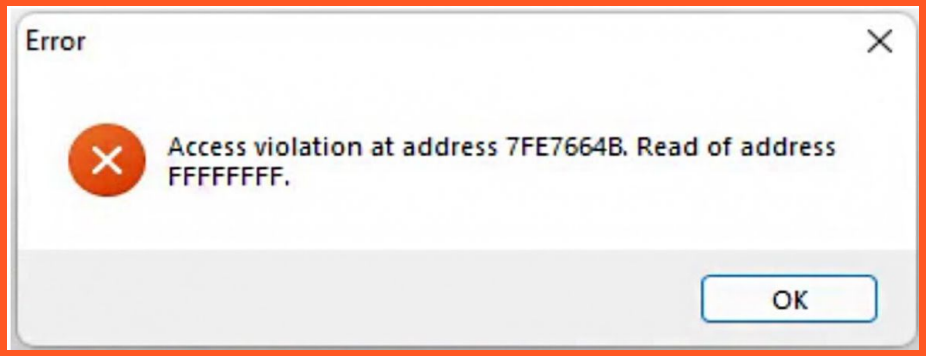
Tem PESADELOS com isso?



Conheça já: Rust!

o mais novo produto das

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

Introdução ao Rust 🦀

Efficiently safe.

v2023 — revisada e aprimorada

Daniel Serezane
@salies

Gustavo Becelli
@becelli

Estrutura do minicurso

Salve Madias ✈️

Dia 1

- O que? (apresentação inicial)
- Por que? (*C++ sucks*)
- Como? (*Rust sucks*)
- + exercícios práticos: sem *live coding*
- + exercício teórico

Dia 2

- Onde? (*MANGA sucks*)
- Quando? (você não é a Google)
- + exemplos práticos “reais”
- + exercício “mão na massa”

Estrutura do minicurso

Salve Madias ✈️

Dia 1

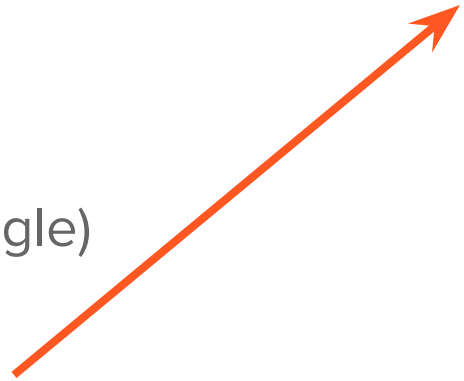
- O que? (apresentação inicial)
- Por que? (*C++ sucks*)
- Como? (*Rust sucks*)
- + exercícios práticos: sem *live coding*
- + exercício teórico

Dia 2

- Onde? (*MANGA sucks*)
- Quando? (você não é a Google)
- + exemplos práticos “reais”
- + exercício “mão na massa”

Odiamos curso de programação.

Chato pra 🌳, improdutivo, além de doer o pescoço! (*Lab 6 sucks*)



Parte 1

Rust: o que, por que e como

O que?

- Linguagem de programação **compilada**, **imperativa** e **estruturada**, criada em 2006 por um funcionário da Mozilla. O projeto foi adotado pela empresa em 2009 e 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 de memória**, contudo também visa ser o mais performática possível. [2]
- É atualmente a linguagem mais adorada pelos programadores (têm sido por 8 anos seguidos). [4]

Um pouco mais de história... ⏳

- Em 2020, com os impactos econômicos da pandemia, ¼ 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.
- Em abril de 2023, a Rust Foundation anunciou mudanças polêmicas em sua marca, causando controvérsias na comunidade e possivelmente impactando a adoção da linguagem.

Por que?

C++ sucks

- 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. **Executáveis mais leves e *memory footprint* baixíssimo.**
- Possui um ecossistema oficial bem integrado e que impõe um certo nível de organização ao projeto, mas fácil de usar. Além disso, possui uma sintaxe modernizada, pouco verbosa, e com atalhos para tarefas frequentes (e.g. for loop)

Por que?

C++ sucks

- 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. **Executáveis mais leves e *memory footprint* baixíssimo.**
- Possui um ecossistema oficial bem integrado e que impõe um certo nível de organização ao projeto, mas fácil de usar. Além disso, possui uma sintaxe modernizada, pouco verbosa, e com atalhos para tarefas frequentes (e.g. for loop)

Principal motivo da popularidade em empresas.

Por que?

C++ sucks

- 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. **Executáveis mais leves e *memory footprint* baixíssimo.**
- Possui um ecossistema oficial bem integrado e que impõe um certo nível de organização ao projeto, mas fácil de usar. Além disso, possui uma sintaxe modernizada, pouco verbosa, e com atalhos para tarefas frequentes (e.g. for loop)

Go algumas das mesmas qualidades de Rust: é uma linguagem compilada, bastante eficiente, e que traz consigo um ambiente moderno.

Por que?

C++ sucks

- 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. **Executáveis mais leves e *memory footprint* baixíssimo.**
- Possui um ecossistema oficial bem integrado e que impõe um certo nível de organização ao projeto, mas fácil de usar. Além disso, possui uma sintaxe modernizada, pouco verbosa, e com atalhos para tarefas frequentes (e.g. for loop)

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

Por que?

C++ sucks

- 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. **Executáveis mais leves e *memory footprint* baixíssimo.**
- Possui um ecossistema oficial bem integrado e que impõe um certo nível de organização ao projeto, mas fácil de usar. Além disso, possui uma sintaxe modernizada, pouco verbosa, e com atalhos para tarefas frequentes (e.g. for loop)
- Por esses motivos, conquistou uma comunidade muito dedicada, mesmo que não tão grande quanto a de Python, por exemplo.

Por que?

C++ sucks

n-body

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

Nota: código em C otimizado com O3!

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

Por que?

C++ sucks

usuário casual de Rust

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:

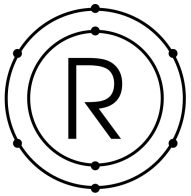
```
int main(void) {
    *((unsigned int*)0) = 0×BEEF;
    return 0;
}
```

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.
- Em suma, Rust é uma linguagem específica criada para solucionar um problema específico. **Se você não tem esse problema, é melhor não usar.**

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.

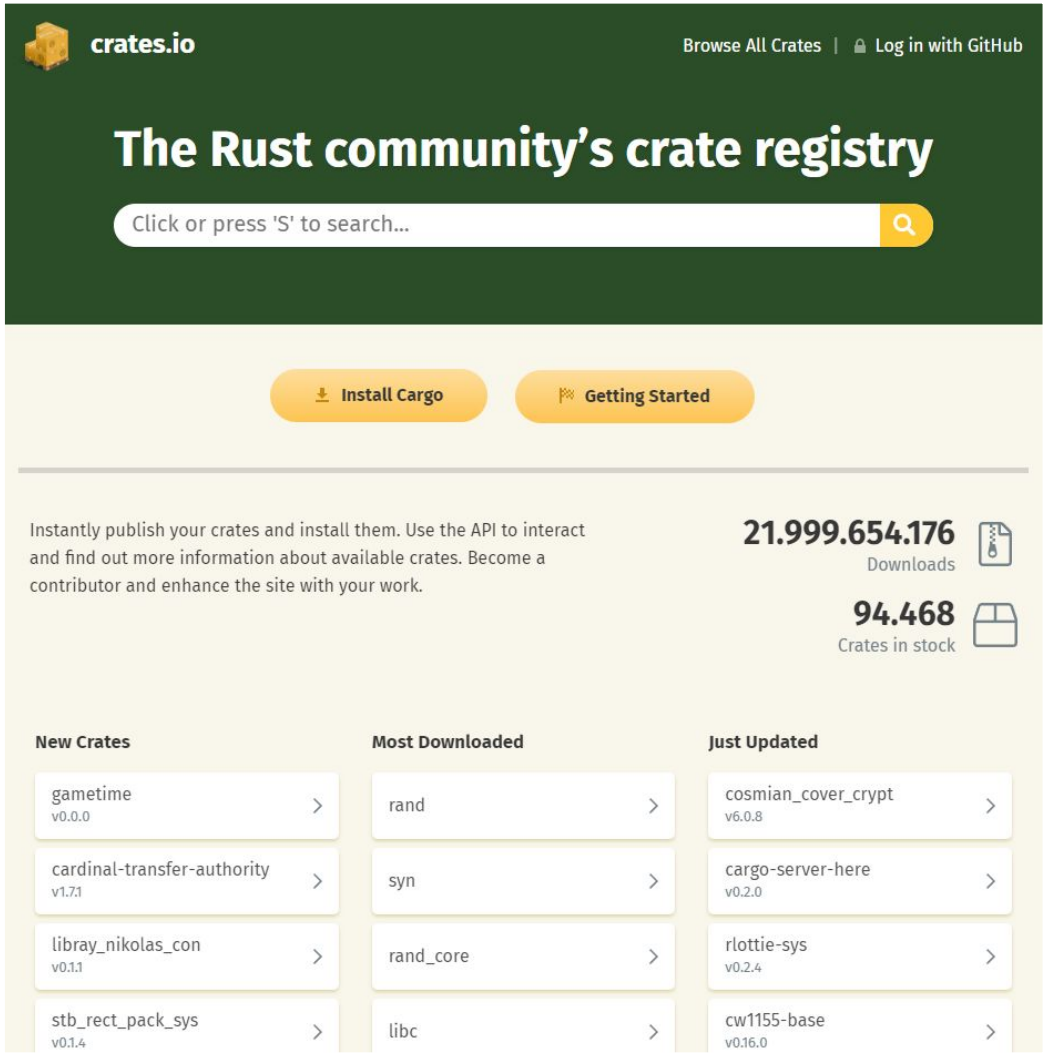
Veloren é um jogo voxel multiplayer RPG escrito em Rust. Inspira-se em jogos como Cube World, Legend of Zelda: Breath of the Wild, Dwarf Fortress e Minecraft.

Como?

Rust sucks

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.



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:

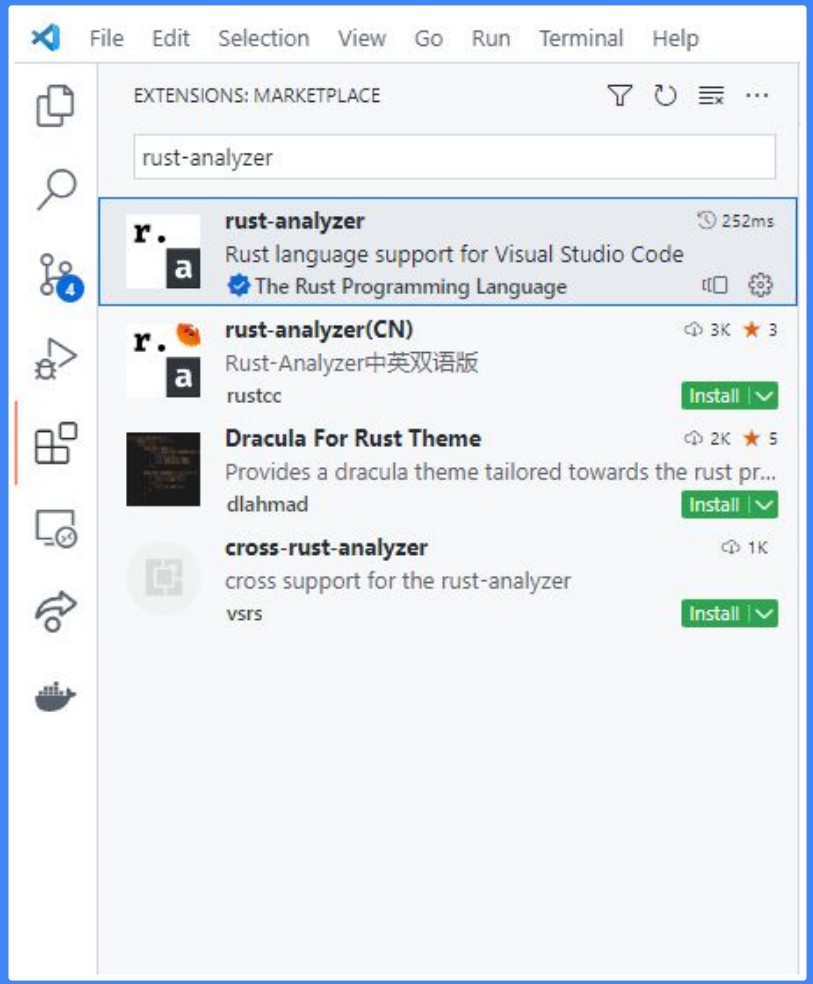
```
secompp-2022 on  main [!] is  v0.1.0 via  v1.64.0
❯ cargo run
   Compiling basics v0.1.0 (D:\dev\secompp-2022)
error[E0382]: borrow of moved value: `name`
  --> src\_10_borrowing.rs:49:11
   |
45 |     let name = String::from("Gustavo Becelli");
   |         ---- move occurs because `name` has type `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
```

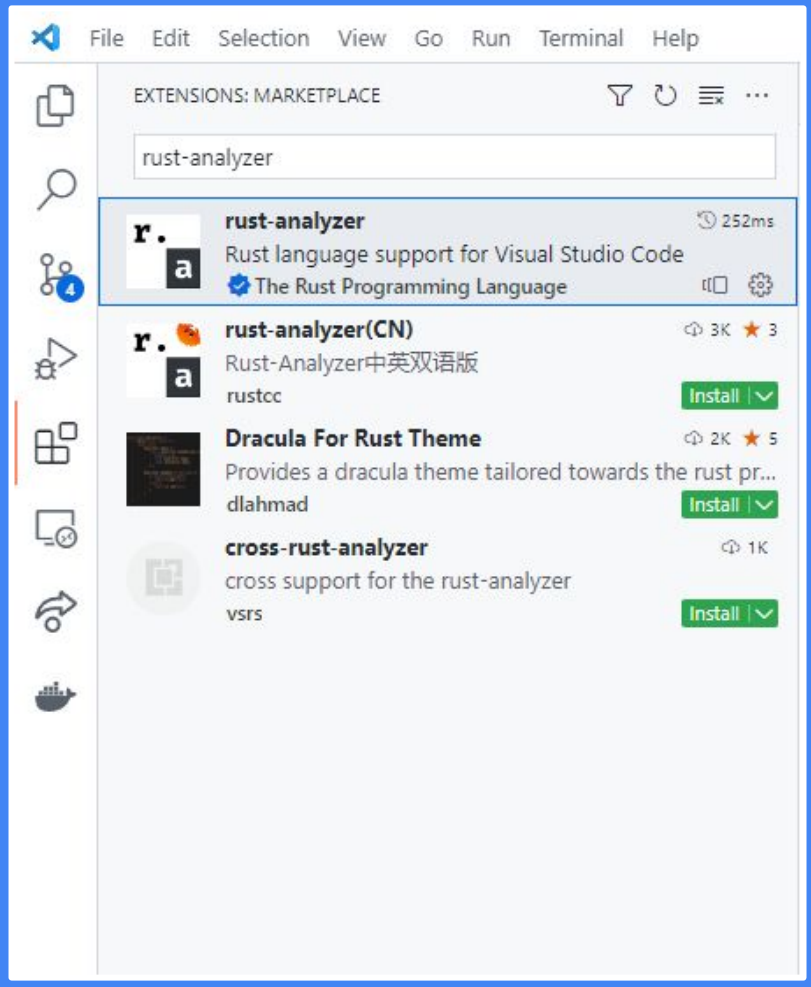
Extensão para o VS Code

rust-analyzer - Visual Studio Marketplace

Provém o LSP e outros utilitários.



~~Extensão para o VS Code~~



Primeira Milestone

Hello, world!

- Função main
- Comentários

```rust
fn main() {
    // Salve salve família
    println!("Hello, world!");
}
```

```
Hello, world!
```

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 que você espera vindo de outras linguagens:

- Caracteres `char` Unicode, suportando valores como 'a', 'α' e '∞' (4 bytes).
- Booleanos `true` e `false`.
- 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`;

```
let decimal: f32 = 65.4321f32;

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

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

Tipos de dados — Primitivos

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

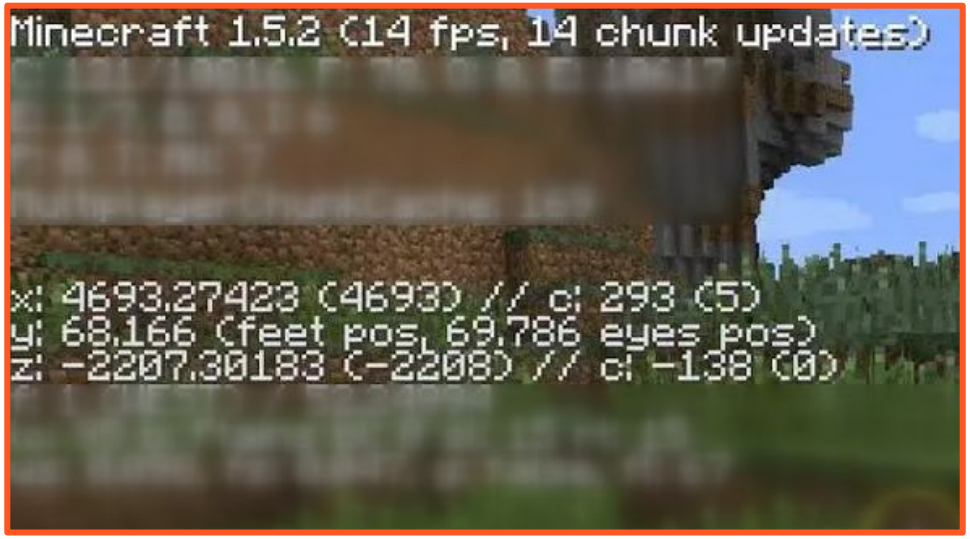
Tipos de dados — Compostos

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

- São estáticas;
- Os tipos precisam ser especificados.

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**
  - **Muito poderoso 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:

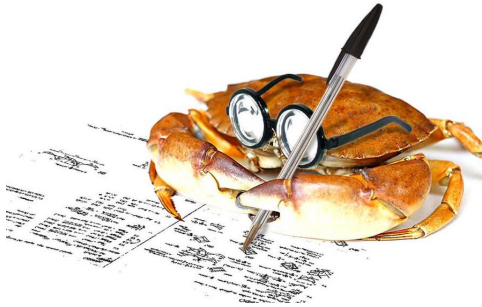
- Toda função inicia um novo escopo;
- O `return` pode ser omitido;
- Pode-se omitir o tipo de retorno caso não haja algum.

```
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.

Hora do exercício!



Copie e complemente os comentários do código abaixo.

Rode no **Rust Playground**.

```rust
fn main() {
    // insira seu comentário aqui
    let msg = "oi";
    println("{}", msg);

    // sinta-se livre para testar outros valores aqui!
    // o que será que acontece?
    let number = 42;

    // o que significa essa atribuição?
    msg = match number {
        666 ⇒ "Então você é do rock? 😈 🤘",
        42 ⇒ "É a resposta.",
        333 ⇒ "Meio besta.",
        // o que a expressão abaixo captura?
        5000..=5999 ⇒ "Você sabia que existem mais de 5,000 tipos diferentes de batatas?!",
        // e essa?
        621 | 177013 ⇒ "Sai de perto de mim.",
        _ ⇒ "Sem graça."
    };

    // qual o valor de msg aqui?
    println!(msg);
}
```

Hora do exercício!

Copie e complemente os comentários do código abaixo.

Rode no **Rust Playground**.

```rust
fn main() {
    // insira seu comentário aqui
    let mut msg = "oi";
    println!("{}", msg);

    // sinta-se livre para testar outros valores aqui!
    // o que será que acontece?
    let number = 42;

    // o que significa essa atribuição?
    msg = match number {
        666 => "Então você é do rock? 😈 🤘",
        42 => "É a resposta.",
        333 => "Meio besta.",
        // o que a expressão abaixo captura?
        5000..=5999 => "Você sabia que existem mais de 5,000 tipos diferentes de batatas?!",
        // e essa?
        621 | 177013 => "Sai de perto de mim.",
        _ => "Sem graça."
    };

    // qual o valor de msg aqui?
    println!("{}", msg);
}
```

Segunda Milestone.

Atributos

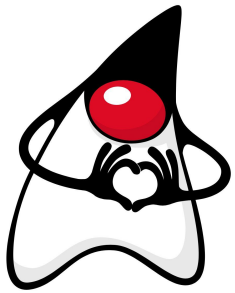
- Aplicam metadados ao código, podendo especificar detalhes de compilação, de interpretação, marcar funções, etc.
- Podem ser relativos à crates ou à configuração do projeto, por exemplo, respectivamente: `#![crate_type = "lib"]` e `#[cfg(target_os = "linux")]`.
- Essencialmente comportam-se como decoradores 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

“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`.

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

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 movidc
// 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, chamar **“greet”** após **“goodbye”** funcionaria sem problemas, pois criariam uma cópia do valor ou apenas o utilizava **inseguramente** como referência.
Em Rust, “name” na função *goodbye* se torna a nova dona do valor passado como parâmetro.

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

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

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

```rust
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
```

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

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

The Borrow Checker

Saiu de escopo, perdeu!
Contudo, essa regra não vale para tipos primitivos/pouco estruturados, onde ocorre passagem por valor.

Lifetimes

Esta é mais uma funcionalidade que a diferencia de outras linguagens. O uso de Lifetimes está diretamente ligado a referências (*borrows*), 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). Outro exemplo seria uma requisição HTTP, a qual pode não completar (sem rede), ser recusada devido à falta de autenticação, entre outros.

Tratamento de Erros

Pergunta: onde o programa abaixo pode falhar?

```
async function fetchUser(url) {
    const response = await fetch(url);
    return response.json();
}

async function main() {
    const user = await fetchUser("https://api.github.com/users/becelli");
    console.log(`Username: ${user.login}`);
    if (user.bio) console.log(`Bio: ${user.bio}`);
}
```

Tratamento de Erros

Pergunta: onde o programa abaixo pode falhar?

```
async function fetchUser(url) {
    const response = await fetch(url); // Falha de rede, Status 400, 500, etc…
    return response.json(); // Falha de parse, JSON inválido, etc ...
}

async function main() {
    const user = await fetchUser("https://api.github.com/users/becelli");
    console.log(`Username: ${user.login}`);
    if (user.bio) console.log(`Bio: ${user.bio}`);
}
```

Tratamento de Erros

Em Rust, o tratamento de erros deve ser explícito

```rust
struct User { login: String, bio: Option<String> }

async fn fetch_user(url: &str) -> Result<User, Box<dyn Error>> {
    let response = reqwest::get(url).await?;
    Ok(response.json().await?)
}

#[tokio::main]
async fn main() {
    match fetch_user("https://api.github.com/users/becelli").await {
        Ok(user) => {
            println!("Username: {}", user.login);
            if user.bio.is_some() {
                println!("Bio: {}", user.bio.unwrap());
            }
        }
        Err(err) => eprintln!("Error fetching user: {}", err),
    }
}
```

Traits

- Permitem a implementação de métodos que operam sobre tipos definidos via `structs`.
- Traits podem apenas provem a assinatura dos métodos, como podem também prover uma implementação padrão, que pode ser sobrescrita. Sim, similarmente à classes abstratas.
- A implementação se dá através do `impl`, que também pode ser chamado isoladamente, para associar métodos a uma `struct`.
- Flexibilidade: é possível usar um mesmo `trait` com diferentes `structs`, e vice-versa.
- Por esse e outros motivos (representação dos dados, por exemplo), **apesar de assemelhar-se a POO, não se trata de uma aplicação deste conceito.**

Unsafe

- Em Rust, o bloco unsafe é utilizado para executar código que “**desativa as verificações de segurança”** (não é exatamente desativar) do compilador, permitindo operações de baixo nível e otimizações que, de outra forma, seriam proibidas pela linguagem, como acessar ponteiros brutos e modificar dados imutáveis.

```
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);
}
```

Unsafe

- Também conhecido como modo f0d@-s3.
- Não use.
- Use C.
- Tá afim de fazer cag@**? Use JavaScript.

```
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 na ampla maioria dos casos, ela não considera *“inseguro”*:

- **Deadlocks**;
- **Condições de corrida**;
- Vazamento de memória;
- Falhar em chamar destrutores;
- **Overflow de inteiros**;
- **Out of bounds;**
- 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.
- **Rust by Example – excelente guia aos principais diferenciais da linguagem 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

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.
- **Rust by Example – excelente guia aos principais diferenciais da linguagem Rust**

Tutoriais:

- **Rustlings — pequenos exercícios para acostumar a ler e escrever códigos Rust.**

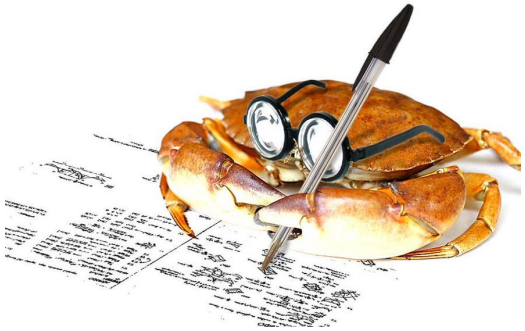
não vamos ficar fazendo exercício aqui, então é uma boa forma de praticar em casa!

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

Hora do exercício!

https://forms.gle/H1RsoUKjR3gzaiQk6



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