

# In Rust we trust



# Idiomatic Rust

- Expressions
- Zero cost abstractions
- Programação funcional
- Projetos
  - Minigrep
  - Gerador de qrcode

# Expressions

Statements	Expressions
Vs	
Como um comando ou declaração	Como uma expressão matemática ou uma pergunta.
performa uma ação sem retorno	Retorna algo ao fim da execução

```
// Em javascript "if" é um statement
// Muitas vezes voce vai precisar de variaveis mutaveis

let temperature = 15;
let feeling; // Variavel mutavel

if (temperature > 20) {
  feeling = "It's hot.";
} else {
  feeling = "It's cold.";
}

// `feeling` foi mudado pelo statement if
console.log(feeling);
```

```
let temperature = 15;

const feeling = temperature > 20 ? "quente" : "Frio";
```

```
// Em rust o if padrão ja é uma expressão
// Não há necessidade de uma sintaxe especial

let temperature = 15;

let feeling = if temperature > 20 {"quente"} else {"frio"};
```

```
// Esta função inteira é uma única expressão `match`.
// O valor que a expressão `match` avalia é retornado diretamente.
fn get_status_category(code: u16) -> &'static str {
  match code {
    // Usamos "ranges" para verificar se um número está em um intervalo.
    200..=299 => "Success",
    300..=399 => "Redirection",
    400..=499 => "Client Error",
    500..=599 => "Server Error",
    // 0 `_` é um padrão "catch-all" para qualquer outro valor.
    _ => "Unknown Category",
  }
}

fn main() {
  let status = get_status_category(404);
  // A variável `status` agora contém "Client Error".
  println!("A categoria do status é: {}", status);

  let other_status = get_status_category(200);
  // A variável `other_status` agora contém "Success".
  println!("A categoria do outro status é: {}", other_status);
}
```

Em rust quase tudo são expressions

- Variaveis e literais
- operadores
- Chamadas de funções e métodos
- Condicionais
- Pattern matching
- Blocos de codigo {}
- Loops
- Outras coisas


E as poucas coisas que não são expressions

- Declarações com let, sturct, trait...
- Expressões declaradas terminadas em ;

# Zero Cost Abstractions

Você não paga pelo que você não usa

e o que você usa é tão eficiente quanto se fosse escrito à mão




```

// Uma função para encontrar o maior i32 em uma slice.
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];
    for &item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

// Uma função quase idêntica para encontrar o maior f64.
fn largest_f64(list: &[f64]) -> f64 {
    let mut largest = list[0];
    for &item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

```




```

// Uma única função genérica que funciona para múltiplos tipos.
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    // O corpo da função é o mesmo de antes.
    let mut largest = list[0];

    for &item in list {
        // O compilador só permite usar `>` porque especificamos `PartialOrd`.
        if item > largest {
            largest = item;
        }
    }
    largest
}

```


# Monomorfização



```
fn largest<T: ...>(...) { /* ... */ }

fn main() {
    let numbers = vec![10, 20, 5];
    largest(&numbers); // <-- Chamada com `i32`

    let floats = vec![1.1, 3.3, 2.2];
    largest(&floats); // <-- Chamada com `f64`
}
```



```
// O compilador escreve uma versão específica para cada tipo que usamos.
fn largest_i32(list: &[i32]) -> i32 { /* ... */ }
fn largest_f64(list: &[f64]) -> f64 { /* ... */ }

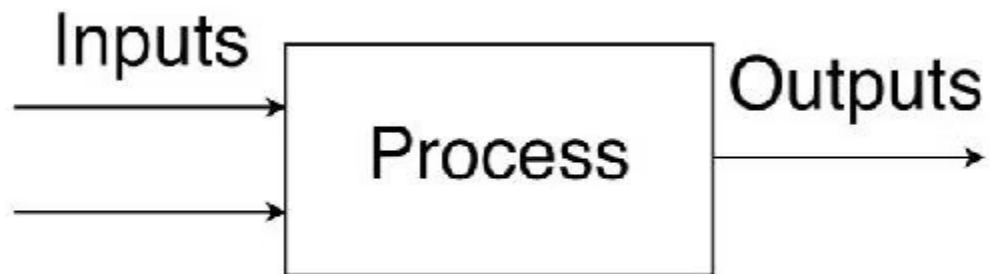
fn main() {
    let numbers = vec![10, 20, 5];
    largest_i32(&numbers); // <-- O compilador substitui a chamada

    let floats = vec![1.1, 3.3, 2.2];
    largest_f64(&floats); // <-- E aqui também
}
```



# Programação Funcional

# O que é uma função?



- Determinística
- Sempre tem um input e output
- Não resulta em efeitos colaterais



# Funções são valores


- Passe como parametro para outras funções (Higher Order Functions)
- Retorne de outras funções
- Atribua a variáveis

```
// Uma função pura
fn greet(name: &str) -> String {
    format!("Hello, {name}")
}

// Uma higher order function (impura por causa do println)
fn greeter(name: &str, greeter: impl Fn(&str) -> String) {
    let greeting = greeter(name);
    println!("{greeting}")
}


fn main() {
    greeter("Diego", greet);
}
```

# Imutabilidade e Transformações



```
// Uma abordagem imperativa que usa mutação
fn find_evens_impure(numbers: &Vec<i32>) -> Vec<i32> {
    let mut even_numbers = Vec::new(); // 1. Cria um novo vetor mutável.

    for &num in numbers {
        if num % 2 == 0 {
            // 2. Efeito colateral: modifica `even_numbers`.
            even_numbers.push(num);
        }
    }
    // 3. Retorna o vetor modificado.
    even_numbers
}
```



```
// Uma abordagem declarativa e funcional
fn find_evens_pure(numbers: &Vec<i32>) -> Vec<i32> {
    // Nós descrevemos a transformação, sem gerenciar o loop ou estado mutável
    numbers
        .iter() // 1. Pega um iterador que "empresta" os itens.
        .copied() // 2. Copia os valores para que a nova coleção os possua.
        .filter(|&num| num % 2 == 0) // 3. Filtra os valores com base em uma função
        .collect() // 4. Coleta os resultados em uma nova coleção.
}
```

# Instalando Rust

# Cargo

- cargo help
- cargo new
- cargo run
- cargo check
- cargo build

## Estrutura do projeto

Cargo.toml - Arquivo de configurações

src/main.rs - Entrypoint para o binário

src/lib.rs - Arquivo principal da biblioteca

src/\*/mod.rs ou src/\*.rs - Um modulo rust

**crates.io e docs.rs**

# Minigrep

# Gerador de qrcode