Conceitos comuns de programação

Diogo Silveira Mendonça

Slides Baseados no Capítulo 3 do Rust Book
https://doc.rust-lang.org/book/ch03-00-common-programming-concepts.html

Introdução

- Serão abordados alguns conceitos comuns de programação e alguns conceitos do Rust.
  - Variáveis e Mutabilidade
  - Constantes
  - Sombreamento
  - Tipos de dados
  - Funções
  - Comentários
  - Controle de Fluxo

- Todos os exemplos podem ser encontrados nesse link.

Variáveis e Mutabilidade

- let x = 5; Definição de uma **variável imutável**.
- x = 6; Tentativa de mudar o valor da **variável imutável**.

```
fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

```
$ cargo run
   Compiling variables v0.1.0
(file:///projects/variables)
error[E0384]: cannot assign twice to immutable
variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding
mutable: `mut x`
3 |     println!("The value of x is: {x}");
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable
variable

For more information about this error, try `rustc
--explain E0384`.
error: could not compile `variables` due to
previous error
```

Variáveis e Mutabilidade

- `let mut x = 5;` Definição de uma **variável mutável**.
- `x = 6;` Mudança do valor da **variável mutável**.

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

```
$ cargo run
   Compiling variables v0.1.0
(file:///projects/variables)
    Finished dev [unoptimized + debuginfo]
target(s) in 0.30s
    Running `target/debug/variables`
The value of x is: 5
The value of x is: 6
```

Constantes

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

- O exemplo mostra uma constante com a quantidade de segundos em 3 horas.
- Constantes são imutáveis.
- É declarado utilizando const.
- Como boa prática, constantes são nomeadas com letras maiúsculas e utilizando “_” para separar.
- Devem possuir o tipo de dado (os tipos de dados serão vistos mais para frente), no caso do exemplo: u32.

Sombreamento

- Permite utilizar o nome de variáveis já definidas, modificar o valor e o tipo de variáveis imutáveis.
- O primeiro let cria x com o valor 5.
- Então x é sombreado por let x, agora o valor de x é 6.
- O terceiro let está dentro de chaves, criando um sombreamento interno, onde x terá o valor de 12.
- Ao sair da chaves, o sombreamento interno termina, então o x fica com o valor 6 novamente.

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner
scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

Sombreamento

```
$ cargo run
   Compiling variables v0.1.0
(file:///projects/variables)
       Finished dev [unoptimized + debuginfo]
target(s) in 0.31s
       Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner
scope is: {x}");
    }

    println!("The value of x is: {x}");
}
```

Sombreamento

- O primeiro exemplo, mostra a utilização do sombreamento para armazenar a quantidade de espaços.

```
let spaces = "   ";
let spaces = spaces.len();
```

- O segundo exemplo, mostra a utilização de uma variável mutável, para tentar armazenar a quantidade de espaços. Esse exemplo não compila, pois spaces armazena uma string e não um número.

```
let mut spaces = "   ";
spaces = spaces.len();
```

Tipos de Dados

- No capítulo 2 adicionamos a linha de código
  `let guess: u32 = "42".parse().expect("Not a number!");`
- Se retirarmos o tipo de dado u32, iremos receber o erro:

```
$ cargo build
   Compiling no_type_annotations v0.1.0 (file:///projects/no_type_annotations)
error[E0282]: type annotations needed
 --> src/main.rs:2:9
  |
2 |     let guess = "42".parse().expect("Not a number!");
  |         ^^^^^
  |
help: consider giving `guess` an explicit type
  |
2 |     let guess: _ = "42".parse().expect("Not a number!");
  |              +++

For more information about this error, try `rustc --explain E0282`.
error: could not compile `no_type_annotations` due to previous error
```

Tipos de dados

- Tipos Escalares
  - Inteiros
  - Ponto-flutuantes
  - Booleanos
  - Carácter
- Tipos Compostos
  - Tupla
  - Array

Inteiros

- São números inteiros.
- Podem sofrer overflow.
- Podem ser signed ou unsigned.
  - Signed, aceita números negativos, tem o tamanho -(2^(n-1)) a 2^(n-1)-1.
    Ex: i8 = -128 a 127
  - Unsigned, aceita somente números positivos, tem o tamanho 0 a 2^(n)-1.
    Ex: i8 = 0 a 255

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

- O valor de arch depende da arquitetura do Sistema operacional.
  - Se a arquitetura for de 32-bit, então arch = 32-bit
  - Se a arquitetura for de 64-bit, então arch = 64-bit

Inteiros

- Podemos escrever números literais, como exemplificado na tabela ao lado.
  - Decimal = 98222
  - Hex = 255
  - Octal = 63
  - Binary = 240
  - Byte = 65

Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

- Podemos utilizar “_” para separar, para facilitar a leitura.
  Exemplo: 1_000 = 1000

Ponto-flutuantes

- Existem dois tipos:
  - f32 com tamanho de 32-bit e precisão de um ponto flutuante.
  - f64 com tamanho de 64-bit e precisão de dois pontos flutuantes.

- Por default o é f64.

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

Operações Numéricas

- Operação de adição
- Operação de diferença
- Operação de multiplicação
- Operação de divisão
- Operação de resto

```
fn main() {
    // addition
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;
    let truncated = -5 / 3; // Results in -1

    // remainder
    let remainder = 43 % 5;
}
```

Booleanos

- Assim como nas outras linguagens, o boolean pode ser verdadeiro ou falso.

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

Carácter

- Rust char tem um tamanho de 4 bytes, e representa um Unicode Scalar Value.
- Pode representar ASCII, letras acentuadas, letras chinesas, japonesas, koreanas, emojis entre outros.
- Unicode Scalar Values armazena U+0000 a U+D7FF e U+E000 a U+10FFFF.

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😻';
}
```

Tupla

- Tupla forma um único componente com diversos valores.
- Tamanho fixo, Tuplas não podem aumentar nem diminuir de tamanho.
- Podemos definir os tipos, para cada posição da Tupla.

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

Tupla

- Podemos decompor uma tupla, como no exemplo ao lado.
  - let tup = (500, 6.4, 1); define a tupla.
  - let (x, y, z) = tup; atribui os valores 500 para x, 6.4 para y e 1 para z.

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

Tupla

- Podemos acessar uma tupla diretamente, utilizando x.posição.
  - x.0 recebe o valor 500
  - x.1 recebe o valor 6.4
  - x.2 recebe o valor 1

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Tupla

- Uma tupla sem nenhum valor é nomeado de unit.
- O valor e o tipo são escritos como “()”
- Representam um valor vazio ou um tipo de retorno vazio.
- Expressões implicitamente retornam o valor 'unit' se elas não retornam nenhum outro valor.

Array

- Diferentemente das outras linguagens, em Rust o Array tem o tamanho fixo.
- Diferentemente da Tupla, o Array possui somente um tipo de dado.
- Podemos definir o Array, declarando o tipo e o tamanho explicitamente.

Exemplo 1:

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

Exemplo2:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

Array

- Podemos acessar o valor dentro de um array, como no exemplo ao lado.

- Se tentarmos acessar um valor maior que o array, o código gerará um erro.

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

Funções

- fn realiza a declaração da função.
- Assim como outras linguagens podemos chamar uma função utilizando o nome dela e seus parâmetros.
- Poderíamos definir another_function antes ou depois de main.

```
fn main() {
    println!("Hello, world!");

    another_function();
}

fn another_function() {
    println!("Another function.");
}
```

Funções

- Para criar um parâmetro, é preciso adicionar o nome da variável e o tipo de dado.
  Ex: (value:i32)

- Podemos declarar mais de um parâmetro.
  Ex: (value: i32, unit_label: char)

- Então podemos chamar a função passando o valor do parâmetro.

```
fn main() {
        print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
        println!("The measurement is: {value}{unit_label}");
}
```

```
$ cargo run
   Compiling functions v0.1.0 (file:///projects/functions)
    Finished dev [unoptimized + debuginfo] target(s) in
0.31s
     Running `target/debug/functions`
The measurement is: 5h
```

Funções

- Statements: são instruções que realizam ações e não retornam um valor.
- Expressão: retornam um valor no final.
- Nota que no exemplo de Expressões, o 5 não termina com “;”, isso é para indicar que 5 é o valor retornado pelo função.

Exemplo de Statements:

```
fn main() {
      let y = 6;
}
```

Exemplo de Expressão:

```
fn five() -> i32 {
    5
}
```

Statements

- No exemplo 1, “let y = 6” não retorna nenhum valor para x.
- Em Rust não é possível escrever x = y = 6.
- Link do código.

Exemplo 1:

```
fn main() {
        let x = (let y = 6);
}
```

Expressão

- Expressões podem ser parte de Statements.
- No exemplo 2, let y recebe o valor 4.
- Nota que “x + 1” não termina em “;”.

Exemplo 2:

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {y}");
}
```

Expressão

- “-> i32” indica o tipo de dado que a função vai retornar.
- O valor de retorno não pode terminar em “;”, se não, a função se torna um statement, não retornando nenhum valor.

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {x}");
}
```

Expressão

- Exemplo de uma Expressão que utiliza o parâmetro “x: i32”.

```
fn main() {
    let x = plus_one(5);

    println!("The value of x is: {x}");
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

Comentários

- Utilizamos “//” para comentar em Rust.
- Para comentar diversas linhas, é preciso adicionar “//” no início de todas elas.

Exemplo 1

```
fn main() {
    // I’m feeling lucky today
    let lucky_number = 7;
}
```

Exemplo 2

```
fn main() {
    // I’m feeling lucky today
    let lucky_number = 7;
}
```

Controle de Fluxo

- If, else if e else
- Repetições com loops
  - Loop
  - While
  - For

If, else if e else

- Definido por “if condição”.
- A condição deve ser boleana.

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

If, else if e else

- Esse código não compila, porque number não é booleano.
- O Rust não tenta converter valores não booleanos em booleanos,

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

```
$ cargo run
   Compiling branches v0.1.0 (file:///projects/branches)
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |    if number {
  |    ^^^^^^ expected `bool`, found integer

For more information about this error, try `rustc --explain E0308`.
error: could not compile `branches` due to previous error
```

If, else if e else

- O código ao lado mostra a utilização do "else if".
- Link do código.

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("number is divisible by 4");
    } else if number % 3 == 0 {
        println!("number is divisible by 3");
    } else if number % 2 == 0 {
        println!("number is divisible by 2");
    } else {
        println!("number is not divisible by 4, 3, or 2");
    }
}
```

Expressão if

- Podemos utilizar if na definição de variáveis.
- No exemplo 2, o código não compila, pois if tem um i32 e o else uma string.
- Os dois precisam ser do mesmo tipo, como no exemplo 1.

Exemplo 1:

```
fn main() {
        let condition = true;
        let number = if condition { 5 } else { 6 };

        println!("The value of number is: {number}");
}
```

Exemplo2:

```
fn main() {
        let condition = true;
        let number = if condition { 5 } else { "six" };

        println!("The value of number is: {number}");
}
```

Loop

- Realiza um loop, até que chegue em um break, que é o comando para parar.

Exemplo 1:

```
fn main() {
    loop {
        println!("again!");
    }
}
```

Exemplo 2:

```
fn main() {
    loop {
        println!("one.");
        break;
    }
}
```

Loop

- Podemos utilizar o loop na definição da variável.
- Podemos adicionar uma variável counter, para realizar a parada do loop.
- “break counter * 2;” sinaliza o valor que o loop retorna.

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

Loop

- Podemos utilizar um loop dentro de um loop.
- “'counting_up: loop” está nomeando o loop de fora.
- “break;” está somente parando o loop de dentro.
- “break 'counting_up;” está parando o loop de fora.
- Link do código.

```
fn main() {
    let mut count = 0;
    'counting_up: loop {
        println!("count = {count}");
        let mut remaining = 10;

        loop {
            println!("remaining = {remaining}");
            if remaining == 9 {
                break;
            }
            if count == 2 {
                break 'counting_up;
            }
            remaining -= 1;
        }

        count += 1;
    }
    println!("End count = {count}");
}
```

While

- É dado por “while condição”.
- Pode ser realizado com um loop, if, else e break.
- index += 1; é equivalente a index = index + 1;
- Link do código.

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;

    while index < 5 {
        println!("the value is: {}", a[index]);
        index += 1;
    }
}
```

For

- Assim como em python, este for percorre todos os elementos do array
- Link do código.

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {element}");
    }
}
```

For

- Nesse caso o for utiliza um range.
- .rev() reverte o range.

```
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
```

Referências

- https://doc.rust-lang.org/book/ch03-00-common-programming-concepts.html