



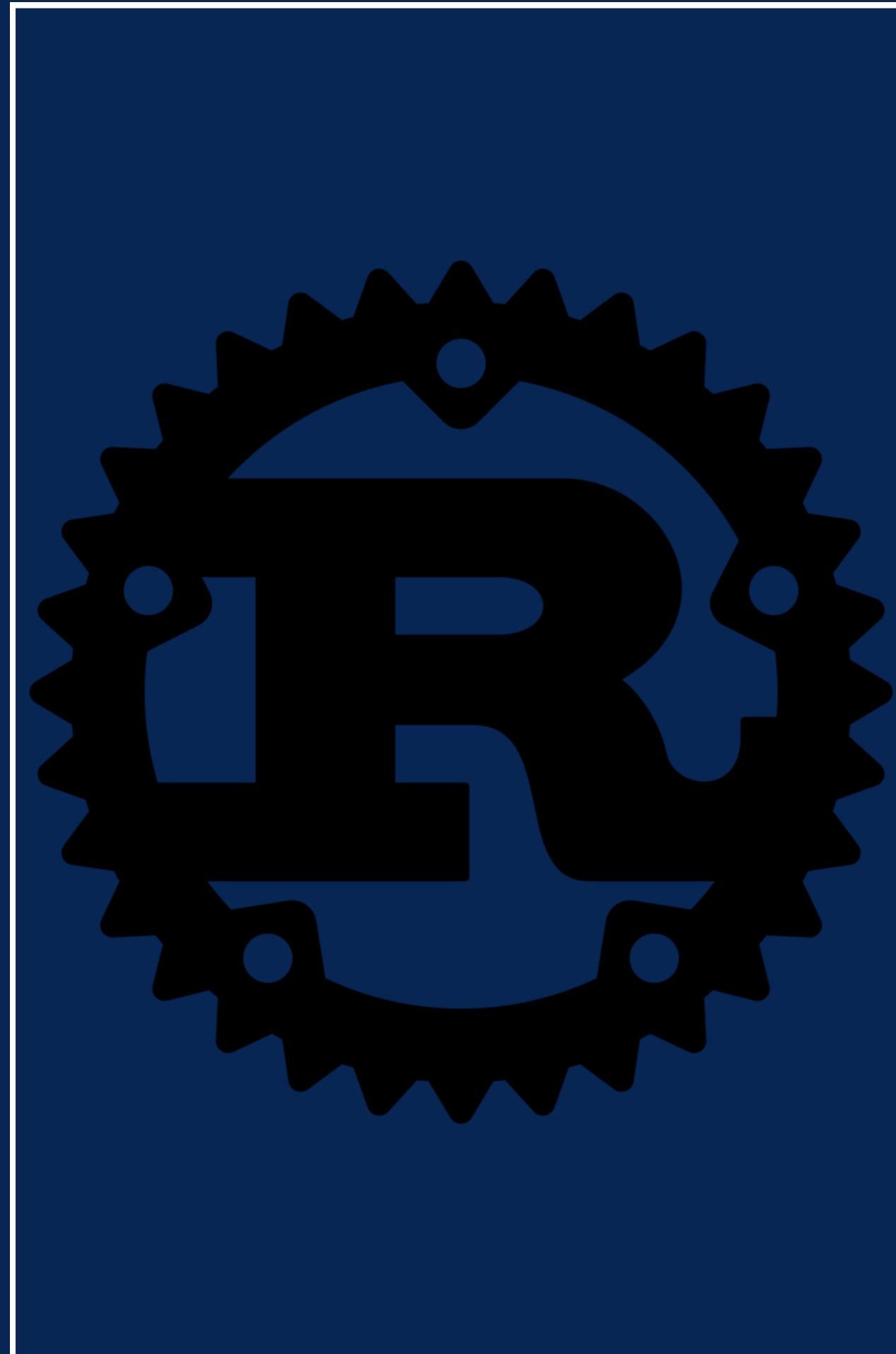
A person wearing a blue shirt and dark pants is working on a bicycle wheel in a workshop. They are holding a tool and appear to be adjusting or assembling the spokes. The background shows shelves with various tools and equipment. The overall lighting is warm and focused on the work area.

LINGUAGENS DE  
PROGRAMAÇÃO

RUST

# HISTÓRIA

Rust nasceu de um projeto paralelo de Graydon Hoare, funcionário da Mozilla, em meados de 2006, buscando uma linguagem mais segura e eficiente para programação de sistemas.



# PARADIGMAS

## Enumerações

Rust possui enumerações de tipagem forte, e suas variantes podem carregar valores diversos.

## Casamento de padrões

Casamento de padrões é muito importante em Rust, pois as enumerações são a base do tratamento de erros.<sup>[45]</sup> Exemplo de declaração de enumerações e o casamento de padrões

## Programação Segura

Rust prioriza segurança desde sua concepção. Ao contrário de linguagens como C/C++, Rust elimina erros comuns de memória em tempo de compilação através de seu sistema de verificações rigorosas.



# CONSTRUTORES

Rust não possui construtores especiais – usa funções associadas (métodos estáticos) seguindo convenções. A função `new()` é o padrão, mas múltiplos métodos como `from()`, `with_*`() ou builders são comuns. Traits como `Default` e `From/Into` fornecem construção padronizada, enquanto validações ocorrem via `Result` para segurança.

```
// 1. Construtor padrão 'new'  
struct Retangulo { largura: u32, altura: u32 }  
impl Retangulo {  
    fn new(l: u32, a: u32) -> Self { Retangulo { largura: l, altura: a } }  
}  
  
// 2. Múltiplos construtores  
impl Retangulo {  
    fn quadrado(lado: u32) -> Self { Retangulo { largura: lado, altura: lado } }  
}  
  
// 3. Builder pattern (simplificado)  
#[derive(Default)]  
struct Conexao { url: String, timeout: u32 }  
struct ConexaoBuilder(Conexao);  
impl ConexaoBuilder {  
    fn new() -> Self { ConexaoBuilder(Conexao::default()) }  
    fn url(mut self, u: &str) -> Self { self.0.url = u.to_string(); self }  
    fn build(self) -> Conexao { self.0 }  
}  
  
// 4. Default trait  
#[derive(Default)]  
struct Config { host: String, port: u16 }  
impl Default for Config {  
    fn default() -> Self { Config { host: "localhost".into(), port: 8080 } }  
}  
  
// 5. From/Into traits  
struct Celsius(f64);  
struct Fahrenheit(f64);  
impl From<Fahrenheit> for Celsius {  
    fn from(f: Fahrenheit) -> Self { Celsius((f.0 - 32.0) * 5.0/9.0) }  
}
```



## LEGIBILIDADE

Em resumo, a legibilidade do Rust é alta para desenvolvedores que já entendem seus princípios fundamentais. Para um programador experiente, o código Rust é legível e expressivo, pois as regras de segurança são explicitamente visíveis no código. No entanto, para um iniciante, a curva de aprendizado íngreme inicial pode fazer com que o código pareça complexo até que os conceitos de ownership sejam dominados.

# CAPACIDADE ESCRITA

A capacidade de escrita da linguagem Rust é considerada desafiadora no início, mas extremamente recompensadora a longo prazo, combinando a expressividade de linguagens de alto nível com o controle e a performance de linguagens de baixo nível como C e C++.

```
use serde::Serialize, Deserialize;

#[derive(Serialize, Deserialize, Debug)]
struct User<'a> {
    name: &'a str,
    email: &'a str
}

impl<'a> User<'a> {
    fn new(name: &'a str, email: &'a str) -> User<'a> {
        User { name, email }
    }
}

fn main() {
    let user :User = User::new( name: "Rust", email: "rust@lang.com");
    let serialized :String = serde_json::to_string_pretty( value: &user).unwrap();

    println!("Serialized User:\n{}", serialized);
}
```

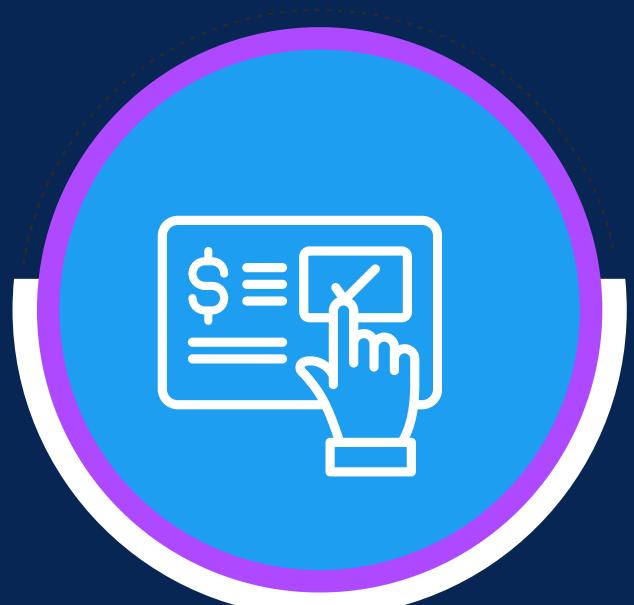


## CONFIABILIDADE

A linguagem de programação Rust é amplamente reconhecida por sua alta confiabilidade, sendo consistentemente classificada como uma das linguagens mais admiradas e seguras pela comunidade de desenvolvedores. Sua confiabilidade deriva de recursos de design exclusivos que previnem classes inteiras de erros comuns de programação em tempo de compilação.



## CUSTO



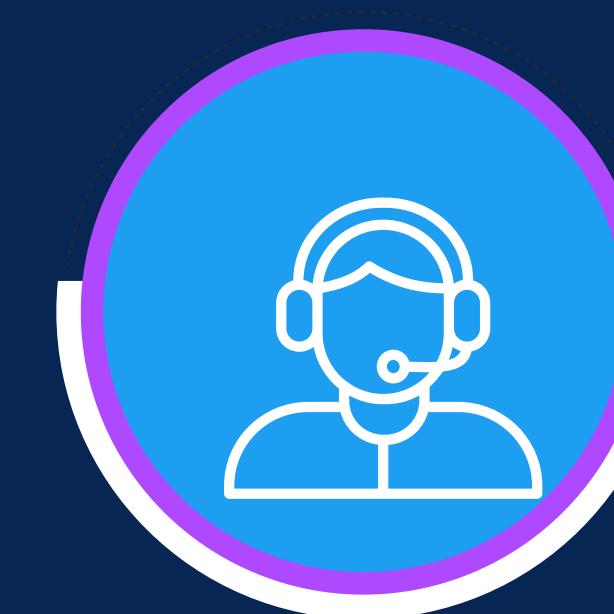
### Treinamento e Curva de Aprendizado

Este é o principal custo. A sintaxe e os conceitos únicos do Rust, como o sistema de ownership (posse de memória), exigem tempo e esforço para serem dominados pelos desenvolvedores, mesmo aqueles experientes em outras linguagens como C++ ou Java.



### Manutenção

O custo de manutenção de software em Rust tende a ser menor a longo prazo, devido à segurança de memória e prevenção de falhas em tempo de compilação, o que resulta em menos bugs em produção. No entanto, os custos iniciais podem ser mais elevados.



### Contratação de Talentos Especializados

Desenvolvedores Rust experientes ainda são um nicho de mercado e, por isso, podem exigir salários mais altos em comparação com programadores de linguagens mais comuns.

## PORTABILIDADE

A portabilidade em Rust é um grande diferencial, permitindo escrever código uma vez e usá-lo em diversas plataformas, de sistemas embarcados (microcontroladores) a servidores, graças à sua natureza compilada para código de máquina nativo e um sistema de tipos robusto que garante segurança de memória sem garbage collector, tornando-o ideal para hardware restrito e sistemas de alto desempenho, além de ter excelente interoperabilidade com C e um ecossistema crescente que facilita o desenvolvimento multi-plataforma.



# PROJETO FINAL

O File Bundler é uma ferramenta de linha de comando (CLI) desenvolvida em Rust que funciona como um "empacotador inteligente" de arquivos.

## Agregação (Empacotamento)

- Entrada: Recebe uma lista de arquivos e/ou pastas
- Processo: Lê todo o conteúdo desses itens sequencialmente
- Saída: Cria um único arquivo contendo tudo

```
fn main() {
    // Parseia os argumentos da linha de comando
    let cli = Cli::parse();

    // Executa o comando apropriado baseado na escolha do usuário
    match &cli.command {
        Commands::Bundle(args) => {
            println!("Iniciando operação 'bundle'...");

            // Chama a função de criação de bundle e trata o resultado
            if let Err(e) = bundle::run_bundle(args) {
                eprintln!("Erro ao criar o bundle: {}", e);
            } else {
                println!("Bundle '{}' criado com sucesso!", args.output.display());
            }
        }
        Commands::Extract(args) => {
            println!("Iniciando operação 'extract'...");

            // Determina o diretório de destino (usa atual se não especificado)
            let output_dir = args.output.as_deref().unwrap_or_else(|| Path::new("."));

            // Chama a função de extração e trata o resultado
            if let Err(e) = extract::run_extract(args, output_dir) {
                eprintln!("Erro ao extrair o bundle: {}", e);
            } else {
                println!("Extração concluída com sucesso!");
            }
        }
    }
}
```