Ownership

Diogo Silveira Mendonça

Slides Baseados no Capítulo 4 do Rust Book
https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html

Introdução

- Veremos os conceitos de:
  - Ownership
  - Escopo de variáveis
  - String
  - Memória e Alocação
  - Referência e Borrowing
  - Tipo Slice

- Os exemplos de código podem ser encontrados nesse link.

O que é Ownership?

- Rust não tem garbage-collector e também não necessita que o usuário explicitamente aloque e libere a memória.

- Em Rust, o gerenciamento de memória é feito por um sistema de ownership, que são um conjunto de regras que o compilador verifica.

- Regras do Ownership:
  - Cada valor tem seu dono;
  - Pode ter somente um dono por vez;
  - Quando o dono sai do escopo, o valor será descartado

Escopo de Variáveis

- O código abaixo, mostra o funcionamento de variáveis em escopo.

- s é uma string literal.

```
fn main() {
    {     // s não é válido aqui, não foi declarada ainda
          let s = "hello";   // s é válida daqui em diante

          // faz alguma coisa com s
    }     // este escopo agora acabou, e s não é mais válido
}
```

String

- Podemos criar uma string utilizando: `let s = String::from("hello");`

- Uma string pode ser mutável

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() appends a literal to a String

println!("{}", s); // This will print `hello, world!`
```

- Se tentarmos utilizar o código acima, com uma string literal, gerará um erro, pois a função junta um literal a uma string.

- Strings e Literais possuem comportamentos diferentes na memória.

Memória e Alocação

- Literais são rápidos e eficientes, pois seu valor é reconhecido em tempo de compilação, sendo injetado diretamente no executável.

- Isso não é possível para um dado que o tamanho pode mudar durante a execução do programa, como strings.

- Então para uma string:
  - A memória deve ser solicitada ao Sistema Operacional.
  - Precisa de uma maneira de retornar o espaço de memória ao Sistema Operacional, quando finalizar com a String.

Memória e Alocação

- let s = String::from("hello"); Solicita o espaço de memória
- O fim do escopo o Rust automaticamente chama a função drop, devolvendo o espaço de memória.

```
{
    let s = String::from("hello"); // s é válido deste ponto para frente


    // faz alguma coisa com s
}   // este escopo acabou e agora não é mais válido
```

Interação de variáveis e dados com Move

- Temos que y = x = 5. Isso é verdadeiro, no caso dos Literais, que possuem tamanho fixo e conhecido, e são colocados na pilha.

```
let x = 5;

let y = x;
```

- Já para strings, isso não é verdadeiro.

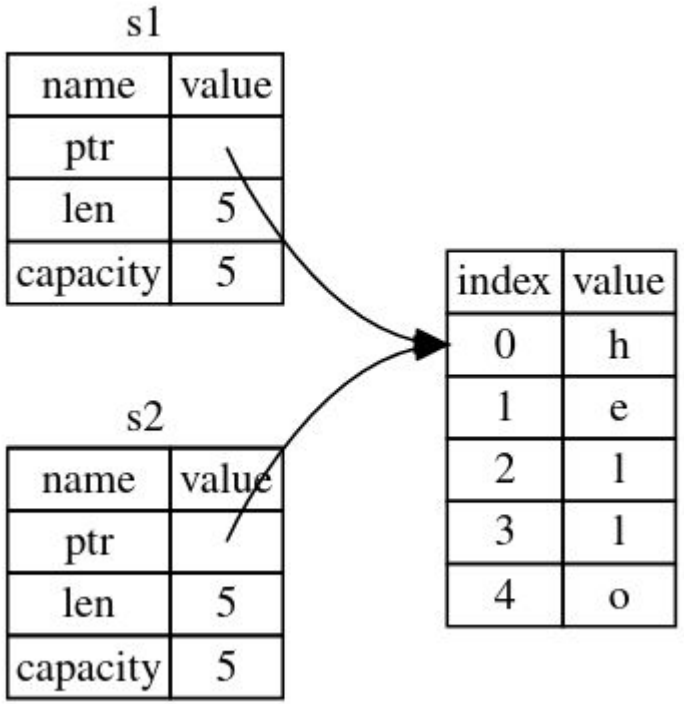
```
let s1 = String::from("hello");

let s2 = s1;
```

Interação de variáveis e dados com Move

- Temos a criação de s1, onde o ponteiro indica o index 0.

- Temos a criação de s2, que copia as propriedades de s1, incluindo o ponteiro.

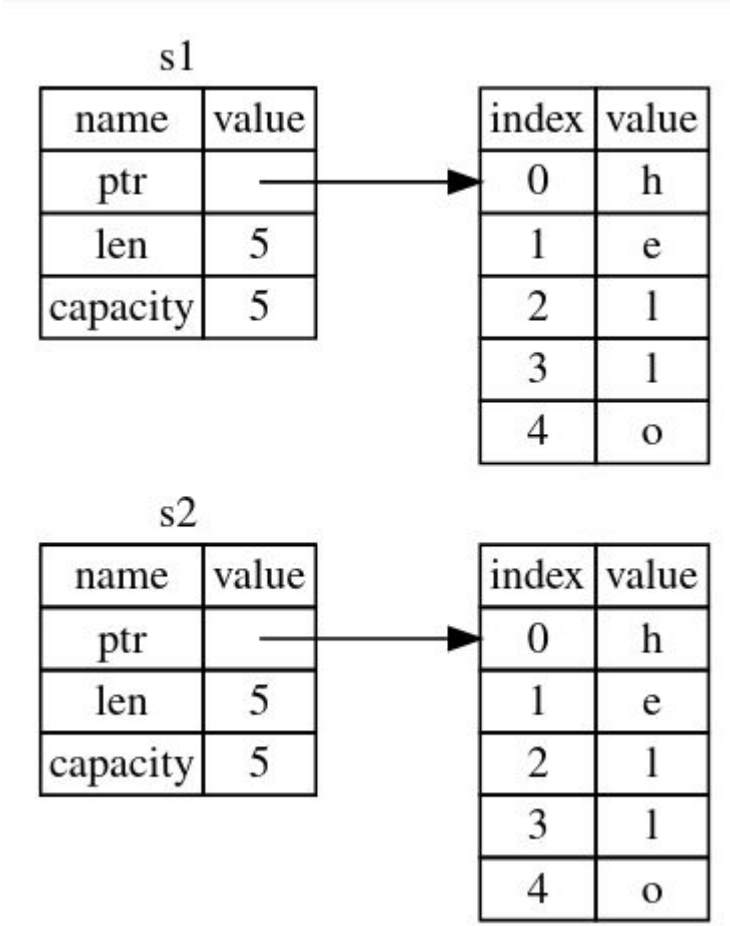
```
let s1 = String::from("hello");

let s2 = s1;
```



Interação de variáveis e dados com Move

- Se o Rust copiasse os valores de s1 para s2, o custo de memória seria maior.

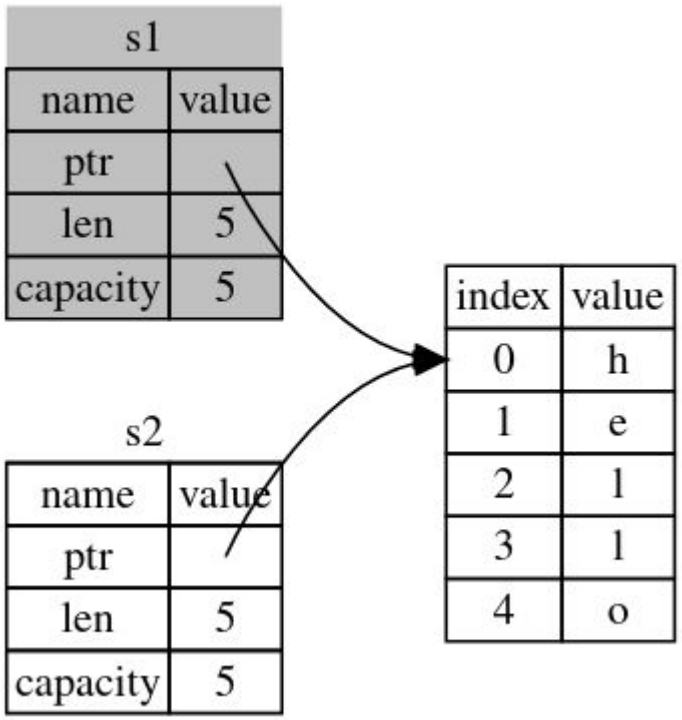
- Isso ocorre para literais, mas não ocorre por default para variáveis que não possuem um tamanho fixo.



Interação de variáveis e dados com Move

- Se o Rust deleta a variável quando sai do escopo. Então s1 e s2 vão tentar apagar o mesmo dado.
- Isso é conhecido como double free error, é um dos bugs de segurança de memória.
- Para solucionar isso, s1 não é mais válido, como podemos perceber no código abaixo que não funciona.

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

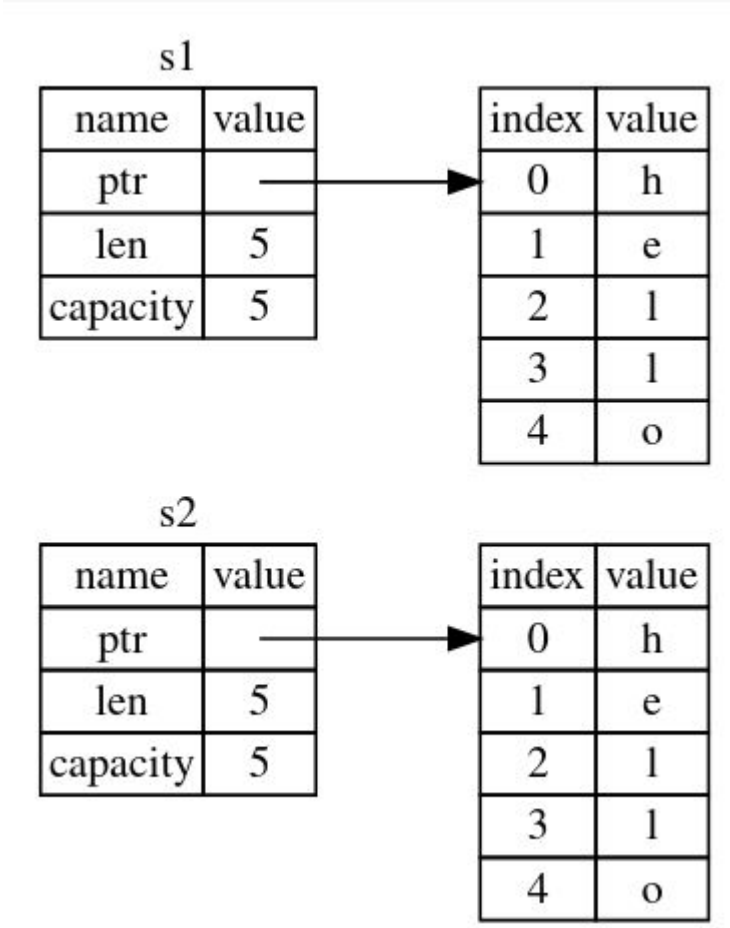


Interação de variáveis e dados com Clone

- let s2 = s1.clone();, realiza uma cópia do valor de s1, para s2.

- Isso faz com que a memória tenha o mesmo comportamento de uma variável literal.

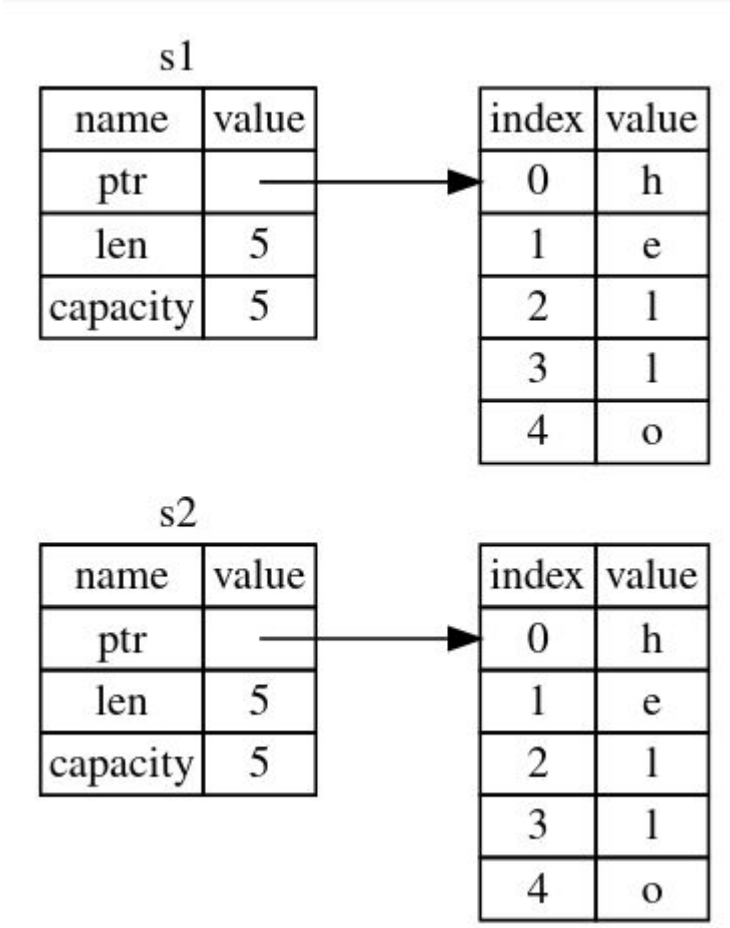
```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```



Dados somente da pilha: copy

- Como dito antes, para variáveis de tamanho fixo, realizam a cópia do valor.
- Isso ocorre, pois variáveis de tamanho fixo são conhecidas em tempo de compilação, sendo armazenadas em uma pilha.
- Por isso copiar o valor é uma forma rápida.

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```



Ownership e Funções

- Quando utilizamos uma variável string como parâmetro para uma função, ela não será um clone, então quando a função terminar a string não estará mais disponível.

- Quando utilizamos um literal como parâmetro, ela realiza um clone então estará disponível após a função.

- Link do próximo código.

Ownership e funções

```
fn main() {
        let s = String::from("hello");  // s entra no escopo

        takes_ownership(s);         // move o valor de s para a função…
                                    // ... então s não é mais válido aqui

        let x = 5;                  // x entra no escopo

        makes_copy(x);              // x poderia mover para a função,
                                    // mas i32 é Cópia, então está ok
                                    // usar x depois daqui

} // Aqui, x sai do escopo, e também s. Contudo, porque o valor de s foi movido, nadade
  // de especial acontece.

fn takes_ownership(some_string: String) { // some_string entra no escopo
        println!("{}", some_string);
} // Aqui, some_string sai do escopoe e `drop` é chamado. A memória
  // é liberada.

fn makes_copy(some_integer: i32) { // some_integer entra no escopo
        println!("{}", some_integer);
} // Aqui, some_integer sai do escpo. Nada especial acontece.
```

Escopo e retorno de valores

- Podemos utilizar o retorno das funções, para ter novamente os valores das strings.
- Podemos utilizar uma tupla, para retornar diversos valores de tipos diferentes.
- Link do código.

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len();
    // len() retorna o compriment da string

    (s, length)
}
```

Referências e Borrowing

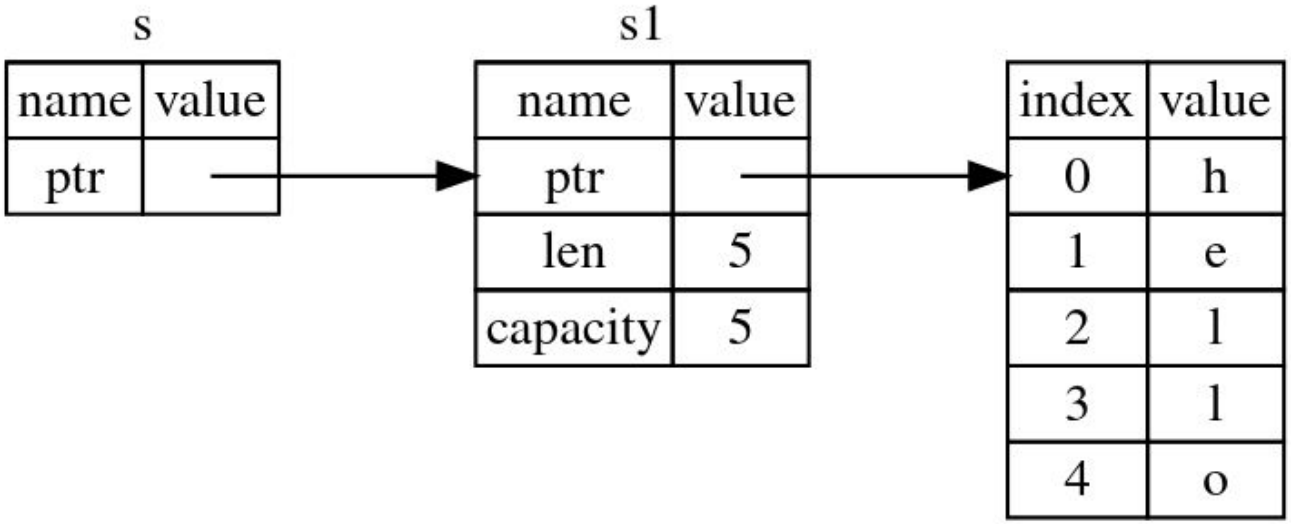
- No exemplo de código anterior, mostra a criação de uma nova variável, para armazenar o mesmo valor que foi passado para a função.

- Podemos também resolver esse problema de outra forma, com a utilização de referências, dado por “&variável”.

```
fn main() {
    let s1 = String::from("texto");

    let tamanho = calcula_tamanho(&s1);

    println!("O tamanho de '{}' é {}.", s1, tamanho);
}

fn calcula_tamanho(s: &String) -> usize {
    s.len()
}
```

Referências e Borrowing

- Quando passamos uma referência, no final da função, “s” será descartado, mantendo s1 e o valor.
- Chamamos essa ação de Borrowing ou Emprestar, já que passando a referência não fará a função ser dono ou ownership do valor.



Referências mutáveis

- Não conseguimos modificar o valor de uma variável imutável, utilizando a referência, como mostra o exemplo 1.
- Podemos modificar o valor de uma variável mutável. Como no exemplo 2

Exemplo 2 :

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Exemplo 1:

```
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Referências mutáveis

- Porém existe uma limitação. Não podemos utilizar duas variáveis mutáveis em sequência, antes de devolver.

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &mut s;
    let r2 = &mut s;

    println!("{}, {}", r1, r2);
}
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here

For more information about this error, try `rustc --explain E0499`.
error: could not compile `ownership` due to previous error
```

Referências mutáveis

- Se a primeira referência for chamada dentro de um escopo e a segunda referência for chamada fora, não terá problema.

```
let mut s = String::from("hello");

{
      let r1 = &mut s;
} // r1 sai do escopo aqui, então podemos fazer uma nova referência.

let r2 = &mut s;
```

Referências mutáveis

- Podemos criar diversas referências imutáveis ao mesmo tempo.
- Não podemos criar uma referência mutável antes da referência imutável devolver.

```
let mut s = String::from("hello");

let r1 = &s; // sem problemas
let r2 = &s; // sem problemas
let r3 = &mut s; // PROBLEMA

println!("{}, {}, and {}", r1, r2, r3);
```

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s; // no problem
  |              -- immutable borrow occurs here
5 |     let r2 = &s; // no problem
6 |     let r3 = &mut s; // BIG PROBLEM
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `ownership` due to previous error
```

Referências mutáveis

- O próximo exemplo mostra a utilização da referência mutável depois que as referências imutáveis devolvem o valor.

```
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{} and {}", r1, r2);
    // variables r1 and r2 will not be used after this point

    let r3 = &mut s; // no problem
    println!("{}", r3);
}
```

Referências soltas

- O Rust garante que não vai existir nenhuma referência solta, isso ocorre quando um ponteiro referência um dado que não existe mais.

```
fn dangle() -> &String {   // dangle retorna a referência para uma String

    let s = String::from("hello"); // s é uma nova String

    &s   // retorna a referência para a string s
}   // Aqui, s sai do escopo, e é eliminada. A memória é liberada.
    // Perigo!
```

Referências soltas

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:16
  |
5 | fn dangle() -> &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
  |
5 | fn dangle() -> &'static String {
  |                +++++++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `ownership` due to previous error
```

Tipo Slice

- Primeiro vamos fazer explicar um programa sem Slice, para explicar o problema que ele resolve.
- let bytes = s.as_bytes(); Transforma “s” em um array.
- iter é um método que retorna cada elemento em uma coleção
- enumerate encapsula o resultado do iter e retorna cada elemento como parte de uma tupla
- O primeiro elemento da tupla é um índice e o segundo uma referência ao valor

Example:

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

Tipo Slice

- for (i, &item) onde “i” é o índice e “&item” é a referência ao valor.
- Se encontrar um espaço, quer dizer que a primeira palavra termina, então retorna o índice.
- Se não encontrar espaço, retorna o tamanho.
- Nota que o valor retornado é um usize.

Example:

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

Tipo Slice

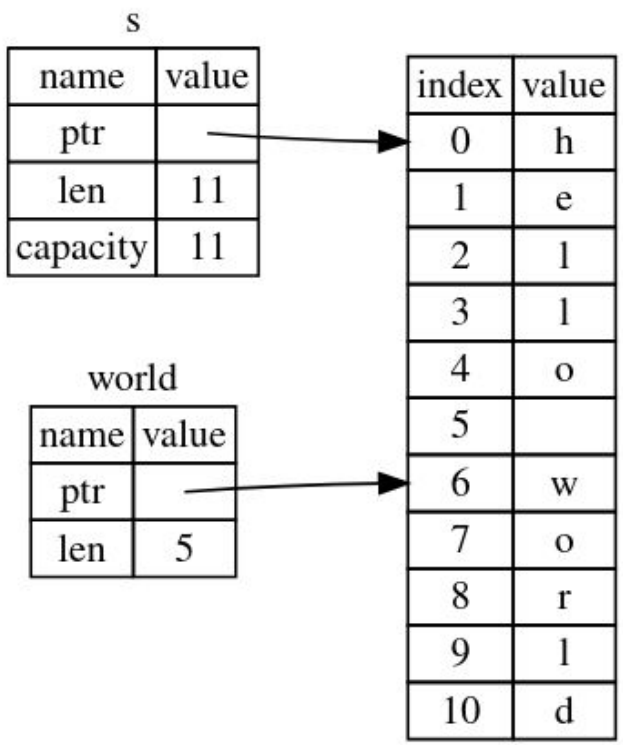
- O valor retornado não garante que ele seja válido no futuro, caso s seja modificado.
- s.clear() esvazia a string.
- a variável word ainda armazena o valor 5, que perde seu propósito, assim que s é esvaziada.
- O código compila, porém existe um bug, já que o valor de word ainda pode ser utilizado.
- Link do código.

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word irá receber o valor 5

    s.clear(); // isto esvazia a String, fazendo ela ser igual a ""

    // word continua ter o valor 5 aqui, mas não há mais string lá
    // não podemos usar corretamente o valor 5!
}
```

Slice de string

- Podemos resolver esse problema, utilizando o tipo Slice.
- Um Slice armazena a posição inicial do índice e o tamanho.
- “[0..5]” sinaliza a posição do índice “[início..fim]”.
- O tamanho do Slice corresponde a: fim - início
- A imagem mostra a referência do Slice.

```
fn main() {
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
}
```



Slice de string

- Podemos representar o Slice de algumas formas diferentes
  - &s[0..2]; é equivalente a &s[..2];
  - &s[3..len]; é equivalente a &s[3..]
  - &s[0..len]; é equivalente a &s[..];
- Temos o função first_word, dessa vez utilizando Slice.
  - &str é o tipo que representa o Slice de string.

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Slice de string

- Utilizando Slice, o Compilador do rust identifica a utilização do Slice, após o s.clear() como um erro.
- Isso impede o bug e permite sabermos dele antes, em tempo de compilação. Como podemos perceber com o código ao lado.
- Como o método clear precisa de uma referência mutável.
- O println! utiliza uma referência imutável.
- Não é possível existir as duas referências simultaneamente.

```
fn first_word(s: &String) -> &str {
        let bytes = s.as_bytes();

        for (i, &item) in bytes.iter().enumerate() {
                if item == b' ' {
                return &s[0..i];
                }
        }

        &s[..]
}

fn main() {
        let mut s = String::from("hello world");

        let word = first_word(&s);

        s.clear(); // error!

        println!("the first word is: {}", word);
}
```

Strings Literais são Slice

```
let s = "Hello, world!";
```

- s é do tipo &str, ou seja é um slice.
- Literais são armazenados dentro do binário.
- O Slice aponta para uma ponto específico do binário .
- Como &str é imutável, então Strings Literais também são imutáveis.

Slice de string como parâmetro

- Podemos utilizar o formato do exemplo 1, onde passamos uma referência da string.
- Podemos utilizar o formato do exemplo 2, onde passamos um slice da string.
- No segundo exemplo, poderemos passar um slice ou string.

Exemplo 1:

```
fn first_word(s: &String) -> &str {
```

Exemplo 2:

```
fn first_word(s: &str) -> &str {
```

Outros slice

- Podemos utilizar slice, para arrays também.
- O tipo desse slice é &[i32].
- Funciona da mesma maneira que o slice de string.

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

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);
```

Referências

- https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html