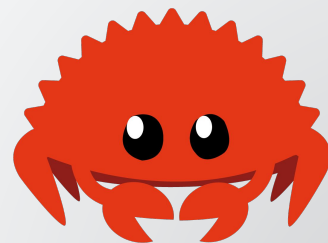


src > main.rs > ...

► Run | Debug

```
1 fn main() {  
2     println!("Hello, world!");  
3 }  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14
```

Linguagem de Programação Rust



Membros do grupo

- Helena Ferreira
- Gabriel Oliveira
- Wanderson Teixeira



PUC Minas

Sumário

- Histórico
- Linha do tempo
- Linguagens relacionadas
- Tipos de Dados
- Características marcantes
 - Compilação
 - Tipagem estática inferida
 - Lifetimes
 - Paradigmas da linguagem
 - Ownership
 - Borrowing
 - Exemplo no playground Rust
 - Traits
- Segurança de memória e alta performance
- Aplicações
- `unsafe{}`
- Tutorial de instalação
- Curiosidades
- Considerações finais
- Referências

HISTÓRICO

Histórico

- Criador: Graydon Hoare
- Desenvolveu a linguagem enquanto trabalhava na Mozilla Research
- Tinha como objetivo oferecer segurança e alto desempenho, evitando erros comuns que ocorriam em outras linguagens



"A linguagem para programadores C++ frustrados"



Graydon Hoare

LINHA DO TEMPO

Graydon Hoare
começa a
trabalhar no
Rust como um
projeto pessoal

2009

Mozilla começa
a financiar o
projeto

Rust é
disponibilizado
para o público

2010

Rust é usado
para construir
uma série de
projetos,
incluindo o
Firefox

2015

É lançada a
primeira versão
estável de Rust

2017

2023

É anunciado
que o kernel do
Windows está
sendo reescrito
em Rust

LINGUAGENS RELACIONADAS

Linguagens relacionadas



Logotipos C, C++, Swift, Haskell e GO respectivamente

TIPOS DE DADOS

Tipos de dados

1. Tipos numéricos:

- **i8, i16, i32, i64, i128**: Inteiros com sinal de diferentes tamanhos.
- **u8, u16, u32, u64, u128**: Inteiros sem sinal de diferentes tamanhos.
- **f32, f64**: Números de ponto flutuante de precisão simples e dupla.

2. Booleanos:

- **bool**: Pode ter os valores true ou false.

3. Caracteres:

- **char**: Representa um único caractere Unicode.

Tipos de dados

4. Tipos de referência:

- **&T**: Referência imutável a um valor do tipo T.
- **&mut T**: Referência mutável a um valor do tipo T.

5. Tipos de propriedade (Ownership):

- **String**: String de texto dinâmica, de propriedade exclusiva.
- **Vec<T>**: Vetor dinâmico de elementos do tipo T, de propriedade exclusiva.

Tipos de dados

6. Tipos compostos:

- **array:** Coleção fixa de elementos do mesmo tipo e tamanho definido em tempo de compilação.
- **tuple:** Coleção de elementos de tipos diferentes e tamanho fixo definido em tempo de compilação.
- **struct:** Tipo de dados personalizado definido pelo usuário que agrupa diferentes campos de dados.
- **enum:** Tipo de dados personalizado que representa um conjunto de variantes possíveis.

Tipos de dados

6. Tipos compostos:

- **enum**: Tipo de dados personalizado que representa um conjunto de variantes possíveis.
 - Em Rust, as enums podem ter campos associados a cada valor, o que significa que cada variante da enumeração pode ter dados associados a ela

Tipos de dados

```
enum FormaGeometrica {
    Retangulo { largura: u32, altura: u32 },
    Circulo(f64),
}

impl FormaGeometrica {
    fn calcular_area(&self) -> f64 {
        match self {
            FormaGeometrica::Retangulo { largura, altura } => f64::from(*largura * *altura),
            FormaGeometrica::Circulo(raio) => std::f64::consts::PI * raio.powi(2),
        }
    }
}

fn main() {
    let retangulo = FormaGeometrica::Retangulo {
        largura: 10,
        altura: 5,
    };
    let circulo = FormaGeometrica::Circulo(3.5);

    println!("Área do retângulo: {:.2}", retangulo.calcular_area());
    println!("Área do círculo: {:.2}", circulo.calcular_area());
}
```

Em Rust não existe null

- Segurança de memória



Em Rust não existe null

- 'Option<T>'

- **Some<T>** / **None<T>**
- Representa valores que podem ou não podem estar presentes
- Forma de tratamento de erro

```
fn find_even(numbers: &[i32]) -> Option<i32> {  
    for &num in numbers {  
        if num % 2 == 0 {  
            return Some(num);  
        }  
    }  
    None  
}  
  
fn main() {  
    let numbers = [1, 3, 5, 7, 2, 9, 4, 6];  
    match find_even(&numbers) {  
        Some(even) => println!("Found an even number: {}", even),  
        None => println!("No even numbers found."),  
    }  
}
```

CARACTERÍSTICAS MARCANTES

Características marcantes

1. Linguagem compilada
2. Abstrações a zero custo
3. Tipagem estática inferida
4. Multiparadigma
5. **Segurança de memória em tempo de compilação**
 - a. Variáveis imutáveis por padrão
 - b. Borrowing - Ownership
6. Não usa coletor de lixo
7. Alta performance

Vamos ver isso melhor nos próximos slides...

Variáveis imutáveis por padrão

```
let x = 5;    // Variável imutável  
x = 10;       // Erro! Tentando modificar uma variável imutável
```

```
let mut y = 5; // Variável mutável  
y = 10;        // OK! Modificando uma variável mutável
```

- Promove a segurança e a prevenção de erros

Linguagem Compilada

- Compilação:
 - Otimização do código
 - Compilação incremental
- Flexibilidade:
 - Pode ser compilada para executáveis, bibliotecas compartilhadas (shared libraries), bibliotecas estáticas (static libraries) e até mesmo WebAssembly (Wasm)

Linguagem Compilada

- Abstrações a zero custo
 - Rust permite expressar abstrações de alto nível sem incorrer em custos adicionais de tempo de execução ou consumo de memória

```
1 ▾ fn sum(numbers: &[i32]) -> i32 {  
2     let mut total = 0;  
3 ▾     for &num in numbers {  
4         total += num;  
5     }  
6     total  
7 }  
8  
9 ▾ fn main() {  
10     let numbers = vec![1, 2, 3, 4, 5];  
11     let result = sum(&numbers);  
12     println!("Sum: {}", result);  
13 }  
14  
15 // Saída:  
16 // Sum 15
```

Linguagem Compilada

- Compilador conhecido por mensagens informativas e úteis

```
1 fn main() {  
2     let x = 5;  
3     let y = "hello";  
4  
5     let z = x + y;  
6  
7     println!("Result: {}", z);  
8 }
```

```
Compiling playground v0.0.1 (/playground)  
error[E0277]: cannot add `&str` to `{integer}`  
--> src/main.rs:5:15  
5 |         let z = x + y;  
   |                   ^ no implementation for `{integer} + &str`  
= help: the trait `Add<&str>` is not implemented for `{integer}`  
= help: the following other types implement trait `Add<Rhs>`:  
    <&'a f32 as Add<f32>>  
    <&'a f64 as Add<f64>>  
    <&'a i128 as Add<i128>>  
    <&'a i16 as Add<i16>>  
    <&'a i32 as Add<i32>>  
    <&'a i64 as Add<i64>>  
    <&'a i8 as Add<i8>>  
    <&'a isize as Add<isize>>  
    and 48 others
```

For more information about this error, try `rustc --explain E0277`.

Linguagem Compilada

- Comparação com C - compilação

```
9  #include <stdio.h>
10
11 int main()
12 {
13     int x = 5;
14     char *y = "hello";
15
16     int z = x + y;
17
18     printf("Result: %i", z);
19
20     return 0;
21 }
```

```
main.c: In function 'main':
main.c:16:13: warning: initialization of 'int' from
'char *' makes integer from pointer without a cast
[-Wint-conversion]
   16 |         int z = x + y;
       |                ^
Result: 2088501257

...Program finished with exit code 0
Press ENTER to exit console.
```


Linguagem Compilada

- Comparação com C - compilação

```
main.c: In function 'main':
main.c:16:13: warning: initialization
  of 'int' from 'char *' makes integer
  from pointer without a cast [-Wint-c
onversion]
   16 |         int z = x + y;
       |         ^
Result: -917196791

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Compiling playground v0.0.1 (/playground)
error[E0277]: cannot add `&str` to `{integer}`
  --> src/main.rs:5:15
   |
5  |         let z = x + y;
   |                   ^ no implementation for `{integer} + &str`
   |
= help: the trait `Add<&str>` is not implemented for `{integer}`
= help: the following other types implement trait `Add<Rhs>`:
   <&'a f32 as Add<f32>>
   <&'a f64 as Add<f64>>
   <&'a i128 as Add<i128>>
   <&'a i16 as Add<i16>>
   <&'a i32 as Add<i32>>
   <&'a i64 as Add<i64>>
   <&'a i8 as Add<i8>>
   <&'a isize as Add<isize>>
   and 48 others
```

For more information about this error, try `rustc --explain E0277`.

Tipagem estática inferida

- O tipo de cada variável deve ser conhecido no momento da compilação

```
let y: i32 = 10;
```

- O compilador pode inferir o tipo de uma variável com base em seu valor

```
let y = 10;
```

Tipagem estática inferida

- É possível redeclarar variáveis em um mesmo escopo

```
fn main() {  
    let age = 25;  
    println!("Age: {}", age); // Age: 25  
  
    let age = "Twenty-five";  
    println!("Age: {}", age); // Age: Twenty-five  
  
    let age = age.chars().count();  
    println!("Age: {}", age); // Age: 11  
}
```

Lifetimes

- Tempo de vida de uma referência em relação aos dados que ela referencia
- Usado para evitar problemas de acesso a memória inválida

```
1 ▾ struct Point<'a> {  
2     x: &'a i32,  
3     y: &'a i32,  
4 }  
5  
6 ▾ fn main() {  
7     let x = 5;  
8     let y = 10;  
9  
10    let point = Point { x: &x, y: &y };  
11  
12    println!("x: {}, y: {}", point.x, point.y);  
13 }
```

PARADIGMAS DA LINGUAGEM

Programação Concorrente e Paralela

Melhor utilização de recursos
de hardware e aumento de
desempenho

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        println!("Olá, eu sou uma thread!");
    });

    handle.join().unwrap();
}
```

Programação Funcional

Suporte a recursão e torna o código mais legível e fácil de testar

```
fn quadrado(n: i32) -> i32 {  
    n * n  
}  
  
fn main() {  
    let numeros = vec![1, 2, 3, 4, 5];  
    let quadrados: Vec<_> = numeros.iter()  
                                     .map(|&n| quadrado(n))  
                                     .collect();  
  
    for quadrado in quadrados {  
        println!("{}", quadrado);  
    }  
}
```

Programação Procedural

Organizar funcionalidades de um programa em trechos de código reutilizáveis chamados procedimentos.

```
fn calcular_soma(a: i32, b: i32) -> i32 {  
    a + b  
}  
  
fn main() {  
    let resultado = calcular_soma(3, 5);  
    println!("O resultado da soma é: {}", resultado);  
}
```


Programação Genérica

Redução de duplicação de código e aumento da flexibilidade

```
fn imprimir_valor<T>(valor: T) {  
    println!("O valor é: {}", valor);  
}  
  
fn main() {  
    imprimir_valor(10);  
    imprimir_valor("Olá, mundo!");  
}
```

Programação Orientada a Objetos

Embora Rust não seja uma linguagem orientada a objetos tradicional, ela oferece recursos semelhantes:

structs, impl → **objetos**
traits → **polimorfismo**

```
struct Pessoa {  
    nome: String,  
    idade: u32,  
}  
  
impl Pessoa {  
    fn new(nome: String, idade: u32) -> Self {  
        Pessoa { nome, idade }  
    }  
  
    fn saudacao(&self) {  
        println!("Olá, meu nome é {} e eu tenho {} anos.", self.nome, self.idade);  
    }  
}  
  
fn main() {  
    let pessoa = Pessoa::new(String::from("João"), 25);  
    pessoa.saudacao();  
}
```

Structs podem ter métodos

- Os métodos em Rust são funções definidas dentro do contexto de uma struct específica e podem ser chamados em instâncias dessa struct.

```
1 struct Retangulo {  
2     largura: u32,  
3     altura: u32,  
4 }  
5  
6 impl Retangulo {  
7     // Método para calcular a área do retângulo  
8     fn area(&self) -> u32 {  
9         self.largura * self.altura  
10    }  
11 }  
12  
13 fn main() {  
14     let retangulo = Retangulo {  
15         largura: 5,  
16         altura: 10,  
17     };  
18  
19     // Chamando o método area() na instância do Retangulo  
20     println!("Área do retângulo: {}", retangulo.area());  
21 }
```

Traits

- Implementa o conceito de interfaces
- Permite a criação de código genérico e reutilizável

```
trait Sound {  
    fn make_sound(&self);  
}  
  
struct Dog;  
struct Cat;  
  
impl Sound for Dog {  
    fn make_sound(&self) {  
        println!("Woof!");  
    }  
}  
  
impl Sound for Cat {  
    fn make_sound(&self) {  
        println!("Meow!");  
    }  
}  
  
fn main() {  
    let dog = Dog;  
    let cat = Cat;  
  
    dog.make_sound(); // Output: Woof!  
    cat.make_sound(); // Output: Meow!  
}
```

Não usa coletor de lixo

Alocação explícita de memória:

O programador deve alocar e liberar memória de forma explícita

Garbage Collection:

Constantemente busca segmentos de memória que já não são mais utilizados enquanto o programa executa

Ownership and Borrowing:

?

SEGURANÇA DE MEMÓRIA E ALTA PERFORMANCE

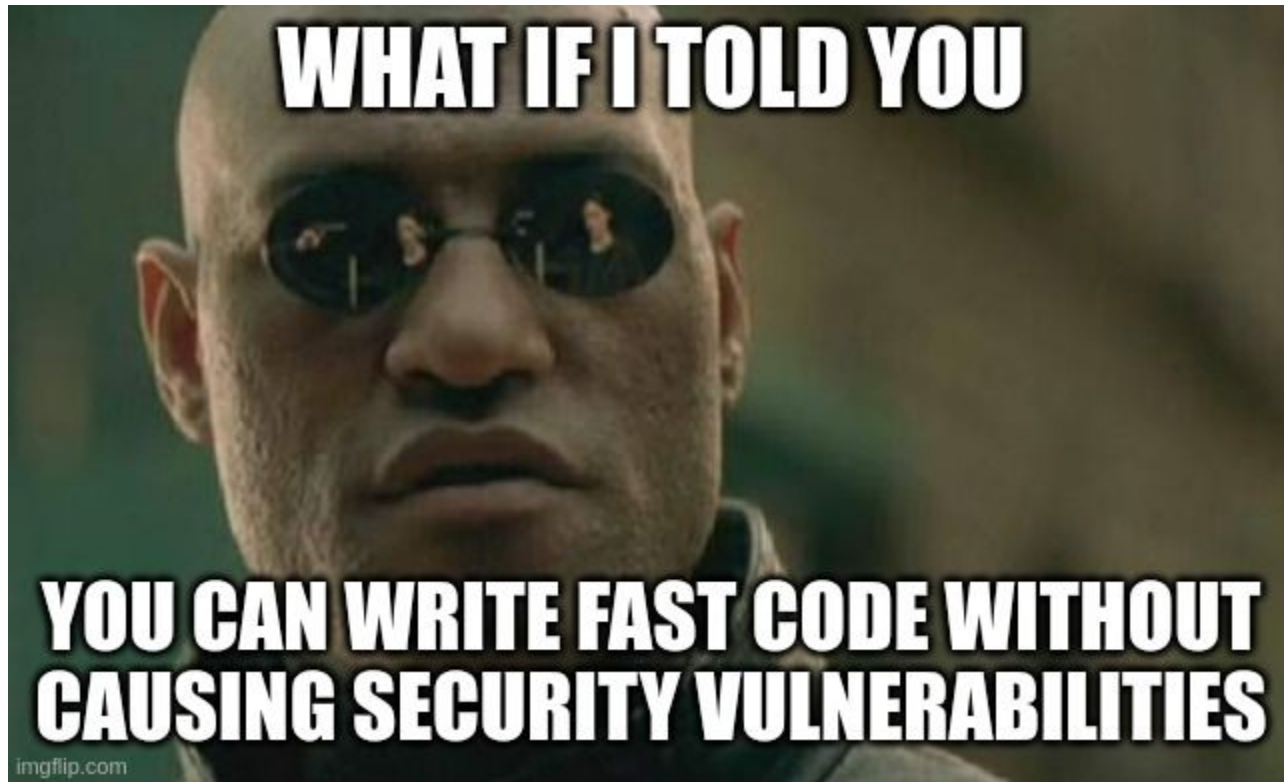
Aproximadamente 60% – 90%

de todas as **vulnerabilidades** de grandes sistemas feitos em C / C++ são problemas de segurança de memória

Vazamentos de memória, acesso inválido à memória, condições de corrida, uso após a liberação, entre outros

Segurança de memória

- Aproximadamente 70% de todas as CVEs (Vulnerabilidades e Exposições Comuns) na Microsoft são problemas de segurança de memória.
 - <https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/>
- Dois terços das vulnerabilidades no kernel Linux são originadas de problemas de segurança de memória.
 - https://static.sched.com/hosted_files/lssna19/d6/kernel-modules-in-rust-lssna2019.pdf
- Um estudo da Apple constatou que 60-70% das vulnerabilidades no iOS e macOS são vulnerabilidades de segurança de memória.
 - <https://langui.sh/2019/07/23/apple-memory-safety/>
- Estima-se que 90% das vulnerabilidades do Android são problemas de segurança de memória.
 - <https://security.googleblog.com/2019/05/queue-hardening-enhancements.html>
- 70% de todos os bugs de segurança no Chrome são problemas de segurança de memória.
 - <https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/>
- Algumas das questões de segurança mais populares de todos os tempos são problemas de segurança de memória:
 - Slammer worm, WannaCry, exploit Trident, HeartBleed, Stagefright, Ghost.



Alta performance

- Controle de memória que elimina a necessidade de um coletor de lixo
- Abstração de baixo nível que permite controle sobre o hardware
- Compilação em um código otimizado antes da execução
- Capacidade de paralelismo

Ownership and Borrowing ou Posse e Empréstimo

Os conceitos de **ownership** e **borrowing** são o que garantem a segurança de memória dos programas em Rust em tempo de compilação.

Recapitulando conceitos...

Como funciona a memória de um programa: uma parte dos dados ficarão salvas na **pilha** e outra no **heap**

Pilha:

Rápida, dados
contidos na pilha
tem tamanho
fixo

Heap:

Mais lenta, é
necessário alocar
um espaço de
memória

OWNERSHIP

Ownership

A memória é gerenciada através de um **sistema de posse**, que tem um conjunto de **regras verificadas em tempo de compilação**

1. Cada valor em Rust possui uma variável que é dita seu owner (sua dona).
2. Pode apenas haver um owner por vez.
3. Quando o owner sai fora de escopo, o valor será destruído.

A memória é automaticamente retornada assim que a variável que a possui sai de escopo

Ownership

```
let x = 5;  
let y = x;
```

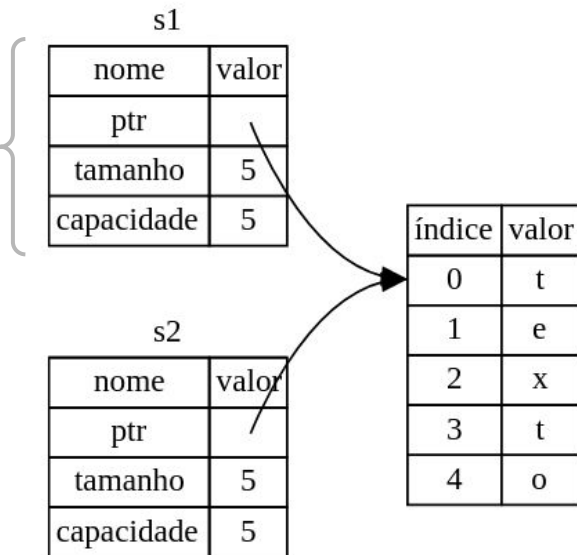
Isso acontece em Rust da mesma forma que aconteceria nas outras linguagens, pois **números inteiros** são valores simples que possuem um tamanho fixo e **são guardados na pilha**

```
let s1 = String::from("texto");  
let s2 = s1;
```

Isso parece similar ao exemplo anterior, mas acontece de forma diferente em Rust, pois a **String não possui tamanho fixo** e consequentemente **ela é guardada na memória de forma diferente**

Ownership

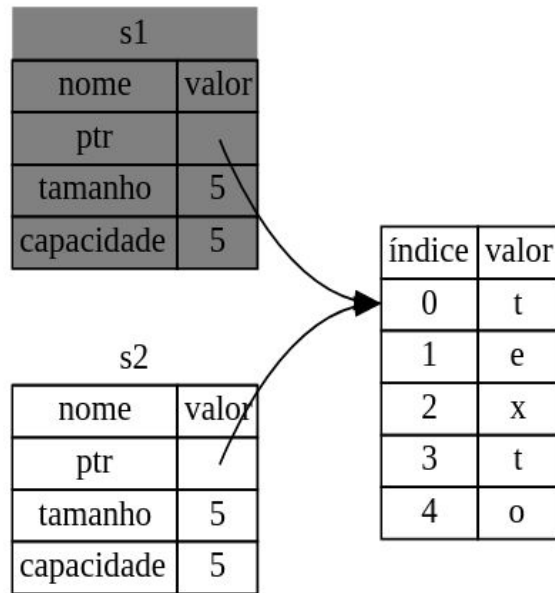
A **String** é feita por **três partes guardadas na pilha**, contendo o tamanho, a capacidade e um ponteiro para o **conteúdo da String** que **vai estar no heap**



Quando atribuímos s1 a s2, não estamos **copiando os dados que estão** na heap e sim os dados que estão **na pilha**

Ownership

Para garantir a **segurança da memória** e **evitar erros de liberação** dupla (double free), o Rust considera que `s1` deixa de ser válida, e portanto, o Rust não precisa liberar nenhuma memória quando `s1` sai de escopo



Ownership

Passar uma variável a uma função irá **mover** ou **copiar**, assim como acontece em uma atribuição.

```
fn main() {  
    let s = String::from("texto"); // s entra em escopo.  
  
    toma_posse(s);                  // move o valor de s para dentro da função...  
                                    // ... e ele não é mais válido aqui.  
  
    let x = 5;                      // x entra em escopo.  
  
    faz_uma_copia(x);               // x seria movido para dentro da função,  
                                    // mas i32 é Copy, então está tudo bem em  
                                    // usar x daqui para a frente.  
  
} // Aqui, x sai de escopo, e depois s. Mas como o valor de s foi movido, nada  
  // de especial acontece.  
  
fn toma_posse(uma_string: String) { // uma_string entra em escopo.  
    println!("{}", uma_string);  
} // Aqui, uma_string sai de escopo, e o método `drop` é chamado. A memória que  
  // guarda seus dados é liberada.  
  
fn faz_uma_copia(um_inteiro: i32) { // um_inteiro entra em escopo.  
    println!("{}", um_inteiro);  
} // Aqui, um_inteiro sai de escopo. Nada de especial acontece.
```

Ownership

Retornar valores
também pode
transferir a posse de
um valor

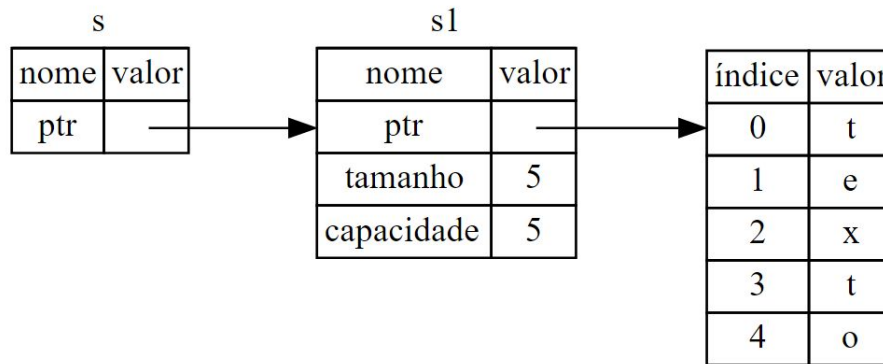
```
fn main() {  
    let s1 = entrega_valor();           // entrega_valor move o valor retornado  
                                         // para s1.  
  
    let s2 = String::from("texto");     // s2 entra em escopo.  
  
    let s3 = pega_e_entrega_valor(s2);  // s2 é movido para dentro da função  
                                         // pega_e_entrega_valor, que também  
                                         // move o valor retornado para s3.  
} // Aqui, s3 sai de escopo e é destruída. s2 sai de escopo, mas já foi movida,  
  // então nada demais acontece. s1 sai de escopo e é destruída.  
  
fn entrega_valor() -> String {         // entrega_valor move o valor  
                                         // retornado para dentro da função  
                                         // que a chamou.  
  
    let uma_string = String::from("olá"); // uma_string entra em escopo.  
  
    uma_string                             // uma_string é retornada e movida  
                                         // para a função que chamou  
                                         // entrega_valor.  
}  
  
// pega_e_entrega_valor vai pegar uma String e retorná-la.  
fn pega_e_entrega_valor(uma_string: String) -> String { // uma_string entra em  
                                                         // escopo.  
  
    uma_string // uma_string é retornada e movida para a função que chamou  
               // pega_e_entrega_valor.  
}
```

BORROWING

Borrowing

Ficar retornando a posse de um valor toda hora que for trabalhar com funções seria tedioso demais, para isso é possível usar referências que permitem que você se **refira a algum valor sem tomar posse**

```
fn main() {  
    let s1 = String::from("texto");  
  
    let tamanho = calcula_tamanho(&s1);  
  
    println!("O tamanho de '{}' é {}.", s1, tamanho);  
}  
  
fn calcula_tamanho(s: &String) -> usize {  
    s.len()  
}
```



Borrowing – Referências Mutáveis

Assim como as variáveis são imutáveis por padrão, referências também são, dessa forma se você quiser mudar essas variáveis você precisa fazer **referências mutáveis**

```
fn main() {  
    let mut s = String::from("texto");  
  
    modifica(&mut s);  
}  
  
fn modifica(uma_string: &mut String) {  
    uma_string.push_str(" longo");  
}
```

Borrowing – Referências Mutáveis

Você **só pode ter uma referência mutável** para um determinado dado **em um determinado escopo**

Isso acontece para prevenir data races em tempo de compilação

Um data race acontece quando esses três fatores ocorrem:

- Dois ou mais ponteiros acessam o mesmo dado ao mesmo tempo.
- Ao menos um dos ponteiros é usado para escrever sobre o dado
- Não há nenhum mecanismo sendo usado para sincronizar o acesso ao dado.

Borrowing – Referências Mutáveis

De forma análoga, nós também **não podemos ter uma referência mutável enquanto temos uma imutável**

Usuários de uma referência imutável não esperam que os valores mudem de repente

Porém, **múltiplas referências imutáveis são permitidas**, pois ninguém que esteja apenas lendo os dados será capaz de afetar a leitura que está sendo feita em outra parte do código

Borrowing – Regras de Referência

Em Rust, o compilador garante que nenhuma referência será uma referência solta!

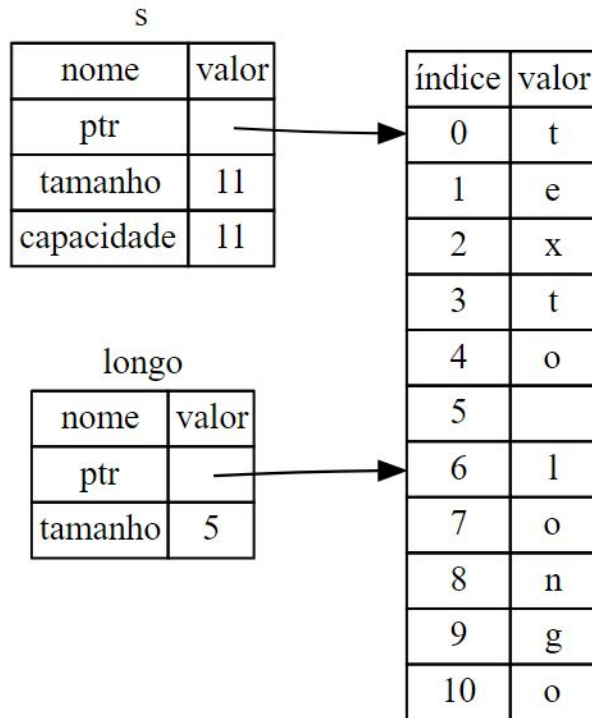
1. Em um dado momento, você pode ter um ou outro, mas não os dois:
 - a. Uma referência mutável.
 - b. Qualquer número de referências imutáveis.
 2. Referências devem ser válidas sempre.
-

Slices

Slices são um tipo de **dado em que não há ownership**.

Eles permitem referenciar uma sequência contígua de elementos em uma coleção em vez de referenciar a coleção inteira

```
let s = String::from("texto longo");  
let texto = &s[0..5];  
let longo = &s[6..11];
```





Exemplificação no playground



APLICAÇÕES

Aplicações

- Muito usado em sistemas de baixo nível, servidores web e infraestrutura de software
- Várias empresas estão reescrevendo suas aplicações em Rust devido às suas vantagens em performance e segurança



Dropbox, Prime Video, Figma, Firefox respectivamente

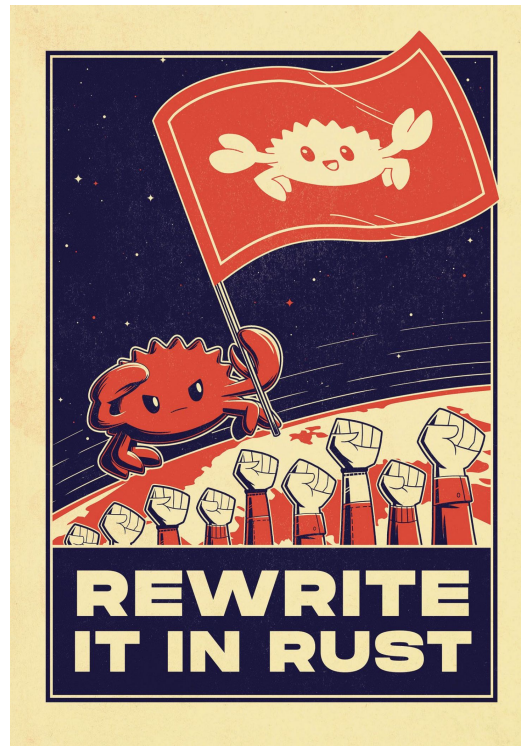


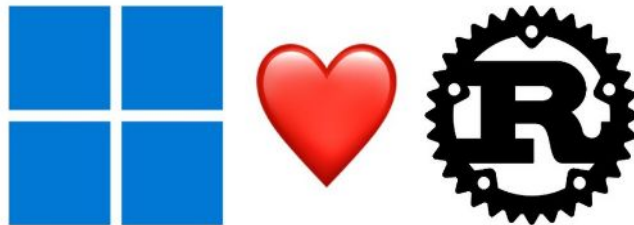
Imagem ilustrativa incentivando a usar Rust

Aplicações



Notícia do Canaltech sobre Rust

**Microsoft está
reescrivendo parte do
Windows em Rust**



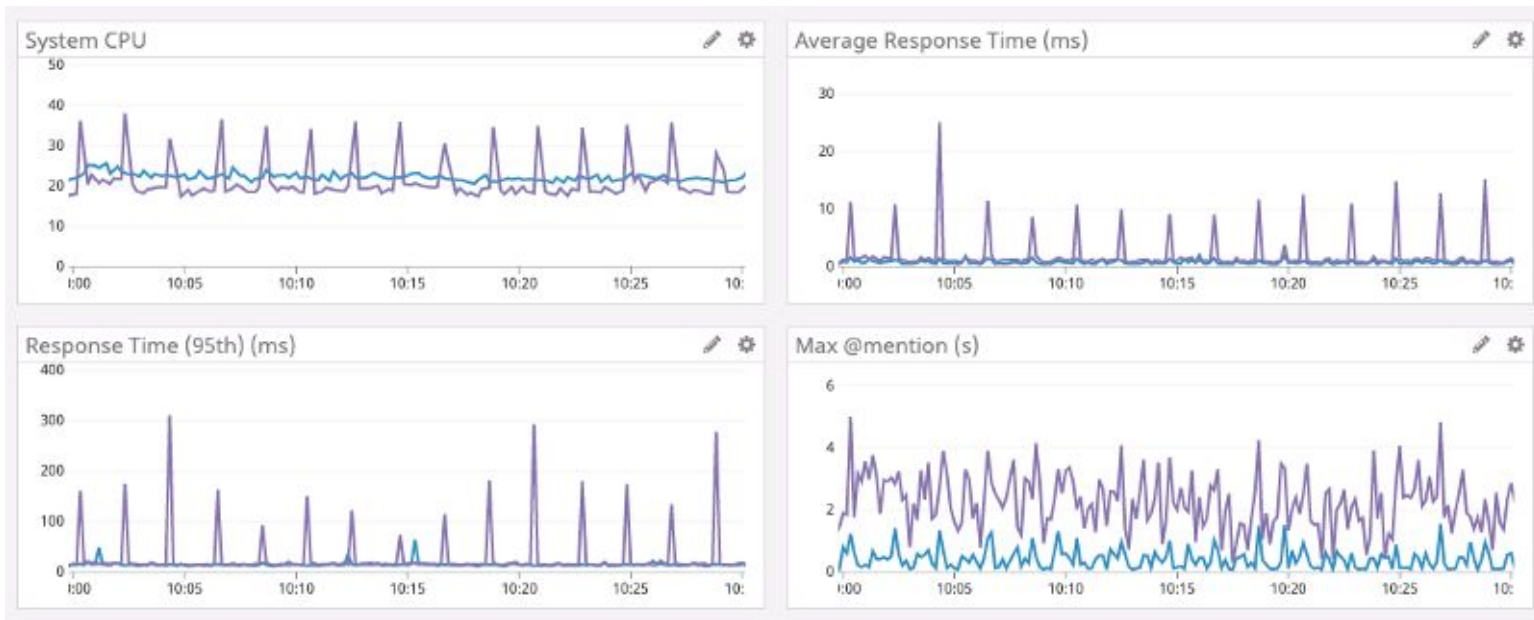
Notícia do Felipe Deschamps sobre Rust

Aplicações



ENGINEERING & DEVELOPERS

POR QUE O DISCORD ESTÁ MUDANDO DE GO PARA RUST



Roxo -> Go | Azul -> Rust

Rust é 100% seguro?

- Infelizmente, nada na vida é perfeito
 - `unsafe{}`

As garantias de segurança de memória do Rust tornam difícil, mas não impossível, criar acidentalmente memória que nunca é liberada (conhecido como vazamento de memória).

- dos autores do livro *The Rust Programming Language*

unsafe{}

- Possibilidade de contornar as restrições de segurança da linguagem
- Seu uso não é recomendado, a menos que seja absolutamente necessário e que o código seja cuidadosamente revisado.

```
fn register_signal_handlers() {  
    // Safe because sigint_handler is async-signal-safe.  
    unsafe { util::set_signal_handlers(&[SIGINT], sigint_handler) };  
    if let Err(err) = block_signal(SIGHUP) {  
        error!("Failed to block SIGHUP: {}", err);  
    }  
}
```

TUTORIAL DE INSTALAÇÃO

Tutorial de instalação Windows

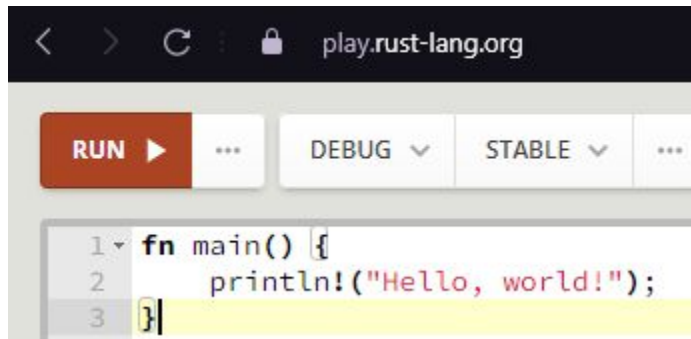
<https://www.rust-lang.org/pt-BR/learn/get-started>

Iniciando

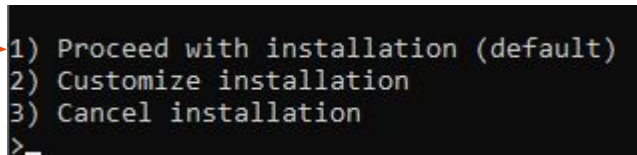
Configure rapidamente seu ambiente de desenvolvimento Rust e escreva uma pequena aplicação!



Site oficial do Rust



Playground do Rust



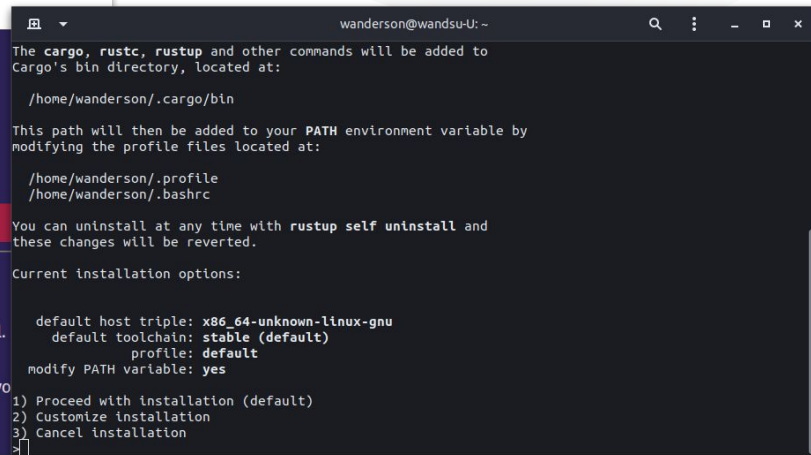
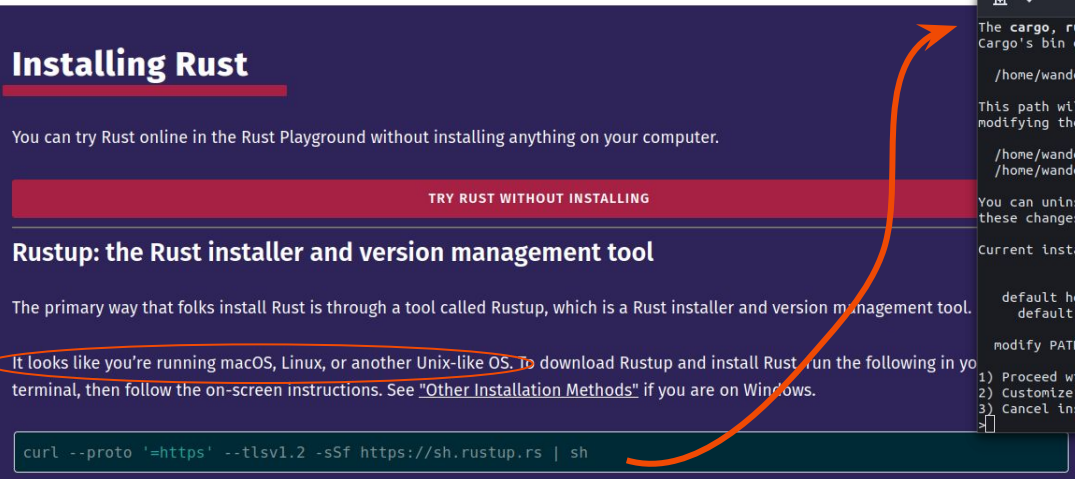
```
1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>
```

Instalação Rust

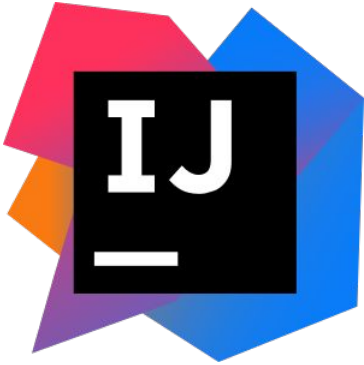
Tutorial de instalação Linux



1. Conferir se o site está recomendando o método de instalação para seu sistema operacional.
2. Copiar o comando da instalação e executar no terminal
3. Escolher a opção 1) para instalação normal



IDEs e ferramentas



Linguagem mais amada e desejada



<https://survey.stackoverflow.co/2022/#most-loved-dreaded-and-wanted-language-want>

Linguagem mais amada e desejada



Considerações finais

- Capacidade do Rust em somar segurança e performance
- Documentação MUITO boa e intuitiva
- A comunidade de Rust é muito engajada!

Referências

The Rust Programming Language - The Rust Programming Language. Disponível em: <<https://doc.rust-lang.org/book/>>. Acesso em: 4 jun. 2023.

A Linguagem de Programação Rust. Disponível em: <<https://github.com/rust-br/rust-book-pt-br>>. Acesso em: 2 jun. 2023.

CANALTECH. Linus Torvalds confirma Rust no kernel do Linux 6.1. Disponível em: <<https://canaltech.com.br/linux/linus-torvalds-confirma-rust-no-kernel-do-linux-61-225803/>>. Acesso em: 2 jun. 2023.

THURROTT, P. Microsoft is Rewriting Parts of the Windows Kernel in Rust. Disponível em: <<https://www.thurrott.com/windows/282471/microsoft-is-rewriting-parts-of-the-windows-kernel-in-rust>>. Acesso em: 2 jun. 2023.

Why Discord is switching from Go to Rust. Disponível em: <<https://discord.com/blog/why-discord-is-switching-from-go-to-rust>>. Acesso em: 2 jun. 2023.

PERGUNTAS?

OBRIGADO!



Leonardo Yvens in Rust Brazil

Conclusão: Rust é difícil porque escrever código correto é difícil.

t.me/rustlangbr/65133

Apr 16 at 15:46