

A linguagem Rust e abstrações de alto nível



SECOMP 2023

Brenno Lemos

-  Syndelis
-  @brenno@fosstodon.org



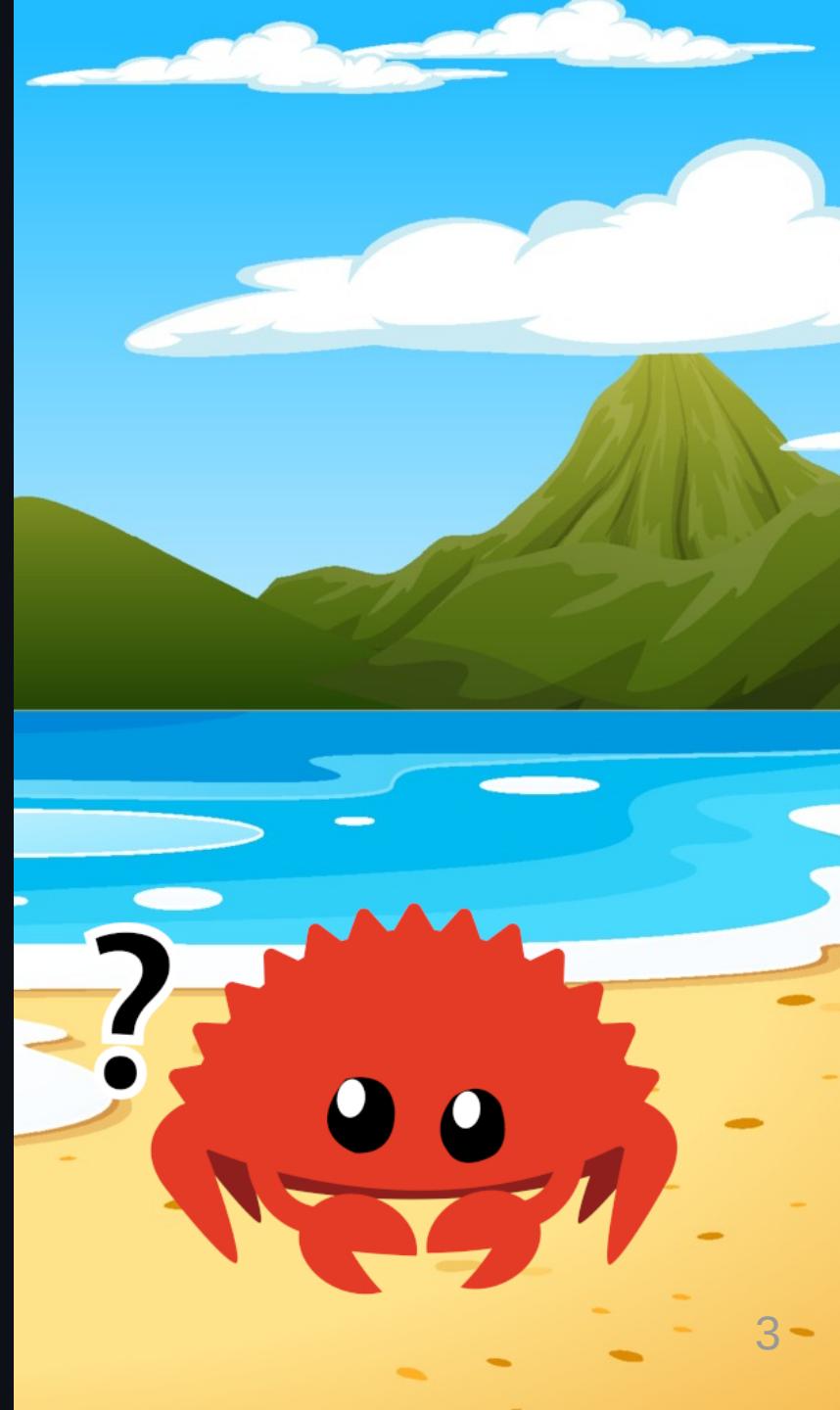
Antes de começarmos

Instale Rust para fazer os exercícios

```
$ curl https://sh.rustup.sh | sh
```

Por que Rust?

- Padrão único de organização estrutural;
- Possui um gerenciador de pacotes oficial;
- Impossibilita* condições de corrida e vazamento de memória;
- É o inimigo Nº 1 do *Segmentation Fault*;
 - Segurança e confiabilidade 🤝



Exemplo: Gerencimanto de Memória Automático

C

```
#include <stdlib.h>
int main() {
    // Alocamos o vetor
    int *vec = (int*) malloc(
        50 * sizeof(int))
    );

    // Usamos o vetor...
    usa_vetor(vec);

    // Liberamos a memória
    free(vec);
}
```

Rust

```
fn main() {
    // Alocamos o vetor
    let vec: Vec<i32> = Vec::new();

    // Usamos o vetor...
    usa_vetor(&vec);

    // A memória é liberada
    // automaticamente
}
```

Índice - O que vamos aprender

1. A Sintaxe de Rust;
 - Comparando com C e Python;
2. Sistema de posse e empréstimo
(ownership & borrowing system);
3. Estruturas e traços
(structs & traits);
4. Implementação "cobertor"
(blanket trait implementation);



1. Um Resumo da Sintaxe

- Similar ao C;
- Parênteses são opcionais e desencorajados;
- `for` genérico ao invés de numérico;
- `return` opcional na maioria dos casos;
- Tipagem pós-fixada ao invés de prefixada;
- Macros explícitos com `!`;

```
fn cinco_ou_maior(x: i32) -> i32 {  
    if x > 5 { x } else { 5 }  
}
```

```
fn main() {  
    for i in 0..10 {  
        println!(  
            "Valor: {}",  
            cinco_ou_maior(i)  
        );  
    }  
}
```

1.1. Declaração de variáveis

- Declaradas com `let`;
- Apesar do nome, não são sempre "variáveis";
 - Por padrão, são **imutáveis**;
- Opcionalmente **mutáveis** com `mut`;
- Podem ser "redefinidas", criando uma nova variável com o mesmo identificador;
 - Dizemos que a variável foi "sombreada" (*shadowed*);
- Tipos podem ser omitidos se *inferíveis*;

Inválido —

```
let x = 10;  
x = 20; // Erro!  
x += 1; // Erro!
```

Válido —

```
let mut x = 10;  
x = 20;  
x += 1;
```

```
let x = 10;  
let x = 20;  
let x = x + 1;
```

2. Posse vs. Empréstimo

- Um dos aspectos mais complicados para iniciantes na linguagem;
- É a "magia" por trás da segurança de Rust;

```
let x = vec![1, 2, 3]; // Dono do dado  
let y = x; // Passagem de posse
```

```
let a = &x[0]; // Erro! `x` não é mais dona do dado!
```

```
let x = vec![1, 2, 3];  
let y = &x; // Empréstimo
```

```
let a = &x[0]; // OK
```

2.1. A Importância do Gerenciamento de Memória Automático

Você consegue dizer qual linha causará um erro?

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *a = (int*)malloc(sizeof(int) * 10);
    int *b = a;

    free(a);

    printf("%d\n", a[5]);
    printf("%d\n", b[9]);

    free(b);
}
```

**Responda
aqui**



3. Estruturas e Traços

- Estruturas nos permitem agrupar e armazenar dados de maneira arbitrária;

```
struct Cpf([u8; 11]);  
  
struct Pessoa {  
    nome: String,  
    cpf: Cpf,  
}
```