# Linguagens de Programação
# Trabalho

Professor:
Christiano Braga

Alunos:
Leonardo Brasil Oliveira Cerne
Victor Teles Araujo Speorin
Vitor Lemos Silva

Maio 2025

## Conteúdo

# 1 Introdução

Este relatório tem como objetivo realizar uma avaliação da linguagem de programação **Rust**, utilizando como base os critérios apresentados nos capítulos 1 e 6 da obra de Robert W. Sebesta, *Conceitos de Linguagens de Programação*. A análise será dividida em duas partes: (i) avaliação geral da linguagem com base em critérios como legibilidade, facilidade de escrita, confiabilidade e custo; e (ii) avaliação detalhada dos tipos de dados suportados pela linguagem, incluindo suas representações, operações, mecanismos de verificação de tipos e gerenciamento de memória.

# 2 Avaliação Geral da Linguagem (Capítulo 1)

## 2.1 Legibilidade

Rust apresenta uma sintaxe moderna e estruturada, inspirada em linguagens como C e C++, mas com melhorias significativas. A presença explícita de tipos e a utilização do sistema de *ownership* contribuem para um código menos ambíguo e mais compreensível a longo prazo. No entanto, a sintaxe das macros e o uso intenso de símbolos (`&`, `*`, `!`) podem gerar confusão para iniciantes.

**Conclusão**: A linguagem é legível, especialmente para programadores experientes com linguagens estáticas e sistemas de tipos sofisticados.

## 2.2 Facilidade de Escrita

Apesar de fornecer recursos poderosos para desenvolvimento seguro e eficiente, Rust possui uma curva de aprendizado acentuada. O sistema de *ownership*, *borrowing* e *lifetimes* exige atenção constante do programador. Contudo, ferramentas como o compilador (`rustc`) e o gerenciador de pacotes (`cargo`) oferecem excelente suporte, com mensagens de erro detalhadas e sugestivas.

**Conclusão**: A escrita de programas em Rust requer maior esforço inicial, mas é auxiliada por um ecossistema robusto e ferramentas bem desenvolvidas.

## 2.3 Confiabilidade

A confiabilidade é um dos maiores destaques da linguagem. Rust previne, em tempo de compilação, diversos tipos de erros comuns em linguagens como C/C++, incluindo *data races*, *null pointers* e *use-after-free*. O sistema de tipos forte e a ausência de coleta de lixo garantem desempenho e segurança.

**Conclusão**: Rust é altamente confiável e adequado para desenvolvimento de software crítico e de baixo nível.

### 2.4 Custo

Em termos de custo, Rust oferece alta performance, comparável a linguagens como C e C++, com um modelo de segurança superior. O tempo de compilação pode ser elevado e a complexidade inicial pode aumentar o custo de treinamento, mas o custo de manutenção tende a ser reduzido devido à robustez da linguagem.

**Conclusão**: Embora o custo de adoção seja inicialmente maior, os benefícios em confiabilidade e manutenção compensam em projetos de médio e longo prazo.

## 3 Avaliação dos Tipos de Dados (Capítulo 6)

### 3.1 Tipos Primitivos

Rust possui um conjunto bem definido de tipos primitivos: inteiros (`i32`, `u32`, etc.), ponto flutuante (`f32`, `f64`), caracteres (`char`) e booleanos (`bool`). Todos os tipos possuem tamanho definido em tempo de compilação e não existem tipos genéricos como `int` ou `float` sem especificação.

### 3.2 Strings

A linguagem possui dois tipos principais para manipulação de cadeias de caracteres:

- `String`: tipo dinâmico, mutável, alocado no heap.
- `&str`: fatia de string, geralmente imutável, usado para literais.

```
let s1 = String::from("hello");
let s2 = &s1; // empréstimo (borrowing)
```

### 3.3 Vetores

Rust fornece o tipo `Vec<T>` para representar vetores dinâmicos:

```
let mut v = vec![1, 2, 3];
v.push(4);
```

### 3.4 Tuplas

As tuplas são estruturas de tamanho e tipos fixos, permitindo agrupar valores heterogêneos:

```
let tup: (i32, f64, u8) = (500, 6.4, 1);
let (x, y, z) = tup;
```

## 3.5 Arrays

Arrays possuem tamanho fixo, conhecido em tempo de compilação:

```
let a = [1, 2, 3, 4];
let a: [i32; 4] = [0; 4];
```

## 3.6 Enums e Pattern Matching

Enums são poderosos e permitem representar variantes de forma segura:

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

## 3.7 Verificação de Tipos

Rust adota um sistema de tipagem estática e forte, o que significa que todas as verificações de tipo são realizadas pelo compilador antes da execução do programa, garantindo que não haja erros decorrentes de operações inválidas em tempo de execução. Esse modelo melhora significativamente a segurança e a previsibilidade do código, evitando problemas como uso indevido de variáveis ou chamadas de funções com argumentos incompatíveis.

Além disso, a linguagem implementa inferência de tipos, permitindo que o compilador deduza automaticamente o tipo de uma variável com base no contexto. Isso reduz a necessidade de declarações explícitas, tornando o código mais conciso e legível. No entanto, há situações em que a inferência pode ser insuficiente, exigindo que o programador especifique os tipos manualmente—especialmente em cenários mais complexos envolvendo genéricos ou múltiplas operações interdependentes.

## 3.8 Gerência de Memória

Rust não possui *garbage collector*. O gerenciamento de memória é feito por meio do sistema de *ownership*:

- Cada valor possui um único dono.
- Quando o dono sai de escopo, o valor é desalocado automaticamente.
- O *borrowing* permite acessos temporários.

```
let s1 = String::from("hello");
let s2 = s1; // s1 é movido, não pode mais ser usado
```

Esse modelo evita vazamentos de memória e condições de corrida sem comprometer o desempenho.

# 4 Conclusão

A linguagem Rust representa uma evolução significativa na construção de software seguro e eficiente. Seu sistema de tipos rigoroso, aliado a um modelo de gerenciamento de memória inovador e sem coleta de lixo, fornece confiabilidade e desempenho superiores às linguagens tradicionais de sistemas. Embora apresente uma curva de aprendizado acentuada, especialmente no que tange aos conceitos de *ownership* e *borrowing*, a linguagem compensa com ferramentas robustas, legibilidade e baixo custo de manutenção. No contexto dos critérios de Sebesta, Rust se destaca principalmente pela confiabilidade e pela robustez de seu sistema de tipos.