# Fundamentos de Rust com Paradigma Funcional

Rust é uma linguagem segura e eficiente, com suporte ao paradigma funcional. Este material cobre os principais fundamentos da linguagem, enfatizando boas práticas funcionais.

## 1. Variáveis e Imutabilidade

Por padrão, as variáveis em Rust são imutáveis. Isso evita efeitos colaterais e torna o código mais previsível.

fn main() {

let x = 10; // Define uma variável imutável

println!("{}", x); // Exibe o valor de x no console

}

Para permitir alteração, use mut:

fn main() {

let mut x = 10; // Define uma variável mutável

x += 5; // Modifica o valor da variável

println!("{}", x); // Exibe o novo valor de x

}

## 2. Entrada e Saída de Dados

A leitura de entrada do usuário pode ser feita com stdin():

use std::io;

fn main() {

let mut entrada = String::new(); // Cria uma string vazia

io::stdin().read\_line(&mut entrada).expect("Erro ao ler linha"); // Lê entrada do teclado

println!("Voce digitou: {}", entrada.trim()); // Exibe a entrada removendo espaços extras

}

## 3. Estruturas de Decisão

O if em Rust funciona de forma similar a outras linguagens:

fn main() {

let numero = 10;

if numero > 5 {

println!("Maior que 5");

} else {

println!("Menor ou igual a 5");

}

}

O if pode ser usado como expressão:

fn main() {

let numero = 10;

let mensagem = if numero > 5 { "Maior" } else { "Menor" };

println!("{}", mensagem); // Exibe "Maior"

}

## 4. Estruturas de Repetição

### Iteradores com for

Percorre um intervalo de valores:

fn main() {

for x in 1..=5 {

println!("{}", x); // Exibe valores de 1 a 5

}

}

### Iteradores com while

Executa enquanto a condição for verdadeira:

fn main() {

let mut x = 0;

while x < 5 {

println!("{}", x);

x += 1;

}

}

## 5. Listas (Vetores)

Vetores são coleções de elementos do mesmo tipo:

fn main() {

let numeros = vec![1, 2, 3, 4, 5]; // Cria um vetor

for n in &numeros {

println!("{}", n); // Itera e exibe os elementos

}

}

Utilizando map para transformar elementos:

fn main() {

let numeros = vec![1, 2, 3, 4, 5];

let dobrados: Vec<i32> = numeros.iter().map(|x| x \* 2).collect();

println!("{:?}", dobrados); // Exibe [2, 4, 6, 8, 10]

}

## 6. Dicionários (HashMap)

O HashMap permite associar chaves a valores:

use std::collections::HashMap;

fn main() {

let mut dicionario = HashMap::new();

dicionario.insert("chave1", "valor1");

dicionario.insert("chave2", "valor2");

for (chave, valor) in &dicionario {

println!("{}: {}", chave, valor); // Exibe as chaves e valores

}

}

## 7. Funções e Closures

Funções são definidas com fn:

fn soma(a: i32, b: i32) -> i32 {

a + b // Retorna a soma dos números

}

fn main() {

println!("{}", soma(2, 3)); // Exibe 5

}

Closures são funções anônimas:

fn main() {

let quadrado = |x: i32| x \* x;

println!("{}", quadrado(4)); // Exibe 16

}

## 8. Manipulação de Strings

O String é uma versão dinamicamente alocada de string:

fn main() {

let texto = String::from("Ola, Rust!");

let tamanho = texto.len();

println!("Tamanho: {}", tamanho); // Exibe o tamanho da string

}

## 9. Trabalhando com Result e Option

O Option<T> representa valores opcionais:

fn dividir(a: i32, b: i32) -> Option<i32> {

if b == 0 {

None

} else {

Some(a / b)

}

}

fn main() {

match dividir(10, 2) {

Some(resultado) => println!("Resultado: {}", resultado),

None => println!("Divisão por zero"),

}

}

O Result<T, E> trata erros explicitamente:

fn dividir(a: i32, b: i32) -> Result<i32, &'static str> {

if b == 0 {

Err("Divisão por zero")

} else {

Ok(a / b)

}

}

fn main() {

match dividir(10, 0) {

Ok(resultado) => println!("Resultado: {}", resultado),

Err(erro) => println!("Erro: {}", erro),

}

}

## Conclusão

Rust permite programar de forma funcional ao priorizar imutabilidade, uso de iteradores e funções puras. Esses conceitos ajudam a escrever código mais seguro e eficiente.

Rust possui um sistema de tipos forte e estático, suportando os seguintes tipos de variáveis:

### ****1. Tipos Escalares****

São valores únicos armazenados em uma variável.

* **Números inteiros (**i8**,** i16**,** i32**,** i64**,** i128**,** isize**)** – Representam números inteiros com sinal.
* **Números inteiros sem sinal (**u8**,** u16**,** u32**,** u64**,** u128**,** usize**)** – Apenas números positivos.
* **Números de ponto flutuante (**f32**,** f64**)** – Representam números decimais.
* **Booleanos (**bool**)** – true ou false.
* **Caracteres (**char**)** – Representam um único caractere Unicode (por exemplo, 'A', 'é', '🚀').

### ****2. Tipos Compostos****

Podem armazenar múltiplos valores.

* **Tuplas (**(T1, T2, ...)**)** – Contêm múltiplos valores de diferentes tipos.

let pessoa: (&str, i32, f64) = ("João", 30, 1.75);

* **Arrays (**[T; N]**)** – Contêm múltiplos valores do mesmo tipo.

let numeros: [i32; 3] = [1, 2, 3];

### ****3. Tipos de Coleção (Dinamicamente Alocados)****

* **Vetores (**Vec<T>**)** – Arrays redimensionáveis.

let mut v = vec![1, 2, 3];

v.push(4);

* **String (**String**)** – Cadeias de caracteres dinâmicas.

let texto = String::from("Olá, Rust!");

* **HashMap (**HashMap<K, V>**)** – Dicionários para mapear chaves a valores.

use std::collections::HashMap;

let mut mapa = HashMap::new();

mapa.insert("chave", 42);

### ****4. Tipos Especiais****

* Option<T> – Representa um valor opcional (Some(valor) ou None).

let talvez\_numero: Option<i32> = Some(10);

* Result<T, E> – Representa operações que podem falhar (Ok(valor) ou Err(erro)).

fn dividir(a: i32, b: i32) -> Result<i32, &'static str> {

if b == 0 {

Err("Divisão por zero")

} else {

Ok(a / b)

}

}