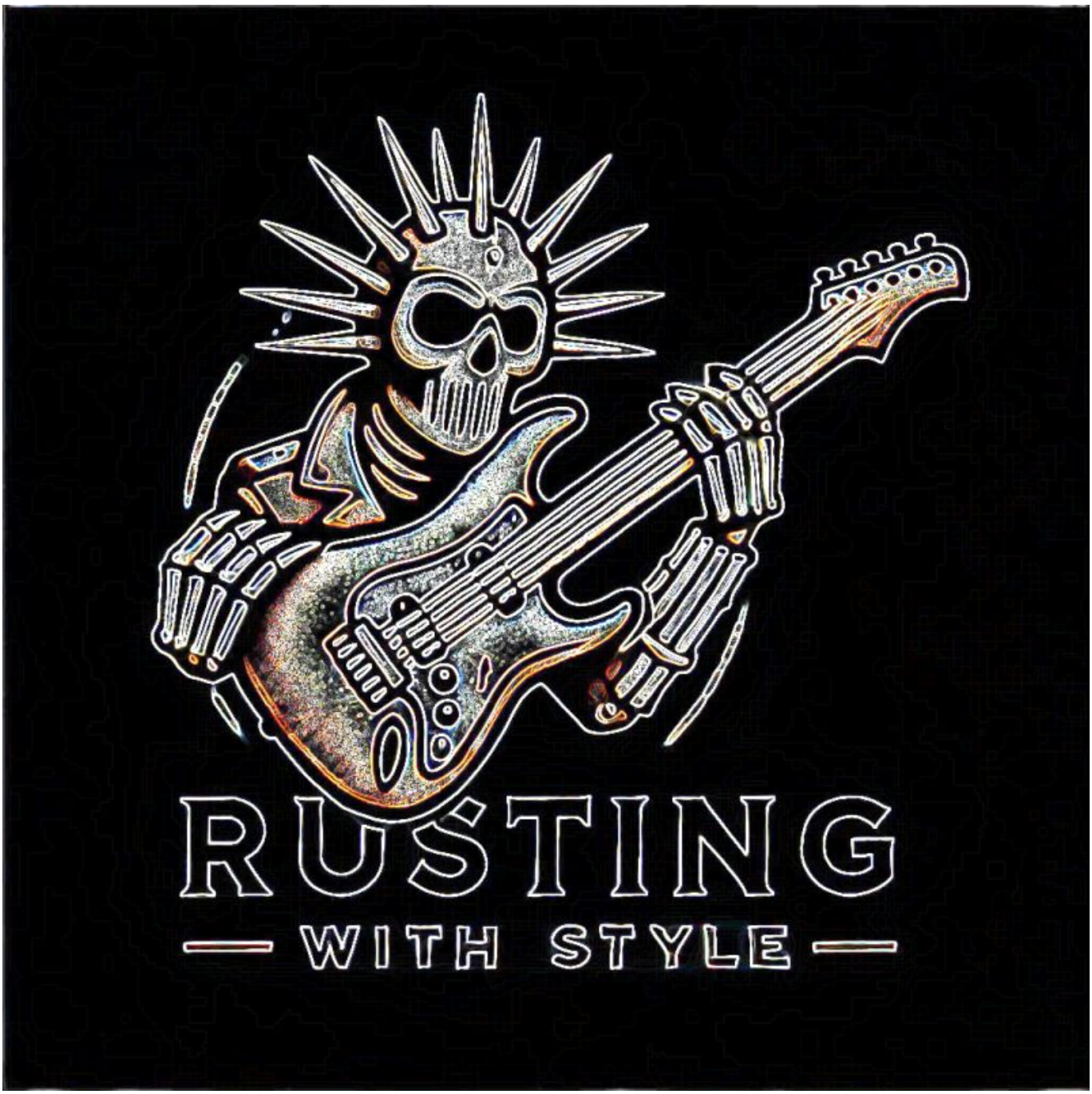

Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Estrutura

A linguagem **Rust** tem a simplicidade estrutural da linguagem **C**. Outras linguagens como **Java** possuem estruturas muito mais rígidas e verbosas.

```rust
// Usamos a instrução "use" para importar uma função ou tipo específico de
outro módulo
use std::cmp::max;

/// Estrutura que representa uma pessoa
struct Pessoa {
    nome: String,
    idade: u8,
}

impl Pessoa {
    /// Método associado que cria uma nova Pessoa
    fn new(nome: String, idade: u8) -> Self {
        Pessoa { nome, idade }
    }

    /// Método que verifica se a pessoa é maior de idade
    fn maior_de_idade(&self) -> bool {
        self.idade >= 18
    }

    /// Método que retorna uma saudação personalizada
    fn saudacao(&self) -> String {
        format!("Olá, meu nome é {} e eu tenho {} anos.", self.nome,
self.idade)
    }
}

/// Função simples que calcula o maior de dois números
fn maior_numero(a: i32, b: i32) -> i32 {
    max(a, b)
}

/// Função principal do programa
fn main() {
    // Criando variáveis simples
    let x = 10;
    let y = 20;

    // Chamando a função maior_numero
    let maior = maior_numero(x, y);
    println!("O maior número entre {} e {} é {}.", x, y, maior);

    // Criando uma instância de Pessoa
    let pessoa = Pessoa::new(String::from("Fulano"), 25);
```

```
    // Usando os métodos da estrutura Pessoa
    println!("{}", pessoa.saudacao());
    if pessoa.maior_de_idade() {
        println!("{} é maior de idade.", pessoa.nome);
    } else {
        println!("{} não é maior de idade.", pessoa.nome);
    }

    // Usando uma macro (println!) para imprimir uma mensagem
    println!("Este é um exemplo de programa Rust!");
}
```

Explicação do Código para Iniciantes em Rust

1. Estrutura de um Programa Rust

- O programa começa com `fn main()`. Esta é a função principal que será executada quando o programa for rodado.
- `main` pode chamar outras funções e métodos.

2. Importando Recursos com `use`

- A instrução `use std::cmp::max` importa a função `max` do módulo `std::cmp`, permitindo comparações entre números.

3. Criando Estruturas com `struct`

- `struct Pessoa` é como um "molde" que descreve uma pessoa, com `nome` e `idade`.
- Os métodos associados (`new`, `maior_de_idade`, `saudacao`) são implementados usando `impl`.

4. Funções Simples

- Funções em Rust são declaradas com `fn`. Exemplo:

```
fn maior_numero(a: i32, b: i32) -> i32 {
    max(a, b)
}
```

  - Esta função recebe dois números inteiros (`a` e `b`) e retorna o maior deles, usando a função `max`.

5. Diferença entre Macros e Funções

Macros

- `println!` é uma **macro**, indicada pelo `!`.
- **Macros** são avaliadas em tempo de compilação e podem aceitar diferentes tipos de argumentos, expandindo para código Rust. Exemplo:

```
println!("O maior número é {}.", maior);
```

Funções

- `max` e `maior_numero` são **funções**, que executam um conjunto fixo de operações em tempo de execução.
- Funções recebem argumentos e retornam valores:

```
fn maior_numero(a: i32, b: i32) -> i32 {
    max(a, b)
}
```

6. Variáveis

- `let` é usado para declarar variáveis. Exemplo:

```
let x = 10;
let y = 20;
```

- Em Rust, as variáveis são imutáveis por padrão, mas você pode torná-las mutáveis com `mut`:

```
let mut z = 30;
z = 40; // Agora é permitido.
```

7. Métodos

- Métodos são funções associadas a uma `struct` e podem acessar seus campos com `self`. Exemplo:

```
fn saudacao(&self) -> String {
    format!("Olá, meu nome é {} e eu tenho {} anos.", self.nome,
self.idade)
}
```

Resumo

Este código mostra como:

1. Criar e usar **funções** e **macros**.
2. Declarar uma **struct** e implementar métodos com `impl`.
3. Usar a instrução `use` para importar funções prontas.
4. Trabalhar com variáveis e lógica básica.

USE ou não USE?

No Rust, a instrução `use` é usada para simplificar o acesso a itens de outros módulos ou bibliotecas. Contudo, nem sempre é necessário usá-la. Aqui está um guia sobre **quando usar `use` e quando não é necessário**:

Quando usar `use`

1. **Para evitar nomes longos e repetitivos**
   - Quando você precisa acessar itens que estão em módulos ou namespaces profundos, `use` pode simplificar o código:

```
// Sem `use`
let maior = std::cmp::max(10, 20);

// Com `use`
use std::cmp::max;
let maior = max(10, 20);
```

2. **Ao usar tipos ou funções várias vezes**
   - Se um item for usado em várias partes do código, é mais eficiente importá-lo:

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

fn main() {
    let mut mapa = HashMap::new();
    mapa.insert("chave", "valor");
}
```

3. **Para organizar dependências externas**
   - Itens de crates externas frequentemente exigem `use` para serem acessados:

```
use serde::Serialize;

#[derive(Serialize)]
struct Pessoa {
    nome: String,
    idade: u8,
}
```

> Um **crate** é a unidade básica de compilação e organização de código no Rust, podendo ser uma biblioteca ou um executável. Ele é o contêiner onde seu código e dependências são agrupados, funcionando como um módulo reutilizável. O **Cargo** é a ferramenta de

> gerenciamento de projetos no Rust que facilita a criação, compilação e gerenciamento de crates, além de resolver dependências automaticamente. A instrução **use** é usada dentro de um crate para importar itens de outros módulos ou de dependências externas gerenciadas pelo Cargo, permitindo acessar funcionalidades sem precisar especificar caminhos completos. Assim, o Cargo gerencia os crates, e o use facilita o acesso ao que eles oferecem.

4. **Ao importar módulos específicos**
   - Você pode usar use para importar apenas partes do módulo, em vez de trazer todo o namespace:

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

fn main() {
    io::stdout().write_all(b"Olá, Rust!").unwrap();
}
```

5. **Com aliases para evitar conflitos**
   - Quando há itens com o mesmo nome, você pode renomeá-los usando as:

```
use std::io::Result as IoResult;

fn escreve() -> IoResult<()> {
    Ok(())
}
```

Quando não é necessário usar use

1. **Para itens no mesmo módulo**
   - Itens como funções ou structs declarados no mesmo módulo podem ser usados diretamente:

```
fn exemplo() {
    println!("Função no mesmo módulo!");
}

fn main() {
    exemplo(); // Não precisa de `use`
}
```

2. **Para itens no módulo std de uso comum**
   - Alguns itens são automaticamente disponíveis (pré-importados) no Rust, como:
     - println!, vec!, String, Option, Result, etc.

```rust
fn main() {
    let nome = String::from("Rust"); // Não precisa de `use`
    println!("Olá, {}!", nome);
}
```

3. **Quando usar um caminho completo**
   - Se você usar o caminho completo, não precisa de `use`:

```rust
fn main() {
    let maior = std::cmp::max(10, 20); // Sem `use`
    println!("O maior número é {}.", maior);
}
```

4. **Para itens de módulos locais ou externos explicitamente referenciados**
   - Você pode acessar itens de um módulo local sem `use`:

```rust
mod util {
    pub fn saudacao() {
        println!("Olá!");
    }
}

fn main() {
    util::saudacao(); // Acesso direto sem `use`
}
```

Usar ou Não `use`?

Use `use` quando:

- Você precisa usar o mesmo item várias vezes.
- Deseja deixar o código mais legível e evitar caminhos completos.
- Está importando itens de bibliotecas externas ou módulos profundos.

Não use `use` quando:

- Você está acessando algo pré-importado ou definido localmente.
- Está usando o caminho completo para itens que aparecem apenas uma vez.

Módulos

Um **módulo** em Rust é uma forma de organizar e agrupar código, permitindo estruturar projetos de maneira hierárquica. Ele é usado para dividir o código em blocos menores, tornando-o mais fácil de entender, reutilizar e manter. Os módulos podem conter funções, structs, enums, constantes, e até outros módulos.

Características dos Módulos:

- São declarados com a palavra-chave `mod`.
- Podem ser definidos no mesmo arquivo ou em arquivos separados.
- Controlam a visibilidade de itens com `pub` (público) ou sem `pub` (privado por padrão).
- Facilitam o uso de caminhos para acessar os itens agrupados.

Exemplo:

```
mod util {
    pub fn saudacao() {
        println!("Olá, Rust!");
    }
}

fn main() {
    util::saudacao(); // Chamando a função dentro do módulo
}
```

Os módulos ajudam a organizar projetos grandes e permitem encapsular lógica, mantendo o código mais limpo e modular.

No Rust, o **caminho** de um item em um módulo é usado para localizar e acessar funções, structs, enums, constantes ou outros itens organizados na estrutura hierárquica de um programa ou biblioteca. O operador `::` é usado para navegar por essa hierarquia, separando os níveis.

Exemplos:

1. **Caminho absoluto**: Começa da raiz do crate ou de um módulo externo:

```
let resultado = std::cmp::max(10, 20);
```

Aqui, `std` é o módulo padrão, `cmp` é um submódulo, e `max` é a função.

2. **Caminho relativo**: Baseado na localização atual do código, usando módulos locais:

```
mod util {
    pub fn saudacao() {
        println!("Olá!");
    }
}

fn main() {
    util::saudacao(); // Caminho relativo
}
```

O operador `::` é, portanto, a maneira de "seguir o caminho" até o item desejado, seja de um módulo local ou externo.

Reusando código

Em outras linguagens de programação, como **Python** é possível reusar código (funções etc) com pouca ou nenhuma "burocracia". Em **Rust** isso é diferente. Há 3 maneiras de você "importar" código de outro programa dentro do seu:

1. Crie um módulo.
2. Crie uma biblioteca "lib".
3. Use a macro `include!`.

Se você precisa criar código que será utilizado em mais de um programa, então coloque esse código em uma biblioteca Rust e utilize a instrução `use` para importar o que desejar.

lib

Se você quiser "importar" código de outro programa em Rust sem usar módulos, normalmente faz isso ao transformar o outro programa em uma **biblioteca** (um `crate` do tipo biblioteca) e adicioná-lo como dependência ao seu projeto principal. Isso é feito com o **Cargo**, o gerenciador de pacotes do Rust.

Aqui estão os passos resumidos:

1. Transforme o programa em uma biblioteca

- No programa que contém o código que você quer usar, ajuste o `Cargo.toml` para criar uma biblioteca. Certifique-se de que ele tem:

```
[lib]
name = "nome_da_biblioteca"
path = "src/lib.rs"
```

- Coloque o código que deseja "importar" no arquivo `src/lib.rs`.

Exemplo de `lib.rs`:

```
pub fn saudacao() {
    println!("Olá do outro programa!");
}
```

2. Adicione como dependência ao programa principal

- No projeto onde você quer usar o código, adicione o outro programa como dependência no `Cargo.toml`. Se o programa estiver em um diretório local:

```
[dependencies]
nome_da_biblioteca = { path = "../caminho_do_outro_programa" }
```

3. Use o código da biblioteca

- Depois de adicionar a dependência, você pode usar a instrução `use` para importar os itens da biblioteca:

```
use nome_da_biblioteca::saudacao;

fn main() {
    saudacao();
}
```

Macro `include!`

Se o outro código não usa módulos ou bibliotecas e está em um arquivo separado, você pode incluir o código com a macro `include!`. Isso essencialmente copia o conteúdo do arquivo durante a compilação.

Exemplo:

- Código no arquivo `outro_programa.rs`:

```
pub fn saudacao() {
    println!("Olá de outro programa!");
}
```

- No seu código principal:

```
include!("../caminho_para/outro_programa.rs");

fn main() {
    saudacao(); // A função do outro arquivo agora está acessível
}
```

> **Nota:** Esta abordagem pode funcionar, mas é considerada um **mau hábito** para projetos grandes, porque não há encapsulamento ou reutilização clara.

Recomendação

Embora seja possível usar abordagens como `include!` ou subprocessos, a melhor prática em Rust é sempre estruturar o código como módulos (`mod`) ou bibliotecas (`lib.rs`). Isso facilita a manutenção, o reuso e garante que o código esteja alinhado com as convenções do Rust. Se o outro código for reutilizável, considere refatorá-lo em uma biblioteca.

Resumo da aula

1. Vimos os elementos que um programa **Rust** pode conter: Funções, structs, importações de módulos, declarações de módulos etc.
2. Vimos como as funções e métodos são declarados e que uma **Struct** é semelhante a uma **Classe**.
3. Vimos brevemente como declarar variáveis.
4. Vimos o que é um **módulo**, como é especificado o **caminho** de seus elementos exportados e como ele pode ser importado dentro de um programa **Rust**.
5. Vimos o que é uma **biblioteca**, como é declarada e como podemos utilizá-la em outros programas **Rust**.