

Ownership e Borrowing

Professor: Pedro Horschulhack

Disciplina: Programação Imperativa

Conteúdos

1. Revisão de ponteiros e referências
2. Ownership (propriedade)
3. Borrowing (empréstimo)

Referências e ponteiros

- Quando nós trabalhamos com referências a valores, o conceito geral era que elas armazenavam o **endereço** de uma variável na memória, ao invés do seu valor
- Vejamos o exemplo abaixo:

```
fn main() {  
    let x: i32 = 42;  
    let y: &i32 = &x;  
    println!("x={} y={:p}", x, y);  
}
```

Referências e ponteiros

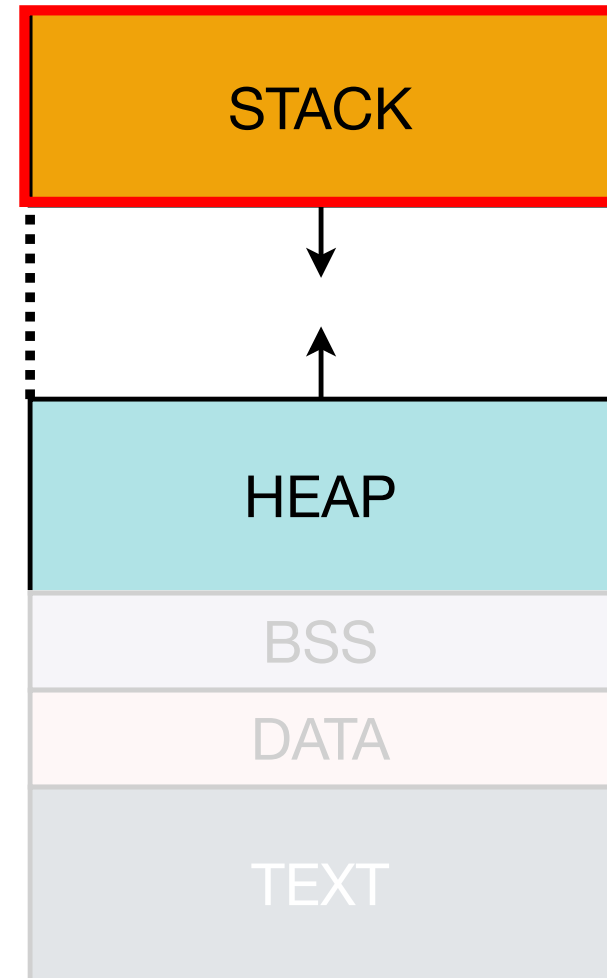
- Até aqui entendemos que nossas variáveis são armazenadas na stack (pilha) de execução
 - Ou seja, assim que terminamos a execução de uma função ou escopo (**código entre chaves**), todo o espaço utilizado é liberado

```
fn main() {  
    let w: i32 = 5;  
    {  
        let s: &str = "olá, mundo!";  
        println!("w={}", w);  
    }  
    println!("s={}", s);  
}
```

Esse código não irá compilar, porque a variável **s** está sendo chamada após ter saído de escopo!

Referências e ponteiros

- Duas coisas importantes!
 - Quando a variável **s** entra no escopo, ela é válida
 - Ela continua existindo até o final de um escopo
- Lembrando:
 - Todas as variáveis são alocadas e armazenadas na (stack) pilha e, quando o escopo acaba, elas são liberadas da memória
 - Isso ocorre porque **sabemos** o tamanho que cada variável ocupará na memória



Referências e ponteiros

- No entanto, nem todas as variáveis são armazenadas **inteiramente** na pilha como, por exemplo, `Strings`.
 - As variáveis também podem ser armazenadas na **heap**!
 - Como assim?
- Como já comentado em algumas aulas, temos dois tipos de strings:
 - Literais (`&str`)
 - Dinâmicas (`String`)

Referências e ponteiros

–Vamos dar uma atenção à este exemplo:

```
fn main() {
```



**Essa variável terá
seu valor
armazenado na
heap!**

```
}
```

Referências e ponteiros

- O que aconteceu no exemplo anterior? Como eu armazeno uma informação que à princípio eu não sei o tamanho?
 - Solicito ao sistema operacional uma quantidade de memória para eu trabalhar
 - Depois que utilizei a memória, preciso devolve-la ao sistema operacional
- Como é feito o processo de “devolução” de memória?
 - Muitas linguagens dependem de um mecanismo chamado Coletor de Lixo (Garbage Collector, GC)
 - O **gerenciamento de memória** não é mais responsabilidade do desenvolvedor
- Sem o GC, nós devemos solicitar e devolver a memória. **Manualmente.**

Referências e ponteiros


- Qual o problema de termos esse controle?
 - Nós desperdiçamos memória, ou seja, deixamos regiões reservadas sem uso
 - Se liberarmos a memória muito cedo, teremos uma variável **inválida**
 - Se liberarmos duas vezes, teremos outro bug
- Para cada chamada de **alocação** temos que ter uma chamada de **liberação**
- Vamos dar uma olhada como isso é feito em Rust

Referências e ponteiros

1. Inicia escopo da função `main()`
2. Inicia um novo escopo
3. Solicita ao Sistema Operacional uma região na heap para armazenar a String
4. Faz alguma coisa
5. Termina escopo e chama, implicitamente, a função `drop()`
6. Termina o código

Aqui a memória
será liberada
para o Sistema
operacional

```
fn main() { 1
    { 2
        3 let s1: String = String::from("Prog. Imperativa");
        4 // Faça alguma coisa com 's1'
        5 }
    6 }
```



Referências e ponteiros

- Em Rust as variáveis interagem com as mesmas informações de diferentes formas, vejamos um exemplo:

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

- Neste caso atribuímos o valor **3** à **x** e copiamos o valor de **x** para **y**, portanto **y = 3**

Referências e ponteiros

–E neste caso? A memória será copiada?

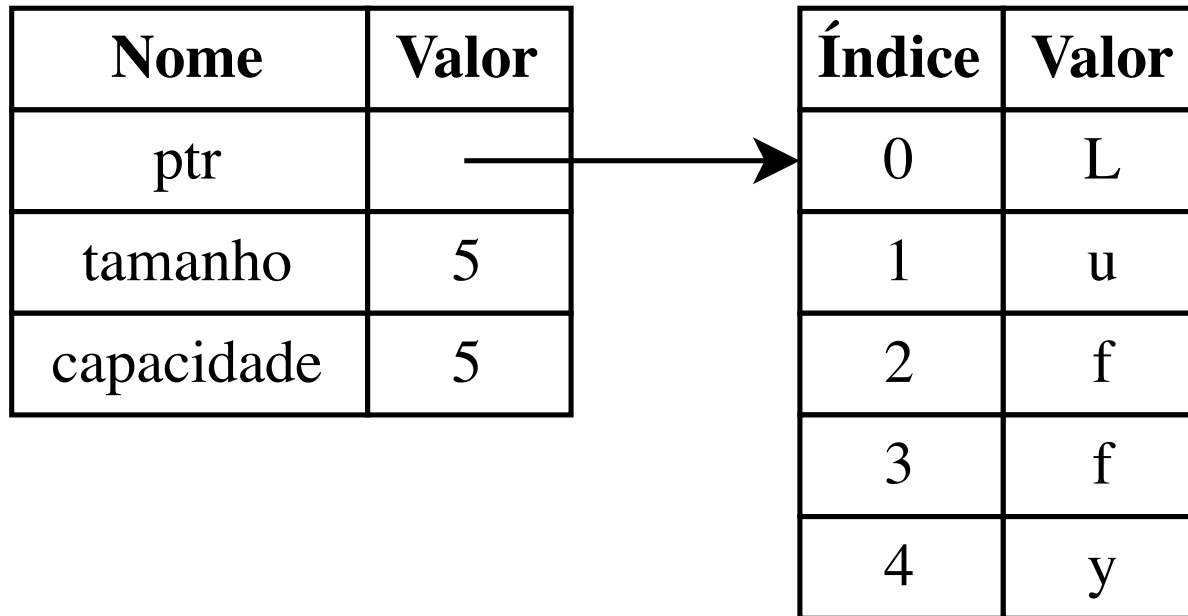
```
fn main() {  
    let nome: String = String::from("Luffy");  
    let nome_copia: String = nome;  
}
```

–Vamos observar minuciosamente...

Referências e ponteiros

- Uma String é uma variável complexa, pois armazena várias informações além do texto. Vejamos a estrutura para a string a seguir:

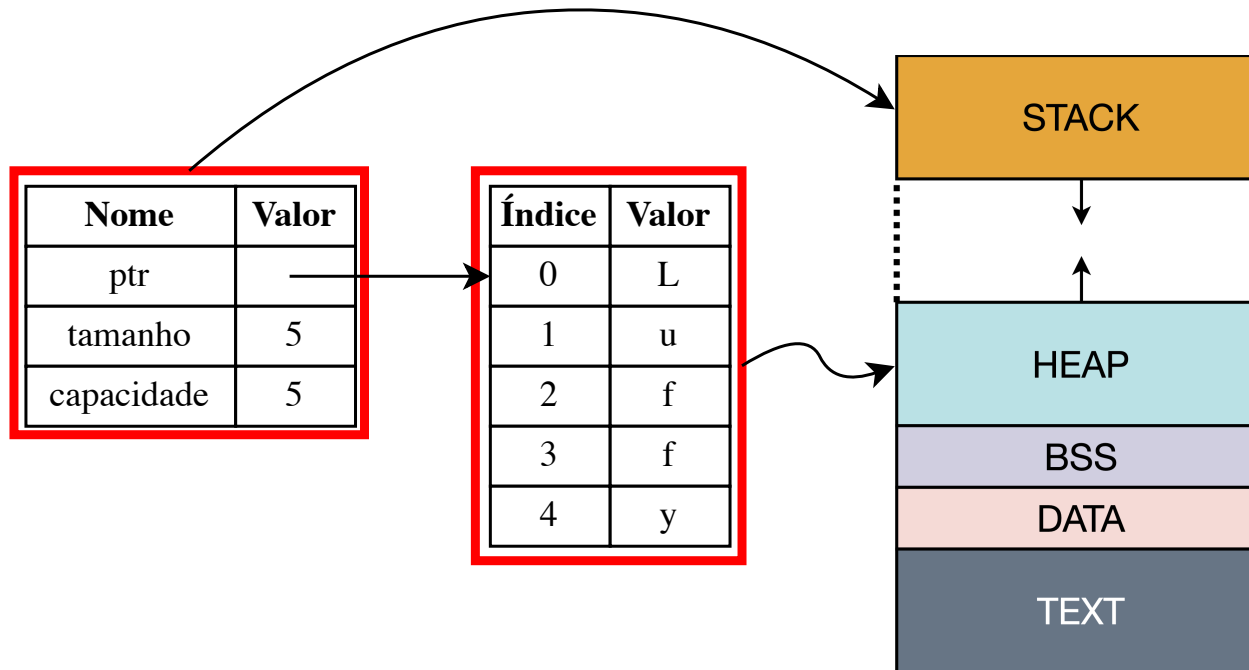
```
let nome: String = String::from("Luffy");
```



Referências e ponteiros

- Uma String é uma variável complexa, pois armazena várias informações além do texto. Vejamos a estrutura para a string a seguir:

```
let nome: String = String::from("Luffy");
```



Referências e ponteiros

- Sim, a memória é copiada! (Parcialmente)
 - Copiamos somente o ponteiro, o tamanho e a capacidade da String, pois elas estão na pilha

```
fn main() {  
    let nome: String = String::from("Luffy");  
    let nome_copia: String = nome;  
}
```

- No Rust os dados que estão na heap não são copiados, principalmente por performance

Referências e ponteiros

- Sim, a memória é copiada! (Parcialmente)
 - Copiamos somente o ponteiro, o tamanho e a capacidade da String, pois elas estão na pilha

```
fn main() {  
    let nome: String = String::from("Luffy");  
    let nome_copia: String = nome;  
}
```

error[E0382]: borrow of moved value: `nome`

--> src/main.rs:5:22

```
2 |     let nome: String = String::from("Luffy");  
   |     ---- move occurs because `nome` has type `String`, which does not implement the `Copy` trait  
3 |     let nome_copia: String = nome;  
   |                             ---- value moved here  
4 |  
5 |     println!("Olá, {}", nome);  
   |                        ^^^^ value borrowed here after move
```

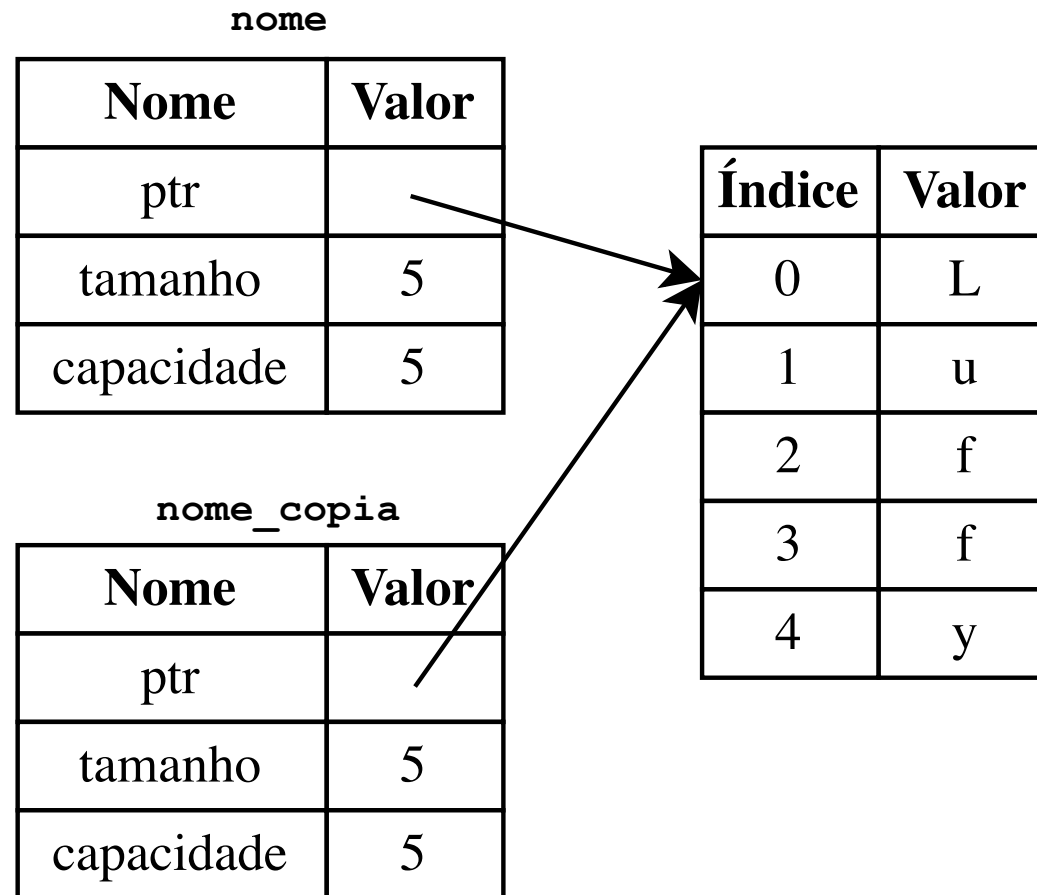

Referências e ponteiros

- Esse erro ocorreu porque **nome** foi **movido** à **nome_copia**
- Um movimento pode ser descrito por nós criarmos uma cópia rasa (somente o ponteiro, tamanho e capacidade da string) e invalidarmos a primeira variável

```
error[E0382]: borrow of moved value: `nome`
--> src/main.rs:5:22
2 |     let nome: String = String::from("Luffy");
   |     ---- move occurs because `nome` has type `String`, which does not implement the `Copy` trait
3 |     let nome_copia: String = nome;
   |                             ---- value moved here
4 |
5 |     println!("Olá, {}", nome);
   |                        ^^^^ value borrowed here after move
```

Referências e ponteiros

– Na prática, é isto que ocorre na memória



Referências e ponteiros

- Nós já vimos que quando uma variável sai de escopo, ela tem sua região da memória liberada
- Mas, já que temos duas variáveis **apontando para o mesmo lugar**, o que ocorre se eu tentar liberar o espaço das duas simultaneamente?
 - Erro de liberação dupla! (Podemos injetar código malicioso)
 - Em linguagens como C, C++ e Assembly isso é permitido e traz consigo problemas de memória

Referências e ponteiros (Exercícios)

1. Crie uma função que recebe uma string como parâmetro e retorne um `usize` que representa o tamanho dela. (Não pode utilizar funções nativas do Rust)
2. Crie uma função que, a partir de um parâmetro que é uma referência à uma variável do tipo `String`, mostre o seu endereço na memória

Ownership

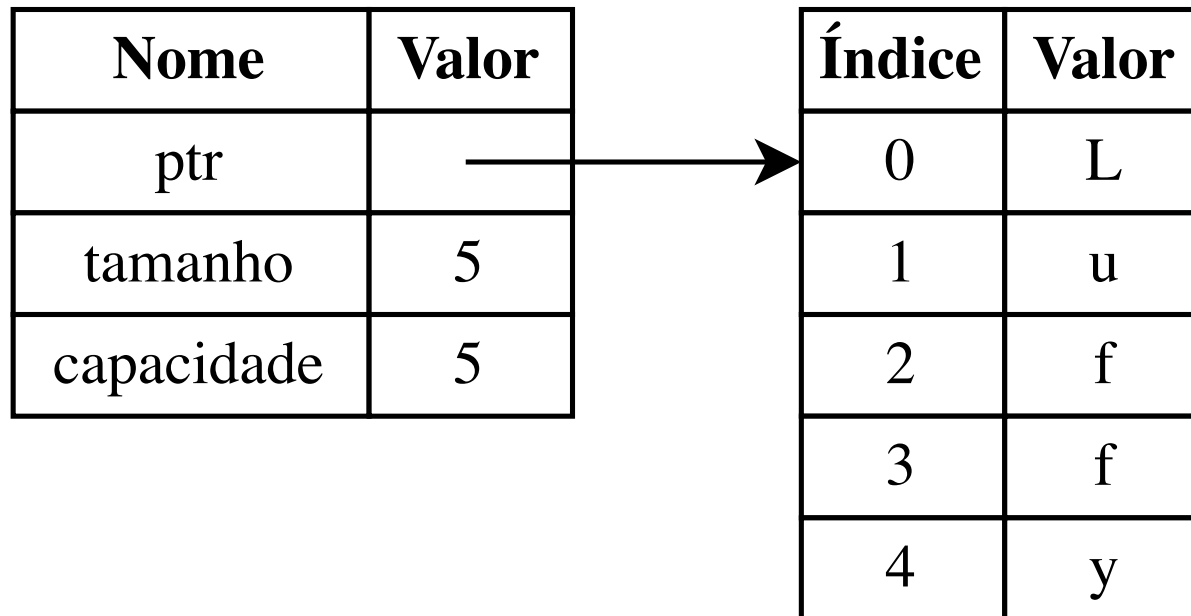
Ownership

- Ownership (propriedade) sobre uma variável ocorre junto a ação de movimento (move) de dados
 - Auxilia na prevenção de liberação de memória indevida
- Essa característica no Rust permite que, por exemplo, uma região de memória é pertencente/propriedade de outra
- Voltemos ao exemplo de strings

Ownership

Aqui podemos dizer que a variável nome é dona da String “Luffy”.

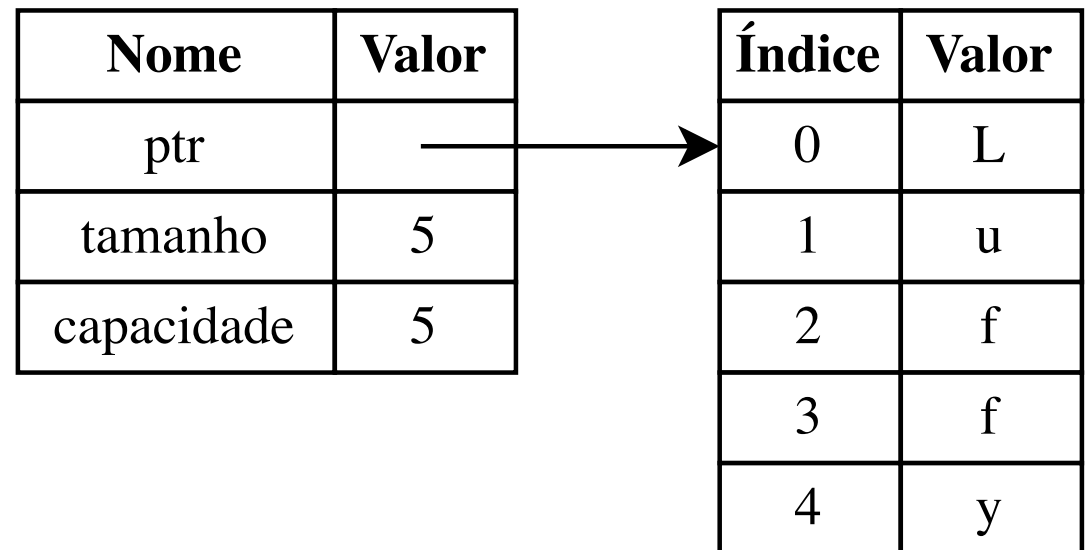
```
let nome: String = String::from("Luffy");
```



Ownership

- Para evitarmos erros de vazamento de memória, por exemplo, devemos **garantir** que os ponteiros devem ser destruídos antes do objeto/variável que é dono dele
- O proprietário determina o tempo de vida da propriedade e todos devem respeitar suas decisões

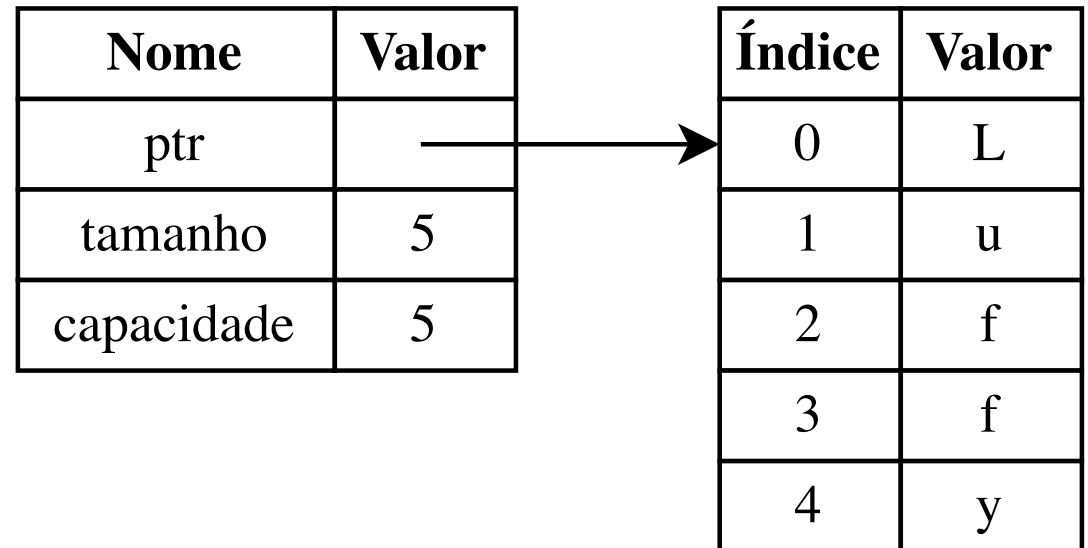
```
let nome: String = String::from("Luffy");
```



Ownership

- Em Rust todo valor possui somente um único dono (owner) que determina seu tempo de vida
- Quando o dono é liberado da memória, então a propriedade é liberada também

```
let nome: String = String::from("Luffy");
```



Ownership

```
fn pega_propriedade(s1: String) {  
    println!("Aqui eu peguei a propriedade da String {} ({:p})", s1, s1.as_ptr());  
}  
  
fn copiar(i: i32) {  
    println!("Copiei o valor {}", i);  
}  
  
fn main() {  
    let s: String = String::from("Luffy");  
    println!("Endereço de s={:p}", s.as_ptr());  
  
    pega_propriedade(s);  
  
    let x: i32 = 5;  
    copiar(x);  
    println!("Valor de x={}", x);  
}
```

A propriedade de **s** passou para a função **pega_propriedade** (variável **s1**)

A partir daqui a variável **s** não é mais válida, pois sua propriedade foi transferida para outro lugar e ela foi liberada assim que terminou o escopo de **pega_propriedade**

Ownership

```
fn pega_propriedade(s1: String) {  
    println!("Aqui eu peguei a propriedade da String {} ({:p})", s1, s1.as_ptr());  
}  
  
fn copiar(i: i32) {  
    println!("Copiei o valor {}", i);  
}  
  
fn main() {  
    let s: String = String::from("Luffy");  
    println!("Endereço de s={:p}", s.as_ptr());  
  
    pega_propriedade(s);  
    println!("Valor de s={}", s);  
  
    let x: i32 = 5;  
    copiar(x);  
    println!("Valor de x={}", x);  
}
```

Ownership

```
fn pega_propriedade(s1: String) {  
    println!("Aqui eu peguei a propriedade da String {} ({:p})", s1, s1.as_ptr());  
}
```

```
fn copiar(i: i32) {
```

error[E0382]: borrow of moved value: `s`

→ src/main.rs:14:28

```
10 |     let s: String = String::from("Luffy");  
    |     - move occurs because `s` has type `String`, which does not implement the `Copy` trait  
...  
13 |     pega_propriedade(s);  
    |     - value moved here  
14 |     println!("Valor de s={}", s);  
    |                       ^ value borrowed here after move
```

```
    let x: i32 = 5;  
    copiar(x);  
    println!("Valor de x={}", x);  
}
```

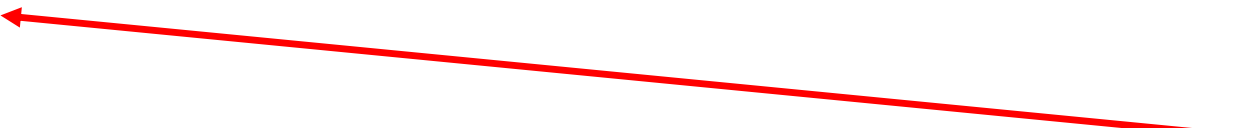
Ownership

- **Relembrando:** Todas as variáveis possuem um tempo de vida
 - Tempo de vida que é o processo de alocação (início) e liberação (fim)
- O gerenciamento do tempo de vida depende do escopo onde ela está, ou seja, quando declarada no início do escopo, ao final terá sua região da memória liberada.
- Portanto, quando passamos a propriedade de uma variável para um escopo ou outra variável, também estamos passando os direitos de alterar esse tempo de vida.

Ownership

– Boas notícias! Temos como recuperar a propriedade de uma variável!

```
fn pega_e_devolve_propriedade(s1: String) -> String {  
    println!("Aqui eu peguei a propriedade da String {} ({:p})", s1, s1.as_ptr());  
    println!("Irei devolvê-la retornando o seu valor");  
    s1  
}  
  
fn main() {  
    let s: String = String::from("Luffy");  
    println!("Endereço de s={:p}", s.as_ptr());  
  
    let s1 = pega_e_devolve_propriedade(s);  
    println!("Valor de s={} e endereço={:p}", s1, s1.as_ptr());  
}
```



Podemos recuperar a propriedade através do retorno do valor emprestado.

Ownership (Exercício)

- Crie uma função **concatenar** que recebe duas Strings como parâmetro e retorne a concatenação das duas. Considere o exemplo a seguir e os conceitos de ownership para resolver

```
fn main() {  
    let a: String = String::from("Olá");  
    let b: String = String::from("Mundo");  
    let c: String = String::from("!");  
  
    let resultado1 = concatenar(a, b); // Olá Mundo  
    let resultado2 = concatenar(resultado1, c);  
    println!("{}", resultado2); // Olá Mundo!  
}
```

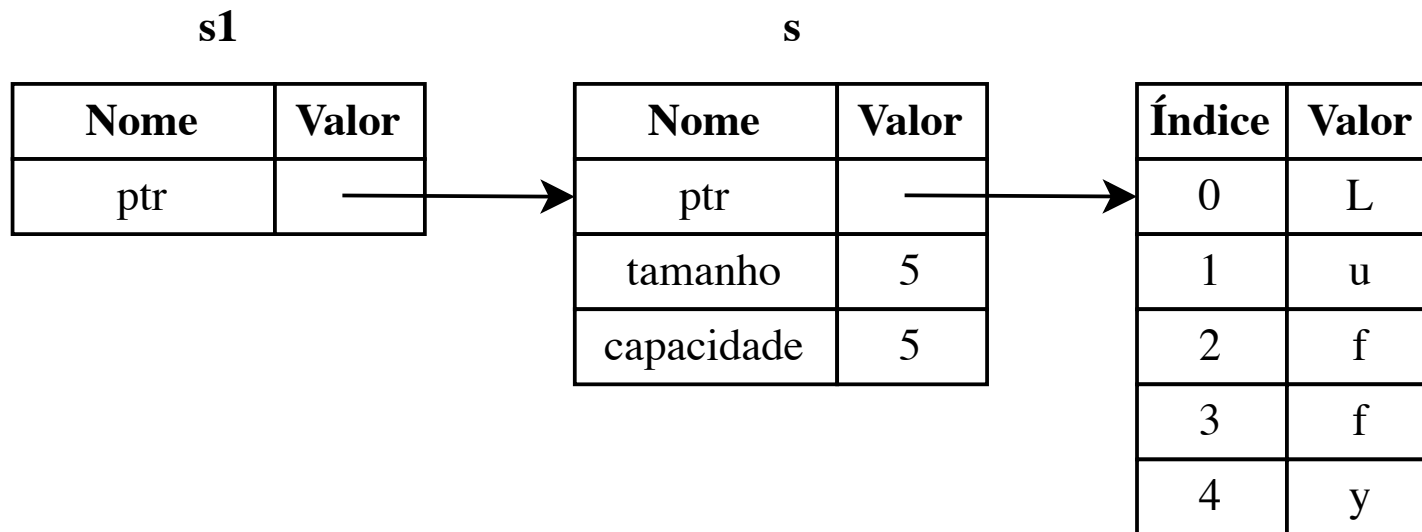
Borrowing

Borrowing

- Até podemos recuperar a propriedade sobre uma variável, porém, muitas vezes, não queremos ficar retornando variáveis para poder reutilizá-las
- Já vimos anteriormente como podemos reutilizar uma região de memória, ou seja, ir para além das regras de propriedade
 - Referências!
- As referências, em Rust, chamamos de borrow (empréstimo)

Borrowing

```
fn empresta(s1: &String) {  
    println!("Aqui eu emprestei a propriedade da String {} ({:p})", s1, s1.as_ptr());  
}  
  
fn main() {  
    let s: String = String::from("Luffy");  
    println!("Endereço de s={:p}", s.as_ptr());  
  
    empresta(&s);  
    println!("Consigo utilizar s normalmente! Olha só o endereço -> {:p}", s.as_ptr());  
}
```



Borrowing

– O que está ocorrendo nesse exemplo?

```
fn calcula_comprimento(s1: &String) -> usize {  
    s1.len()  
}
```

Pega referência (empresta)
à uma `String`

Calcula e retorna o
tamanho de `s1`

`s1` sai de escopo. Como
não temos a propriedade
dele, nada acontece.

Borrowing

- Por padrão, tudo em Rust é **imutável**. Isso se aplica, também, à variáveis emprestadas
- Quando tentamos alterar um valor que foi emprestado, o código produzirá um erro

```
fn mudar(s1: &String) {  
    s1.push_str(" Luffy");  
}  
  
fn main() {  
    let s: String = String::from("Monkey D.");  
    mudar(&s);  
}
```

Borrowing

– Por padrão, tudo em Rust é **imutável**. Isso se aplica, também, à variáveis emprestadas

```
error[E0596]: cannot borrow `*s1` as mutable, as it is behind a `&` reference
```

```
--> src/main.rs:10:5
```

```
10 |     s1.push_str(" Luffy");
```

```
~~~~~ `s1` is a `&` reference, so the data it refers to cannot be borrowed as mutable
```

```
help: consider changing this to be a mutable reference
```

```
9 | fn mudar(s1: &mut String) {  
    +++
```

```
fn main() {  
    let s: String = String::from("Monkey D.");  
    mudar(&s);  
}
```

Borrowing

- Resolvemos colocando `mut` na frente do operador de empréstimo/referência (&)

```
fn mudar(s1: &mut String) {  
    s1.push_str(" Luffy");  
}  
  
fn main() {  
    let mut s: String = String::from("Monkey D.");  
    mudar(&mut s);  
}
```

Borrowing

- No entanto, referências mutáveis possuem uma restrição: você só pode ter uma referência mutável à um pedaço específico de dados em um escopo específico.
- Isso permite um maior controle do comportamento do seu programa, porque evita ***data races*** no tempo de compilação, quando:
 - Dois ou mais ponteiros acessam o mesmo dado simultaneamente
 - Ao menos um dos ponteiros está sendo usado para escrever à região de memória
 - Não existem mecanismos utilizados para sincronizar o acesso aos dados
- Vejamos o exemplo a seguir

Borrowing

```
fn main() {  
    let mut s = String::from("Olá");  
  
    let r1 = &mut s;  
    let r2 = &mut s;  
  
    r1.push_str(", mundo");  
}
```


Borrowing

error[E0499]: cannot borrow `s` as mutable more than once at a time

--> src/main.rs:7:14

6 | let r1 = &mut s;

----- first mutable borrow occurs here

7 | let r2 = &mut s;

^^^^^ second mutable borrow occurs here

8 |
9 | r1.push_str(", mundo");

----- first borrow later used here

Borrowing

- Também não podemos ter uma combinação entre referências mutáveis e imutáveis
- Não é esperado que referências imutáveis, repentinamente, tenham seu valor alterado
- No entanto, múltiplas ocorrências de referências imutáveis são **ok** porque nenhuma delas conseguem afetar a leitura dos dados
- Vejamos o exemplo na sequência

Borrowing

```
fn main() {  
    let mut s = String::from("Olá");  
  
    let r1 = &s; // Sem problemas  
    let r2 = &s; // Sem problemas  
    let r3 = &mut s; // PROBLEMA  
}
```

Borrowing

- Além de tudo isso temos também *dangling pointers*, ou ponteiros pendentes, que é um ponteiro que faz referência a uma localização na memória que pode ter sido concedida a outra parte, liberando parte da memória enquanto se mantém um ponteiro para essa memória.
- Em Rust o compilador garante que as referências nunca serão "referências pendentes": se você tiver uma referência a algum dado, o compilador garantirá que os dados não sairão de escopo antes que a referência aos dados saia de escopo.

Borrowing

```
fn pender_ponteiro() -> &String {  
    let s = String::from("Luffy");  
    &s  
}  
  
fn main() {  
    let referencia_para_nada = pender_ponteiro();  
}
```

Borrowing

error[E0106]: missing lifetime specifier

--> src/main.rs:2:25

```
2 | fn pender_ponteiro() -> &String {  
    ^ expected named lifetime parameter
```

= help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from

help: consider using the ``static`` lifetime

Borrowing

```
fn pender_ponteiro() -> &String {  
    let s = String::from("Luffy");  
    &s  
}
```

**Especifica que retornará
uma referência de String**

Cria uma String

**Retornamos a referência à
String; s**

**Aqui a variável `s` sairá do
escopo, portanto será
liberada da memória.**

Borrowing

Solução: Mover a propriedade para fora da função

```
fn não_pender() -> String {  
    let s = String::from("Luffy");  
    s  
}
```


Borrowing (Exercício)

1. Crie uma função `completar` que recebe uma referência `String` e retorne a mesma `String` concatenando um ponto (.) no final.
2. Crie uma função que receba uma `String` como parâmetro e retorne uma tupla dos tipos `usize` e `usize`, onde o primeiro valor representa o tamanho da `String` e o segundo a capacidade. Na sequência, na função `main`, imprima a `String` **após** chamar a função criada. Justifique o porquê de dar errado.
3. Crie uma função `contar_vogais` que aceite uma referência imutável a uma `String` como entrada e retorne o número de vogais na `String`. A função deve ser projetada para **não possuir** a `String`, apenas lê-la.

Contato: p.horchulhack@pucpr.br