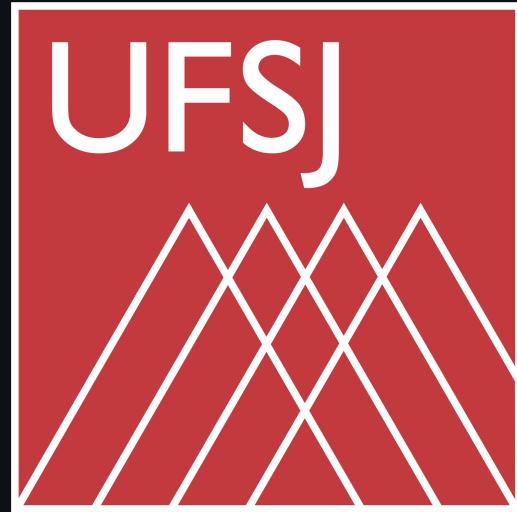


# A linguagem Rust e abstrações de alto nível



SECOMP 2023

Brenno Lemos

-  Syndelis
-  @brenno@fosstodon.org



# Antes de começarmos

**Instale Rust para fazer os exercícios**

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

# Aprenda Rust!

[secomp2023.brenno.codes](https://secomp2023.brenno.codes)



"O Livro"

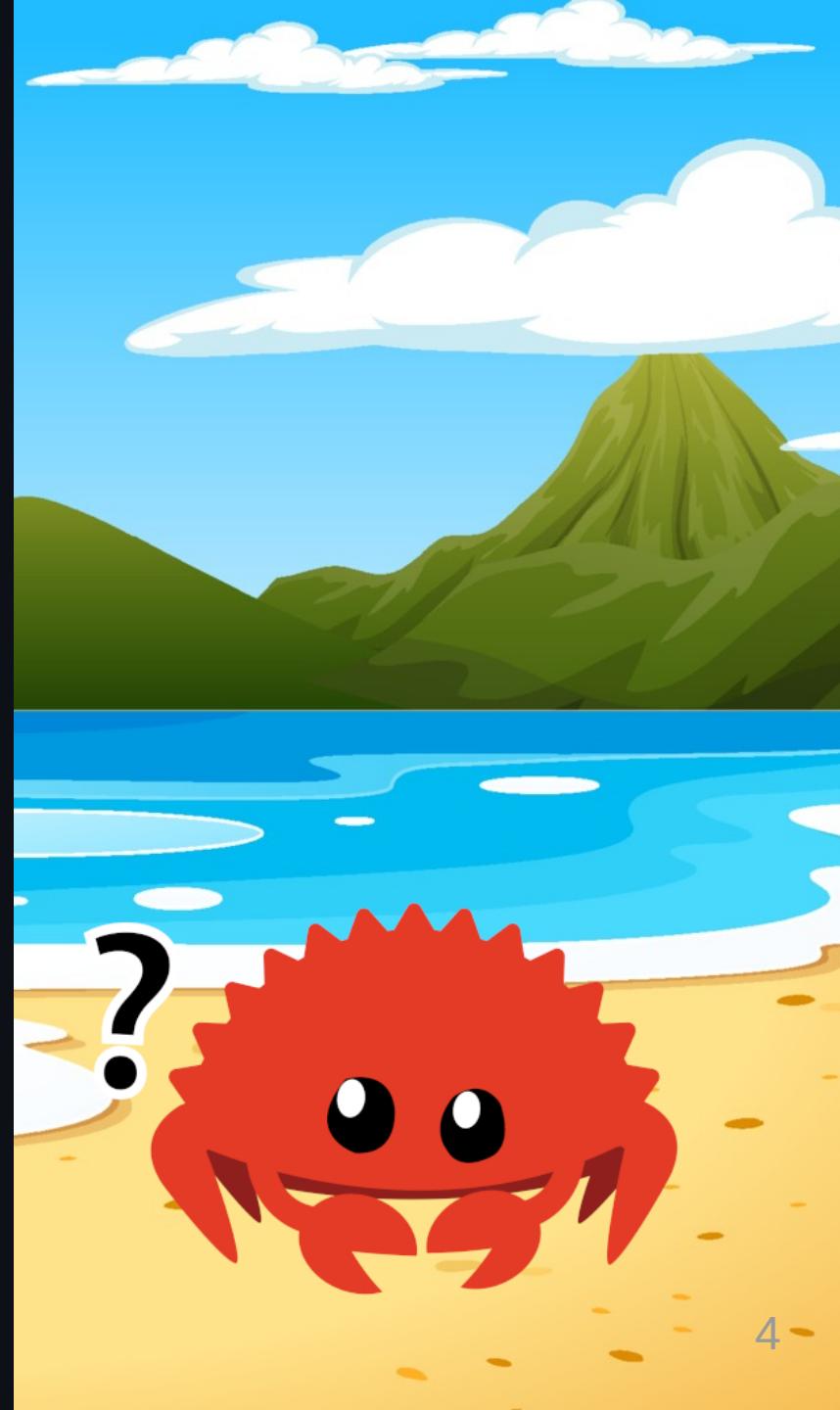


Rustlings



# Por que Rust?

- Padrão único de organização estrutural;
  - Manutenibilidade;
- 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
}
```

# O que fazer com Rust?

- CLIs;
  - Bibliotecas: *Clap*;
- Sistemas Operacionais;
- Backend e Frontend Web;
  - Bibliotecas: *Axum*, *Yew*, *Leptos*;
- Jogos;
  - Bibliotecas: *Bevy*;
- *Data Science*;
  - Bibliotecas: *Polars*, *ndarray*;
- Shaders;
  - Bibliotecas: *rust-gpu*;
- Sistemas embarcados;
- Muitas outras aplicações...

# 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, enumeradores e implementações (*structs, enums & impl*);
4. Traços (*traits*);
5. Monomorfismo e Polimorfismo;



# 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 cédulas de cada valor para que o saque seja realizado. Considere todas as cédulas disponíveis no 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;

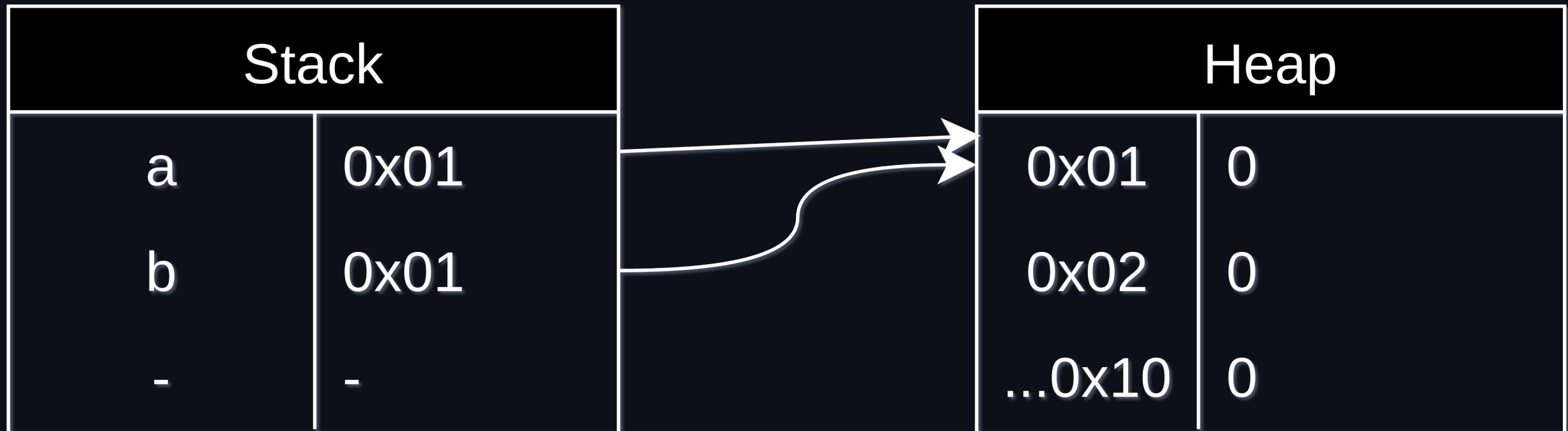
    free(a);

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

    free(b);
}
```

### Dica

Compile com  
-fsanitize=address  
para evitar surpresas.



## Exercício 2: Caixa eletrônico com cédulas em falta

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

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

Use funções para listar as cédulas disponíveis e para calcular as cédulas entregues 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  
        )  
    }  
}
```

## Exercício 3: Cadastro de Usuário

Crie uma struct que represente um usuário com nome e e-mail e implemente os seguintes métodos:

- Um método para imprimir no terminal o usuário no formato 'nome <e-mail>';
- Um método para registrar um novo usuário a partir de leitura do terminal;

## 3.2. Enumeradores

Em Rust, o `enum` é o que chamamos de união discriminada (*tagged union*). Com ele, é possível definir não somente um nome para um valor constante, mas também incluir valores nas variantes do enumerador.

```
enum FormaGeometrica {  
    Circulo { raio: f32 },  
    Quadrado { lado: f32 },  
    Retangulo { altura: f32, largura: f32 },  
}
```

## 3.2. Acessando enumeradores

```
fn main() {  
    let mut entrada = String::new();  
    let stdin = std::io::stdin();  
  
    stdin.read_line(&mut entrada);  
  
    let x = entrada.trim().parse::<i32>();  
  
    match x {  
        Ok(x) => println!("x é {}", x),  
        Err(e) => println!("Erro: {}", e),  
    }  
}
```

Antes de acessarmos o valor de um enum, é necessário discriminar a variante.

Podemos fazer isso de várias maneiras, sendo a mais comum com o comando `match`.

## *Mini Exercício 4: Calculando a área das figuras geométricas*

Escreva uma implementação para o `enum FormaGeometrica` que imprima a área da forma no terminal.

# Apéndice 3.1: *Tagged Unions* em C e C++

C

```
#include <stdio.h>

typedef struct {
    enum { RETANGULO, QUADRADO, CIRCULO } tipo;
    union {
        struct { float altura, largura; } retangulo;
        struct { float lado; } quadrado;
        struct { float raio; } circulo;
    };
} FiguraGeometrica;

int main() {
    FiguraGeometrica fig = {
        .tipo = QUADRADO,
        .quadrado = { .lado = 2.0 }
    };

    printf("%.2f\n", fig.quadrado.lado);
}
```

C++17

```
#include <iostream>
#include <variant>

struct Retangulo { int largura, altura; };
struct Quadrado { int lado; };
struct Circulo { int raio; };

using FiguraGeometrica = std::variant<
    Retangulo, Quadrado, Circulo
>;

int main() {
    auto fig = FiguraGeometrica {
        Retangulo { .largura = 10, .altura = 20 }
    };

    std::cout
        << "largura: "
        << std::get<Retangulo>(fig).largura
        << std::endl;
}
```

## 4. Traços

Como vimos anteriormente, Rust não é uma linguagem Orientada a Objetos. Contudo, ela oferece um recurso familiar aos programadores **OO** para a reutilização de código (*dentro de inúmeras outras funções*): os traços.

Traços descrevem uma série de métodos que devem ser implementados por uma **struct** ou **enum**.

## 4.1. O esqueleto de um traço

```
trait Animal {  
    fn ameacar(&self);  
}  
  
struct Cachorro;  
  
struct Gato;  
  
impl Animal for Cachorro {  
    fn ameacar(&self) {  
        println!("Grrr");  
    }  
}  
  
impl Animal for Gato {  
    fn ameacar(&self) {  
        println!("Hiss");  
    }  
}
```

```
fn main() {  
    let c = Cachorro;  
    let g = Gato;  
  
    c.ameacar();  
    g.ameacar();  
}
```

## 4.2. Implementações padrão

Traços também podem provir implementações padrão para os métodos especificados, de tal forma que não seja necessário re-implementá-los para todas as estruturas que quiserem implementá-los.

```
trait Animal {  
    fn ameacar(&self);  
    fn ameacar_e_atacar(&self) {  
        self.ameacar();  
        println!("Slash!");  
    }  
}
```

Neste exemplo, tanto as estruturas `Cachorro` e `Gato` terão o método `.ameacar_e_atacar` auto-definido.

## 4.3. Macros `derive`

Vimos previamente que funções pós-fixadas com `!` são funções-macro. Em Rust, 3 tipos de macro existem, no total, sendo um dos mais importantes o `derive`.

```
fn main() {
    let c = Carro {
        modelo: "Fusca",
        numero_portas: 2
    };

    dbg!(c); // Imprime `Carro { ... }`
}

#derive[Debug]
struct Carro {
    modelo: String,
    numero_portas: i32,
}
```

Estes macros comumente são utilizados para prover funcionalidades trivialmente implementáveis. Exemplos:

- `Debug`: Possibilita impressão dos dados da estrutura;
- `Eq`: Possibilita comparação de igualdade entre estruturas;
- `Hash`: Possibilita que a estrutura seja usada como chave de `HashMap`;

## *Mini Exercício 5: Printando nossa estrutura com Display e Debug*

Faça uma estrutura que represente um aluno, com pelo menos 3 campos de tipos diferentes. Utilize o `derive` para implementar `Debug` na estrutura e realizar a impressão de depuração.

Após isso, implemente `Display` para definir como um aluno deve ser apresentado no SIGAA. Siga este template para a impressão:

```
Olá, {aluno.nome}. Sua matrícula é {aluno.matricula}.
```

## 5. Escrevendo código reutilizável

Como vimos anteriormente, uma maneira fácil de escrever código reutilizável é agrupar "tipos" que aceitam operações em comum num enumerador, como no caso de **FiguraGeometrica**.

Contudo, existem certas ocasiões nas quais é preferível a utilização de estruturas e traços para agrupar operações em comum.

Neste capítulo, veremos como podemos escrever funções, estruturas e traços que aceitem múltiplos tipos diferentes baseados no comportamento dos tipos aceitáveis.

## 5.1. Tipos Genéricos | Monomorfismo

Rust oferece tipos genéricos de forma similar a C++ ou Java. Com estes tipos, é possível escrever funções que atuam em múltiplos tipos de dados diferentes, contanto que estes tipos de dados possuam alguma funcionalidade em comum.

```
use std::fmt::Display;

fn imprime_array<T: Display>(arr: &[T]) {
    for (i, t) in arr.iter().enumerate() {
        println!("{}: {}", i, t)
    }
}

fn main() {
    let x = [10, 20, 30, 40];
    imprime_array(&x);
    let y = ["Olá", "Mundo"];
    imprime_array(&y);
}
```

## 5.1.1. Tipos Genéricos em estruturas

```
struct Cliente<M: MeioDeContato> {  
    nome: String,  
    contato: M  
}  
  
trait MeioDeContato {  
    fn envia_mensagem(  
        &self, mensagem: String  
    );  
}  
  
struct Email(String);  
struct Celular {  
    ddd: [2; u8],  
    numero: [9; u8]  
}
```

```
impl MeioDeContato for Email {  
    fn envia_mensagem(  
        &self, mensagem: String  
    ) {  
        envia_email(&self.0, mensagem);  
    }  
}  
  
impl MeioDeContato for Celular {  
    fn envia_mensagem(  
        &self, mensagem: String  
    ) {  
        envia_sms(  
            &self.ddd,  
            &self.numero,  
            &mensagem[..=256]  
        );  
    }  
}
```

## 5.1.2. Tipos Genéricos em enumeradores

```
enum Option<T> {  
    Some(T),  
    None,  
}
```

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

## 5.1.3. Tipos Genéricos em traços

Tipos Genéricos podem ser utilizados para implementar traços automaticamente.

```
use std::fmt::Display, iter::IntoIterator;

fn main() {
    [10, 20, 30].imprime();
    ["Olá", "Mundo!"].imprime();
}

trait Imprime {
    fn imprime(self);
}

impl<T: Display, It: IntoIterator<Item = T>> Imprime for It {
    fn imprime(self) {
        for i in self.into_iter() {
            println!("{}"), i
        }
    }
}
```

## 5.3. Polimorfismo (*type erasure*)

É possível **apagar** as informações de um tipo por meio de ponteiros ( `Box` ) ou referências. Quando usados, é possível omitir o tamanho que um tipo ocupa na pilha (*stack*) e, portanto, não permitem que accedemos os campos internos quando utilizados.

Por este motivo, é necessário definir operações que podem ser utilizadas por meio de traços

```
fn main() {
    imprime(&4);
    imprime(&"Olá");
}

fn imprime(obj: &dyn std::fmt::Display) {
    println!("{}", obj);
}
```