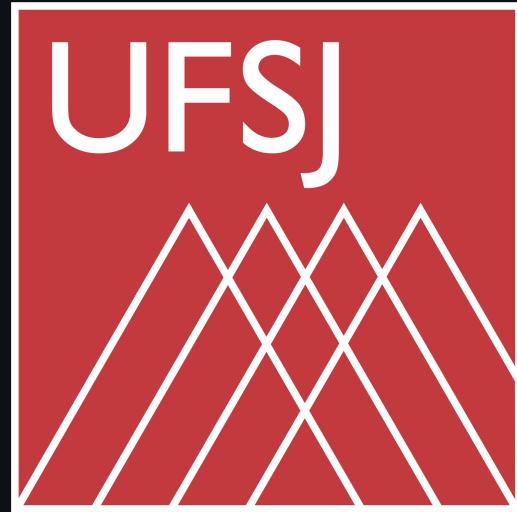


A linguagem Rust e abstrações de alto nível



SECOMP 2023

Brenno Lemos

-  Syndelis
-  @brenno@fosstodon.org



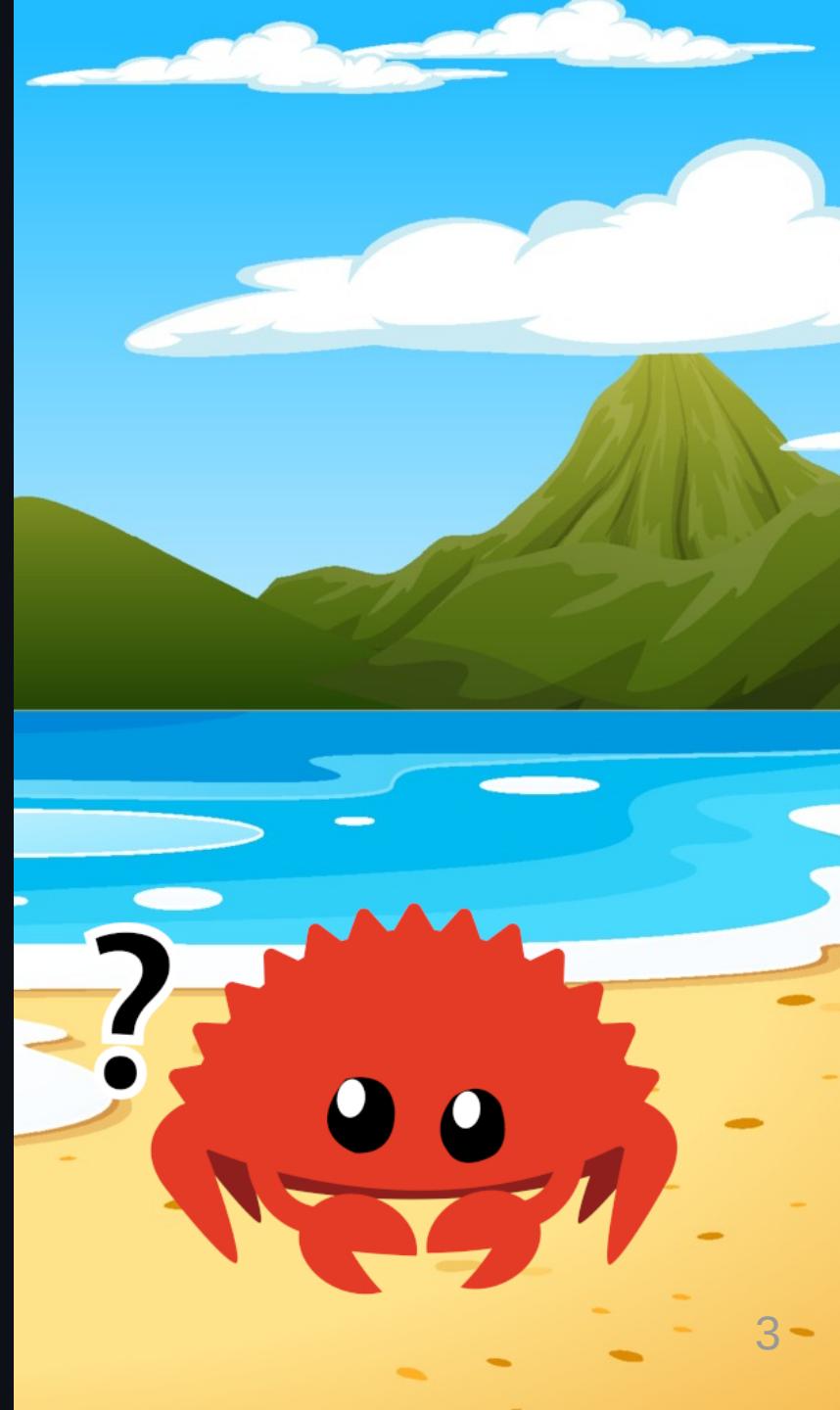
Antes de começarmos

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;
```

Exercício 1: Caixa eletrônico

Dado um valor inteiro X que o usuário deseja sacar, imprima no terminal a quantidade de notas de cada valor para que o saque seja realizado. Considere os valores de notas do Brasil: R\$ 200, R\$ 100, R\$ 50, R\$ 20, R\$ 10, R\$ 5 e R\$ 2.

Lendo valores do terminal

```
fn main() {
    let mut ent = String::new();
    std::io::stdin().read_line(&mut ent);

    let x: i32 = ent.trim().parse().unwrap();

    println!("Você digitou {}", x);
}
```

Compile e Execute

```
$ rustc caixa.rs
$ ./caixa
```

Apêndice 1.1: Sobre leitura de dados do terminal

Por que tantos comandos foram usados para ler um inteiro do terminal?

```
// Rust
fn main() {
    let mut ent = String::new();
    std::io::stdin().read_line(&mut ent);

    let x: i32 = ent.trim().parse().unwrap();

    println!("Você digitou {}", x);
}
```

```
// C
#include <stdio.h>
int main() {
    int x;
    scanf("%d", &x);

    printf("Você digitou %d\n", x);
}
```

Apêndice 1.1: Simples, Segurança!

```
// C
#include <stdio.h>
int main() {
    int x;

    // Em caso de erro, `scanf` retorna `1` e coloca `0` no valor
    // da variável
    scanf("%d", &x);

    printf("Você digitou %d\n", x);
}
```

- gcc scanf-test.c -o scanf-test
 - ./scanf-test
- asd
Você digitou 0

Apêndice 1.1.1: Quando estiver desenvolvendo em C, leia o manual!

```
$ man scanf
```

SYNOPSIS

```
#include <stdio.h>
int scanf(const char *restrict format, ...);
```

RETURN VALUE

On success, these functions return the number of input items successfully matched and assigned; this can be fewer than provided for, **or even zero, in the event of an early matching failure.**

Apêndice 1.2: Macros explícitos? Por quê?

Neste ponto do curso você deve estar se perguntando por que para imprimir no terminal usamos uma "função" que tem um ! no nome.

Diferentemente de printf do C, println! é um macro, e em Rust, macros (macro-funções, mais especificamente) são pós-fixados de ! .

Para entender o porquê, vejamos esse exemplo de código em C e sua saída.

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

#define max(a, b) (a) > (b) ? (a) : (b)

int main() {
    for (int i = 0; i < 10; i++)
        printf("%d\n", max(rand()%10, 5));
}
```

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. Comparativo com C: Por que Rust é chato sobre posse e empréstimo?

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;

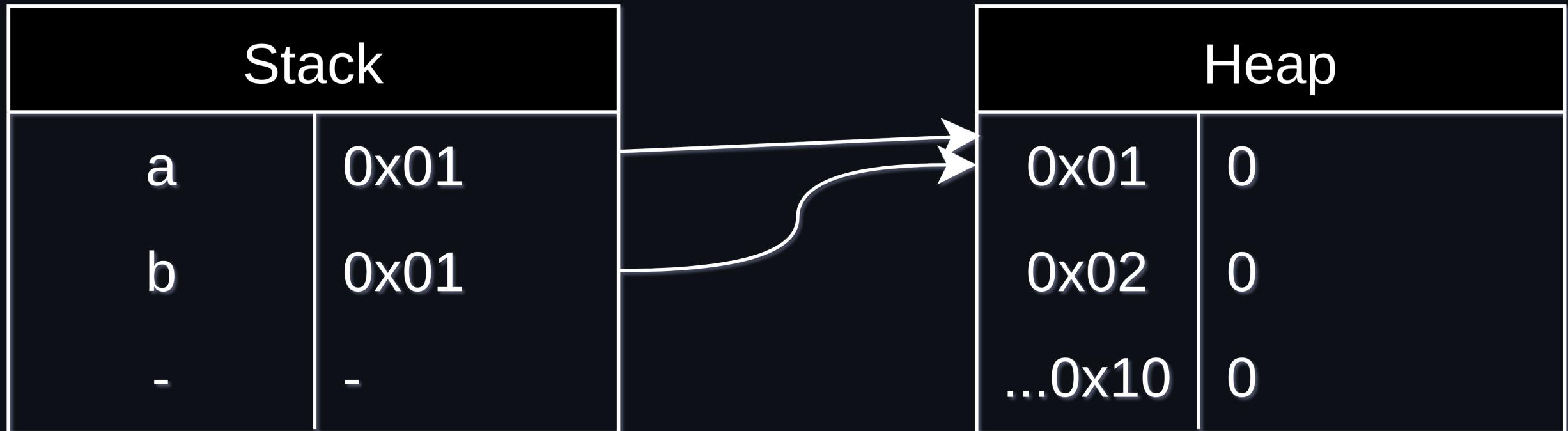
    free(a);

    printf("%d\n", a[5]);
    b[9] = 10;
    printf("%d\n", b[9]);

    free(b);
}
```

**Responda
aqui**





Exercício 2: Caixa eletrônico com notas faltantes

Baseando-se no exercício 1, altere o código do seu caixa eletrônico e remova as notas de R\$ 100 e R\$ 10 reais.

Sempre que o programa começar, avise ao usuário quais são as notas disponíveis.

Use funções para listar as notas disponíveis e para calcular as notas usadas no saque.

2.2. Empréstimo Único vs. Empréstimo Compartilhado

- Temos duas formas de emprestar valores em Rust;
 - `&` são referências imutáveis (ou compartilhadas);
 - `&mut` são referências mutáveis (ou únicas);
- É permitido que existam, ao mesmo tempo, infinitas referências compartilhadas ou uma referência única;
- Por quê? Para evitar condições de corrida em ambientes parallelizados;

3. Estruturas e Implementações

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

```
struct Cpf([u8; 11]);
```

```
struct Pessoa {  
    nome: String,  
    cpf: Cpf,  
}
```

- Dizemos que `Cpf` é uma "estrutura-tupla" (*tuple-struct*);
 - `Cpf` possui um array de 11 elementos inteiros sem sinal de 8 bits;

3.1. Implementações

Implementações nos permitem associar código a determinadas estruturas. Você pode pensar em implementações como paralelos a métodos em linguagens Orientadas a Objetos; a diferença é que estrutura e código são definidos em blocos diferentes.

```
struct Pessoa {  
    nome: String,  
    sobrenome: String,  
}
```

```
impl Pessoa {  
    fn nome_completo(  
        &self  
    ) -> String {  
        format!("{} {}",  
            self.nome,  
            self.sobrenome  
        )  
    }  
}
```