Rusting with style - Curso básico de linguagem Rust

Cleuton Sampaio

Veja no GitHub

Menu do curso

[**VÍDEO DESTA AULA**]

Variáveis

Declaração versus inicialização

Podemos declara uma variável especificando apenas o seu tipo de dados (o seu domínio), sem inicializá-la:

```
let a: i32;
```

Porém, se tentarmos utilizá-la sem termos atribuído um valor, tomaremos um erro:

```
fn main() {
    let a: i32; // Declaração de variável sem inicialização
    println!("a = {}", a); // Erro: variável não inicializada
}
```

Se inicializarmos antes de tentar obter seu valor, o erro desaparece:

```
    let a: i32; // Declaração de variável sem inicialização
    a = 5; // Inicialização da variável
    println!("a = {}", a); // Ok
```

Podemos declarar uma variável sem declarar seu tipo, que é inferido a partir da inicialização:

```
let a = 5;
```

Qual seria o tipo dessa variável?

```
fn mostrar_tipo<T>(_: &T) {
    println!("{}", std::any::type_name::<T>());
}

fn main() {
    let a = 5;
    mostrar_tipo(&a);
}
```

Seria `i32`: Inteiro sinalizado de 32 bits. Para mostrar o tipo de uma variável é preciso criar uma função template como essa `mostrar_tipo()`.

Mutável ou não

Essa variável é imutável:

```
    let a: i32; // Declaração de variável sem inicialização
    a = 5; // Inicialização da variável
    println!("a = {}", a); // Ok
```

Depois de atribuído o primeiro valor a ela, seu conteúdo não pode ser alterado.

```
fn main() {
    let a: i32; // Declaração de variável sem inicialização
    a = 5; // Inicialização da variável
    println!("a = {}", a); // Ok
    a = 1; // Erro
}
```

Se uma variável é mutável, seu conteúdo pode ser alterado no escopo onde foi declarada. Mas e se quisermos alterar em uma `função`?

```
fn nao_altera(x: i32) {
    println!("x = {}", x);
}

fn altera(x: i32) {
    x += 1;
}
```

```
fn main() {
    let mut a = 5;
    nao_altera(a);
    altera(a);
}
```

Vai dar erro! Apesar da variável ser mutável, a função `altera()` a recebe por valor. Para podermos alterar, ela teria que receber uma referência mutável da variável:

```
fn nao_altera(x: i32) {
    println!("x = {}", x);
}

fn altera(x: &mut i32) {
    *x += 1;
}

fn main() {
    let mut a = 5;
    nao_altera(a);
    altera(&mut a);
    nao_altera(a);
}
```

Quando passamos uma referência estamos "emprestando" a variável (no jargão do Rust). Guarde isso para mais tarde.

Vec versus vetores traditionais

Vamos declarar um vetor de inteiros com 5 elementos:

```
fn main() {
    let v: [i32; 5] = [200, 300, 400, 500, 600];
    for x in 0..v.len() {
        println!("O valor da posição {} é {}", x, v[x]);
    }
}
```

Declaramos um vetor `v` de inteiros (`i32`), com 5 posições e o inicializamos. Depois, um loop `for` nos permitiu pegar cada elemento do vetor.

Poderia ser assim também, caso não desejássemos obter o valor do índice:

```
    for valor in &v {
        println!("O valor é {}", valor);
    }
```

Por que esse `&` no nome do vetor? Porque eu quero pegar "emprestado" seus valores, portanto, preciso oficializar isso. É uma das diferenças do Rust para outras linguagens. Daria para fazer sem isso se não tentássemos associar uma posição de `v` à variável `valor`, como fizemos no primeiro `for`.

Mais variáveis e observações

Em Rust um **String** pode ser modificado, ao contrário de outras linguagens, como **Java**. Vejamos alguns tipos de variáveis neste outro exemplo:

```
fn main() {
    // Variáveis de tipos inteiros
    let inteiro: i32 = 42;
    let pequeno_inteiro: u8 = 255; // Unsigned 8-bit integer

    // Variáveis de ponto flutuante
    let flutuante: f64 = 3.1415;
    let flutuante_menor: f32 = 2.718;

    // Variável booleana
    let verdadeiro: bool = true;
    let falso: bool = false;

    // Variáveis de string
    let texto: &str = "Olá, mundo!"; // String slice
    let mut string: String = String::from("Rust é incrível!");

    // Exibindo os valores
    println!("Número inteiro: {}", inteiro);
    println!("Pequeno inteiro (u8): {}", pequeno_inteiro);
    println!("Número de ponto flutuante (f64): {}", flutuante);
    println!("Número de ponto flutuante menor (f32): {}", flutuante_menor);
    println!("Valor booleano verdadeiro: {}", verdadeiro);
    println!("Valor booleano falso: {}", falso);
    println!("Texto: {}", texto);
    println!("String mutável: {}", string);

    // Modificando a string mutável
    string.push_str(" Vamos aprender Rust!");
    println!("String após modificação: {}", string);
}
```