

# The Rust Programming Language

# Referências técnicas

- Editor online: <https://play.rust-lang.org/>
- Livro oficial: <https://doc.rust-lang.org/stable/book/>
- Documentações: <https://docs.rs/>
- Exemplos: <https://doc.rust-lang.org/rust-by-example/>
- Exercícios: <https://github.com/rust-lang/rustlings>
- Ótimo canal no youtube: <https://www.youtube.com/@letsgorusty>

# Origens

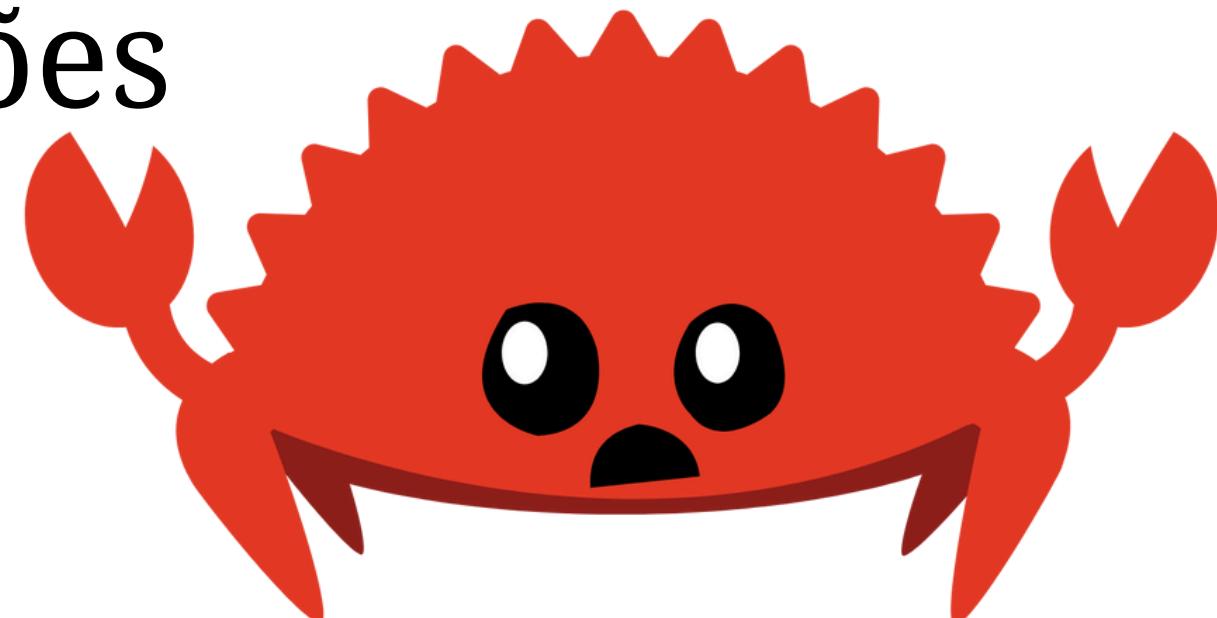
- Iniciado em 2006 por Graydon Hoare, funcionário da mozilla
- O incentivo para sua criação foi o elevador de seu prédio que tinha problemas de software constantes, provavelmente por problemas de memória
- Sua indignação com o elevador o levou a imaginar uma linguagem com segurança de memória garantida, descrevendo-a como “overengineered for survival”
- Em 2009 a Mozilla entrou na jogada, financiando o até então projeto pessoal
- Em 2015 foi lançada sua primeira versão estável, prometendo estabilidade sem estagnação

# Inspirações e bases

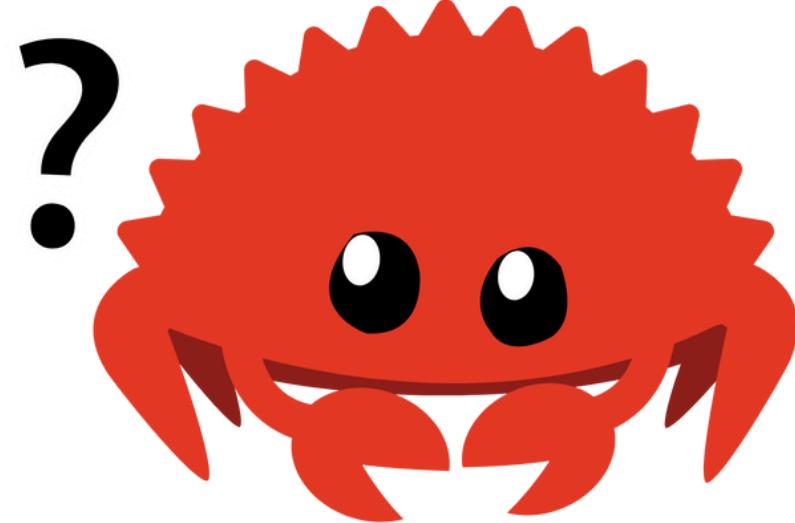


# O que rust não tem?

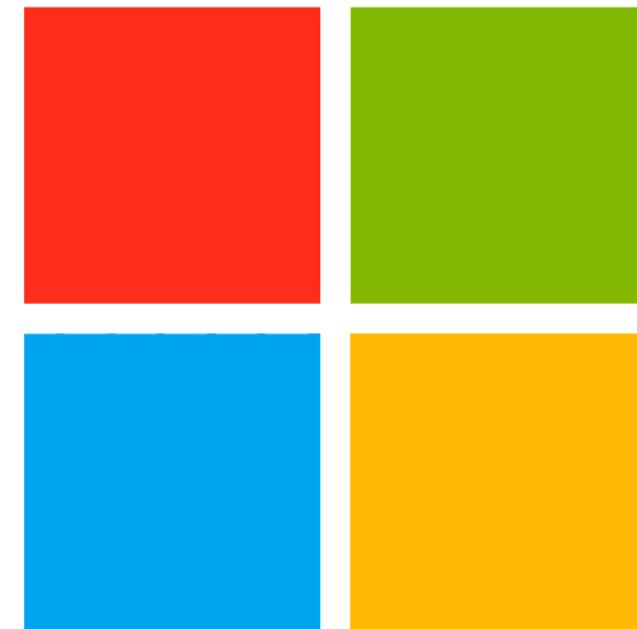
- Herança
- Exceptions e try catch
- Garbage Collector
- Conceito de NULL
- Sobrecarga de funções
- Operador ternário



# Quem usa rust?

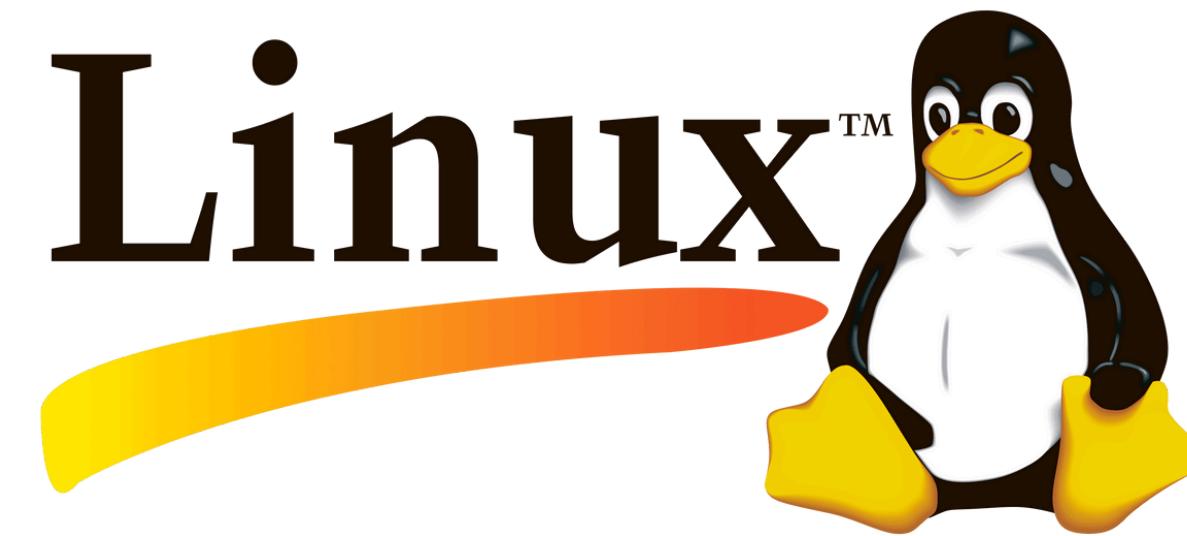


- Pequenas startups
- Projetos open source irrelevantes
- Produtos nichados e desconhecidos
- Desempregados



# Microsoft

- Reescrevendo partes centrais do windows em rust
- De acordo com o CTO da azure estão “all in” em rust
- Um dos principais membros e financiadores da rust foundation



- Além de C, Rust é a unica linguagem aceita para o desenvolvimento do kernel
- Primeiro ponto de adoção em outubro de 2022
- Apesar de muito drama da comunidade, Rust é aprovado pelo Linus Torvalds
- Empresas focadas em linux como RedHat, Canonical e System76 foram arrastadas para o uso de rust



**cloudwalk**

- Fintech unicórnio brasileira dona de produtos como Infinite Pay e JIM
- Desenvolve a blockchain Stratus por trás de seus sistemas em Rust
- Anfitriã da Rust-SP Meetup

# Rust Foundation

## Platinum



## Silver



## Associate



# O futuro é escrito em rust



>Nushell





# Por que Rust?

## Performance

Rust é atualmente considerada uma das linguagens mais rápidas do mundo, por muitas vezes vencendo dos gigantes e não mais inabaláveis C e C++

## Produtividade

Rust é uma linguagem moderna, com ferramentas modernas, tal como Typescript ou Golang. Liderando o ranking de linguagens mais amadas do stackoverflow **todos os anos desde 2016**

## Confiabilidade

Se o código compila, ele provavelmente funciona. Vários problemas comuns são **impossíveis** em rust, como null pointer exceptions ou tentar usar uma variavel builder depois de chamar build

# Muito papo pouco código

Vamos começar a brincadeira!

Inicialmente pelo playground <https://play.rust-lang.org/>

# O compilador é seu amigo

Cansado de erros idiotas ocupando todo seu tempo? Isso não é um problema em rust, ele sabe onde está faltando o ; ou o a chave aberta nunca fechada, e vai te dizer, vamos brincar com o hello world em rust e javascript

```
1 // Abrimos a chave e nunca fechamos
2 fn main() {
3     println!("Hello, world!");
4 }
```

```
1 // Mesma coisa, aberto e nunca fechado
2 function funcaojs (){
3     console.log("hello")
4 }
```

O compilador do rust  
identifica o problema e te  
diz exatamente como  
corrigir

```
Compiling playground v0.0.1 (/playground)
error: this file contains an unclosed delimiter
--> src/main.rs:3:32
2 | fn main() {
   |         - unclosed delimiter
3 |     println!("Hello, world!");
   | ^
error: could not compile `playground` (bin "playground") due to 1 previous error
```

O interpretador do javascript  
cospe algum tipo de magia  
ancestral que pode até fazer  
sentido pra ele mas não pra  
qualquer pessoa sã

```
SyntaxError: Unexpected end of input
    at wrapSafe (node:internal/modules/cjs/loader:1666:18)
    at Module._compile (node:internal/modules/cjs/loader:1708:20)
    at Object..js (node:internal/modules/cjs/loader:1899:10)
    at Module.load (node:internal/modules/cjs/loader:1469:32)
    at Module._load (node:internal/modules/cjs/loader:1286:12)
    at TracingChannel.traceSync (node:diagnostics_channel:322:14)
    at wrapModuleLoad (node:internal/modules/cjs/loader:235:24)
    at Module.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:152:5)
    at node:internal/main/run_main_module:33:47

Node.js v24.2.0
```

# Variaveis

```
1 fn main(){  
2     // Declarando e atribuindo uma variavel  
3     let x = 5;  
4  
5     // Mostrando o valor de x formatado  
6     println!("O valor de x é {x}");  
7  
8     // Agora vamos mudar o valor de x  
9     x = 6;  
10    | println!("O valor de x é {x}");  
11 }
```

# Tipos

- i32 → int
- u32 → unsigned int
- i64 → long
- f32 → float
- f64 → Double
- char → Caracter utf8
- String → **Owned** string
- &str → Referencia de string
- (U, W, U) → Tupla
- [T; x] → Array de tipo T e tamanho x

# Ownership

- O que são os tais “Owners”
- O que é um escopo
- Como os “Owners” são responsáveis por seus recursos
- Como isso é diferente de todo resto

## **Um valor, um dono**

- Cada valor garantidamente possui um dono que o gerencia
- So pode haver um dono por vez
- Quando o dono sai de escopo, o valor é descartado

# Um valor, um dono

```
1 fn main(){  
2     // s1 se tornou dona do valor completamente valido dessa string  
3     let s1 = String::from("Uma belissima string");  
4     println!("O valor de s1 é: {s1}");  
5     let s2;  
6  
7     // s1 deixou de se tornar dono de seu valor, que foi passado para s2  
8     // Rust usa o termo "moved" para essa situação, a propriedade do valor  
9     // e responsabilidade por ele passa a ser de s2  
10    s2 = s1;  
11  
12    // Válido, s2 agora é uma dona  
13    println!("O valor de s2 é: {s2}");  
14  
15    println!("Um pequeno errinho de compilação {s1}");  
16 }
```

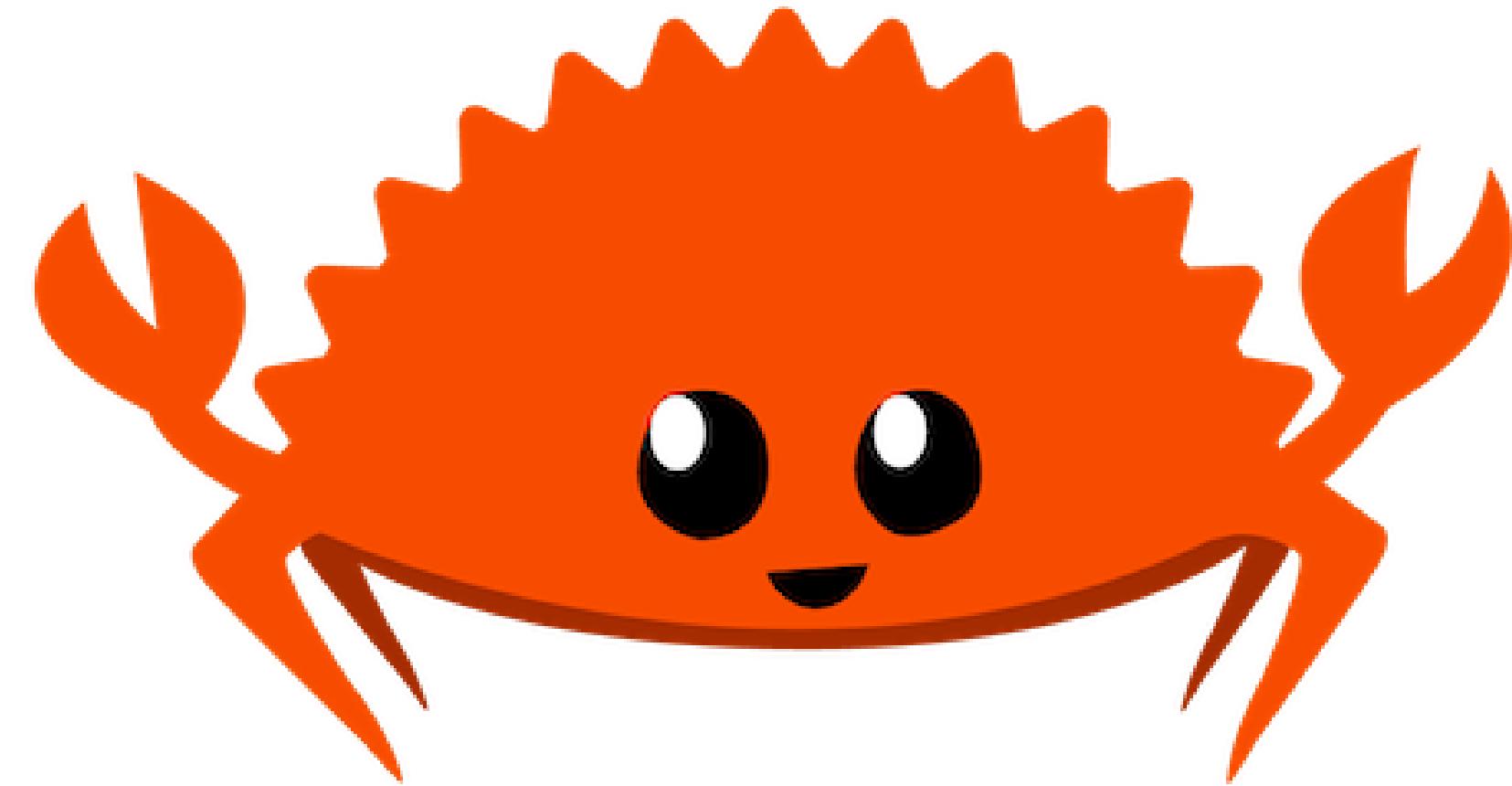
# Donos e escopos

```
1 fn main() {
2     println!("--- Início do main ---");
3
4     let s1 = String::from("Eu sou s1");
5     println!("s1 foi criado :D");
6
7     { // Escopo 'a
8         let s2 = String::from("Eu sou s2"); // s2, viva somente dentro do escopo 'a
9         println!("s2 foi criado :D");
10        println!("--- Fim do escopo interno ---");
11    } // Fim do escopo 'a, toda memória consumida por valores associados a ele
12    // são limpos AGORA, nesse caso a variavel s2
13
14    println!("O escopo interno acabou, mas s1 ainda existe :)");
15    println!("--- Fim do main ---");
16
17 } // Fim do programa, todos os valores são liberados AGORA,
18 //nesse caso a variável s1
```

# Movendo de escopo

```
1 fn main(){  
2     // s1 se tornou dona do valor completamente valido dessa string  
3     let s1 = String::from("Uma belissima string");  
4     println!("O valor de s1 é: {s1}");  
5     let s2;  
6  
7     // s1 deixou de se tornar dono de seu valor, que foi passado para s2  
8     // Rust usa o termo "moved" para essa situação, a propriedade do valor  
9     // e responsabilidade por ele passa a ser de s2  
10    s2 = s1;  
11  
12    // Válido, s2 agora é uma dona  
13    println!("O valor de s2 é: {s2}");  
14  
15    println!("Um pequeno errinho de compilação {s1}");  
16 }
```

# Como o modelo Rust é diferente de C e C++ ou Java e C#



# Funções

```
1 fn main() {  
2     let resultado = soma(5, 10);  
3     println!("A soma é: {resultado}");  
4 }  
5  
6 // fn nome(parametro:tipo_do_parametro) -> tipo_do_retorno {corpo}  
7 fn soma(x: i32, y: i32) -> i32 {  
8     // Retorno feio mas funciona  
9     return x + y;  
0 }
```

# Funções

```
1 fn main() {  
2     let s = String::from("meu texto");  
3  
4     // Passar `s` para a função...  
5     toma_posse(s); // ...MOVE a posse de `s` para o parâmetro da função.  
6  
7     // Tentamos usar `s` aqui, mas a posse já se foi!  
8     // Esta linha não vai compilar.  
9     println!("Na main, tentando usar s: {}", s);  
10 }  
11  
12 fn toma_posse(uma_string: String) { //Escopo da função 'a  
13 // `uma_string` se torna a dona e está associada ao escopo 'a.  
14     println!("Na função, eu sou a dona de: {}", uma_string);  
15 } // `uma_string` sai do escopo e é "dropada". A memória é liberada.
```

# Referencias

- O conceito de borrowing e como se relaciona ao sistema de ownership
- Como são diferente dos ponteiros e referências de C e C++
- Como o rust sabe que referências ainda são válidas
- O temido borrow checker



uff referências 🔥

# O que é borrowing no contexto de ownership

```
1 fn main() {
2     let mut x = 5; // Um i32 na stack.
3
4     // Usamos o operador `&` para criar uma referência para `x`.
5     // O tipo de `y` não é `i32`, é `&i32` (uma "referência a um i32").
6     let y = &x;
7
8     println!("O valor de x é: {x}");
9     println!("O valor de y é: {y}");
10    println!("O valor de y é uma referência para x.");
11
12    // O que acontece se tentarmos mudar o valor de x enquanto y ainda
13    // está o referenciando?
14    x = 12;
15    println!("O valor de x é: {x}");
16    // O que acontece se y tentar mudar o valor emprestado de x?
17    // Precisamos dereferenciar y com * para mudar seu valor
18    *y = 23;
19    println!("O valor de y é: {y}");
20
21    // Podemos provar que `y` é um ponteiro/endereço.
22    // A formatação `{:p}` nos mostra o endereço de memória para o qual `y` aponta.
23    println!("O endereço de memória que y aponta é: {:p}", y);
24 }
```

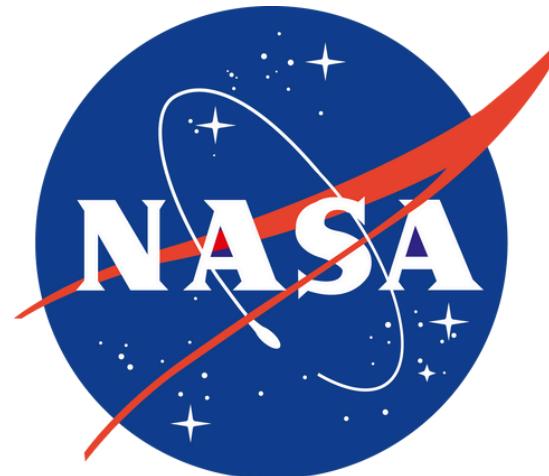
# Como isso é diferente de C e C++?

Este é um código  
perfeitamente válido  
em C, e cheio de  
problemas

Parece algo besta mas  
diversos incidentes  
graves foram causados  
por esse tipo de  
comportamento que o  
Rust nasceu para  
evitar

```
50 #include <stdio.h>
29
28 int main() {
27
26     // Declaramos dois ponteiros para um inteiro
25     int *ptr;
24     int *ptr2;
23
22     { // Início de um novo escopo
21         int x = 5;
20         ptr = &x; // O ponteiro agora aponta para o endereço de x
19         printf(format: "Dentro do escopo, o ponteiro aponta para o valor: %d\n", *ptr);
18
17     // Em C podemos mudar o valor de ponteiros livremente
16     *ptr = 32;
15     printf(format: "O valor de x agora é %d\n", x);
14 } // Fim do escopo. A variável 'x' é destruída. Sua memória na stack
13 // é marcada como livre para ser usada por outra coisa.
12
11     ptr2 = ptr;
10
9     *ptr2 = 90;
8
7     // O ponteiro 'ponteiro' ainda existe e aponta para aquele endereço de memória...
6     // que agora é inválido.
5     printf(format: "Fora do escopo, o que há no endereço do ponteiro? Valor: %d\n", *ptr);
4     // Isto é um "Dangling Pointer" / "Use-After-Free".
3
2     return 0;
1 }
```

# Por que o jeito C é problematico?



Crash do Mars Pathfinder (C, 1997)

- **Causa:** Race condition. Múltiplas tarefas com prioridades diferentes tentavam aceder a um recurso partilhado sem a sincronização adequada.
- **Consequências:** A sonda em Marte sofria reboots completos do sistema, arriscando a perda da missão.
- **Prevenção em Rust:** O conceito de Ownership e o borrow checker impedem data races (um tipo de race condition) em tempo de compilação, forçando o uso de primitivas de sincronização seguras como Mutex<T> para gerir o acesso a dados partilhados.



CROWDSTRIKE

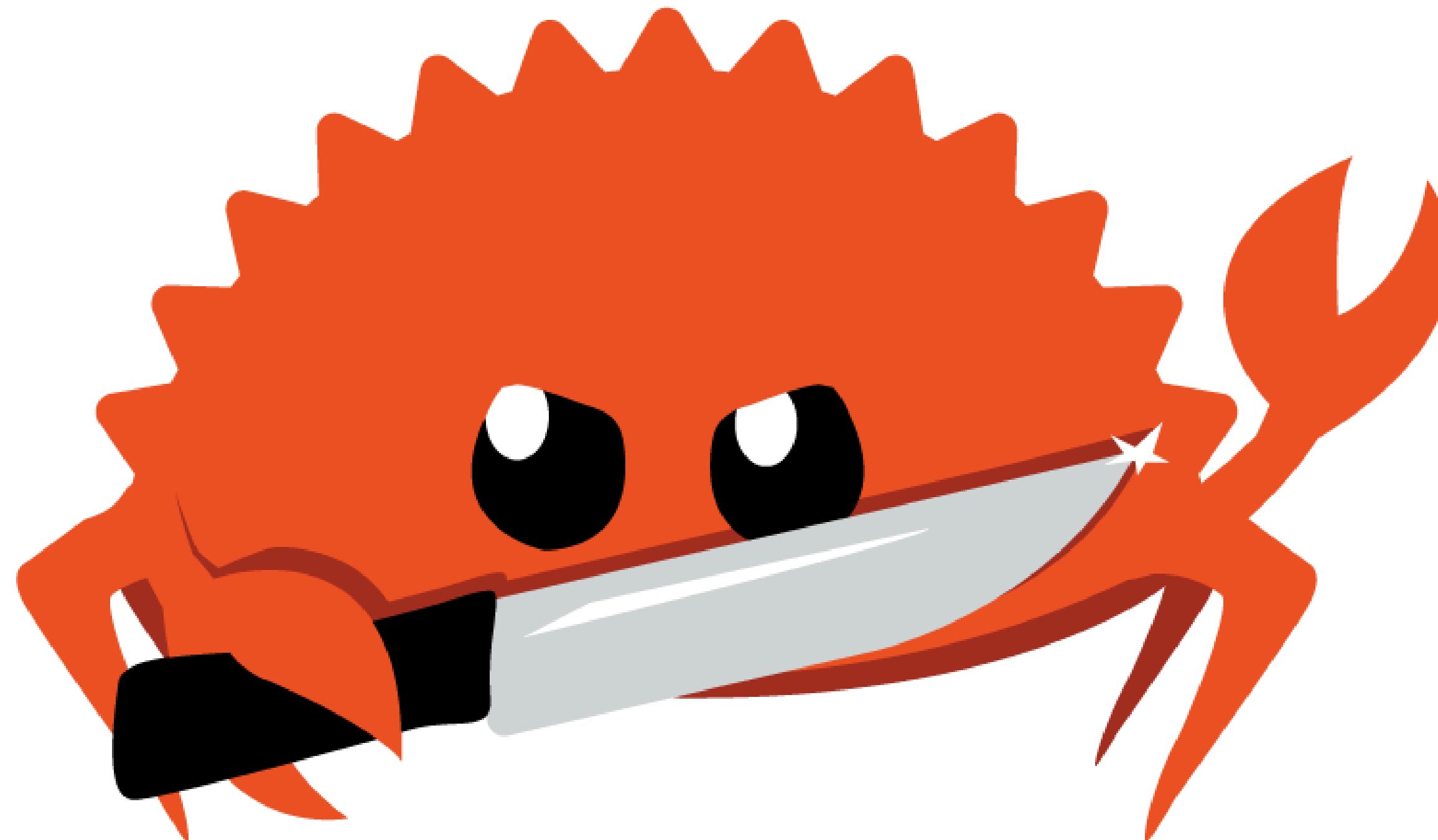
CrowdStrike Blackout (C e C++, 2024)

- **Causa:** Um erro lógico numa atualização de driver levou a um acesso inválido à memória no nível do kernel. Um erro de verificação de limites (bounds check)
- **Consequências:** Paralisia global, bilhões em prejuízo
- **Prevenção em Rust:** O borrow checker e as verificações de limites automáticas do Rust em acessos a slices teriam tornado este tipo de acesso inválido à memória um erro em tempo de compilação ou um panic controlado, prevenindo a falha catastrófica do kernel.

# Validade das referencias

```
1 fn main() {
2     let string1 = String::from("abcd");
3     {
4         |
5
6     let string2 = String::from("xyz");
7
8     let resultado = maior_string(&string1, &string2);
9     println!("A maior string é {}", resultado);
10    }
11
12    println!("A primeira string é: {}", string1);
13 }
14
15 // Esta função recebe duas referências de string (&str)
16 // e tenta retornar uma referência para uma delas.
17 fn maior_string<'a>(x: &String, y: &String) -> &String { // O que vai aqui?
18     if x.len() > y.len() {
19         x
20     } else {
21         y
22     }
23 }
```

# O temido borrow checker



**Ele só é mal compreendido**



# Collections

```
1 fn main() {  
2     // Vamos criar um vetor de inteiro  
3     // O vec é dono de todos os valores que ele contém  
4     let lista: Vec<i32> = Vec::new();  
5     lista.push(1);  
6     lista.push(2);  
7     lista.push(3);  
8     println!("{}{lista:?}");  
9  
10    // Há um jeito ainda mais fácil de criar um vetor  
11    // com o macro vec![]  
12    let lista: Vec<i32> = vec![4,3,2];  
13    println!("{}{lista:?}");  
14 }
```

# Slices where?

```
1 // Esta função retorna o ÍNDICE do byte onde a primeira palavra termina.
2 fn indice_primeira_palavra(s: &String) -> usize {
3     let bytes = s.as_bytes(); // Converte a String em um array de bytes
4     for (i, &item) in bytes.iter().enumerate() {
5         if item == b' ' { // `b' '` é a sintaxe para um byte literal
6             return i; // Encontramos um espaço, retornamos a posição
7         }
8     }
9
10    s.len() // Se não houver espaços, a palavra é a string inteira
11 }
12
13 fn main() {
14     let mut frase = String::from("olá mundo");
15
16     let indice = indice_primeira_palavra(&frase);
17
18     // O que acontece se a String original mudar?
19     frase.clear(); // Esvaziamos a String. `frase` agora é ""
20
21     // O que o nosso `indice` (que ainda tem o valor 4) significa agora?
22     // Ele não tem mais nenhuma conexão com o estado atual da `frase`.
23     // Usar este índice agora levaria a um bug lógico ou a um crash.
24     // E o compilador NÃO nos ajudou aqui!
25     println!("O índice é {}", indice);
26 }
```

# ♥♥♥ Slices ♥ ♥

```
1 // A função agora retorna um `&str`, um "string slice".
2 fn primeira_palavra(s: &String) -> &str {
3     let bytes = s.as_bytes();
4
5     for (i, &item) in bytes.iter().enumerate() {
6         if item == b' ' {
7             // Retornamos um slice do começo da string até o índice i
8             return &s[0..i];
9         }
10    }
11    // Retornamos um slice da string inteira
12    &s[..]
13 }
14
15 fn main() {
16     let mut frase = String::from("olá mundo");
17
18     let palavra = primeira_palavra(&frase);
19
20     // Vamos tentar limpar a frase...
21     //frase.clear(); // <-- ISTO NÃO COMPILA!
22
23     // O compilador nos impede de obter uma referência mutável (`clear`)
24     // porque já existe uma referência imutável (`palavra`) em uso.
25     println!("A primeira palavra é: {}", palavra);
26 }
```

# Construindo seus próprios tipos

- Structs
- Metodos
- Funções associadas
- Traits
- Enums

# Structs

```
1 // Macro que nos permite mostrar o struct
2 // com println, porém só formatado com :?
3 #[derive(Debug)]
4 struct Usuario{
5     nome: String,
6     idade: u8,
7     ativo: bool
8 }
9
10 fn main() {
11     let u = Usuario{
12         nome: "Diego".to_string(),
13         idade: 19,
14         ativo: true
15     };
16     println!("{}:?}", u)
17 }
```

# Métodos e funções associadas

```
1 // No struct você define somente os DADOS que seu tipo precisa
2 #[derive(Debug)]
3 struct Retangulo{
4     altura: i32,
5     largura: i32,
6 }
7
8 // Na implementação você define o COMPORTAMENTO do seu tipo
9 // Structs em rust são similares a classes em outras linguagens
10 // Porém com as partes que definem dados e comportamento sendo separadas e independentes
11 impl Retangulo{
12     // Isso é uma função associada, uma função construtora
13     // Self representa o mesmo tipo onde a função está implementada
14     // nesse caso Retangulo, usar Self ou Retangulo tem o mesmo
15     // efeito
16     fn new(altura: i32, largura:i32) -> Self{
17         Self{
18             // Se o valor que voce quer atribuir possui o mesmo nome do campo, você não precisa especificar
19             altura,
20             largura
21         }
22     }
23
24     // Isso é um metodo, a grande diferença pra uma função associada
25     // é que um metodo possui o primeiro parametro self, self é o
26     // equivalente ao this
27     fn area(&self) -> i32{
28         self.altura * self.largura
29     }
30 }
31
32 fn main() {
33     let r = Retangulo::new(2,5);
34     let area = r.area();
35     println!("A area de {r:?} é {area}");
36 }
```

# Traits

- **Definem um comportamento**
- **Permitem polimorfismo através de dynamic dispatch**
- **Desacoplam a definição do comportamento de sua implementação**

# Traits: Em structs

```
1 // Estamos definindo o comportamento que queremos, a forma como ele é implementado não interessa
2 // Traits se traduzem diretamente pra interfaces em outras linguagens
3 trait Forma{
4     fn area(&self) -> i32;
5 }
6
7 // Debug tambem é uma trait, mas nós o implementamos automaticamente através do macro derive
8 #[derive(Debug)]
9 struct Retangulo{
10     altura: i32,
11     largura: i32,
12 }
13
14 // A unica diferença pra implementação anterior é que indicamos a trait primeiro e quem a implementa
15 // Se não implementarmos todos os comportamentos definidos o código não irá compilar
16 impl Forma for Retangulo{
17     fn area(&self) -> i32{
18         self.altura * self.largura
19     }
20 }
21
22 #[derive(Debug)]
23 struct Quadrado{
24     lado: i32
25 }
26
27 impl Forma for Quadrado{
28     fn area(&self) -> i32{
29         self.lado * self.lado
30     }
31 }
```

# Traits: Uso e polimorfismo

```
34 fn main() {
35     let r = Retangulo{altura: 2, largura: 5};
36     let q = Quadrado {lado:3};
37
38     // Pra brincar, vamos escolher pelo terminal o que queremos ver
39     let mut input = String::new();
40
41     println!("Digite 1 pra quadrado ou qualquer coisa pra retangulo: ");
42     // Lendo uma linha do terminal
43     std::io::stdin()
44         .read_line(&mut input)
45         .expect("Falha ao ler a linha."); // Lida com erros de I/O
46
47     // 3. Parse: Tentamos converter o texto para um número.
48     //     `trim()` remove espaços em branco e a quebra de linha `\n` no final.
49     //     `parse()` tenta converter para o tipo que especificamos (aqui, u32).
50     //     `parse()` retorna um `Result`, então precisamos de lidar com o erro.
51     let n: u32 = input
52         .trim()
53         .parse()
54         .unwrap_or(0); // Lida com erros de parsing
55
56     // Vamos decidir o que é a forma?
57     let forma: &dyn Forma = if n == 1 {&q} else {&r};
58     let area = forma.area(); // Qualquer q seja a forma, tem uma área
59     println!("A área da forma é {area}, mas não sei que tipo de forma é essa");
60 }
```

# Enums e pattern matching



# Enums

- Enums em rust carregam dados
- Enums são exaustivos
- Tudo é um enum
- Enum é vida

# Enums com dados

```
1 // Definimos um enum `IpAddrKind` (Tipo de Endereço IP)
2 // que pode ser ou `V4` ou `V6`.
3 enum IpAddrKind {
4     V4,
5     V6,
6 }
7
8 // Podemos usar nosso enum em um struct.
9 struct IpAddr {
10     kind: IpAddrKind,
11     address: String,
12 }
13
14 // Mas em Rust, podemos fazer melhor!
15 // Podemos colocar os dados DIRETAMENTE dentro das variantes do enum.
16 enum IpAddrMelhorado {
17     V4(u8, u8, u8, u8), // A variante V4 agora tem 4 bytes associados.
18     V6(String),         // A variante V6 tem uma String associada.
19 }
20
21 fn main() {
22     // Criamos instâncias do nosso enum melhorado.
23     let casa = IpAddrMelhorado::V4(127, 0, 0, 1);
24     let loopback = IpAddrMelhorado::V6(String::from("::1"));
25 }
26 |
```

# Pattern matching

Rust não possui o conceito de nulo, a forma de representarmos valores opcionais é com o enum Option e suas variantes Some(T) e None

```
1 // Esta função recebe uma idade opcional e retorna um Resultado.
2 // O tipo de sucesso é uma String, e o tipo de erro também é uma String.
3 fn verificar_acesso(idade_opcional: Option<u32>) -> Result<String, String> {
4     // Primeiro, usamos `match` para lidar com o Option.
5     match idade_opcional {
6         // Caso 1: Nós TEMOS uma idade.
7         Some(idade) => {
8             // Agora, verificamos a idade para decidir o Result.
9             if idade >= 18 {
10                 // Sucesso! Retornamos um Ok() com uma mensagem.
11                 Ok(String::from("Acesso permitido. Bem-vindo!"))
12             } else {
13                 // Falha! Retornamos um Err() com uma mensagem de erro.
14                 Err(String::from("Acesso negado. Idade insuficiente."))
15             }
16         }
17         // Caso 2: Nós NÃO temos uma idade.
18         None => {
19             // Falha! Retornamos um Err() com outra mensagem de erro.
20             Err(String::from("Acesso negado. Idade não fornecida."))
21         }
22     }
23 }
24
25 fn main() {
26     // Cenário 1: Sucesso
27     let resultado1 = verificar_acesso(Some(20));
28     println!("Tentativa 1: {:?}", resultado1);
29
30     // Cenário 2: Falha por idade
31     let resultado2 = verificar_acesso(Some(15));
32     println!("Tentativa 2: {:?}", resultado2);
33
34     // Cenário 3: Falha por ausência de dados
35     let resultado3 = verificar_acesso(None);
36     println!("Tentativa 3: {:?}", resultado3);
37
38     // Lidando com o resultado de forma mais elegante com `match`
39     println!("\n--- Lidando com a Tentativa 1 ---");
40     match resultado1 {
41         Ok(mensagem_sucesso) => println!("Sucesso: {}", mensagem_sucesso),
42         Err(mensagem_erro) => println!("Erro: {}", mensagem_erro),
43     }
44 }
```

Rust não possui conceitos especiais de exceptions e try catch, a forma como representamos erros é com o enum Result e suas variantes Ok(T) e Err(E)

# O que mais há de rust?

- Statements vs expressions
- Macros
- Smart pointers
- Generics
- Async await
- Modulos e projetos

# Vamo explorar

