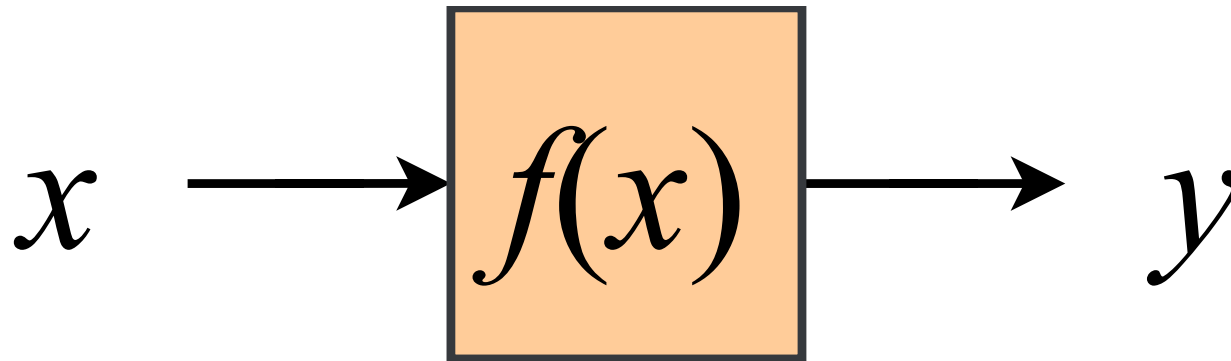


# Funções

# Funções

- Podemos modularizar nosso código através de **funções**, onde dividimos podemos dividir nossos programas em diversos componentes



# Funções

- Em Rust, como já vimos a função `main()`, declaramos funções da seguinte forma:

```
fn NOME_DA_FUNCAO(param1: TIPO, param2: TIPO) -> RETORNO {  
    // Código  
}
```

# Funções

- Declarando uma função que retorna a soma de dois números inteiros

```
fn soma_inteiros(n1: i32, n2: i32) -> i32 {  
    n1 + n2  
}
```

# Funções

- Quando criamos uma função e **esperamos um valor de retorno**, podemos especificar o valor retornado de duas formas diferentes:
  - Através da diretiva **return**
  - Chamar o valor de retorno na última linha do escopo da função sem ponto e vírgula (;)

# Funções

Última linha sem ponto-e-vírgula (;)

- Declarando uma função que retorna a soma de dois números inteiros

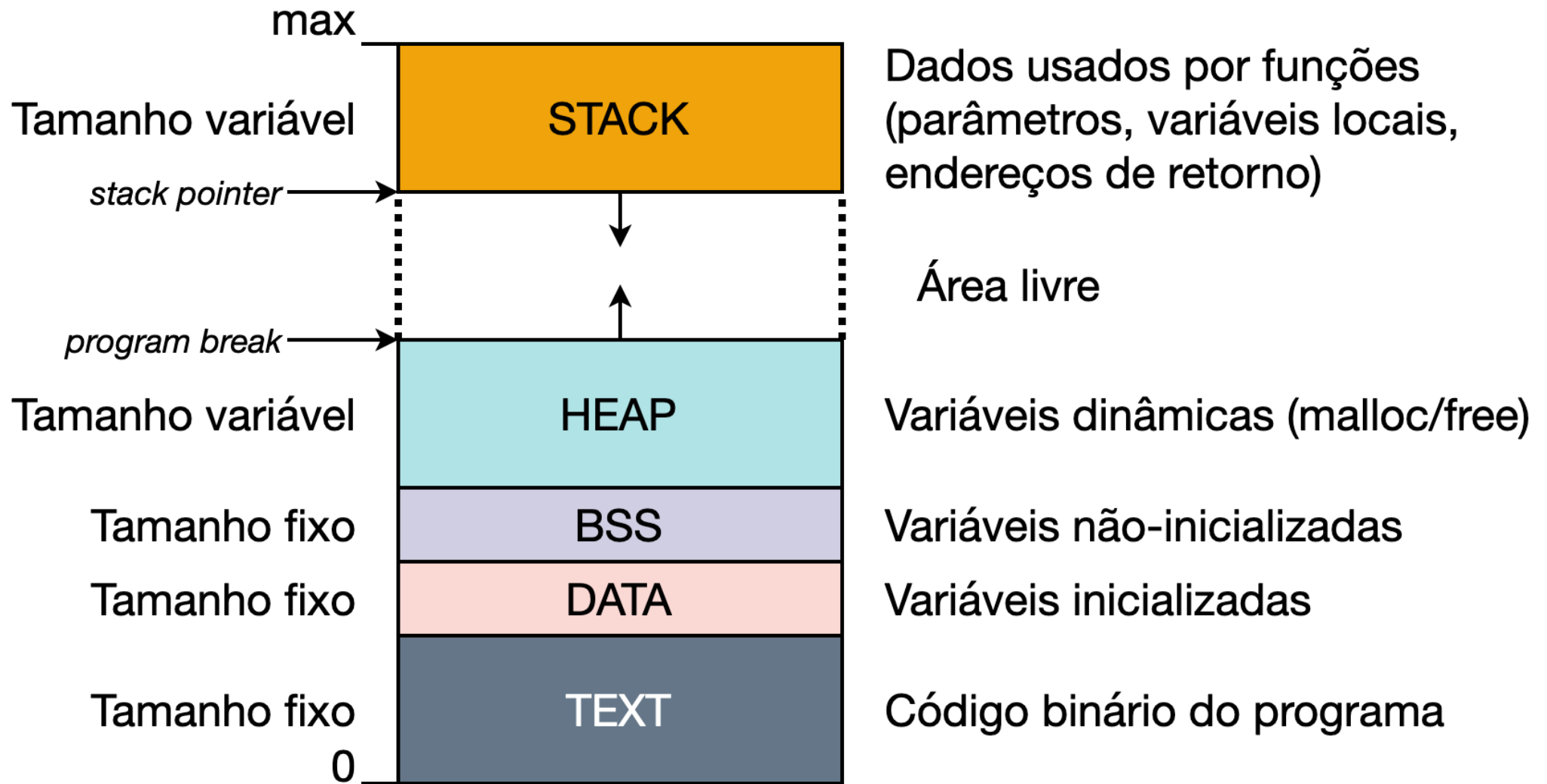
```
fn soma_inteiros(n1: i32, n2: i32) -> i32 {  
    n1 + n2  
}
```

# Funções

## Diretiva `return`

- Declarando uma função que retorna a soma de dois números inteiros

```
fn soma_inteiros(n1: i32, n2: i32) -> i32 {  
    return n1 + n2  
}
```

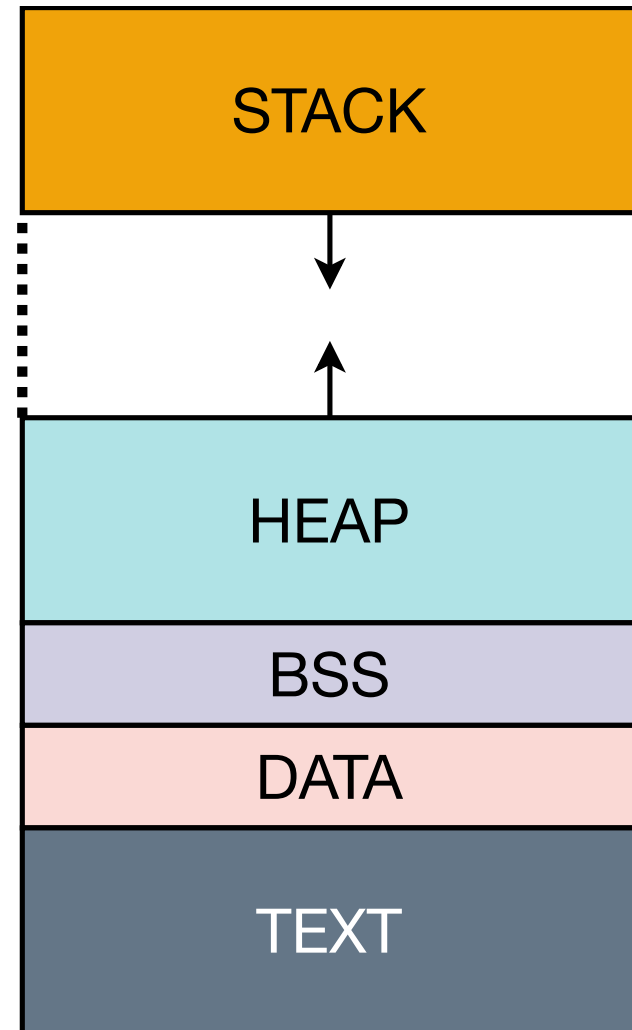




# Funções

## Estrutura na memória

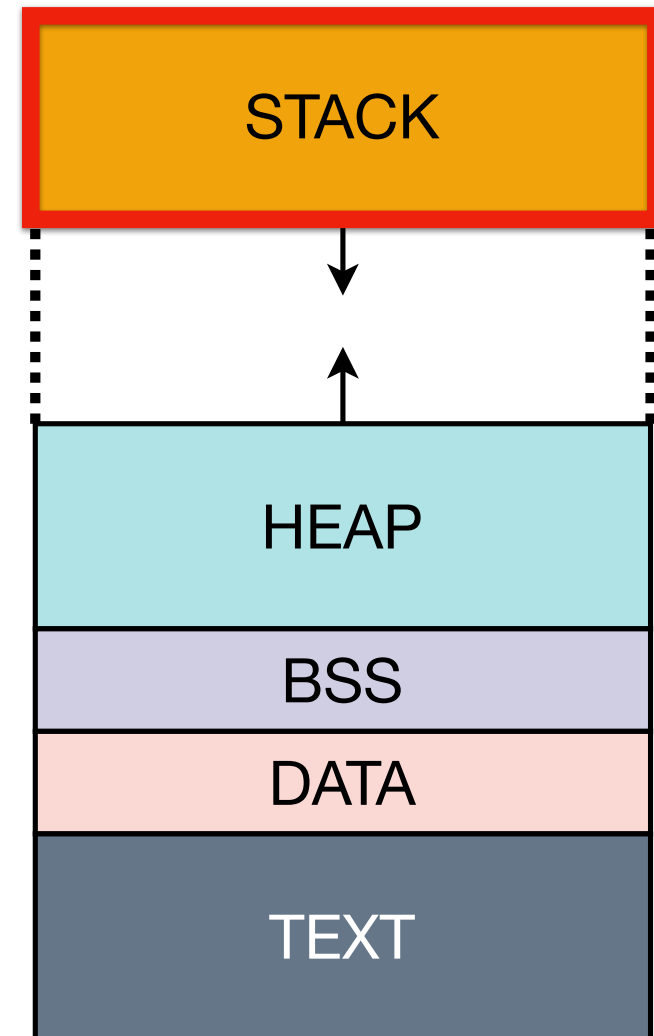
- Temos duas regiões de memória principais onde nosso código estará manipulando dados
  - Stack
  - Heap



# Funções

## Stack

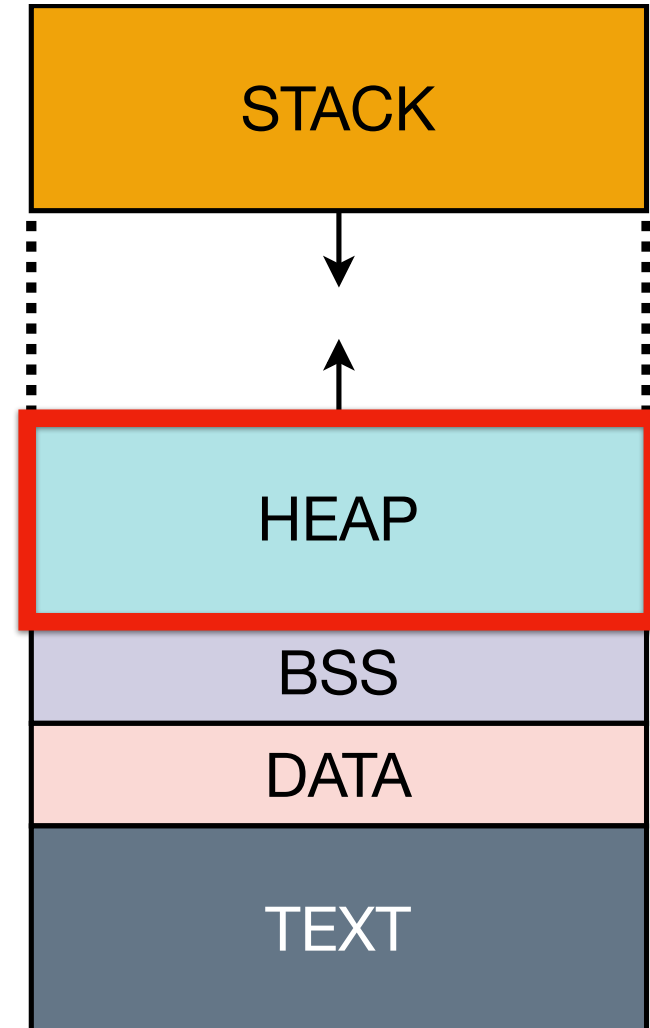
- Utilizada comumente para armazenar informações relacionadas à chamadas de funções
- Rápida, porém possui tamanho limitado
- Armazena informações como:
  - Valores dos parâmetros
  - Endereço de retorno à quem chamou a função
  - Variáveis locais da função



# Funções

## Heap

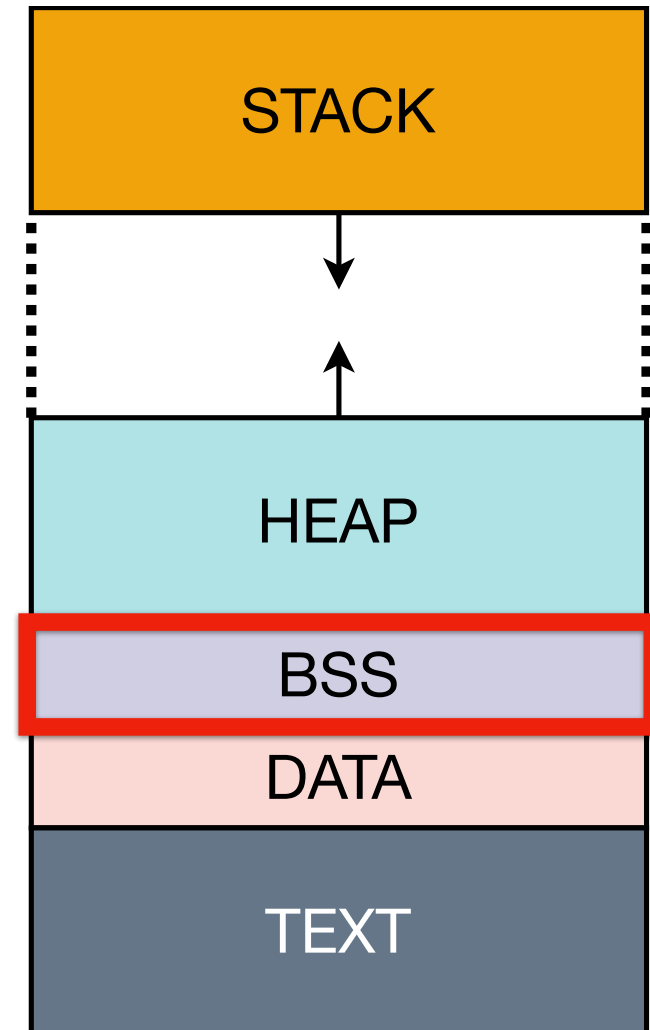
- Utilizada pelo programa para alocar memória dinamicamente
- É mais lenta que a stack, pois os dados geralmente não estão organizados sequencialmente
- Tamanho ilimitado



# Funções

## Block Started by Symbol (BSS)

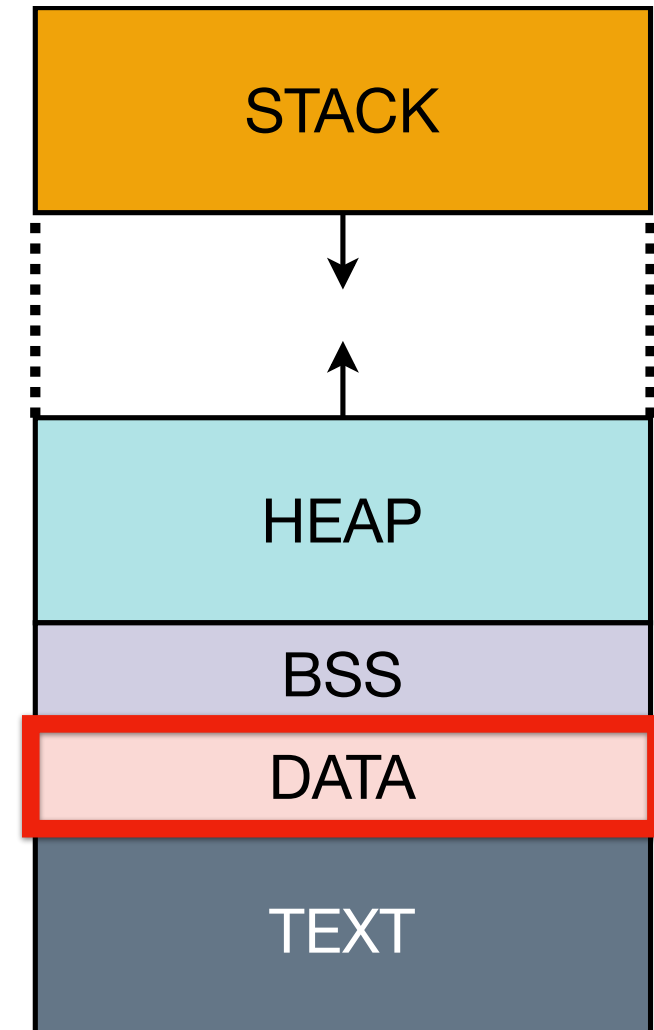
- Destinada às variáveis não inicializadas



# Funções

## Data

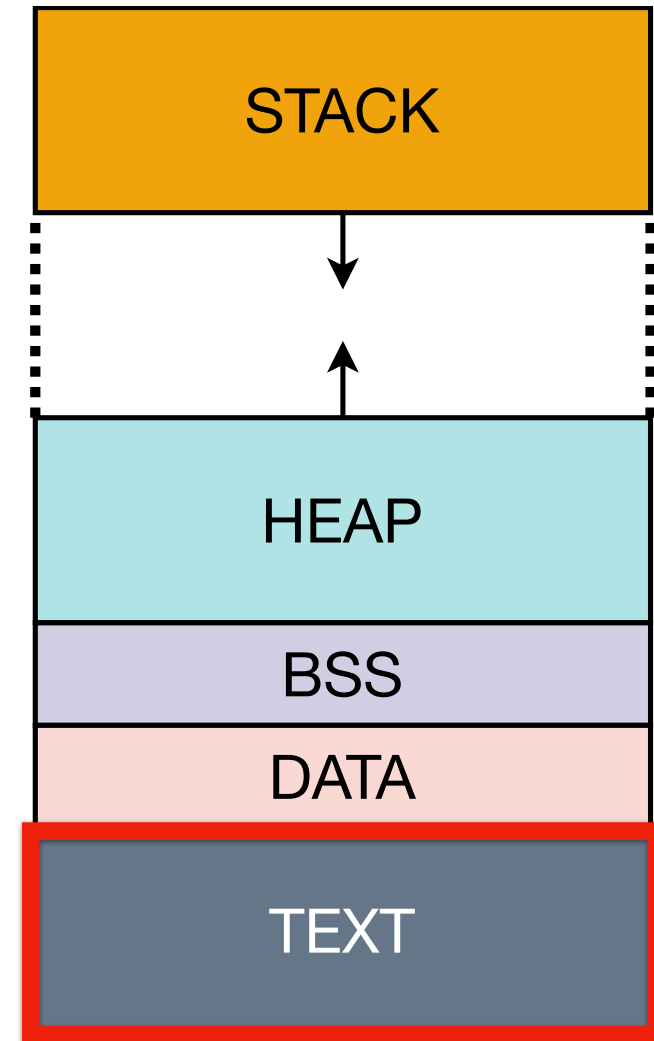
- Destinada às variáveis inicializadas



# Funções

## Text

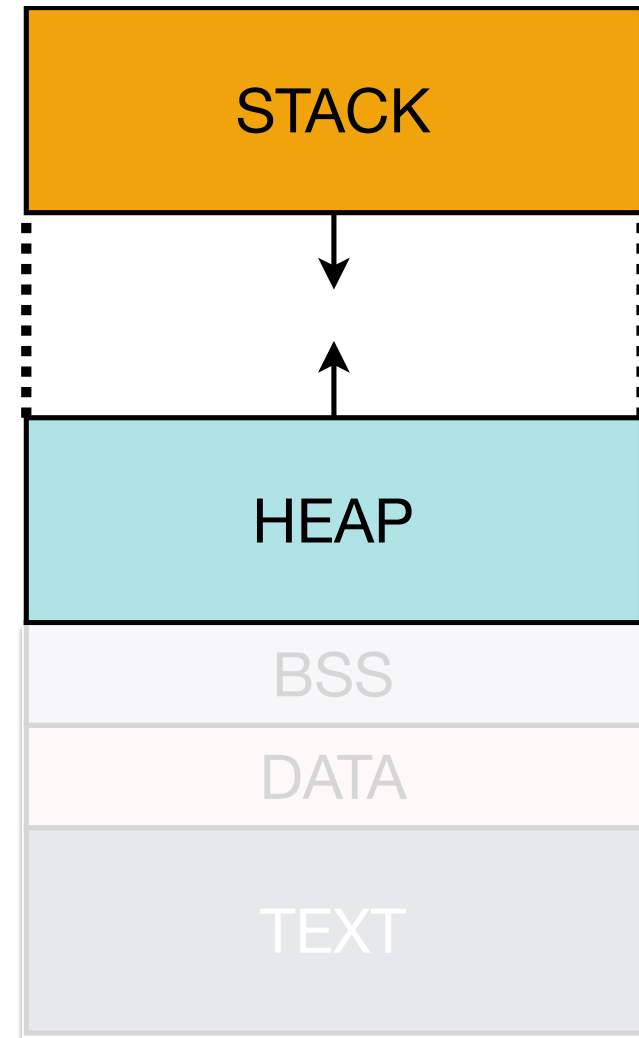
- Espaço reservado (somente leitura) para as instruções do programa em linguagem de máquina



# Funções

## Estrutura na memória

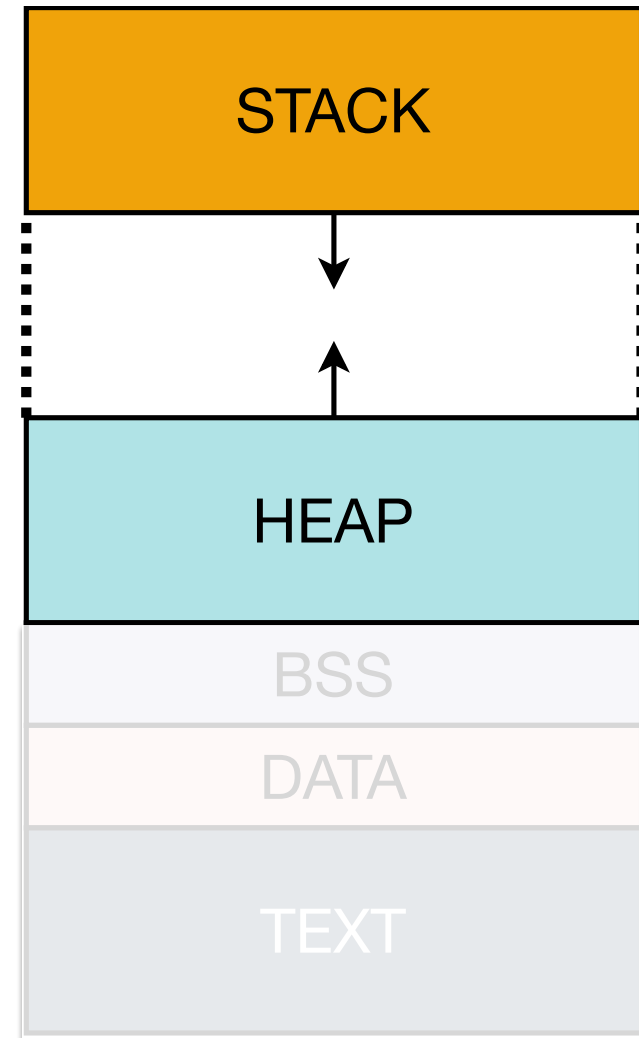
- Durante a disciplina estaremos focando nessas estruturas principais (Stack e Heap)



# Funções

## Estrutura na memória

- O que acontece “nos bastidores” quando chamamos uma função?





# Funções (sem argumentos)

## Estrutura na memória


```
fn pilha() {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha();  
}
```

**Stack (pilha)**

Endereço	Nome	Valor

# Funções (sem argumentos)

## Estrutura na memória

```
fn pilha() {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;   
  
    pilha();  
}
```

Stack (pilha)

Endereço	Nome	Valor
0	a	20

# Funções (sem argumentos)

## Estrutura na memória

```
fn pilha() {  
    let b = 30;  
    let c = 40;  
}
```

```
fn main() {  
    let a = 20;
```

```
    pilha();  
}
```




Stack (pilha)

Endereço	Nome	Valor
0	a	20

# Funções (sem argumentos)

## Estrutura na memória


```
fn pilha() {  
    let b = 30;   
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha();  
}
```

Stack (pilha)

Endereço	Nome	Valor
1	b	30
0	a	20

# Funções (sem argumentos)

## Estrutura na memória

```
fn pilha() {  
    let b = 30;  
    let c = 40;   
}  
  
fn main() {  
    let a = 20;  
  
    pilha();  
}
```

Stack (pilha)

Endereço	Nome	Valor
2	c	40
1	b	30
0	a	20

# Funções (sem argumentos)

## Estrutura na memória

```
fn pilha() {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha();  
}
```

← Fim da função  
pilha

Stack (pilha)

Endereço	Nome	Valor
2	c	40
1	b	30
0	a	20

# Funções (sem argumentos)

## Estrutura na memória

```
fn pilha() {  
    let b = 30;  
    let c = 40;  
}
```

```
fn main() {  
    let a = 20;  
  
    pilha();  
}
```

← Fim do programa

Stack (pilha)

Endereço	Nome	Valor
0	a	20

# Funções (sem argumentos)

## Estrutura na memória

```
fn pilha() {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha();  
}
```

**Stack (pilha)**

Endereço	Nome	Valor



# Funções (com argumentos)

## Estrutura na memória


```
fn pilha(d: i32) {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha(50);  
}
```

Stack (pilha)

Endereço	Nome	Valor

# Funções (com argumentos)

## Estrutura na memória

```
fn pilha(d: i32) {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;   
  
    pilha(50);  
}
```

Stack (pilha)

Endereço	Nome	Valor
0	a	20

# Funções (com argumentos)

## Estrutura na memória

```
fn pilha(d: i32) {  
    let b = 30;  
    let c = 40;  
}
```

```
fn main() {  
    let a = 20;
```

```
    pilha(50);  
}
```

Stack (pilha)

Endereço	Nome	Valor
0	a	20

# Funções (com argumentos)

## Estrutura na memória

```
fn pilha(d: i32) { ←  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha(50);  
}
```


Stack (pilha)

Endereço	Nome	Valor
1	d	50
0	a	20

# Funções (com argumentos)

## Estrutura na memória

```
fn pilha(d: i32) {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha(50);  
}
```




Stack (pilha)

Endereço	Nome	Valor
2	b	30
1	d	50
0	a	20

# Funções (com argumentos)

## Estrutura na memória

```
fn pilha(d: i32) {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha(50);  
}
```



Stack (pilha)

Endereço	Nome	Valor
3	c	40
2	b	30
1	d	50
0	a	20

# Funções (com argumentos)

## Estrutura na memória

```
fn pilha(d: i32) {  
    let b = 30;  
    let c = 40;  
}
```

```
fn main() {  
    let a = 20;  
  
    pilha(50);  
}
```

← Fim da função  
pilha

Stack (pilha)

Endereço	Nome	Valor
3	c	40
2	b	30
1	d	50
0	a	20

# Funções (com argumentos)

## Estrutura na memória

```
fn pilha(d: i32) {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha(50);  
}
```

Stack (pilha)

Endereço	Nome	Valor
0	a	20

← Fim do programa



# Funções (com argumentos)

## Estrutura na memória

```
fn pilha(d: i32) {  
    let b = 30;  
    let c = 40;  
}  
  
fn main() {  
    let a = 20;  
  
    pilha(50);  
}
```

Stack (pilha)

Endereço	Nome	Valor

# Funções

## Passagem de parâmetros/argumentos

- Quando especificamos os parâmetros de uma função, existem dois mecanismos que podemos aplicar
  - Cópia de valores
  - Referência à valores
- A maioria das linguagens de programação permitem a passagem de parâmetros **copiando** valores

# Funções

## Passagem de parâmetros/argumentos - Cópia de valores

- Nesse mecanismo os parâmetros são alocados como variáveis locais dentro do escopo da função, de modo que os valores passados como parâmetros sejam **copiados** às variáveis de parâmetro
- Vejamos o exemplo a seguir:

```
fn dobro(i: i32) -> i32 {  
    i * 2  
}
```

# Funções

## Passagem de parâmetros - Cópia de valores

```
fn dobro(i: i32) -> i32 {  
    i * 2  
}  
  
fn main() {  
→ let x = 3;  
  let y = dobro(x);  
  println!("Dobro de {} é {}", x, y);  
  // Dobro de 3 é 6  
}
```

Stack (pilha)

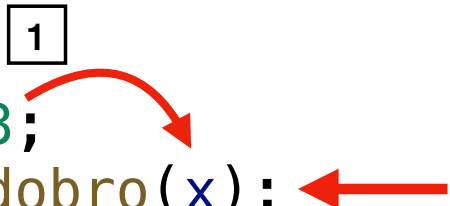
Endereço	Nome	Valor
0	x	3

# Funções

## Passagem de parâmetros - Cópia de valores

```
fn dobro(i: i32) -> i32 {  
    i * 2  
}
```

```
fn main() {  
    let x = 3;  
    let y = dobro(x);  
    println!("Dobro de {} é {}", x, y);  
    // Dobro de 3 é 6  
}
```



Stack (pilha)

Endereço	Nome	Valor
0	x	3

# Funções

## Passagem de parâmetros - Cópia de valores

```
fn dobro(i: i32) -> i32 { ←
```

*(A arrow points to the opening curly brace of the function definition)*

```
    i * 2  
}
```

```
fn main() { 1  
    let x = 3;  
    let y = dobro(x);  
    println!("Dobro de {} é {}", x, y);  
    // Dobro de 3 é 6  
}
```

*(A curved arrow points from the boxed '1' to the function call 'dobro(x)')  
(Another curved arrow points from the '3' in 'let x = 3;' to the 'x' in 'dobro(x)')  
(A third curved arrow points from the '3' in 'println!' to the 'x' in 'x, y')  
(A fourth curved arrow points from the '6' in 'println!' to the 'y' in 'x, y')*

Stack (pilha)

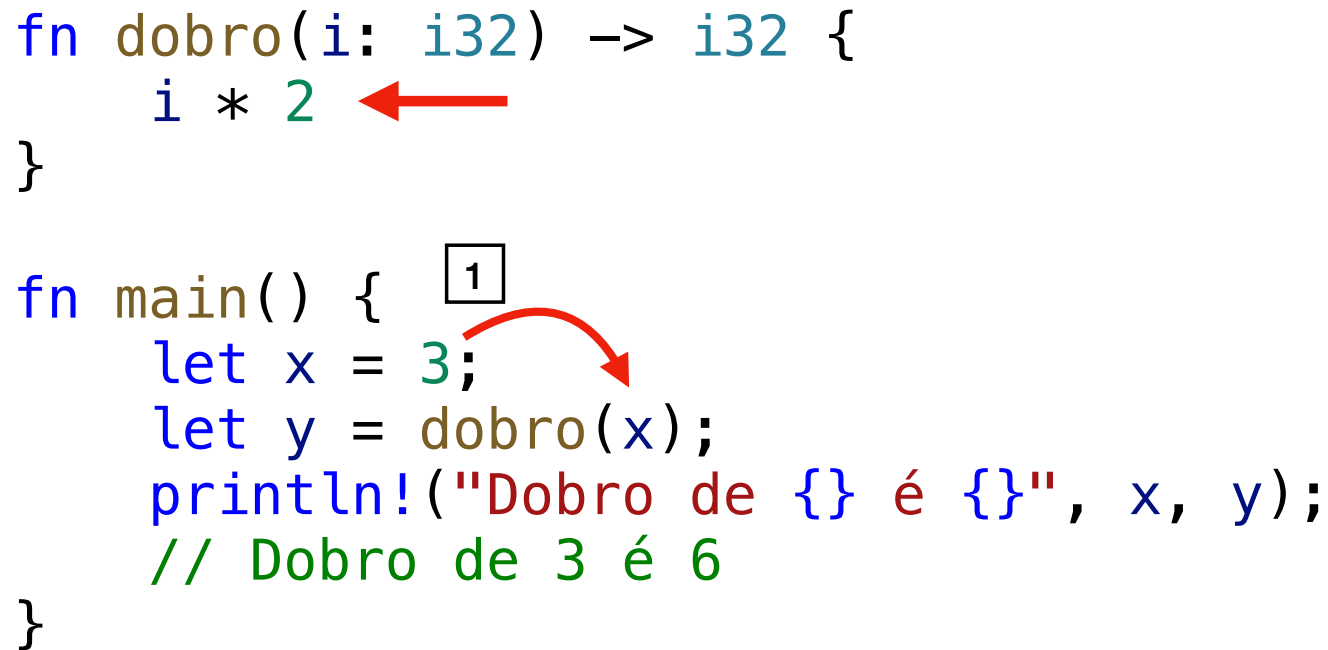
Endereço	Nome	Valor
1	i	3
0	x	3

*(A red arrow points to the '3' in the 'Valor' column for 'i')  
(Another red arrow points to the '3' in the 'Valor' column for 'x')*

# Funções

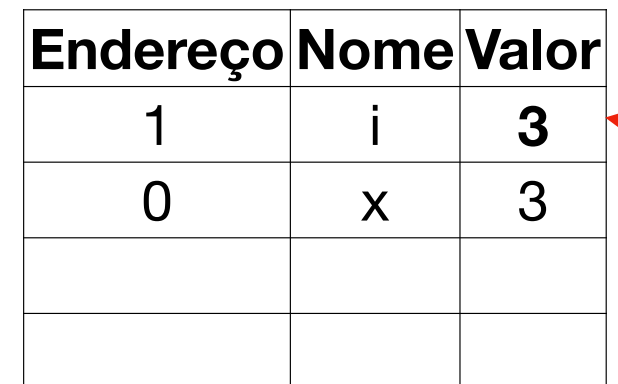
## Passagem de parâmetros - Cópia de valores

```
fn dobro(i: i32) -> i32 {  
    i * 2  
}  
  
fn main() {  
    let x = 3;  
    let y = dobro(x);  
    println!("Dobro de {} é {}", x, y);  
    // Dobro de 3 é 6  
}
```



Stack (pilha)

Endereço	Nome	Valor
1	i	3
0	x	3



# Funções

## Passagem de parâmetros - Cópia de valores

```
fn dobro(i: i32) -> i32 {  
    i * 2  
} ←
```

```
fn main() {  
    let x = 3;  
    let y = dobro(x);  
    println!("Dobro de {} é {}", x, y);  
    // Dobro de 3 é 6  
}
```

Diagram illustrating the call to `dobro(x)` in `main`. A box labeled `1` is placed above the opening curly brace of `main`. A red arrow points from this box to the argument `x` in the function call `dobro(x)`.

Stack (pilha)

Endereço	Nome	Valor
0	x	3



# Funções

## Passagem de parâmetros - Cópia de valores

```
fn dobro(i: i32) -> i32 {  
    i * 2  
}  
  
fn main() {  
    let x = 3;  
    let y = dobro(x);  
    println!("Dobro de {} é {}", x, y);  
    // Dobro de 3 é 6  
}
```

Diagram illustrating the call to the `dobro` function from `main`. A box labeled **1** is next to the `main` function definition. A box labeled **2** is next to the line `let y = dobro(x);`. A red arrow points from `x` in `main` to the parameter `i` in `dobro`. Another red arrow points from the return value of `dobro(x)` to `y` in `main`.

Stack (pilha)

Endereço	Nome	Valor
1	y	6
0	x	3

# Funções

## Passagem de parâmetros - Cópia de valores

```
fn dobro(i: i32) -> i32 {  
    i * 2  
}  
  
fn main() {  
    let x = 3;  
    let y = dobro(x);  
    println!("Dobro de {} é {}", x, y);  
    // Dobro de 3 é 6  
}
```

Diagram illustrating the call to the `dobro` function from `main`. A red arrow shows the argument `x` (value 3) being passed to the parameter `i` of the `dobro` function. A box labeled '1' is next to the `main` function definition, and a box labeled '2' is next to the `println` statement. A red arrow points from the `println` statement to the `Stack (pilha)` table.

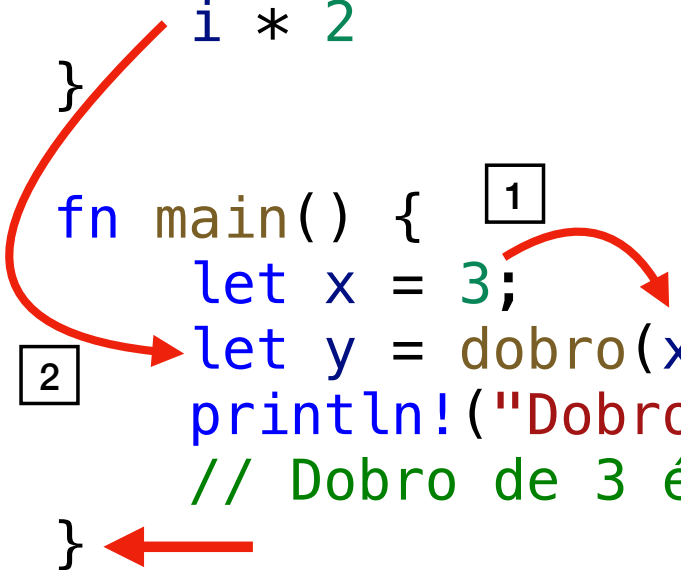
Stack (pilha)

Endereço	Nome	Valor
1	y	6
0	x	3

# Funções

## Passagem de parâmetros - Cópia de valores

```
fn dobro(i: i32) -> i32 {  
    i * 2  
}  
  
fn main() {  
    let x = 3;  
    let y = dobro(x);  
    println!("Dobro de {} é {}", x, y);  
    // Dobro de 3 é 6  
}
```



Stack (pilha)

Endereço	Nome	Valor

# Funções

## Exercícios

- Crie uma função que receba um vetor como parâmetro e retorne a sua média
- Crie uma função que calcule o fatorial de um número fornecido como parâmetro

# Funções

## Passagem de parâmetros - Referência à valores

- Aqui passamos uma **referência**, ou seja, a localização na memória daquela variável em específico. Acessamos **diretamente** a variável, ao invés de copiarmos o valor para uma nova variável.
- Uma referência à variável é análogo à um endereço na memória, ou seja, a sua exata localização.
- No caso anterior nós copiávamos valores, implicando em maior consumo de memória. Referência à valores pode nos ajudar a “economizar” recursos e a reutilizar variáveis por maiores períodos.

# Funções

## Passagem de parâmetros - Referência à valores

- Associado à isso, teremos dois operadores importantes em Rust:
  - `&`: Indica o endereço da memória (referência) à uma variável
  - `*` : Além de ser um operador aritmético (multiplicação), podemos utilizá-lo para acessarmos o valor do endereço da memória (referência)

# Funções

## Passagem de parâmetros - Referência à valores

- Vejamos o exemplo anterior:

```
fn dobro(i: i32) -> i32 {  
    i * 2  
}
```

# Funções

## Passagem de parâmetros - Referência à valores

```
fn dobro(i: i32) -> i32 {  
    i * 2  
}
```

```
fn main() {  
    let x = 3;  
    let y = dobro(x);  
    println!("Dobro de {} é {}", x, y);  
    // Dobro de 3 é 6  
}
```



# Funções

## Passagem de parâmetros - Referência à valores


```
fn dobro(i: &mut i32) {  
    *i *= 2  
}  
  
fn main() {  
    let mut x = 3;  
    print!("0 dobro de {} ", x);  
    dobro(&mut x);  
    println!("é {}", x);  
}
```

# Funções

## Passagem de parâmetros - Referência à valores

```
fn dobro(i: &mut i32) {  
    *i *= 2  
}
```

Parâmetro mutável (**mut**)  
que é uma referência à um valor (&)




```
fn main() {  
    let mut x = 3;  
    print!("O dobro de {} ", x);  
    dobro(&mut x);  
    println!("é {}", x);  
}
```

# Funções

## Passagem de parâmetros - Referência à valores


```
fn dobro(i: &mut i32) {  
    *i *= 2  
}  
  
fn main() {  
    let mut x = 3;  
    print!("0 dobro de {} ", x);  
    dobro(&mut x);  
    println!("é {}", x);  
}
```

Operador de dereferência  
(retorna o valor associado ao endereço)



# Funções

## Passagem de parâmetros - Referência à valores

```
fn dobro(i: &mut i32) {  
    *i *= 2  
}  
  
fn main() {  
    let mut x = 3;   
    print!("0 dobro de {} ", x);  
    dobro(&mut x);  
    println!("é {}", x);  
}
```

Stack (pilha)

Endereço	Nome	Valor
0	x	3

# Funções

## Passagem de parâmetros - Referência à valores


```
fn dobro(i: &mut i32) {  
    *i *= 2  
}  
  
fn main() {  
    let mut x = 3;  
    print!("0 dobro de {} ", x);  
    dobro(&mut x);  
    println!("é {}", x);  
}
```

Stack (pilha)

Endereço	Nome	Valor
0	x	3

# Funções

## Passagem de parâmetros - Referência à valores

```
fn dobro(i: &mut i32) {  
    *i *= 2  
}  
  
fn main() {  
    let mut x = 3;  
    print!("0 dobro de {} ", x);  
    dobro(&mut x);   
    println!("é {}", x);  
}
```

Stack (pilha)

Endereço	Nome	Valor
0	x	3

# Funções

## Passagem de parâmetros - Referência à valores

```
fn dobro(i: &mut i32) { ←
    *i *= 2
}


fn main() {
    let mut x = 3;
    print!("0 dobro de {} ", x);
    dobro(&mut x);
    println!("é {}", x);
}
```

Stack (pilha)

Endereço	Nome	Valor
1	i	→0
0	x	<b>3</b>

# Funções

## Passagem de parâmetros - Referência à valores

```
fn dobro(i: &mut i32) {  
    *i *= 2   
}  
  
fn main() {  
    let mut x = 3;  
    print!("0 dobro de {} ", x);  
    dobro(&mut x);  
    println!("é {}", x);  
}
```

Stack (pilha)

Endereço	Nome	Valor
1	i	→0
0	x	6



# Funções

## Passagem de parâmetros - Referência à valores


```
fn dobro(i: &mut i32) {  
    *i *= 2  
} ←  
  
fn main() {  
    let mut x = 3;  
    print!("0 dobro de {} ", x);  
    dobro(&mut x);  
    println!("é {}", x);  
}
```

Stack (pilha)

Endereço	Nome	Valor
0	x	6

# Funções

## Passagem de parâmetros - Referência à valores

```
fn dobro(i: &mut i32) {  
    *i *= 2  
}  
  
fn main() {  
    let mut x = 3;  
    print!("0 dobro de {} ", x);  
    dobro(&mut x);   
    println!("é {}", x);  
}
```

Stack (pilha)

Endereço	Nome	Valor
0	x	6

# Funções

## Passagem de parâmetros - Referência à valores

```
fn dobro(i: &mut i32) {  
    *i *= 2  
}  
  
fn main() {  
    let mut x = 3;  
    print!("0 dobro de {} ", x);  
    dobro(&mut x);  
    println!("é {}", x); ←  
}
```


Stack (pilha)

Endereço	Nome	Valor
0	x	6

# Funções

## Passagem de parâmetros - Referência à valores

```
fn dobro(i: &mut i32) {  
    *i *= 2  
}  
  
fn main() {  
    let mut x = 3;  
    print!("0 dobro de {} ", x);  
    dobro(&mut x);  
    println!("é {}", x);  
}
```



Stack (pilha)

Endereço	Nome	Valor

# Funções

## Exercícios

- Escreva uma função que receba dois valores (qualquer tipo), que sejam mutáveis e troque seus valores. Utilize referências para realizar a troca.
- Escreva uma função que calcule a média de um vetor de números inteiros. A função deve receber uma referência ao vetor como parâmetro e retornar a média.
- Crie uma função que verifique se uma palavra é um palíndromo, ou seja, ela é a mesma quando é lida de trás para frente. A função deve receber uma referência à uma string como parâmetro e retornar um booleano indicando se é um palíndromo ou não.