

**E se seus programas
nunca crashassem?**



<https://salies.github.io/secompp-rust/>

Introdução ao Rust

Efficiently safe.

Gustavo Becelli do Nascimento
[@becelli](#)

Daniel Henrique Serezane Pereira
[@salies](#)

Parte 1

Rust: o que, por que e como

O que?

- Linguagem de programação compilada, imperativa e estruturada, criada em 2006 como um projeto pessoal de Graydon Hoare, funcionário da Mozilla. O projeto foi adotado pela empresa em 2009. A linguagem foi publicada em 2010. Seu compilador tornou-se capaz de compilar a si mesmo em 2011. [1] [2]
- Teve sua primeira versão estável em 15 de maio de 2015. [3]
- Seu principal foco é **segurança**, contudo também visa ser o mais performática possível. [2]
- É atualmente a linguagem mais adorada pelos programadores (têm sido por 7 anos seguidos). [4]

IMPORTANTE — Como assim “segurança”?

- É estaticamente tipada. [5]
- **Impõe segurança de memória:** principal diferencial da linguagem

Um pouco mais de história...

- Em 2020, com os impactos econômicos da pandemia, $\frac{1}{4}$ da equipe da Mozilla foi desbancada, introduzindo incertezas sobre o futuro da linguagem;
- Em 2021, a AWS, Huawei, Google, Microsoft e a Mozilla anunciam a “Rust Foundation”, que patrocinam juntas o projeto.

Por que?

- Desenvolvida por “programadores C++ frustrados”, visando sanar as principais dificuldades da linguagem. [2]
- Criada com *systems programming* em mente. [2]
- De acordo com a Microsoft, 70% dos CVEs (vulnerabilidades e exposições comuns de software) são causados por erros de segurança de memória. [6][7]
- Rust impõe a segurança de memória sem utilizar-se de um garbage collector, em oposição à sua “rival”, Go.
- Possui um ecossistema oficial bem integrado e que impõe um certo nível de organização ao projeto, mas fácil de usar.



Por que?

- Desenvolvida por “programadores C++ frustrados”, visando sanar as principais dificuldades da linguagem. [2]
- Criada com *systems programming* em mente. [2]
- De acordo com a Microsoft, 70% dos CVEs (vulnerabilidades e exposições comuns de software) são causados por erros de segurança de memória. [6][7]
- Rust impõe a segurança de memória sem utilizar-se de um garbage collector, em oposição à sua “rival”, Go.
- Possui um ecossistema oficial bem integrado e que impõe um certo nível de organização ao projeto, mas fácil de usar.

Principal motivo da popularidade em empresas.



Por que?

- Desenvolvida por “programadores C++ frustrados”, visando sanar as principais dificuldades da linguagem. [2]
- Criada com *systems programming* em mente. [2]
- De acordo com a Microsoft, 70% dos CVEs (vulnerabilidades e exposições comuns de software) são causados por erros de segurança de memória. [6][7]
- Rust impõe a segurança de memória sem utilizar-se de um garbage collector, em oposição à sua “rival”, Go.
- Possui um ecossistema oficial bem integrado e que impõe um certo nível de organização ao projeto, mas fácil de usar.

Principal motivo da popularidade com desenvolvedores independentes, em pequenos projetos ou em projetos pessoais.



Por que?

- Desenvolvida por “programadores C++ frustrados”, visando sanar as principais dificuldades da linguagem. [2]
- Criada com *systems programming* em mente. [2]
- De acordo com a Microsoft, 70% dos CVEs (vulnerabilidades e exposições comuns de software) são causados por erros de segurança de memória. [6][7]
- Rust impõe a segurança de memória sem utilizar-se de um garbage collector, em oposição à sua “rival”, Go.
- Possui um ecossistema oficial bem integrado e que impõe um certo nível de organização ao projeto, mas fácil de usar.
- Por esses motivos, conquistou uma comunidade muito dedicada, mesmo que não tão grande quanto a de Python, por exemplo.



Por que?

n-body

source	mem	gz	cpu
<u>C gcc #9</u>	992	1633	2.12
<u>Rust #9</u>	1,052	1874	2.89
<u>Rust #7</u>	1,052	1753	3.24
<u>Rust #6</u>	1,028	1790	4.05
<u>C gcc #8</u>	8	1391	4.10
<u>C gcc #4</u>	1,304	1490	4.44

Nota: código em C otimizado com O3!

Fonte: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html>

Crashando um programa em C!

Hora do speedrun! Segue abaixo um tutorial de como crashar um programa em C em menos de 10 segundos:

```
int main(void) {  
    int* n = 0;  
    *n = 0;  
    return 0;  
}
```

Versão elegante em apenas uma linha:

```
int main(void) {  
    *((unsigned int*)0) = 0xDEAD;  
    return 0;  
}
```

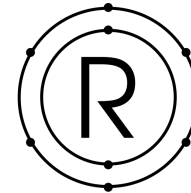
Por que não?

- Lidar com as imposições da linguagem pode ser mais difícil do que parece. Veremos mais sobre isso em seguida, assim como outros “por que não” e “por que não... ainda” na segunda parte do minicurso. [8]
- Apesar de possuir estruturas de dados muito robustas, não é orientada a objetos. Se isto é necessário no seu projeto, Rust não é uma opção recomendável.

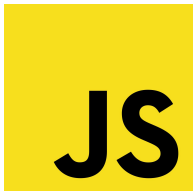
Onde ela está sendo utilizada?



No Discord, o serviço de “Read States” que verifica quais mensagens você leu na plataforma foi **migrado de Go para Rust**. [9]



RedoxOS é um **sistema operacional** de código aberto majoritariamente desenvolvido utilizando Rust.



Rust está sendo incorporado no ecossistema JavaScript, como **no framework Next.JS** e o **runtime Deno**. Além disso, é amplamente utilizado em **aplicações WebAssembly**.



Veloren é um jogo voxel multiplayer RPG escrito em Rust. Inspira-se em jogos como Cube World, Legend of Zelda: Breath of the Wild, Dwarf Fortress e Minecraft.



Sem mais enrolações...

Como?

...let's dive in!

Apresentando o ambiente...

Assim como o NPM de JavaScript, PIP de Python, GEM de Ruby e outros repositórios de pacotes, Rust possui o crates.io. Neste local, pode-se encontrar pacotes produzidos pela comunidade (com diversas licenças) para se utilizar em seu projeto. Dentre elas, está a **Diesel**, futuramente explorada.

The screenshot shows the crates.io website. At the top, there's a dark green header with the crates.io logo (a yellow cube) and the text "crates.io". To the right, there are links for "Browse All Crates" and "Log in with GitHub". Below the header, the main title "The Rust community's crate registry" is displayed in white. Underneath the title is a search bar with the placeholder text "Click or press 'S' to search...". Below the search bar, there are two yellow buttons: "Install Cargo" and "Getting Started".

Below the buttons, there's a section with text: "Instantly publish your crates and install them. Use the API to interact and find out more information about available crates. Become a contributor and enhance the site with your work." To the right of this text, there are two statistics: "21.999.654.176 Downloads" (with a download icon) and "94.468 Crates in stock" (with a crate icon).

Below the statistics, there are three columns of crates:

- New Crates**:
 - gametime v0.0.0
 - cardinal-transfer-authority v1.7.1
 - libray_nikolas_con v0.1.1
 - stb_rect_pack_sys v0.1.4
- Most Downloaded**:
 - rand
 - syn
 - rand_core
 - libc
- Just Updated**:
 - cosmian_cover_crypt v6.0.8
 - cargo-server-here v0.2.0
 - rlottie-sys v0.2.4
 - cw1155-base v0.16.0

Cargo — gerenciador de pacotes



Comandos básicos:

- Adicionar uma dependência ao projeto
`> cargo add crate_name`
- Compilar o código e gerar um executável
`> cargo build`
- Compilar o código e executar o binário
`> cargo run`

Por padrão, o **cargo** assume um ambiente de desenvolvimento, entregando baixos tempos de compilação e maior informação para debugging.

Com a flag `--release`, o código é mais otimizado para velocidade de execução e tamanho do arquivo, em troca de consumo de mais tempo para a compilação.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL GITLENS JUPYTER

```
basics on  master [?] is v0.1.0 via v1.64.0
• > cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
    Running `target\debug\basics.exe`
In a calm sea every man is a pilot.

basics on  master [?] is v0.1.0 via v1.64.0
• > █
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL GITLENS JUPYTER

```
basics on  master [?] is v0.1.0 via v1.64.0
• > cargo run --release
    Finished release [optimized] target(s) in 0.01s
    Running `target\release\basics.exe`
In a calm sea every man is a pilot.

basics on  master [?] is v0.1.0 via v1.64.0
• > █
```

rustc — Rust compiler

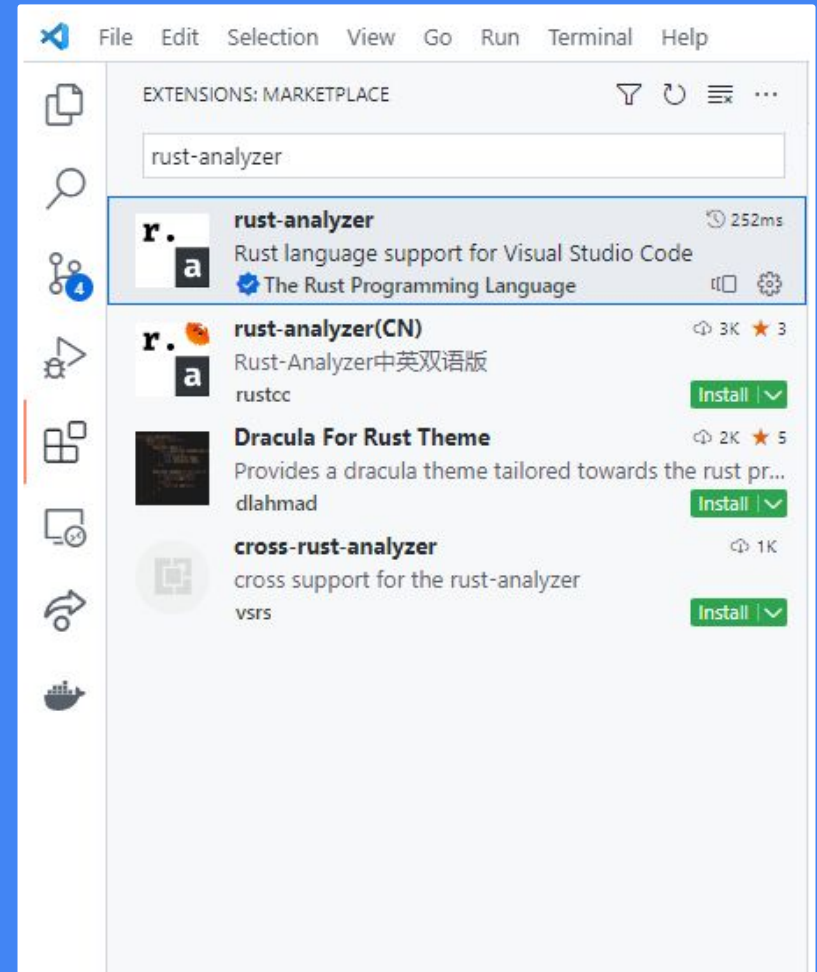
Outro diferencial em Rust que o destaca das demais linguagens é seu excelentíssimo compilador. Veja você mesmo:

```
secompp-2022 on  main [!] is  v0.1.0 via  v1.64.0
❌ > cargo run
   Compiling basics v0.1.0 (D:\dev\secompp-2022)
error[E0382]: borrow of moved value: `name`
  --> src\_10_borrowing.rs:49:11
   |
45 |     let name = String::from("Gustavo Becelli");
   |         ---- move occurs because `name` has type `String`, which does not implement the `Copy` trait
   |
...
48 |     goodbye(name);
   |         ---- value moved here
49 |     greet(&name); //Não funciona porque o nome foi movido
   |         ^^^^^ value borrowed here after move
```

For more information about this error, try `rustc --explain E0382`.
error: could not compile `basics` due to previous error

Extensão para o VS Code

rust-analyzer - Visual Studio Marketplace
Provém o LSP e outros utilitários.



Primeira Milestone

Hello World!

- Função main
- Comentários

Printando coisas...

- Macro `println!`
- Print formatado
- A forma com que uma estrutura de dados em Rust é formatada baseia-se na `impl Display` relativa a estrutura do módulo `fmt` da biblioteca padrão.
- Existem outros tipos de print

Tipos de dados — Primitivos

Existem um conjunto de primitivas escalares disponíveis:

- Inteiros com sinal `i8`, `i16`, `i32`, `i64`, `i128` e `isize*`;
- Inteiros SEM sinal `u8`, `u16`, `u32`, `u64`, `u128` e `usize*`;
- Pontos flutuantes `f32` e `f64`;
- Caracteres `char` Unicode, suportando valores como `'a'`, `'α'` e `'∞'` (4 bytes).
- Booleanos `true` e `false`.

Importante: ela não realiza conversão implícita de tipos primitivos, mas permite *casting* entre tipos compatíveis.

```
let decimal: f32 = 65.4321f32;
```

```
let integer: u8 = decimal as u8;
```

**`isize` e `usize` se referem
a tamanhos de ponteiro.*

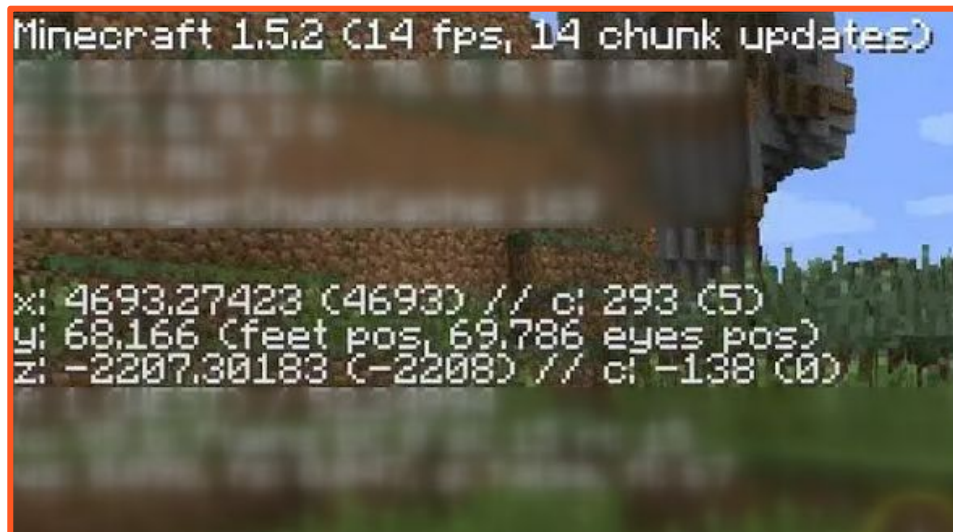
Tipos de dados — Compostos

Existem também os tipos compostos:

- Arrays, como `[T; size]`.
 - São coleções de objetos de um mesmo tipo `T` (genérico);
 - Possuem tamanho fixo sendo alocado na região da *stack* (mais rápida);
 - Mesmo declarando a variável como mutável, não se pode alterar o tamanho da estrutura.
- Vetores, como `Vec<T>`
 - São coleções assim como array;
 - Possuem tamanho dinâmico, alocados na *heap* (mais lenta).

Tipos de dados — Compostos

- Tuplas, como (T1, T2);
 - Coleção de valores de quaisquer tipos (inclusive, outra tupla), com qualquer quantidade de elementos;
 - Após definida, não pode alterar-se o tipo ou tamanho.
 - Úteis para representar alguns conceitos específicos, como as coordenadas de um ponto no espaço.
 - (x: i32, y: i32, z: i32, b: Block);



Tipos de dados — Slices e referências

- Fatias (slices) — `&[T]` ou `&str: &[i32]`
 - São similares a arrays, com quantidade de elementos finitos;
 - Tamanho desconhecido em tempo de compilação;
 - São compostas por um ponteiro a um dado e o tamanho da fatia;
 - strings literais são implementadas utilizando slices não mutáveis.
- Referências `&T`
 - São como ponteiros em C, referenciam um valor de uma variável;
 - Todas as fatias são referências, mas não o contrário.
 - Serão mais exploradas na seção do Borrow Checker.

Tipos de dados — Customizados

- Enumeráveis (`enum`)
 - Representam possibilidades de um conceito, por exemplo, as direções principais: esquerda, direita, frente e trás.
 - São amplamente utilizados para códigos de erros.
- Estruturas (`struct`)
 - Representam estruturas mais complexas ou objetos;
 - Possui atributos/campos;
 - Quando combinadas com (`traits`), podem ser usadas para construir estruturas análogas à classes.

Tipos de dados — Customizados

- Apelidos (aliases).
 - Definem um “nome” par uma composição de tipos. Por exemplo, uma cor em um pixel RGB.

```
type Rgba = (u8, u8, u8, u8);  
type Image = Vec<Vec<Rgba>>;  
type Point = (i32, i32);  
type Point3d = (i32, i32, i32);
```
 - Precisam ser UpperCamelCase, ou o compilador mostrará um warning (isso não vale para tipos primitivos: `usize`, `f32`, etc).

Variáveis e constantes

Variáveis

Instanciadas com `let`

```
1 fn main() {  
2     let name: String = String::from("Gustavo");  
3     println!("Seja bem-vindo, {}!", name);  
4 }
```

- São IMUTÁVEIS por padrão;
- Fortemente tipadas;
- Tipos podem omitidos;

Constantes

Instanciadas com `const`

```
1 fn main() {  
2     ...  
3     const PI: f32 = 3.141592;  
4     println!("O valor de PI: {}", PI);  
5 }
```

- São estáticas;
- Os tipos precisam ser especificados.

Escopo

Variáveis, além de armazenarem dados na stack, também possuem ‘recursos próprios’ na heap;

Para evitar vazamentos de memória, caso ela vá para fora de escopo, seus recursos são liberados;

Um escopo é simplesmente um bloco, com seu próprio contexto, representado por um par de chaves.

Blocos

Blocos definem novos escopos, como observamos no slide anterior.

Em Rust, eles também são considerados expressões, logo podem retornar algum valor.

Em todo bloco, uma expressão sem ; é considerado seu retorno.

Variáveis: mais conceitos

- Shadowing
- Declaração posterior
- Congelamento (freezing)

Operações com tipos

- Rust não tem conversão implícita
- Casting (conversão de tipos primitivos)
- Comportamento dos literals numéricos
- Inferência
- Conversão de tipos não primitivos (via `traits`)
- Conversão de tipos quaisquer para `String` é feito através do `impl Display`, já citado, de forma bem similar a linguagens POO [10]

Expressões

- Como em qualquer linguagem de programação
- Principal diferencial é que blocos também são expressões

Fluxo de controle

- Não usam parênteses no condicional, portanto o chaveamento é necessário
- São blocos
- **if**
 - Igual à maioria das outras linguagens de programação
- **loop**
 - Loop infinito! (sim, o Rust formaliza o loop infinito)
- **while**
- **for**
 - Iteradores
- **match**
 - Pode ser usado como um switch em C
 - Porém é mais voltado para pattern matching
- Existem usos mais complexos de fluxo de controle, como `if let` e `while let`

Funções

As funções são declaradas com `fn`, os parâmetros devem ser especificados e caso possuam retorno, devem ser especificados após uma seta `→`.

```
fn diff(a: i32, b: i32) → i32 {  
    return a - b;  
}
```

É importante saber que:

- Toda função inicia um novo escopo;
- O `return` pode ser omitido;
- Pode-se omitir o tipo de retorno caso não haja algum.

```
fn sum(a: i32, b: i32) → i32 {  
    ...  
    a + b  
}
```

Módulos

- Rust provê um sistema de módulos para favorecer reaproveitamento de código e a criação de bibliotecas.
- Visibilidade (tomar cuidado quando usar structs)
- Podem ser compilados em bibliotecas para virarem crates. Em formato de crates, podem ser distribuídos pelo cargo.
- Não vamos ensinar a criar bibliotecas, muito menos distribuí-las, mas vamos usar modularização na segunda parte do minicurso.

Segunda Milestone.



Atributos

- Aplicam metadados ao código, podendo especificar detalhes de compilação, de interpretação, marcar funções, etc.
- Podem ser relativos à crates ou à configuração do projeto, por exemplo, respectivamente: `#![crate_type = "lib"]` e `#[cfg(target_os = "linux")]`.
- Essencialmente comportam-se como decoradores de Python.
- Nas versões mais novas de Rust é possível criar algo como atributos personalizados com Custom Derive, mas é uma funcionalidade avançada que não detalharemos aqui.

“Eu realmente gosto de como o Rust não esconde a complexidade dos desenvolvedores por trás dos *trade-offs*, mas oferece ferramentas úteis para gerenciar a complexidade, bem como padrões sensatos.”



Duke Skookum

The Borrow Checker

O Borrow Checker é de longe a funcionalidade mais distinta e única de Rust.

Ele permite que Rust crie garantias de segurança de memória sem precisar de um Garbage Collector, ou alocar e desalocar memória manualmente.

Essa característica gera grande impacto de como o código é codificado, apesar de ele definir apenas três regras simples.

The Borrow Checker

1. Todo valor em Rust tem um ***dono***;
2. Só é possível existir um **dono** por vez;
3. Quando o **dono** deixa de existir (sai de escopo), o valor é descartado.

The Borrow Checker

1. Todo valor em Rust tem um dono;

Esta é a propriedade mais simples de se compreender. Para todo valor, existe um dono.

Contudo, sabemos que compartilhar é importante. Para fazer isso, a linguagem proporciona o recurso de referência ou ***borrowing***, que traduzido da gringa significa emprestar.

Para fazer isso, basta preceder uma variável por &.

```
let my_name: String = String::from("Gustavo Becelli");
```

```
let name_to_congratulate: &String = &my_name;
```

```
congratulate(name_to_congratulate);
```

The Borrow Checker

A propósito, vimos que strings literais são representados como um slice (referência) imutável `&str`.

Isso permite que o compilador saiba o tamanho da string no momento da compilação, sem precisar alocar em execução.

`String`, por outro lado, é uma implementação de string que é “dona” do que armazena.

Desta forma, consegue ser definida em tempo de execução, por exemplo, um input variável de um usuário, e ser **mutável**.

The Borrow Checker

Mas o que de fato é *emprestar* na linguagem?

É simplesmente permitir que **outra variável** utilize seu valor, contudo, sem se tornar **dona** dele. Por padrão, os empréstimos são de forma imutável, requerendo a keyword `&mut` para ser mutável.

- Um fator importante a se considerar:

Caso uma variável dona armazene um valor imutável, o ato de criar uma referência mutável é proibida pelo compilador.

Além disso, só é possível existir uma referência mutável para um valor. Por que senão, como vou saber quem quebrou meu PlayStation?



The Borrow Checker

2. Só é possível existir um dono por vez;

Diferentemente de linguagens que suportam ponteiros, um valor não pode possuir mais que um dono por vez.

O exemplo mais comum de para entender isso, são funções:

```
let name: String = String::from("Gustavo Becelli");  
greet(&name);  
greet(&name);  
goodbye(name);  
// greet(&name); Não funciona porque o nome foi movido  
// e não existe mais
```

```
fn greet(name: &String) {  
    println!("Hello, {}!", name);  
}
```

```
fn goodbye(name: String) {  
    println!("Goodbye, {}!", name);  
}
```

The Borrow Checker

Na maioria das linguagens, o código da direita funcionaria sem problemas, pois criariam uma cópia do valor ou apenas o utilizava não seguramente como referência.

Neste caso, “name” da função *goodbye* se torna a nova dona do valor passado como parâmetro.

```
let name: String = String::from("Gustavo Becelli");  
greet(&name);  
greet(&name);  
goodbye(name);  
// greet(&name); Não funciona porque o nome foi movido  
// e não existe mais
```

```
fn greet(name: &String) {  
    println!("Hello, {}!", name);  
}
```

```
fn goodbye(name: String) {  
    println!("Goodbye, {}!", name);  
}
```

The Borrow Checker

3. Quando o dono deixa de existir (sai de escopo), o valor é descartado.

Da mesma forma que a primeira, a regra é simples! Saiu de escopo, perdeu! Assumindo o exemplo anterior, quando o `name` é passado para a `goodbye`, `name` deixa de ser válido no escopo maior.

Caso a variável não retorne (como na `goodbye`), seu valor é perdido ao finalizar a função.

```
let name: String = String::from("Gustavo Becelli");
greet(&name);
greet(&name);
goodbye(name);
// greet(&name); Não funciona porque o nome foi movido
// e não existe mais
```

```
fn greet(name: &String) {
    println!("Hello, {}!", name);
}
```

```
fn goodbye(name: String) {
    println!("Goodbye, {}!", name);
}
```

Lifetimes

Esta é mais uma funcionalidade que a diferencia de outras linguagens. O uso de Lifetimes está diretamente ligado a referências (*borrow*s), escopos, e desambiguação.

Para que o programa compile, às vezes é necessário anotar o código com lifetimes (tempos de vida). Essas anotações são apóstrofos seguidos de uma string, geralmente caracteres do alfabeto.

```
fn longest<'a>(x: &'a str, y: &'a str) → &'a str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}
```


Tratamento de Erros

Qualquer programa que realiza IO (entrada/saída) de dados, seja obtendo um input de um usuário, lendo ou escrevendo em um banco de dados, ou realizando uma conversão de tipos, arrisca falhar em alguma destas operações.

Por exemplo, meu programa pode requisitar um usuário no banco de dados e ele retornar nulo (ou nenhum resultado).

Tratamento de Erros

Em uma linguagem convencional, C, Python, JavaScript, entre outras, normalmente você chama uma função que te diz que irá retornar o valor desejado e, se você for um programador minimamente experiente, checará se o valor é nulo.

```
def print_user_name(user_id):  
    user: User = get_user(user_id)  
    if user: # Se o usuário existir...  
        print(user.name)  
  
def get_user(conn, user_id) → User:  
    c = conn.cursor()  
    c.execute(f"SELECT * FROM users WHERE id = {user_id}")  
    user: User = c.fetchone()  
    return user
```

Exemplo de Python

Traits

- Não é POO!
- Praticamente idênticas às interfaces de POO
- Mas não é POO!
- São abstratas
- Uma coleção de métodos definidos para um tipo qualquer.
- Os métodos definidos podem ser “virtuais” (não é POO!), ou podem ser implementados. Independentemente, podem ser sobrescritos com o `impl`.
- Quando combinados com `structs`, essencialmente constroem classes.
mas não é POO
- Para usos avançados, verifiquem a documentação ou o Rust by Example

Unsafe

- Não use
- Use C
- Ou use, sei lá, seja feliz.

```
fn main() {  
    let i: u64 = 3;  
    let o: u64;  
    unsafe {  
        asm!(  
            "mov {0}, {1}",  
            "add {0}, 5",  
            out(reg) o,  
            in(reg) i,  
        );  
    }  
    assert_eq!(o, 8);  
}
```

Então não existem comportamentos inseguros na linguagem?

Não é bem assim...

Apesar de assegurar a segurança de memória em diversos casos, ela não considera inseguro:

- **Deadlocks;**
- **Condições de corrida;**
- Vazamento de memória;
- Falhar em chamar destrutores;
- **Overflow de inteiros;**
- Abortar o programa;

Recomendações para estudo

Leituras

- [The Rust Programming Language Book](#) — livro e documentação oficial da linguagem, escrito em linguagem fácil, com diversos exemplos e explicações.
- [Rust Cookbook](#) — “livro de receitas” que apresenta boas práticas em tarefas comuns de programação, utilizando os crates do ecossistema Rust.

Tutoriais:

- [Rustlings](#) — pequenos exercícios para acostumar a ler e escrever códigos Rust.

Vídeos

- [Let's Get Rusty](#) — canal dedicado exclusivamente a Rust, cobrindo as atualizações, tutoriais, dicas, entre outros.
- [NoBoilerplate](#) — pequenos vídeos técnicos, apresentando conceitos e exemplos



Referências

1. Rust. Frequently Asked Questions.
2. InfoQ. Interview on Rust, a Systems Programming Language Developed by Mozilla.
3. Rust Blog. Announcing Rust 1.0.
4. StackOverflow. What is Rust and why is it so popular?
5. The Rust Programming Language. Data Types.
6. Microsoft. We need a safer systems programming language.
7. Microsoft. Why Rust for safe systems programming.
8. GitLab. A guide to the Rust programming language.
9. Discord. Why Discord is switching from Go to Rust.
10. Rust. To and From Strings.

Material baseado em:

1. Rust by Example
2. The Rust Programming Language Book