

Estruturas e implementações

Professor: Pedro Horschulhack

Disciplina: Programação Imperativa

Conteúdo

1. Estruturas
2. Implementações
3. Exercícios

Estruturas

Estruturas

- Quando lidamos com muitos tipos de dados ao mesmo tempo, podemos utilizar vetores ou tuplas
- Para obtermos o valor em alguma posição, podemos fazer através do `.` seguido do índice que queremos acessar.

```
fn main() {  
    let pessoa: (String, i32) = (String::from("Bob"), 28);  
    println!("A pessoa se chama {} e tem {} anos", pessoa.0, pessoa.1);  
}
```

Estruturas

- Neste caso, podemos, também, passar uma tupla como parâmetro de função e acessar os seus membros
- No entanto, temos um problema: O acesso aos membros da tupla são ordenados de acordo com a sua definição.

```
fn informacoes_pessoa(pessoa: (String, i32)) {  
    println!("A pessoa se chama {} e tem {} anos", pessoa.0, pessoa.1);  
}  
  
fn main() {  
    let pessoa: (String, i32) = (String::from("Bob"), 28);  
    informacoes_pessoa(pessoa);  
}
```

Estruturas

- No exemplo abaixo queremos acessar o índice 0 (zero), que corresponde ao nome.
- Mas e se eventualmente mudarmos?
 - O resultado não sairia como esperado! Teremos que mudar, inclusive, a forma que acessamos os valores

```
fn informacoes_pessoa(pessoa: (i32, String)) {  
    println!("A pessoa se chama {} e tem {} anos", pessoa.0, pessoa.1);  
}  
  
fn main() {  
    let pessoa: (i32, String) = (28, String::from("Bob"));  
    informacoes_pessoa(pessoa);  
}
```


Estruturas

- Estruturas (`structs`) são elementos presentes em diversas linguagens de programação
- Elas auxiliam na organização e agrupamento de um conjunto de informações
- Além disso, podem ser bastante semelhantes às tuplas, porém veremos que elas são mais eficazes no que tange à organização do código

Estruturas

– Temos structs com as mesmas variantes de enumerações:

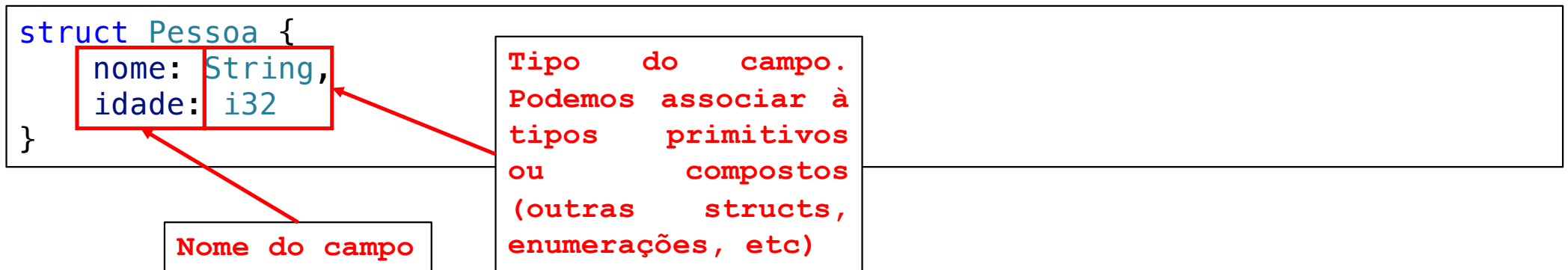
- Estruturas
- Tuplas
- Unidades



| | |
|-------------|-------------|
| Na | disciplina |
| estaremos | trabalhando |
| somente | com essa |
| variante de | estrutura |

Estruturas

- Em uma estrutura nós **definimos** o nome do dado que iremos acessar. Dessa forma, não precisamos nos ater na ordem em que eles são declarados.
- A definição de uma estrutura está logo abaixo:



Estruturas

- Podemos entender que esse trecho de código representa uma ideia, um conceito, dentro do nosso código. Ele servirá como base para **instanciarmos** uma pessoa **concreta** dentro do nosso código.
- Vejam os exemplos na sequência

Estruturas

```
struct Pessoa {  
    nome: String,  
    idade: i32  
}
```

```
fn main() {  
    let pessoa: Pessoa = Pessoa{  
        nome: String::from("Bob"),  
        idade: 28  
    };  
}
```

Instância

Neste código criamos a mesma pessoa, porém de outra forma

Note que a variável `pessoa` é imutável, portanto não conseguiremos alterar os seus valores internos após a sua criação

Estruturas

- Ao invés de criarmos criar um “atalho” para inicializarmos os valores de uma struct através de funções que retornam structs do mesmo tipo.

```
struct Pessoa {  
    nome: String,  
    idade: i32  
}  
  
fn new_pessoa(nome: String, idade: i32) -> Pessoa {  
    Pessoa { nome: nome, idade: idade }  
}  
  
fn main() {  
    let nome: String = String::from("Bob");  
    let idade: i32 = 28;  
    let pessoa: Pessoa = new_pessoa(nome, idade);  
}
```

Podemos simplificar a atribuição dos campos da struct com os seus valores, desde que ambos (campos e valores) sejam do mesmo tipo e tenham o mesmo nome.


Pessoa { nome, idade }

Instância

Estruturas

- Também conseguimos passar estruturas como parâmetro de função

Pegamos propriedade de p



```
fn mostrar_pessoa(p: Pessoa) {  
    println!("A pessoa se chama {} e tem {} anos", p.nome, p.idade);  
}  
  
fn new_pessoa(nome: String, idade: i32) -> Pessoa {  
    Pessoa { nome: nome, idade: idade }  
}  
  
fn main() {  
    let pessoa: Pessoa = new_pessoa(String::from("Bob"), 28);  
    mostrar_pessoa(pessoa); // A pessoa se chama Bob e tem 28 anos  
}
```

Implementações

Implementações

- Ao longo dos nossos códigos em Rust já nos deparamos com chamadas de funções de algumas variáveis.
- Por exemplo, quando colocamos um valor em um vetor (`v.push(e)`), quando queremos o tamanho de uma String (`s.len()`), quando vamos remover a quebra de linha (`\n`) de uma String (`s.trim()`), etc.
- Esse tipo de função se chama **método** e que são **associadas** a uma struct (funções associadas). Funções normais, como criamos até agora, são chamadas de funções livres (não possuem associação).

Implementações

- Nós declaramos a associação através de blocos de código **impl**, onde dentro dele criamos funções e podemos acessar os membros de uma determinada instância de uma struct.
- Antes havíamos implementado uma função que criava uma pessoa dessa forma:

Implementações

```
struct Pessoa {  
    nome: String,  
    idade: i32  
}  
  
fn new_pessoa(nome: String, idade: i32) -> Pessoa {  
    Pessoa { nome, idade }  
}  
  
fn main() {  
    let nome: String = String::from("Bob");  
    let idade: i32 = 28;  
    let pessoa: Pessoa = new_pessoa(nome, idade);  
}
```

Implementações

Aqui vamos usar o bloco `impl`

```
struct Pessoa {  
    nome: String,  
    idade: i32  
}  
  
impl Pessoa {  
    fn new(nome: String, idade: i32) -> Pessoa {  
        Pessoa{nome, idade}  
    }  
}  
  
fn main() {  
    let nome: String = String::from("Bob");  
    let idade: i32 = 28;  
    let pessoa: Pessoa = Pessoa::new(nome, idade);  
}
```

Por que aqui acessamos a função associada por meio de `::`?

Implementações

- Nós acessamos aquela função associada (`new`) através do operador `::` porque ela é uma função **estática**, isto é, chamamos ela direto da **definição** da struct.
- Ainda, não precisamos de uma **instância** da struct para chamar o método estático.
- Agora vamos criar um método que seja possível chamar de uma instância da struct.

Implementações

```
struct Pessoa {  
    nome: String,  
    idade: i32  
}  
  
impl Pessoa {  
    fn new(nome: String, idade: i32) -> Pessoa {  
        Pessoa{nome, idade}  
    }  
  
    fn falar(&self, frase: String) {  
        println!("{}", disse '{}', self.nome, frase);  
    }  
}  
  
fn main() {  
    let pessoa: Pessoa = Pessoa::new(String::from("Bob"), 28);  
    pessoa.falar("Olá, mundo".to_string()); // Bob disse 'Olá, mundo'  
}
```

O que quer dizer esse
&self?

Implementações

- O **self** representa a **instância** da struct que está chamando o método (função)

É uma referência (ponteiro) para a struct instanciada (pessoa), portanto é possível acessar os campos nome e idade.

```
impl Pessoa {  
    fn falar(&self, frase: String) {  
        println!("{}", disse '{}', self.nome, frase);  
    }  
}
```

Aqui nós emprestamos self (&self) porque devemos sempre seguir as regras de ownership do Rust.

```
fn main() {  
    let pessoa: Pessoa = Pessoa::new(String::from("Bob"), 28);  
    pessoa.falar("Olá, mundo".to_string()); // Bob disse 'Olá, mundo'  
}
```

Exercício 1

- Implemente uma estrutura chamada Pessoa e que esteja de acordo com a seguinte especificação:
 - Possui três campos: nome, do tipo String; idade, do tipo u8 e altura, do tipo f32.
- Uma função estática new, que retorna uma estrutura Pessoa, com os seguintes parâmetros: nome, idade e altura. Ao final, retorne uma instância de Pessoa inicializada com esses parâmetros.
- Crie uma função associada a estrutura Pessoa chamada falar, com um parâmetro mensagem do tipo String e sem retorno, que mostre na tela “A pessoa <NOME> disse <MENSAGEM>”, substituindo NOME pelo campo nome e MENSAGEM pelo parâmetro mensagem.
- Crie uma função main para criar uma pessoa através do método new e chamar a função falar fornecendo uma mensagem arbitrária como parâmetro.

Exercício 2

- Implemente uma estrutura chamada `Permissao` com três campos booleanos: `escrita`, `leitura` e `execução`
 - Crie uma função estática `new`, que retorna uma estrutura `Permissao`, com os seguintes parâmetros: `escrita`, `leitura` e `execucao`. Ao final, retorne uma instância de `Permissao` inicializada com esses parâmetros.
- No mesmo código, implemente outra estrutura chamada `Arquivo` com três campos: `permissao` (`Permissao`), `nome` (`String`) e `tamanho` (`usize`).
 - Crie uma função estática `new`, que retorna uma estrutura `Arquivo`, inicializando seus campos. Ao final, retorne uma instância de `Arquivo` inicializada.
- Na função `main`, crie quatro arquivos a partir das funções criadas.

Contato: p.horchulhack@pucpr.br