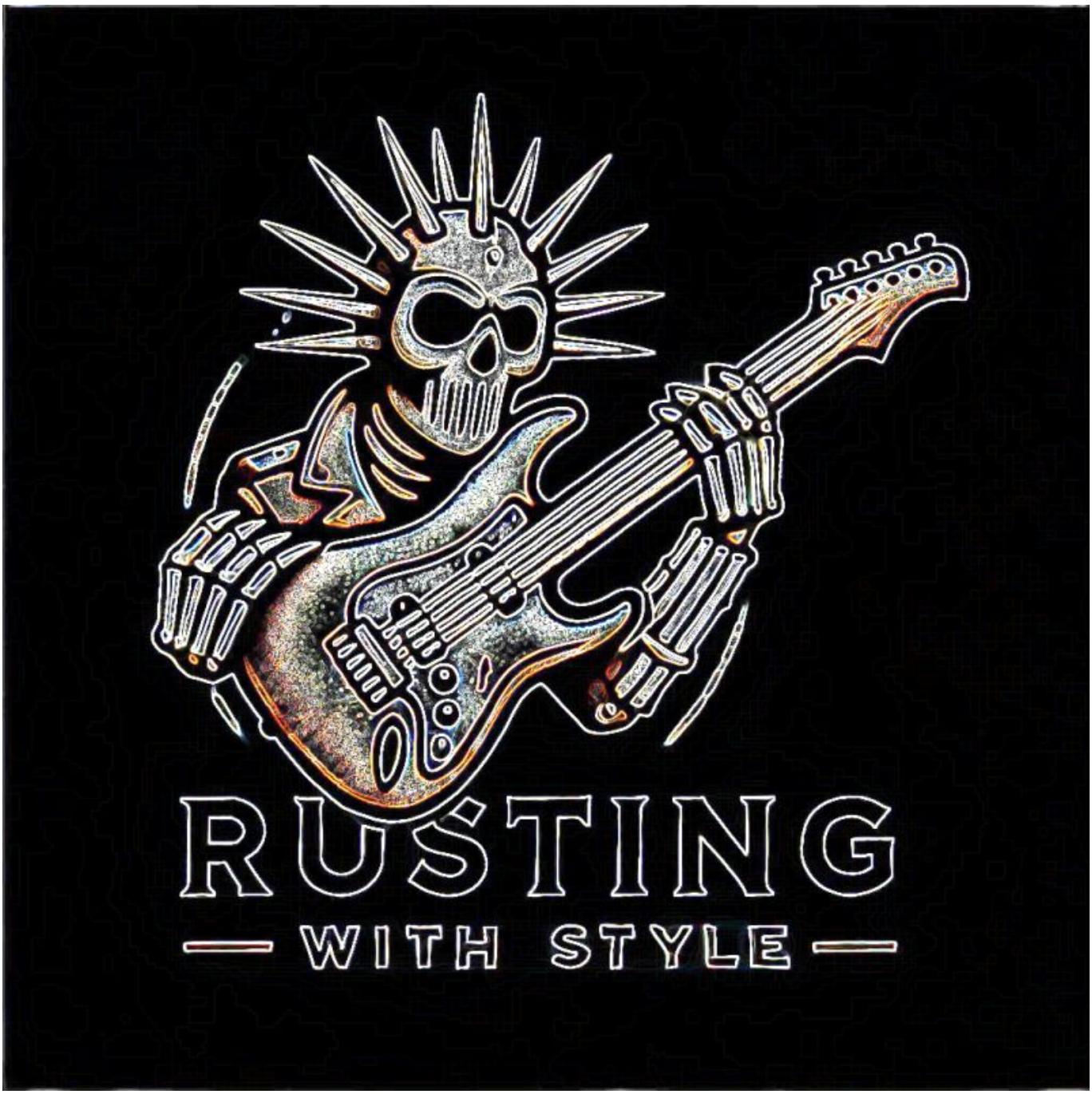


Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

Certificado do curso

Este curso é totalmente gratuito, exceto pelo **certificado**. Você pode fazer o curso e nem se preocupar com o certificado. Porém, se quiser um certificado de conclusão, precisará agendar **NESTE LINK** abra o menu **contatos** e preencha o formulário solicitando o agendamento.

- O certificado é obtido após entrevista ao vivo previamente agendada.
- Esta entrevista será cobrada.
- Caso o aluno não obtenha grau de aprovação, poderá reagendar uma vez gratuitamente.
- O objetivo é evitar o uso de IA, portanto, alunos que tentarem utilizar serão desclassificados.
- Ao agendar a entrevista de certificação, você receberá um pacote de informações.

Conhecendo Rust

VÍDEO DESTA AULA

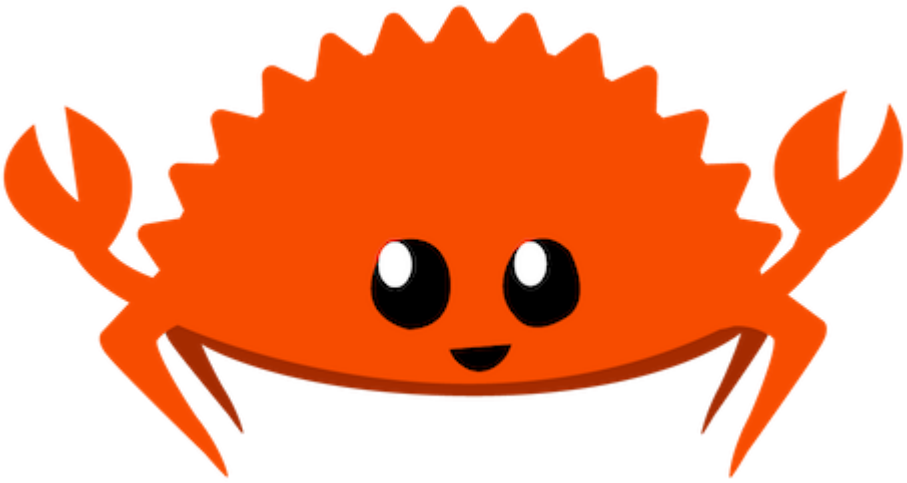
Rust é uma linguagem de programação criada pela Mozilla Research em 2010, liderada inicialmente por **Graydon Hoare**.

Sua proposta surgiu como uma resposta à necessidade de criar software seguro, eficiente e concorrente, sem os riscos comuns de linguagens como C e C++. Rust foi projetada com um foco especial na segurança de memória, eliminando problemas como ponteiros nulos e condições de corrida através de um modelo de propriedade único, que combina verificações em tempo de compilação com alta performance em tempo de execução.

A linguagem cresceu em popularidade ao ser adotada em projetos como o Servo (um engine de browser experimental) e outros sistemas críticos, consolidando-se como uma escolha moderna para desenvolvimento de sistemas, aplicações web e muito mais.

O "mascote" da linguagem **Rust** é o carangueijo **Ferris**. Ele é um simpático caranguejo, geralmente desenhado em estilo minimalista e com um sorriso amigável. Ferris não foi criado oficialmente pela equipe principal de Rust, mas surgiu da comunidade de desenvolvedores e rapidamente foi adotado como um símbolo da linguagem devido ao seu apelo carismático.

O nome **Ferris** é uma brincadeira com a palavra "ferric", que remete ao ferro, conectando ao nome "Rust" (**ferrugem**). Ele representa a energia e a camaradagem da comunidade Rust, refletindo o espírito colaborativo e acolhedor dos desenvolvedores que utilizam a linguagem.



Características da linguagem Rust

1. **Segurança de memória sem coletor de lixo**: Rust previne erros como ponteiros nulos e uso de memória após liberação (use-after-free) usando seu sistema de propriedade. Ele verifica tudo em tempo de compilação, eliminando a necessidade de um coletor de lixo (garbage collector) que muitas linguagens usam.

2. **Concorrência sem medo**: Rust facilita a criação de programas que usam múltiplas threads ao garantir segurança contra condições de corrida (race conditions). Seu sistema de tipos verifica que variáveis compartilhadas entre threads estão protegidas, sem deixar brechas.

3. **Performance comparável a C/C++**: Como não usa coletor de lixo e compila diretamente para código nativo, Rust entrega a mesma eficiência que linguagens como C ou C++, sendo ideal para sistemas onde a velocidade é crucial.

4. **Gerenciamento automático de memória**: Apesar de não ter coletor de lixo, Rust automatiza o gerenciamento de memória com o conceito de **ownership** (propriedade). Isso permite que você trabalhe de forma eficiente sem precisar alocar ou liberar memória manualmente.

5. **Comunidade e ferramentas robustas**: Rust possui um ecossistema rico, com gerenciadores de pacotes como o Cargo e uma comunidade ativa. O compilador fornece mensagens de erro detalhadas e úteis, tornando o aprendizado e a depuração mais fáceis para desenvolvedores.

Diferenças para outras linguagens

Em **Rust**, os conceitos de variáveis, funções, structs e traits oferecem uma abordagem que combina paradigmas procedurais e funcionais, contrastando com linguagens mais orientadas a objetos como **Java**.

- **Variáveis**: Em Rust, as variáveis são imutáveis por padrão (usando `let`). Para permitir a mutabilidade, é necessário usar `mut`. Isso incentiva um estilo de programação mais seguro, ao contrário de Java, onde as variáveis são mutáveis por padrão. Além disso, o sistema de propriedade de Rust controla o uso de memória, algo ausente em Java.

```
let x = 10; // Imutável
let mut y = 20; // Mutável
```

- **Funções**: Rust adota uma abordagem funcional e procedural. As funções são definidas com a palavra-chave `fn` e podem ter tipos explícitos para os parâmetros e o retorno. Ao contrário de Java, Rust não exige que as funções estejam dentro de uma classe, permitindo maior flexibilidade.

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

- **Structs**: As `structs` em Rust são semelhantes a classes em Java, mas são mais simples, pois não possuem métodos nem herança diretamente embutidos. Métodos associados a structs são definidos separadamente, usando blocos `impl`. Diferentemente de Java, structs não possuem visibilidade pública por padrão, incentivando o encapsulamento explícito.

```
struct Pessoa {
    nome: String,
    idade: u32,
}

impl Pessoa {
    fn nova(nome: String, idade: u32) -> Self {
        Self { nome, idade }
    }
}
```

- **Traits**: Traits em Rust são semelhantes às interfaces em Java, mas mais poderosas. Elas permitem definir comportamento compartilhado para diferentes tipos e podem ter métodos padrão implementados. Diferentemente de Java, Rust não usa herança tradicional; em vez disso, a composição de traits é usada para compartilhar funcionalidades entre tipos.

```
trait Saudacao {
    fn saudar(&self) -> String;
}

struct Pessoa {
    nome: String,
}

impl Saudacao for Pessoa {
    fn saudar(&self) -> String {
        format!("Olá, meu nome é {}", self.nome)
    }
}
```

Enquanto Java é altamente centrado em classes e herança, Rust oferece uma abordagem mais modular, com forte foco em segurança, eficiência e abstração através de traits. Isso o torna mais adequado para sistemas de baixo nível e concorrência segura, enquanto Java é amplamente usado em sistemas corporativos e aplicativos onde a herança e a orientação a objetos predominam.

Turbo setup

Chega de papo furado! Bora trabalhar! Vamos instalar **Rust** e criar um projeto simples. Instalar Rust é seguir o que diz o site **rustup.rs**. Em plataformas **Linux** ou **macOS**:

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

Se usa Microsoft Windows veja o **suporte para outros instaladores**.

Depois de instalar, se estiver utilizando algum sabor de **Unix**, feche e abra um terminal novamente. Depois é só testar:

```
cargo --version
```

Cargo é o gerenciador de projetos e dependências do Rust. Com ele você pode: Criar novos projetos, compilar, executar e adicionar dependências. Para criar um projeto basta:

```
cargo new fibo --bin
```

Este comando cria uma pasta chamada `fibo` contendo um projeto **Rust** de executável (`--bin`) em vez de biblioteca. Nesta pasta você verá:

```
fibo
|
+--Cargo.toml
+--src
   |
   +--main.rs
```

Nesta pasta vemos o arquivo do projeto `Cargo.toml` (pronuncia-se: "tom-el") que descreve seu projeto e suas dependências:

```
[package]
name = "fibo"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
```

Como pode ver, não temos dependências. E é gerado um arquivo simples `main.rs` com um "hello world":

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

Para executar o programa vá para a pasta `fibo` e rode o comando:

```
cargo run
```

Agora, para implementar o código de hoje, substitua o conteúdo de `main.rs` por:

```
fn fibo(n: u32) -> u32 {
    if n < 2 {
        return n;
    }
    return fibo(n - 1) + fibo(n - 2);
}

fn main() {
    println!("Fibo de 0: {}", fibo(0));
    println!("Fibo de 1: {}", fibo(1));
    println!("Fibo de 2: {}", fibo(2));
    println!("Fibo de 3: {}", fibo(3));
    println!("Fibo de 8: {}", fibo(8));
}
```

E execute. O resultado deve ser:

```
$ cargo run
   Compiling fibo v0.1.0
(/home/cleuton/Documents/projetos/rustingcrab/rusting_with_style_PTBR/lição
1 - Conhecendo/codigo/fibo)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.13s
     Running `target/debug/fibo`
Fibo de 0: 0
Fibo de 1: 1
Fibo de 2: 1
Fibo de 3: 2
Fibo de 8: 21
```

Explicação do código fonte

Sei que nada vimos sobre Rust mas quero explicar esse código fonte. Esse código implementa uma função recursiva em Rust para calcular números da sequência de Fibonacci. Vamos explicá-lo passo a passo de forma simples:

Função `fibo`

```rust
fn fibo(n: u32) -> u32 {
    if n < 2 {
        return n;
    }
    return fibo(n - 1) + fibo(n - 2);
}
```

1. A função `fibo` recebe um número inteiro positivo (`u32`) como entrada (`n`) e também retorna um número inteiro positivo (`u32`).
2. **Caso base**: Se `n` for menor que 2 (ou seja, 0 ou 1), a função retorna o próprio número `n`. Isso reflete o fato de que os dois primeiros números da sequência de Fibonacci são 0 e 1.
3. **Passo recursivo**: Para números maiores ou iguais a 2, a função calcula o valor atual somando os dois números anteriores da sequência, chamando `fibo(n - 1)` e `fibo(n - 2)`.

Função `main`

```rust
fn main() {
    println!("Fibo de 0: {}", fibo(0));
    println!("Fibo de 1: {}", fibo(1));
    println!("Fibo de 2: {}", fibo(2));
    println!("Fibo de 3: {}", fibo(3));
    println!("Fibo de 8: {}", fibo(8));
}
```

1. A função `main` é o ponto de entrada do programa. Tudo começa a ser executado a partir daqui.
2. **Imprimindo resultados**: Usa-se `println!` para mostrar no console o resultado da função `fibo` para diferentes valores de entrada (0, 1, 2, 3, 8).
3. O `{}` dentro da string é um **placeholder**. Ele será substituído pelo valor retornado por `fibo()` para cada número.

O que o código faz

Quando você executa o programa, ele calcula e imprime os números de Fibonacci para as entradas 0, 1, 2, 3 e 8:

- **Fibonacci de 0**: `fibo(0)` retorna 0.
- **Fibonacci de 1**: `fibo(1)` retorna 1.

- **Fibonacci de 2**: `fibo(2)` chama `fibo(1) + fibo(0)` e retorna 1.
- **Fibonacci de 3**: `fibo(3)` chama `fibo(2) + fibo(1)` e retorna 2.
- **Fibonacci de 8**: Calcula a soma recursiva dos dois números anteriores até chegar ao resultado final, que é 21.

Resultado no console

Quando você rodar o código, verá algo como:

```
Fibo de 0: 0
Fibo de 1: 1
Fibo de 2: 1
Fibo de 3: 2
Fibo de 8: 21
```

Esse é um exemplo simples, mas mostra como Rust lida bem com recursão, mesmo com tipos fortes e verificações em tempo de compilação.

> **Por que o arquivo se chama main?**

O arquivo principal em um projeto Rust se chama **`main.rs`** por convenção, mas isso pode variar dependendo da estrutura do projeto. Vamos entender por quê e se é possível usar outro nome:

1. **Por que `main.rs`?**
   - No contexto de Rust, o arquivo principal do código é aquele que contém a função `main`, que é o ponto de entrada do programa.
   - Se você criar um novo projeto com o `cargo new`, o gerenciador de projetos do Rust cria automaticamente um arquivo `main.rs` dentro da pasta `src`. Isso facilita a organização e segue uma convenção que outros desenvolvedores entendem facilmente.
2. **Pode ter outro nome?**
   - Sim, mas com limitações.
   - O compilador `rustc` permite compilar qualquer arquivo com código válido. Por exemplo, você pode ter um arquivo chamado `meu_programa.rs` e compilá-lo com:

     ```
     rustc meu_programa.rs
     ```

   - No entanto, se você estiver usando o `Cargo`, ele espera que o arquivo principal de um projeto binário esteja no diretório `src` e seja chamado `main.rs`. Alterar o nome exigiria modificar a estrutura padrão do projeto.
3. **Modificando o nome com Cargo**
   - Se você quiser usar um nome diferente com `Cargo`, pode criar um projeto com múltiplos binários. Nesse caso, você organiza o código na pasta `src/bin`, e cada arquivo nessa pasta

pode ter um nome diferente.
- Exemplo:

```
src/
  main.rs
  bin/
    outro_nome.rs
```

Nesse caso, `Cargo` identificará os arquivos em `src/bin` como binários adicionais.

4. **Por que seguir a convenção?**
   - Embora seja possível usar outro nome, seguir a convenção `main.rs` torna seu projeto mais fácil de entender por outros desenvolvedores e ferramentas que interagem com Rust.

Se não houver um motivo específico para mudar, é uma boa prática manter o arquivo principal como `main.rs`.