust

```
impl Retangulo {
    fn area(&self) -> u32 { // &self é uma abreviação para self: &Self
        self.largura * self.altura
    }

    fn pode_conter(&self, outro: &Retangulo) -> bool {
        self.largura > outro.largura && self.altura > outro.altura
    }
}

fn main() {
    let ret1 = Retangulo { largura: 30, altura: 50 };
    println!("A área é {}", ret1.area()); // Chamada de método
}
```

- &self: Empresta a instância imutavelmente.
- &mut self: Empresta a instância mutavelmente.
- self: Toma ownership da instância (raro, usado quando o método transforma self em outra coisa).

Rust realiza referenciação e desreferenciação automática em chamadas de método. Se você tem objeto.metodo(), Rust automaticamente adiciona &, &mut, ou * para que objeto corresponda à assinatura do método. Isso torna o código mais ergonômico.[38]

5.7.2. Funções Associadas

Funções dentro de um bloco impl que não têm self como primeiro parâmetro são chamadas de *funções associadas* (não métodos). Elas são frequentemente usadas como construtores, convencionalmente chamadas new. São chamadas usando o

nome da struct e ::, como Retangulo::quadrado(3).

```rust
impl Retangulo {
    fn quadrado(tamanho: u32) -> Self { // Self é um alias para o tipo Retangulo
        Self { largura: tamanho, altura: tamanho }
    }
}
```

Uma struct pode ter múltiplos blocos impl.[38]

Capítulo 6: Enums e Pattern Matching

Enums (enumerações) permitem definir um tipo enumerando suas possíveis *variantes*. Pattern matching, através da construção match, permite executar código diferente dependendo da variante de um enum.[26]

6.1. Definindo um Enum

Enums são úteis quando um valor pode ser uma de várias possibilidades distintas.

```rust
enum IpAddrKind {
    V4,
    V6,
}

fn main() {
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;
    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
}
```

```
fn route(ip_kind: IpAddrKind) {}
```

IpAddrKind::V4 e IpAddrKind::V6 são do mesmo tipo: IpAddrKind.[39]

6.1.1. Valores de Enum

Variantes de enum podem armazenar dados diretamente. Isso é mais conciso do que usar uma struct separada para associar dados a um tipo de enum.

Rust

```
enum IpAddr {
    V4(String), // Variante V4 armazena uma String
    V6(String), // Variante V6 armazena uma String
}

let home = IpAddr::V4(String::from("127.0.0.1"));
let loopback = IpAddr::V6(String::from("::1"));
```

Cada variante pode ter diferentes tipos e quantidades de dados associados:

Rust

```
enum IpAddrComTiposDiferentes {
    V4(u8, u8, u8, u8),
    V6(String),
}
```

A biblioteca padrão IpAddr usa structs dentro das variantes do enum.[39]

Um enum Message pode ter variantes diversas:

Rust

```
enum Message {
    Quit, // Sem dados associados
    Move { x: i32, y: i32 }, // Campos nomeados como uma struct
    Write(String), // Inclui uma String
    ChangeColor(i32, i32, i32), // Inclui três i32
}
```

Assim como structs, enums podem ter métodos definidos em um bloco impl.[39]

6.1.2. O Enum Option e Suas Vantagens Sobre Valores Nulos

Rust não tem o conceito de "nulo" como em muitas outras linguagens. Em vez disso, usa o enum Option<T> para codificar o conceito de um valor que pode estar presente ou ausente.

Rust

```
enum Option<T> {
    None,    // Indica ausência de valor
    Some(T), // Indica presença de um valor do tipo T
}
```

Option<T> está no prelúdio, então suas variantes Some e None podem ser usadas diretamente. O T é um parâmetro de tipo genérico.

Rust

```
let algum_numero = Some(5);
let alguma_string = Some("uma string");
let numero_ausente: Option<i32> = None; // Anotação de tipo necessária para None
```

A principal vantagem de Option<T> sobre nulos é a segurança de tipo. Option<T> e T são tipos diferentes, e o compilador não permitirá usar um Option<T> como se fosse definitivamente um T válido sem antes converter o Option<T> para T. Isso previne

muitos bugs comuns relacionados a nulos.[39] Para usar um valor Option<T>, é necessário lidar com ambas as variantes Some e None, geralmente usando uma expressão match.[39]

6.2. A Construção match

A construção match permite comparar um valor com uma série de padrões e executar código com base no padrão que corresponde. Padrões podem ser literais, nomes de variáveis, wildcards, etc..[26]

Rust

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState), // Variante Quarter armazena um UsState
}

# // Para que possamos inspecionar o estado
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => { // state se vincula ao valor UsState
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

- match é seguido por uma expressão (coin).
- Cada braço do match tem um padrão e código associado (=>).
- O código associado é uma expressão, e o valor resultante dessa expressão é retornado para toda a expressão match.
- Os braços são avaliados em ordem. O primeiro padrão que corresponder terá seu código executado.[26]

6.2.1. Padrões que Vinculam Valores

Braços de match podem vincular-se a partes dos valores que correspondem ao padrão. No exemplo Coin::Quarter(state), state se vincula ao valor UsState dentro da variante Quarter.[26]

6.2.2. Combinando com Option<T>

match é frequentemente usado com Option<T>:

Rust

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}
```

.[26]

6.2.3. Matches São Exaustivos

Os braços de match devem cobrir todas as possibilidades. Se um caso não for tratado, Rust emitirá um erro de compilação. Isso garante que todos os cenários sejam considerados.[26]

6.2.4. O Placeholder _

Para casos em que não se deseja executar código para todas as outras possibilidades, o padrão _ (underscore) pode ser usado como um catchall. Ele corresponderá a qualquer valor, mas não o vinculará a uma variável. Se nada deve

acontecer, usa-se () (o tipo unidade) como código para o braço _.

Rust

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(), // _ ignora o valor
}
// Ou, se nada deve acontecer:
// _ => (),
```

.[26]

6.3. Controle de Fluxo Conciso com if let

A sintaxe if let é uma forma mais curta de lidar com valores que correspondem a um padrão e ignorar todos os outros. É açúcar sintático para uma expressão match que executa código em apenas um padrão.

Rust

```
let config_max = Some(3u8);

// Usando match
match config_max {
    Some(max) => println!("O máximo configurado é {}", max),
    _ => (),
}

// Usando if let
if let Some(max) = config_max {
    println!("O máximo configurado é {}", max);
}
```

if let é menos verboso que match quando se está interessado em apenas um caso. Pode-se incluir um else com if let, que funciona da mesma forma que o braço _ em uma expressão match correspondente.42
A desvantagem de if let é que o compilador não verifica a exaustividade como faz com match.42

Capítulo 7: Gerenciando Projetos em Crescimento com Pacotes, Crates e Módulos

À medida que um projeto cresce, organizá-lo bem se torna cada vez mais importante. Rust oferece um sistema de módulos que permite dividir um projeto em partes menores e mais gerenciáveis, controlando o escopo e a privacidade dos itens.[12]

7.1. Pacotes e Crates

- Um **crate** é a menor unidade de compilação em Rust. Ele pode ser um *crate binário* (um programa executável com uma função main) ou um *crate de biblioteca* (código destinado a ser usado por outros programas, sem main). O arquivo fonte a partir do qual o compilador Rust começa é chamado de *raiz do crate* (src/main.rs para binários, src/lib.rs para bibliotecas, por convenção).[12]
- Um **pacote** (package) é um conjunto de um ou mais crates que fornecem um conjunto de funcionalidades. Um pacote contém um arquivo Cargo.toml que descreve como construir esses crates. Um pacote deve conter pelo menos um crate, seja ele de biblioteca ou binário. Ele pode ter no máximo um crate de biblioteca, mas pode ter múltiplos crates binários (colocando arquivos em src/bin).[12]

7.2. Definindo Módulos para Controlar Escopo e Privacidade

Módulos permitem organizar o código dentro de um crate em grupos para legibilidade e fácil reutilização. Eles também controlam a *privacidade* dos itens, determinando se um item pode ser usado por código externo (público) ou se é um detalhe de implementação interno e não disponível para uso externo (privado).[43]

- **Declaração de Módulos**: Usa-se a palavra-chave mod. O código do módulo pode estar inline entre chaves {} ou em outro arquivo.

```Rust
// Em src/lib.rs ou src/main.rs
mod front_of_house {
    mod hosting { // Submódulo
        fn add_to_waitlist() {}
    }
}
```

Isso cria uma árvore de módulos, com crate como a raiz implícita.[43]

- **Privacidade Padrão**: Por padrão, todos os itens (funções, structs, enums, módulos, constantes) são privados para seus módulos pais.
- **Tornando Público com pub**: Para tornar um item público, usa-se a palavra-chave pub antes de sua declaração. Se um módulo é público (pub mod), seus itens internos ainda são privados, a menos que também sejam marcados com pub.

```Rust
mod front_of_house {
    pub mod hosting { // Módulo hosting é público
        pub fn add_to_waitlist() {} // Função add_to_waitlist é pública
        fn seat_at_table() {} // Função seat_at_table é privada
    }
}
```

.[43]

7.3. Caminhos para Referenciar um Item na Árvore de Módulos

Para usar um item de um módulo, é preciso especificar seu caminho na árvore de módulos.

- **Caminhos Absolutos**: Começam da raiz do crate, usando a palavra-chave crate para o crate atual ou o nome de um crate externo.

```Rust
// Supondo que front_of_house está na raiz do crate
crate::front_of_house::hosting::add_to_waitlist();
```

- **Caminhos Relativos**: Começam do módulo atual, usando self, super, ou um identificador no módulo atual.
  - super refere-se ao módulo pai, similar a .. em sistemas de arquivos.

```Rust
fn serve_order() {}
mod back_of_house {
    fn fix_incorrect_order() {
        super::serve_order(); // Chama serve_order no módulo pai (raiz do crate)
    }
}
```

    - self refere-se ao módulo atual (menos comum para caminhos).
- Os componentes do caminho são separados por ::.[45]

7.4. Trazendo Caminhos para o Escopo com a Palavra-Chave use

A palavra-chave use cria atalhos para caminhos, evitando a repetição de caminhos longos.

- **Funcionamento**: use caminho::para::item; torna item (ou o último componente do caminho) um nome válido no escopo atual.

```Rust
use crate::front_of_house::hosting; // Traz o módulo hosting para o escopo
// use crate::front_of_house::hosting::add_to_waitlist; // Alternativa: traz a função

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist(); // Usando o atalho
    // add_to_waitlist(); // Se a função foi trazida diretamente
}
```

- **Idiomas para use**:
    - Para funções: é idiomático trazer o módulo pai da função para o escopo (e.g., use std::collections; e depois collections::HashMap).
    - Para structs, enums e outros itens: é idiomático trazer o item completo para o escopo (e.g., use std::collections::HashMap;).
    - Se houver conflito de nomes, pode-se usar o módulo pai para desambiguar ou usar as para renomear o tipo importado: use std::fmt::Result; use std::io::Result as IoResult;.[46]
- **Reexportando Nomes com pub use**: Combinar pub com use torna o item importado parte da API pública do módulo atual, permitindo que código externo use o atalho. Isso é útil para criar uma API pública diferente da estrutura interna do crate.

```Rust
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}
pub use crate::front_of_house::hosting; // Reexporta o módulo hosting
// Código externo pode agora usar restaurante::hosting::add_to_waitlist()
```

.[46]

- **Usando Caminhos Aninhados**: Para importar múltiplos itens do mesmo crate ou módulo, caminhos aninhados podem limpar listas use longas.
  Rust
  ```
  // Em vez de:
  // use std::cmp::Ordering;
  // use std::io;
  // Pode-se usar:
  use std::{cmp::Ordering, io};

  // Para importar std::io e std::io::Write:
  use std::io::{self, Write}; // self refere-se ao módulo pai (io)
  ```

.[46]

- **O Operador Glob ***: Traz todos os itens públicos de um caminho para o escopo. Deve ser usado com cautela, pois pode dificultar a identificação da origem dos nomes. É comum em módulos de teste (use super::*;) e no padrão *prelude*.
  Rust
  ```
  use std::collections::*;
  ```

.[46]

7.5. Separando Módulos em Arquivos Diferentes

Para organizar projetos maiores, módulos podem ser movidos para seus próprios arquivos.

- **Declaração mod nome_modulo;**: No arquivo pai (e.g., src/lib.rs), substitua o corpo do módulo mod nome_modulo {... } por mod nome_modulo;.
- **Localização do Arquivo do Módulo**:
  - O compilador procurará o código do módulo em src/nome_modulo.rs.
  - Ou, para um estilo mais antigo (ainda suportado), em src/nome_modulo/mod.rs. A convenção moderna prefere nome_modulo.rs para evitar muitos arquivos mod.rs.
- **Submódulos**: Se src/nome_modulo.rs contém mod nome_submodulo;, o compilador procurará por src/nome_modulo/nome_submodulo.rs (ou src/nome_modulo/nome_submodulo/mod.rs). A estrutura de arquivos deve espelhar a árvore de módulos.[47]

O layout padrão de um projeto Cargo inclui [48]:

- Cargo.toml e Cargo.lock na raiz.
- src/lib.rs para a raiz de um crate de biblioteca.
- src/main.rs para a raiz de um crate binário principal.
- src/bin/ para múltiplos crates binários.
- examples/ para exemplos.
- tests/ para testes de integração.
- benches/ para benchmarks.

Capítulo 8: Coleções Comuns

A biblioteca padrão de Rust inclui várias estruturas de dados muito úteis chamadas *coleções*. A maioria dos outros tipos de dados representa um valor específico, mas as coleções podem conter múltiplos valores. Ao contrário dos tipos array e tupla embutidos, os dados que essas coleções apontam são armazenados no heap, o que significa que a quantidade de dados não precisa ser conhecida em tempo de compilação e pode crescer ou diminuir à medida que o programa é executado.

8.1. Armazenando Listas de Valores com Vetores (Vec<T>)

Vec<T>, ou vetor, permite armazenar um número variável de valores um ao lado do outro.[49]

- **Criação**:
  - Vec::new(): Cria um vetor vazio. Requer anotação de tipo se os tipos não puderem ser inferidos.
    ```Rust
    let v: Vec<i32> = Vec::new();
    ```
  - vec! macro: Cria um vetor com valores iniciais. Rust pode inferir o tipo.
    ```Rust
    let v = vec!;
    ```

- **Atualização**:
  - push(): Adiciona um elemento ao final do vetor. O vetor deve ser mutável.
    ```Rust
    let mut v = Vec::new();
    v.push(5);
    ```

- **Leitura de Elementos**:
  - Indexação (&v[indice]): Retorna uma referência ao elemento. Causa pânico se

o índice estiver fora dos limites.

```rust
let v = vec!;
let terceiro: &i32 = &v; // terceiro é &3
```

- Método get(indice): Retorna um Option<&T>. Retorna None se o índice estiver fora dos limites, sem pânico.

```rust
let v = vec!;
match v.get(2) {
    Some(terceiro) => println!("O terceiro elemento é {terceiro}"),
    None => println!("Não há terceiro elemento."),
}
```

As regras de empréstimo de Rust se aplicam: não se pode ter referências mutáveis e imutáveis para o mesmo vetor no mesmo escopo se houver risco de modificação que invalide as referências (e.g., push pode realocar).[49]

- **Iterando sobre Vetores**:
  - Referências imutáveis:

```rust
let v = vec!;
for i in &v {
    println!("{i}");
}
```

  - Referências mutáveis:

```rust
let mut v = vec!;
for i in &mut v {
    *i += 50; // Usa * para desreferenciar e modificar
}
```

- **Usando um Enum para Armazenar Múltiplos Tipos**: Como um vetor só pode armazenar valores do mesmo tipo, um enum pode ser usado como um wrapper para armazenar diferentes tipos de dados em um único vetor.

```rust
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
```

```
}
let linha = vec!;
```

- **Descarte (Drop)**: Quando um vetor sai do escopo, todos os seus elementos também são descartados, e a memória que ele usa é liberada.[49]

A documentação da API para std::vec::Vec detalha muitos outros métodos úteis como pop, insert, remove, iter_mut, with_capacity, etc..[50] Por exemplo, Vec::with_capacity(capacity) constrói um novo vetor vazio com pelo menos a capacidade especificada, o que pode evitar realocações se o número de elementos for conhecido antecipadamente.[50]

8.2. Armazenando Texto UTF-8 com Strings (String)

O tipo String de Rust é uma string UTF-8, de propriedade, mutável e que pode crescer. É implementada como um wrapper em torno de um Vec<u8>.[51]

- **Criação**:
  - String::new(): Cria uma nova string vazia.
  - Método to_string(): Disponível em qualquer tipo que implemente o trait Display (como literais de string).
    Rust
    ```
    let s = "conteúdo inicial".to_string();
    ```
  - String::from(): Equivalente a to_string para literais de string.
    Rust
    ```
    let s = String::from("conteúdo inicial");
    ```
- **Atualização**:
  - push_str(&str): Anexa um slice de string. Não toma ownership.
    Rust
    ```
    let mut s = String::from("foo");
    s.push_str("bar"); // s agora é "foobar"
    ```
  - push(char): Anexa um único caractere.
    Rust
    ```
    let mut s = String::from("lo");
    s.push('l'); // s agora é "lol"
    ```
- **Concatenação**:
  - Operador +: Usa o método add, que toma ownership da primeira string e

anexa uma cópia da segunda (que deve ser um &str devido a deref coercion).

```Rust
let s1 = String::from("Olá, ");
let s2 = String::from("mundo!");
let s3 = s1 + &s2; // s1 foi movida aqui e não pode mais ser usada
```

- Macro format!: Mais legível para múltiplas strings, não toma ownership dos parâmetros.

```Rust
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");
let s = format!("{s1}-{s2}-{s3}"); // s é "tic-tac-toe"
```

- **Indexação em Strings**: Rust não permite indexar String diretamente (e.g., s) porque a indexação de strings UTF-8 é complexa. Um índice de byte pode não corresponder a um caractere Unicode válido, e determinar o n-ésimo caractere não é uma operação O(1).[51]
  - Rust reconhece três formas de interpretar dados de string: bytes (u8), valores escalares Unicode (char) e agrupamentos de grafemas (o mais próximo de "letras" humanas).
- **Slicing Strings**: Slices de string (&str) podem ser criados usando ranges de bytes, mas os limites do slice devem cair em fronteiras de caracteres UTF-8 válidos para evitar pânico.

```Rust
let ola = "Здравствуйте";
let s = &ola[0..4]; // s será "Зд" (cada caractere tem 2 bytes)
// let s = &ola[0..1]; // Pânico!
```

- **Métodos para Iterar sobre Strings**:
  - .chars(): Itera sobre os valores escalares Unicode (char).
  - .bytes(): Itera sobre os bytes brutos (u8). Para agrupamentos de grafemas, crates externos são necessários.[51]

A API std::string::String oferece métodos como insert_str, replace, to_lowercase, trim, entre outros.[52] replace cria uma nova String substituindo todas as ocorrências de um padrão. to_lowercase retorna o equivalente em minúsculas como uma nova String. trim retorna um slice de string com espaços em branco no início e no fim removidos.[52]

8.3. Armazenando Chaves com Valores Associados em Hash Maps (HashMap<K,

V>)

O tipo HashMap<K, V> armazena um mapeamento de chaves do tipo K para valores do tipo V usando uma função de hashing para determinar como colocar essas chaves e valores na memória. Hash maps são úteis quando você deseja procurar dados não por um índice, mas usando uma chave que pode ser de qualquer tipo.[53]

- **Criação**:
  - HashMap::new(): Cria um hash map vazio. Requer use std::collections::HashMap;.
    ```Rust
    use std::collections::HashMap;
    let mut pontuacoes = HashMap::new();
    pontuacoes.insert(String::from("Azul"), 10);
    ```
  - Hash maps armazenam seus dados no heap. Chaves devem ser todas do mesmo tipo, e valores devem ser todos do mesmo tipo.[53]
- **Acesso a Valores**:
  - Método get(&key): Retorna um Option<&V>.
    ```Rust
    let nome_time = String::from("Azul");
    let pontuacao = pontuacoes.get(&nome_time).copied().unwrap_or(0);
    ```
    .copied() converte Option<&i32> para Option<i32>, e .unwrap_or(0) define pontuacao como 0 se a chave não existir.[53]
  - Iteração:
    ```Rust
    for (chave, valor) in &pontuacoes {
        println!("{chave}: {valor}");
    }
    ```
    A ordem de iteração é arbitrária.[53]
- **Hash Maps e Ownership**:
  - Para tipos que implementam Copy (como i32), os valores são copiados para o hash map.
  - Para valores possuídos (como String), os valores são movidos, e o hash map se torna o dono. As variáveis originais não podem mais ser usadas.
  - Se referências são inseridas, os valores referenciados não são movidos, mas devem permanecer válidos enquanto o hash map existir.[53]
- **Atualizando um Hash Map**:
  - **Sobrescrevendo um Valor**: Inserir uma chave que já existe sobrescreverá o valor antigo.
  - **Adicionando uma Chave e Valor Apenas se uma Chave Não Estiver**

Presente: O método entry(key).or_insert(value) insere value se key não existir, ou retorna uma referência mutável ao valor existente.

```Rust
pontuacoes.entry(String::from("Amarelo")).or_insert(50); // Insere Amarelo: 50
pontuacoes.entry(String::from("Azul")).or_insert(50);    // Azul já existe, não altera
```

- **Atualizando um Valor com Base no Valor Antigo**: entry e or_insert podem ser usados para obter uma referência mutável e modificar o valor.

```Rust
let texto = "olá mundo maravilhoso mundo";
let mut mapa = HashMap::new();
for palavra in texto.split_whitespace() {
    let contador = mapa.entry(palavra).or_insert(0);
    *contador += 1; // Desreferencia e incrementa
}
```

or_insert(0) fornece 0 para novas palavras. *contador desreferencia a &mut V para modificar o valor.[53]

- **Funções de Hashing**: Por padrão, HashMap usa SipHash, que oferece resistência a ataques de Denial of Service (DoS) em tabelas hash. Pode-se especificar um hasher diferente implementando o trait BuildHasher se o desempenho for crítico.[53]

A API std::collections::HashMap é rica, incluindo métodos como get_mut, remove, keys, values, with_capacity, contains_key, etc..[54] Por exemplo, get_mut(&key) retorna uma referência mutável ao valor, remove(&key) remove um par chave-valor, e entry(key) fornece uma API Entry para manipulação local de uma entrada.[54]

Capítulo 9: Tratamento de Erros

Rust agrupa erros em duas categorias principais: *recuperáveis* e *irrecuperáveis*. Erros recuperáveis são aqueles que podem ser razoavelmente respondidos ou reportados, como um arquivo não encontrado. Erros irrecuperáveis são sempre sintomas de bugs, como tentar acessar uma localização além do final de um array, e resultam em pânico.[55]

9.1. Erros Irrecuperáveis com panic!

A macro panic! sinaliza que o programa está em um estado do qual não pode se recuperar. Quando panic! é executada, o programa imprime uma mensagem de falha, desenrola e limpa a pilha, e então sai.

- **Desenrolando a Pilha ou Abortando**: Por padrão, quando ocorre um pânico,

Rust *desenrola* a pilha. Isso significa que Rust percorre a pilha para trás e limpa os dados de cada função que encontra. Alternativamente, pode-se configurar Rust para *abortar* imediatamente ao entrar em pânico, o que encerra o programa sem limpar. O tamanho do binário resultante é menor com aborto. Isso pode ser configurado em Cargo.toml:

```
Ini, TOML
[profile.release]
panic = 'abort'
```

- **Backtraces com panic!**: Um backtrace é uma lista de todas as funções que foram chamadas para chegar ao ponto do pânico. Para obter um backtrace, as informações de depuração devem estar habilitadas (o que é padrão para builds de depuração). Executar com a variável de ambiente RUST_BACKTRACE=1 (ou qualquer valor diferente de 0) imprimirá o backtrace.

```
Rust
fn main() {
    let v = vec!;
    v; // Causa pânico: acesso a índice fora dos limites
}
```

Executar RUST_BACKTRACE=1 cargo run mostrará o caminho das chamadas de função que levaram ao erro.[55]

9.2. Erros Recuperáveis com Result<T, E>

A maioria dos erros não é suficientemente séria para exigir que o programa pare completamente. Em vez disso, uma função pode falhar de uma forma que o código chamador possa interpretar e responder. O enum Result<T, E> é usado para esses casos.[56]

```
Rust
enum Result<T, E> {
    Ok(T),  // Contém um valor do tipo T em caso de sucesso
    Err(E), // Contém um valor de erro do tipo E em caso de falha
}
```

- **Combinando com Result**:

```
Rust
```

```rust
use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("ola.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("ola.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problema ao criar o arquivo: {:?}", e),
            },
            other_error => {
                panic!("Problema ao abrir o arquivo: {:?}", other_error);
            }
        },
    };
}
```

Este exemplo tenta abrir um arquivo. Se falhar porque o arquivo não existe (ErrorKind::NotFound), ele tenta criar o arquivo. Para outros erros, ele entra em pânico.[56]

- **Atalhos para Pânico em Erro: unwrap e expect**:
  - unwrap(): Se o Result for Ok, retorna o valor dentro de Ok. Se for Err, chama panic! com uma mensagem de erro padrão.
    ```rust
    let greeting_file = File::open("ola.txt").unwrap();
    ```
  - expect("mensagem"): Similar a unwrap, mas permite fornecer uma mensagem de pânico personalizada, o que é útil para depuração.
    ```rust
    let greeting_file = File::open("ola.txt")
        .expect("ola.txt deveria estar incluído neste projeto");
    ```

  Estes métodos são úteis para prototipagem ou quando se tem certeza de que a operação não falhará.[56]
- **Propagando Erros com o Operador ?**: Quando uma função tem uma falha que deve ser tratada pelo código chamador, o erro pode ser propagado. O operador ?

simplifica isso.

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

fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("ola.txt")?; // Propaga erro se File::open falhar
    let mut username = String::new();
    username_file.read_to_string(&mut username)?; // Propaga erro se read_to_string falhar
    Ok(username)
}
```

Se o valor de Result for Ok, o valor dentro de Ok é retornado da expressão, e o programa continua. Se for Err, o valor Err é retornado de toda a função como se return Err(e) tivesse sido usado. O operador ? também pode ser usado com Option<T>.31
O operador ? chama a função from definida no trait From no tipo de erro, o que permite converter diferentes tipos de erro no tipo de erro retornado pela função atual.56
O operador ? só pode ser usado em funções cujo tipo de retorno seja compatível com o valor em que ? é usado (ou seja, Result ou Option ou outro tipo que implemente FromResidual).56 A função main também pode retornar Result<(), E>.56

9.3. Entrar em Pânico ou Não?

A decisão entre panic! e Result<T, E> depende da recuperabilidade do erro e do contexto.[57]

- **Result<T, E> como Padrão**: É a escolha padrão para funções que podem falhar, dando ao chamador a opção de tentar recuperar ou entrar em pânico.
- **Quando Usar panic!**:
  - Quando o código pode terminar em um "estado ruim" inesperado (suposição, garantia, contrato ou invariante quebrado).
  - Quando o código subsequente depende de não estar nesse estado ruim.
  - Quando não há uma boa maneira de codificar essa informação nos tipos usados.
  - Em exemplos, código de protótipo e testes, onde um pânico é aceitável ou desejado para indicar uma falha.
  - Quando se tem mais informações que o compilador e se tem certeza de que

um Result será Ok (usando unwrap ou expect com uma mensagem que documenta a suposição).
    - Quando a falha indica um bug no lado do chamador (violação de contrato da função).
- **Quando Retornar Result**:
    - Quando a falha é esperada (e.g., dados malformados, limite de taxa atingido).
- **Criando Tipos Personalizados para Validação**: Para impor invariantes e reduzir verificações repetitivas, pode-se criar tipos personalizados com lógica de validação em seus construtores. Se a validação falhar, o construtor pode entrar em pânico.

```
Rust
pub struct Palpite {
    valor: i32,
}
impl Palpite {
    pub fn new(valor: i32) -> Palpite {
        if valor < 1 |
```

```
| valor > 100 {
panic!("O valor do palpite deve estar entre 1 e 100, obteve {}.", valor);
}
Palpite { valor }
}
pub fn valor(&self) -> i32 {
self.valor
}
}
```

Funções que requerem um número entre 1 e 100 podem então aceitar um Palpite como parâmetro, eliminando a necessidade de verificações adicionais.57

Capítulo 10: Tipos Genéricos, Traits e Lifetimes

Estes são recursos avançados que permitem escrever código flexível e reutilizável, garantindo a segurança e correção em tempo de compilação.

10.1. Tipos de Dados Genéricos

Tipos genéricos permitem escrever definições de funções, structs, enums e métodos que funcionam com muitos tipos de dados concretos diferentes, evitando duplicação de código.[58]

- **Em Definições de Função**: Parâmetros de tipo genérico são declarados na assinatura da função, entre colchetes angulares (<>), antes da lista de parâmetros.

```Rust
fn maior<T: std::cmp::PartialOrd>(lista: &) -> &T {
    let mut maior_item = &lista;
    for item in lista {
        if item > maior_item { // Requer que T implemente PartialOrd
            maior_item = item;
        }
    }
    maior_item
}
```

  Aqui, T é um tipo genérico. A restrição T: std::cmp::PartialOrd (um *trait bound*) é necessária porque o corpo da função usa o operador >.[58]
- **Em Definições de Struct**: Permite que campos de struct armazenem valores de tipos genéricos.

```Rust
struct Ponto<T> { // Genérico sobre um tipo T
    x: T,
    y: T,
}
let inteiro = Ponto { x: 5, y: 10 };
let flutuante = Ponto { x: 1.0, y: 4.0 };

struct PontoMultiplo<T, U> { // Genérico sobre dois tipos T e U
    x: T,
    y: U,
}
let ambos_tipos = PontoMultiplo { x: 5, y: 4.0 };
```

.[58]

- **Em Definições de Enum**: Enums podem conter tipos genéricos em suas variantes. Option<T> e Result<T, E> são exemplos proeminentes da biblioteca padrão.

```Rust
enum Option<T> {
```

```
    Some(T),
    None,
}
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

.[58]

- **Em Definições de Método**: Métodos implementados em structs e enums genéricos também usam tipos genéricos. O parâmetro de tipo genérico deve ser declarado após impl.

```
Rust
impl<T> Ponto<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
// Métodos podem ser implementados apenas para tipos concretos específicos:
impl Ponto<f32> {
    fn distancia_da_origem(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

  Parâmetros de tipo genérico em uma definição de struct podem ser diferentes daqueles usados nas assinaturas de método dessa struct.[58]
- **Desempenho de Tipos Genéricos**: Rust implementa genéricos usando *monomorfização* em tempo de compilação. O compilador substitui o código genérico por código específico para cada tipo concreto usado, resultando em nenhum custo de tempo de execução para genéricos.[58]

10.2. Traits: Definindo Comportamento Compartilhado

Um trait define funcionalidade que um tipo particular pode ter e que pode ser compartilhada com outros tipos. Traits são semelhantes a interfaces em outras linguagens.[60]

- **Definindo um Trait**: Usa-se a palavra-chave trait seguida do nome do trait e, entre chaves, as assinaturas dos métodos que descrevem os comportamentos.

```rust
pub trait Sumario {
    fn resumir_autor(&self) -> String; // Método obrigatório
    fn resumir(&self) -> String { // Método com implementação padrão
        format!("(Leia mais de {}...)", self.resumir_autor())
    }
}
```

- **Implementando um Trait em um Tipo**: Usa-se um bloco impl NomeDoTrait for Tipo {... }.

```rust
pub struct ArtigoDeNoticia { /*... */ }
impl Sumario for ArtigoDeNoticia {
    fn resumir_autor(&self) -> String {
        format!("{}", self.autor)
    }
    // resumir() usará a implementação padrão se não for sobrescrito
}
```

  A regra da órfandade (orphan rule) dita que se pode implementar um trait em um tipo se o trait ou o tipo (ou ambos) forem locais ao crate atual.[60]
- **Implementações Padrão**: Traits podem fornecer implementações padrão para alguns ou todos os seus métodos. Essas podem ser sobrescritas ao implementar o trait. Implementações padrão podem chamar outros métodos do mesmo trait.[60]
- **Traits como Parâmetros**: A sintaxe impl Trait permite que uma função aceite qualquer tipo que implemente um trait específico.

```rust
pub fn notificar(item: &impl Sumario) {
    println!("Notícia de última hora! {}", item.resumir());
}
```

  Isso é açúcar sintático para uma forma mais longa chamada *trait bound*:

```rust
pub fn notificar_com_bound<T: Sumario>(item: &T) { /*... */ }
```

  Trait bounds são úteis para forçar múltiplos parâmetros a serem do mesmo tipo genérico que implementa o trait. Múltiplos trait bounds podem ser especificados com + (e.g., item: &(impl Sumario + Display)). Cláusulas where podem tornar assinaturas complexas mais legíveis.[60]
- **Retornando Tipos que Implementam Traits**: A sintaxe impl Trait também pode

ser usada na posição de retorno para retornar um valor de algum tipo que implementa um trait, sem nomear o tipo concreto. Isso é útil para closures e iteradores. A função deve retornar um único tipo concreto que implemente o trait.[60]

```Rust
fn retorna_sumarizavel() -> impl Sumario {
    ArtigoDeNoticia { /*... */ }
}
```

- **Usando Trait Bounds para Implementar Métodos Condicionalmente**: Pode-se implementar métodos em um tipo genérico T apenas se T implementar certos traits.

```Rust
struct Par<T> { x: T, y: T }
impl<T: Display + PartialOrd> Par<T> {
    fn cmp_display(&self) { /*... */ }
}
```

  Também é possível implementar condicionalmente um trait para qualquer tipo que implemente outro trait (implementações blanket), como ToString para qualquer tipo que implemente Display.[60]

10.3. Validando Referências com Lifetimes

Lifetimes garantem que as referências sejam válidas enquanto são necessárias, prevenindo referências pendentes.[62]

- **Prevenindo Referências Pendentes**: O *borrow checker* de Rust compara escopos (lifetimes) para garantir que os dados não saiam do escopo antes de suas referências. Se o lifetime de um valor emprestado ('b) for menor que o lifetime da referência ('a), o compilador rejeitará o código.[62]
- **Sintaxe de Anotação de Lifetime**: Lifetimes são um tipo de genérico. Anotações de lifetime começam com um apóstrofo (') e são tipicamente nomes curtos em minúsculas (e.g., 'a). Elas descrevem as relações entre os lifetimes de múltiplas referências.
  - &'a i32: Uma referência com lifetime 'a.
  - &'a mut i32: Uma referência mutável com lifetime 'a.[62]
- **Anotações de Lifetime em Assinaturas de Função**: Declara-se parâmetros de lifetime genéricos entre <> após o nome da função.

```Rust
fn mais_longo<'a>(x: &'a str, y: &'a str) -> &'a str {
```

```
    if x.len() > y.len() { x } else { y }
}
```

Isso diz a Rust que para algum lifetime 'a, a função recebe dois slices de string que vivem pelo menos tanto quanto 'a', e o slice retornado também viverá pelo menos tanto quanto 'a'. O lifetime do valor retornado será o menor dos lifetimes dos argumentos.[62]

- **Anotações de Lifetime em Definições de Struct**: Se uma struct armazena referências, sua definição precisa de uma anotação de lifetime para cada referência.

```
Rust
struct ExcerptImportante<'a> {
    parte: &'a str,
}
```

Uma instância de ExcerptImportante não pode sobreviver à referência que ela detém em seu campo parte.[62]

- **Regras de Elisão de Lifetime**: O compilador Rust pode inferir lifetimes em muitos casos comuns através de regras de elisão, reduzindo a necessidade de anotações explícitas.
  1. Cada parâmetro que é uma referência recebe seu próprio parâmetro de lifetime.
  2. Se houver exatamente um parâmetro de lifetime de entrada, esse lifetime é atribuído a todos os parâmetros de lifetime de saída.
  3. Se houver múltiplos parâmetros de lifetime de entrada, mas um deles for &self ou &mut self (indicando um método), o lifetime de self é atribuído a todos os parâmetros de lifetime de saída. Se a ambiguidade persistir após aplicar essas regras, o compilador exigirá anotações explícitas.[62]
- **Anotações de Lifetime em Definições de Método**: Seguem a mesma sintaxe de tipos genéricos. Lifetimes que são parte do tipo da struct são declarados após impl.

```
Rust
impl<'a> ExcerptImportante<'a> {
    fn level(&self) -> i32 { 3 } // Lifetime de &self elidido
}
```

.[62]

- **O Lifetime Estático ('static)**: Indica que a referência pode viver por toda a

duração do programa. Literais de string têm o lifetime 'static porque seu texto é embutido diretamente no binário do programa.

```Rust
let s: &'static str = "Eu tenho um lifetime estático.";
```

Usar 'static deve ser considerado cuidadosamente; geralmente, sugere um problema subjacente com referências pendentes ou incompatibilidade de lifetimes.[62]

- **Parâmetros de Tipo Genérico, Trait Bounds e Lifetimes Juntos**: Todos podem ser usados em uma única assinatura de função.

```Rust
use std::fmt::Display;
fn mais_longo_com_anuncio<'a, T: Display>(x: &'a str, y: &'a str, ann: T) -> &'a str {
    println!("Anúncio! {}", ann);
    if x.len() > y.len() { x } else { y }
}
```

.[62]

- **Conceitos Avançados de Lifetime** (Capítulo 19.2):
  - **Subtipagem de Lifetime ('long <: 'short)**: Um lifetime mais longo é um subtipo de um lifetime mais curto se a região do lifetime mais longo contiver completamente a região do mais curto. Isso permite que uma referência com um lifetime mais longo seja usada onde uma com um lifetime mais curto é esperada.[31]
  - **Variância**: Define como as relações de subtipagem são propagadas através de parâmetros genéricos (covariante, contravariante, invariante). &'a T é covariante sobre 'a. &'a mut T é invariante sobre T para prevenir use-after-free.[66]
  - **Trait Bounds de Lifetime (T: 'a)**: Especifica que todos os parâmetros de lifetime de referência no tipo T devem viver pelo menos tanto quanto o lifetime 'a.[31]
  - **Elisão de Lifetime em Cabeçalhos impl**: As regras de elisão também se aplicam a cabeçalhos impl.[31]
  - **Higher-Rank Trait Bounds (HRTBs) (for<'a> F: Trait<'a>)**: Expressa que um trait bound deve valer para *todos* os lifetimes possíveis. Usado principalmente com os traits Fn para closures e funções que operam sobre dados com lifetimes variados.[31]

Capítulo 11: Escrevendo Testes Automatizados

Testes automatizados são cruciais para verificar se o código funciona corretamente e para manter essa correção à medida que o código evolui. Rust possui suporte de primeira classe para escrever testes.

11.1. Como Escrever Testes

Testes em Rust são funções que verificam se o código não-teste está funcionando da maneira esperada. Os corpos das funções de teste tipicamente executam três ações: configurar quaisquer dados ou estado necessários, executar o código que se quer testar e, então, afirmar que os resultados são o que se espera.[69]

- **O Atributo #[test]**: Para transformar uma função regular em uma função de teste, adiciona-se #[test] na linha anterior a fn. Quando cargo test é executado, Rust constrói um executável de teste que executa as funções anotadas e reporta seu status.[69]

```Rust
#[cfg(test)] // Compila e executa código de teste apenas com `cargo test`
mod tests {
    use super::*; // Traz itens do módulo pai para o escopo
    #[test]
    fn funciona() {
        let resultado = 2 + 2;
        assert_eq!(resultado, 4);
    }
}
```

- **Verificando Resultados com as Macros assert!, assert_eq! e assert_ne!**:
  - assert!(expressao): Entra em pânico se a expressão for false.
  - assert_eq!(esquerda, direita): Testa se os valores são iguais. Imprime ambos os valores na falha.
  - assert_ne!(esquerda, direita): Testa se os valores são desiguais. Para assert_eq! e assert_ne!, os valores comparados devem implementar os traits PartialEq e Debug.[69]
- **Adicionando Mensagens de Falha Personalizadas**: Argumentos opcionais podem ser passados para as macros de asserção para fornecer mensagens de erro mais contextuais.

```Rust
assert!(resultado.contains("Carol"),
    "A saudação não continha o nome, o valor era `{}`", resultado
```

```
);
```

.[69]

- **Verificando Pânicos com should_panic**: O atributo #[should_panic] em uma função de teste faz o teste passar se o código dentro da função entrar em pânico, e falhar se não entrar.
  ```Rust
  #[test]
  #[should_panic]
  fn maior_que_100() {
      Palpite::new(200);
  }
  ```
  Pode-se adicionar um parâmetro expected a #[should_panic] para verificar se a mensagem de pânico contém um texto específico.[69]
- **Usando Result<T, E> em Testes**: Funções de teste podem retornar Result<T, E>. Isso permite usar o operador ? dentro dos testes para propagar erros de forma concisa. Um teste passa se retorna Ok(()) e falha se retorna Err. Não se pode usar #[should_panic] em testes que retornam Result<T, E>. Para afirmar que uma operação retorna Err, use assert!(valor.is_err()).[69]

11.2. Controlando Como os Testes São Executados

O comando cargo test compila o código em modo de teste e executa o binário de teste resultante. O comportamento padrão pode ser modificado com opções de linha de comando.[71]

- **Executando Testes em Paralelo ou Consecutivamente**: Por padrão, testes são executados em paralelo usando threads. Para executar sequencialmente (e.g., se os testes compartilham estado ou interferem uns com os outros), use cargo test -- --test-threads=1.[71]
- **Mostrando Saída de Testes Bem-Sucedidos**: Por padrão, a saída (e.g., de println!) de testes que passam é capturada e não exibida. Para ver a saída de testes bem-sucedidos, use cargo test -- --show-output.[71]
- **Executando um Subconjunto de Testes por Nome**:
  - Para executar um único teste: cargo test nome_do_teste_funcao.
  - Para filtrar e executar múltiplos testes: cargo test nome_parcial (executa todos os testes cujo nome contenha nome_parcial). O nome do módulo também faz parte do nome do teste.[71]
- **Ignorando Alguns Testes a Menos que Especificamente Solicitado**:

    - Anote testes demorados com #[ignore].
    - Execute apenas testes ignorados: cargo test -- --ignored.
    - Execute todos os testes, incluindo os ignorados: cargo test -- --include-ignored.[70]

11.3. Organização de Testes

Testes em Rust são categorizados em testes de unidade e testes de integração.[72]

- **Testes de Unidade**:
    - **Localização**: No diretório src, no mesmo arquivo do código que testam.
    - **Convenção**: Dentro de um módulo tests anotado com #[cfg(test)]. Esta anotação instrui Rust a compilar e executar o código de teste apenas com cargo test.
    - **Testando Funções Privadas**: As regras de privacidade de Rust permitem que testes acessem funções privadas definidas no mesmo módulo ou em módulos pais (usando use super::*;).[72]
- **Testes de Integração**:
    - **Localização**: No diretório tests no nível superior do projeto, ao lado de src. Cargo trata cada arquivo no diretório tests como um crate separado.
    - **Compilação**: Cada arquivo em tests é compilado como um crate individual.
    - **Sem #[cfg(test)]**: Não é necessário, pois Cargo compila o diretório tests apenas ao executar cargo test.
    - **Importando o Crate**: É necessário importar o crate da biblioteca sob teste em cada arquivo de teste de integração (e.g., use nome_do_crate;).
    - Eles testam apenas a API pública da biblioteca.[70]
- **Submódulos em Testes de Integração**: Para compartilhar código entre arquivos de teste de integração (e.g., helpers), crie um módulo como tests/common/mod.rs. Este arquivo não será tratado como um crate de teste separado. Outros arquivos de teste podem então usar mod common;.[70]
- **Testes de Integração para Crates Binários**: Crates binários (apenas src/main.rs) não podem ser testados diretamente por testes de integração da mesma forma que crates de biblioteca. A prática comum é ter a lógica principal em src/lib.rs e fazer com que src/main.rs chame essa lógica. Assim, os testes de integração podem testar o crate de biblioteca.[72]
- **Testes de Documentação**: Rust compila e executa exemplos de código em comentários de documentação (///) como testes. Isso garante que a documentação e os exemplos permaneçam precisos e funcionais.[23]

Esta estrutura de testes incentiva uma cobertura abrangente, desde unidades individuais até a interação entre componentes e a correção da documentação,

contribuindo para a robustez do software Rust.

Capítulo 12: Projeto de E/S: Construindo um Programa de Linha de Comando

Este capítulo foca na aplicação prática dos conceitos aprendidos para construir um programa de linha de comando funcional. Um exemplo comum para este tipo de projeto em "The Rust Programming Language" é uma versão simplificada da ferramenta grep, chamada minigrep. Este projeto envolveria:

1. **Aceitando Argumentos de Linha de Comando**:
    - Usar std::env::args() para coletar os argumentos passados ao programa. args() retorna um iterador sobre os argumentos.
    - Analisar os argumentos para obter, por exemplo, uma query de busca e um nome de arquivo.
2. **Lendo o Conteúdo de um Arquivo**:
    - Usar std::fs::File::open() para abrir um arquivo.
    - Usar std::io::Read e seu método read_to_string() para ler o conteúdo do arquivo em uma String.
3. **Tratamento Robusto de Erros**:
    - Usar Result<T, E> para todas as operações que podem falhar (e.g., abertura de arquivo, leitura).
    - Propagar erros usando o operador ? ou lidar com eles usando match ou métodos como unwrap_or_else.
    - Imprimir mensagens de erro para stderr usando eprintln!.
4. **Escrevendo na Saída Padrão ou Erro Padrão**:
    - Usar println! para a saída normal.
    - Usar eprintln! para mensagens de erro, para que possam ser redirecionadas separadamente da saída bem-sucedida.
5. **Usando Variáveis de Ambiente**:
    - Ler variáveis de ambiente com std::env::var() para, por exemplo, habilitar buscas case-insensitive.
6. **Estruturando a Aplicação com Módulos**:
    - Separar a lógica de análise de argumentos, lógica de leitura de arquivo e lógica de busca em módulos distintos para melhor organização.
    - Definir uma struct Config para agrupar os parâmetros de configuração.
7. **Escrevendo Testes**:
    - Testes de unidade para as funções de busca e análise.
    - Testes de integração para o comportamento geral do programa (embora testar binários diretamente seja mais complexo, a lógica da biblioteca pode

ser testada exaustivamente).

Embora não haja snippets diretos para um projeto minigrep específico nos materiais fornecidos para este capítulo, os conceitos dos capítulos anteriores seriam aplicados intensivamente. Por exemplo, String e Vec (Capítulo 8) para manipular dados, Result e panic! (Capítulo 9) para tratamento de erros, e módulos (Capítulo 7) para organização.

Capítulo 13: Recursos de Linguagem Funcional: Iteradores e Closures

Rust incorpora muitos recursos de linguagens funcionais, permitindo um estilo de programação expressivo e conciso. Iteradores e closures são dois desses recursos poderosos.

13.1. Closures: Funções Anônimas que Podem Capturar Seu Ambiente

Closures são funções anônimas que podem ser salvas em variáveis ou passadas como argumentos para outras funções. Uma característica distintiva é sua capacidade de capturar valores do escopo em que são definidas.[73]

- **Sintaxe e Características**:
  - Definidas com |parametros| corpo.
  - Tipos de parâmetros e retorno podem ser inferidos pelo compilador, mas podem ser anotados explicitamente: |param: Tipo| -> TipoRetorno { corpo }.
  - Chaves {} são opcionais para um corpo de expressão única.[73]

```
Rust
let adicionar_um_v2 = |x: u32| -> u32 { x + 1 };
let adicionar_um_v3 = |x| { x + 1 }; // Tipos inferidos
let adicionar_um_v4 = |x| x + 1  ; // Chaves opcionais
```

- **Capturando o Ambiente**: Closures podem acessar variáveis do escopo onde são criadas.
  - **Empréstimo Imutável**: Se o closure apenas lê o valor.
  - **Empréstimo Mutável**: Se o closure modifica o valor.
  - **Tomando Ownership (com move)**: A palavra-chave move força o closure a tomar ownership dos valores que usa do ambiente. Isso é crucial ao passar closures para novas threads, garantindo que os dados permaneçam válidos.[73]

```
Rust
let lista = vec!;
println!("Antes de definir closure: {:?}", lista);
let apenas_imprime = |
```

```
| println!("Da closure: {:?}", lista); // Empréstimo imutável
apenas_imprime();



let mut lista_mut = vec!;
let mut empresta_e_modifica = |

| lista_mut.push(7); // Empréstimo mutável
// empresta_e_modifica(); // Não pode emprestar mutavelmente duas vezes no mesmo escopo
```

- **Traits Fn, FnMut e FnOnce**: A forma como um closure captura e manipula valores do ambiente determina qual dos traits Fn ele implementa. Estes traits são como os tipos de closures são especificados em assinaturas de função.
  1. **FnOnce**: Aplica-se a closures que podem ser chamadas apenas uma vez. Todos os closures implementam FnOnce. Um closure que move valores capturados para fora de seu corpo só implementará FnOnce.
  2. **FnMut**: Aplica-se a closures que não movem valores capturados para fora, mas podem mutar os valores capturados. Podem ser chamadas mais de uma vez.
  3. **Fn**: Aplica-se a closures que não movem valores capturados para fora nem mutam valores capturados (ou não capturam nada do ambiente). Podem ser chamadas múltiplas vezes, inclusive concorrentemente. Um closure implementa Fn se puder implementar FnMut, e implementa FnMut se puder implementar FnOnce.[73]
- **Ponteiros de Função como Closures**: Funções regulares (fn) podem ser passadas onde um closure é esperado, pois fn implementa todos os três traits Fn (desde que não capturem o ambiente, o que funções não fazem).[75]

13.2. Processando uma Série de Itens com Iteradores

O padrão iterador permite realizar uma tarefa em uma sequência de itens por vez. Em Rust, iteradores são *preguiçosos* (lazy), o que significa que não têm efeito até que se chame métodos que consomem o iterador.[76]

- **O Trait Iterator e o Método next**: Todos os iteradores implementam o trait Iterator, definido na biblioteca padrão.
  Rust
  ```
  pub trait Iterator {
  ```

```
    type Item; // Tipo associado que representa o tipo dos itens do iterador
    fn next(&mut self) -> Option<Self::Item>;
    // outros métodos com implementações padrão...
}
```

O único método obrigatório é next, que retorna um item do iterador por vez, encapsulado em Some, e retorna None quando a iteração termina.[76]

- **Criando Iteradores**:
  - iter(): Produz um iterador sobre referências imutáveis (&T).
  - into_iter(): Produz um iterador que toma ownership da coleção e retorna valores possuídos (T).
  - iter_mut(): Produz um iterador sobre referências mutáveis (&mut T). .[76]
- **Adaptadores Consumidores (Consuming Adaptors)**: Métodos que chamam next e consomem o iterador.
  - sum(): Toma ownership do iterador e soma seus itens.

    ```
    Rust
    let v1 = vec!;
    let iterador = v1.iter();
    let total: i32 = iterador.sum(); // iterador não pode mais ser usado
    assert_eq!(total, 6);
    ```

  - collect(): Consome o iterador e coleta os valores resultantes em uma estrutura de dados de coleção.

    ```
    Rust
    let v1: Vec<i32> = vec!;
    let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
    assert_eq!(v2, vec!);
    ```

.[76]

- **Adaptadores de Iterador (Iterator Adaptors)**: Métodos que transformam um iterador em um novo tipo de iterador. São preguiçosos e precisam de um adaptador consumidor para ter efeito.
  - map(|closure|...): Aplica um closure a cada elemento para produzir um novo iterador com os elementos transformados.
  - filter(|closure|...): Aplica um closure a cada elemento e produz um novo iterador que contém apenas os elementos para os quais o closure retorna true.

```
Rust
```

```
// Exemplo usando map e filter [78, 79]
let numeros = (0..).map(|n| n * n)          // Quadrados dos números naturais
                .take_while(|&n_quad| n_quad < 1000) // Pega enquanto menor que 1000
                .filter(|&n_quad| n_quad % 2!= 0) // Filtra os ímpares
                .collect::<Vec<u32>>();     // Coleta em um vetor
```

O método fold (similar a reduce em outras linguagens) também é um adaptador consumidor poderoso para resumir um iterador a um único valor.[78]

- **Usando Closures que Capturam Seu Ambiente com Iteradores**: Closures passadas para adaptadores de iterador podem capturar variáveis de seu ambiente.

```
Rust
#
struct Sapato {
    tamanho: u32,
    estilo: String,
}

fn sapatos_no_tamanho(sapatos: Vec<Sapato>, tamanho_sapato: u32) -> Vec<Sapato>
{
    sapatos.into_iter().filter(|s| s.tamanho == tamanho_sapato).collect()
}
```

Aqui, o closure em filter captura tamanho_sapato do ambiente da função sapatos_no_tamanho.[76]

A combinação de iteradores, closures e traits Fn torna o código Rust funcionalmente expressivo e eficiente, pois muitas operações de iterador são otimizadas para evitar alocações desnecessárias e podem até ser compiladas para loops imperativos eficientes.

Funções de ordem superior (Higher Order Functions

Works cited

1. Blockchain In Supply Chain Market Size | CAGR of 44.5% - Market.us, accessed May 29, 2025, https://market.us/report/blockchain-in-supply-chain-market/
2. Blockchain Market Size to Surpass USD 988.83 Billion by 2032 | SNS Insider, accessed May 29, 2025, https://www.globenewswire.com/news-release/2025/02/19/3028897/0/en/Blockchain-Market-Size-to-Surpass-USD-988-83-Billion-by-2032-SNS-Insider.html
3. The Rust Programming Language: Klabnik, Steve, Nichols, Carol - Amazon.com, accessed May 30, 2025, https://www.amazon.com/Rust-Programming-Language-Steve-Klabnik/dp/15932

78284
4. The Rust Programming Language, 3rd Edition by Carol Nichols, Chris Krycho: 9781718504448 | PenguinRandomHouse.com: Books, accessed May 30, 2025, https://www.penguinrandomhouse.com/books/790517/the-rust-programming-language-3rd-edition-by-carol-nichols-and-chris-krycho/
5. Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/
6. Introduction - The rustup book, accessed May 30, 2025, https://rust-lang.github.io/rustup/
7. Getting Started - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch01-00-getting-started.html
8. Hello, World! - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch01-02-hello-world.html
9. The Cargo Book - MIT, accessed May 30, 2025, https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/cargo/print.html
10. The Cargo Book - GitHub Pages, accessed May 30, 2025, https://ebarnard.github.io/2019-06-03-rust-smaller-trait-implementers-docs/cargo/print.html
11. Hello, Cargo! - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch01-03-hello-cargo.html
12. Packages and Crates - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html
13. Programming a Guessing Game - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch02-00-guessing-game-tutorial.html
14. Common Programming Concepts - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch03-00-common-programming-concepts.html
15. Variables and Mutability - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html
16. Variable Bindings - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/variable_bindings.html
17. Data Types - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch03-02-data-types.html
18. Primitives - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/primitives.html
19. Types - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/types.html
20. Functions - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch03-03-how-functions-work.html
21. Functions - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/fn.html
22. Comments - The Rust Programming Language - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/book/ch03-04-comments.html
23. Publishing a Crate to Crates.io - The Rust Programming Language, accessed May

30, 2025, https://doc.rust-lang.org/book/ch14-02-publishing-to-crates-io.html
24. Control Flow - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch03-05-control-flow.html
25. Flow of Control - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/flow_control.html
26. The match Control Flow Construct - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch06-02-match.html
27. match - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/flow_control/match.html
28. What is Ownership? - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
29. References and Borrowing - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html
30. Borrowing - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/scope/borrow.html
31. accessed December 31, 1969, https://doc.rust-lang.org/rust-by-example/error_handling/result.html
32. The Slice Type - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch04-03-slices.html
33. RAII - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/scope/raii.html
34. Lifetimes - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/scope/lifetime.html
35. Defining and Instantiating Structs - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch05-01-defining-structs.html
36. Structures - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/custom_types/structs.html
37. An Example Program Using Structs - The Rust Programming ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch05-02-example-structs.html
38. Method Syntax - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch05-03-method-syntax.html
39. Defining an Enum - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html
40. Enums - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/custom_types/enum.html
41. accessed December 31, 1969, https://doc.rust-lang.org/rust-by-example/error_handling/option.html
42. Concise Control Flow with if let and let else - The Rust Programming ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch06-03-if-let.html
43. Defining Modules to Control Scope and Privacy - The Rust ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html
44. Modules - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/mod.html

45. Paths for Referring to an Item in the Module Tree - The Rust ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch07-03-paths-for-referring-to-an-item-in-the-module-tree.html
46. Bringing Paths Into Scope with the use Keyword - The Rust ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch07-04-bringing-paths-into-scope-with-the-use-keyword.html
47. Separating Modules into Different Files - The Rust Programming ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch07-05-separating-modules-into-different-files.html
48. Package Layout - The Cargo Book, accessed May 30, 2025, https://doc.rust-lang.org/cargo/guide/project-layout.html
49. Storing Lists of Values with Vectors - The Rust Programming ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch08-01-vectors.html
50. Vec in std::vec - Rust, accessed May 30, 2025, https://doc.rust-lang.org/std/vec/struct.Vec.html
51. Storing UTF-8 Encoded Text with Strings - The Rust Programming ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch08-02-strings.html
52. String in std::string - Rust, accessed May 30, 2025, https://doc.rust-lang.org/std/string/struct.String.html
53. Storing Keys with Associated Values in Hash Maps - The Rust ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch08-03-hash-maps.html
54. HashMap in std::collections - Rust, accessed May 30, 2025, https://doc.rust-lang.org/std/collections/struct.HashMap.html
55. Unrecoverable Errors with panic! - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch09-01-unrecoverable-errors-with-panic.html
56. Recoverable Errors with Result - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html
57. To panic! or Not to panic! - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch09-03-to-panic-or-not-to-panic.html
58. Generic Data Types - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch10-01-syntax.html
59. Generics - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/generics.html
60. Traits: Defining Shared Behavior - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch10-02-traits.html
61. Traits - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/trait.html
62. Validating References with Lifetimes - The Rust Programming ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html
63. Functions - Rust By Example - Rust Documentation, accessed May 30, 2025,

https://doc.rust-lang.org/rust-by-example/scope/lifetime/fn.html
64. Structs - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/scope/lifetime/struct.html
65. accessed December 31, 1969, https://doc.rust-lang.org/rust-by-example/scope/lifetime/static.html
66. Subtyping and Variance - The Rustonomicon, accessed May 30, 2025, https://doc.rust-lang.org/nomicon/subtyping.html
67. accessed December 31, 1969, https://doc.rust-lang.org/book/ch19-02-advanced-lifetimes.html
68. Higher-Rank Trait Bounds - The Rustonomicon, accessed May 30, 2025, https://doc.rust-lang.org/nomicon/hrtb.html
69. How to Write Tests - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch11-01-writing-tests.html
70. Testing - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/testing.html
71. Controlling How Tests Are Run - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch11-02-running-tests.html
72. Test Organization - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch11-03-test-organization.html
73. Closures: Anonymous Functions that Capture Their Environment ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch13-01-closures.html
74. Closures - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/fn/closures.html
75. Advanced Functions and Closures - The Rust Programming Language, accessed May 30, 2025, https://doc.rust-lang.org/book/ch19-05-advanced-functions-and-closures.html
76. Processing a Series of Items with Iterators - The Rust Programming ..., accessed May 30, 2025, https://doc.rust-lang.org/book/ch13-02-iterators.html
77. accessed December 31, 1969, https://doc.rust-lang.org/rust-by-example/iter.html
78. Higher Order Functions - Rust By Example - MIT, accessed May 30, 2025, https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/rust-by-example/fn/hof.html
79. Higher Order Functions - Rust By Example - Rust Documentation, accessed May 30, 2025, https://doc.rust-lang.org/rust-by-example/fn/hof.html