

# *Amarração de Nomes*

Estrutura de Linguagens

<https://github.com/fsantanna-uerj/EDL/>

Francisco Sant'Anna

[francisco@ime.uerj.br](mailto:francisco@ime.uerj.br)



# Amarração de Nomes

# Amarração de Nomes

- Associa ou vincula um nome ou identificador a uma entidade no programa

# Amarração de Nomes

- Associa ou vincula um nome ou identificador a uma entidade no programa
  - `int a = 1`

# Amarração de Nomes

- Associa ou vincula um nome ou identificador a uma entidade no programa
  - `int a = 1`
  - `print(a)`

# Amarração de Nomes

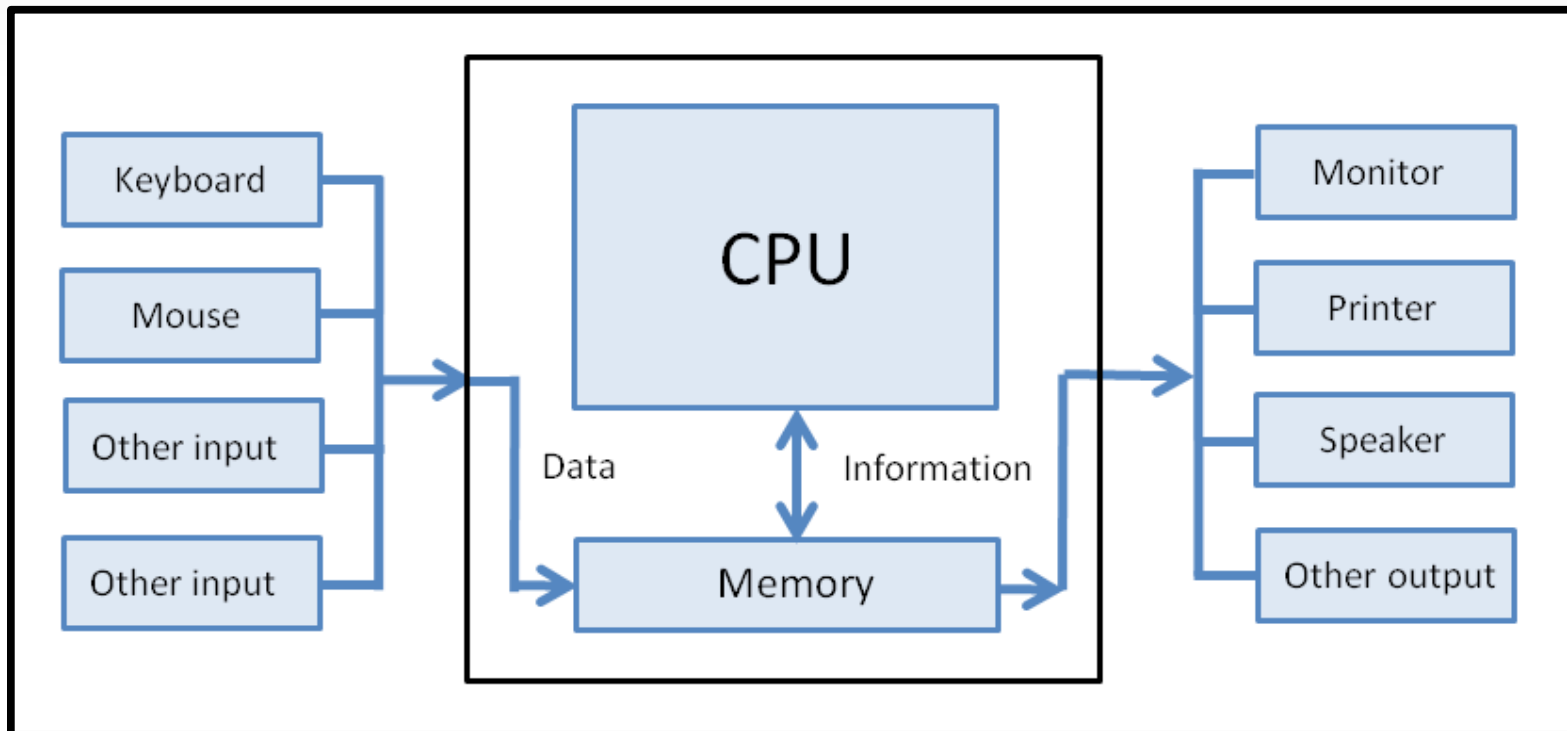
- Associa ou vincula um nome ou identificador a uma entidade no programa
  - `int a = 1`
  - `print(a)`
- Presente em todas as linguagens de alto nível

# Linguagem como Abstração

```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```

# Linguagem como Abstração

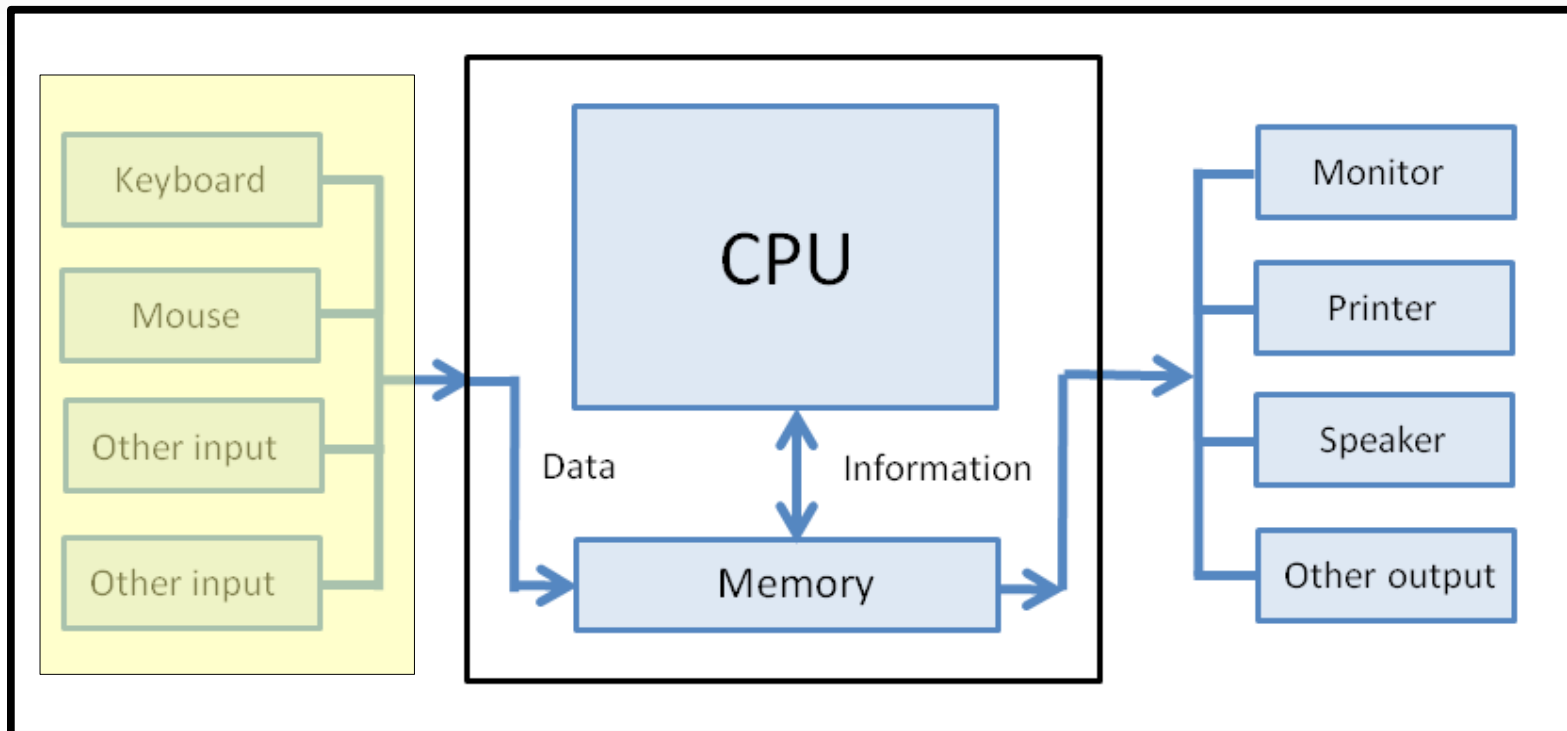
```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```





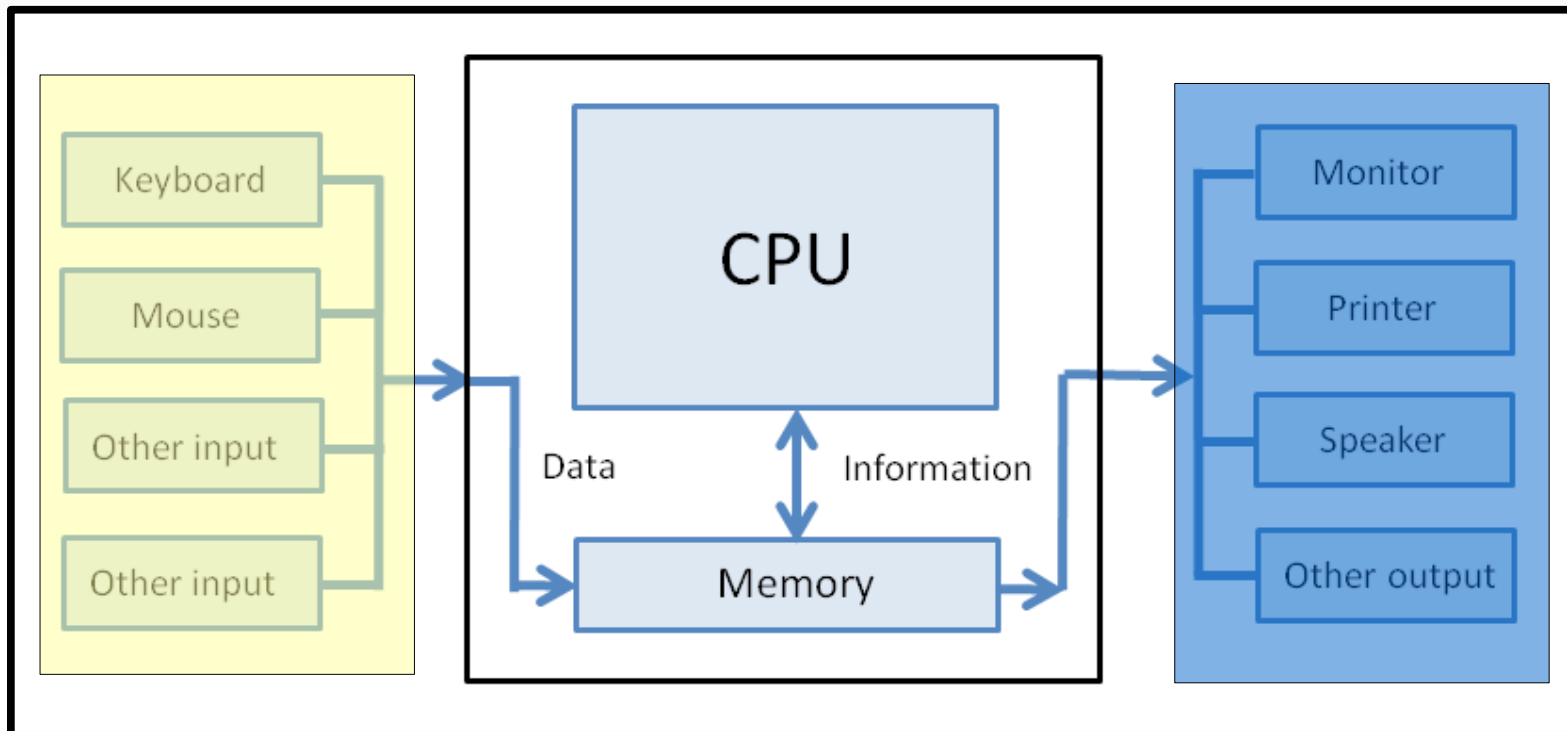
# Linguagem como Abstração

```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```



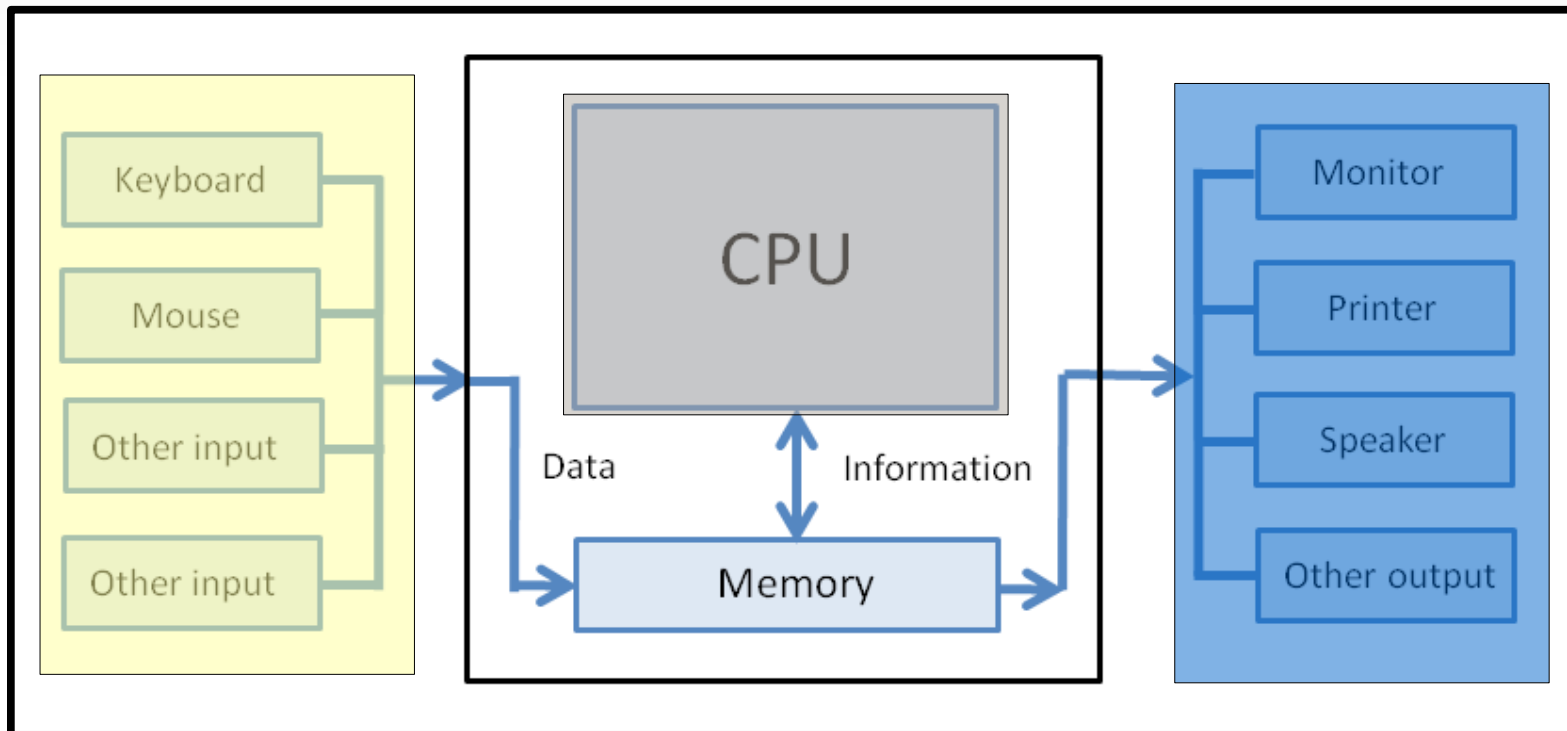
# Linguagem como Abstração

```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```



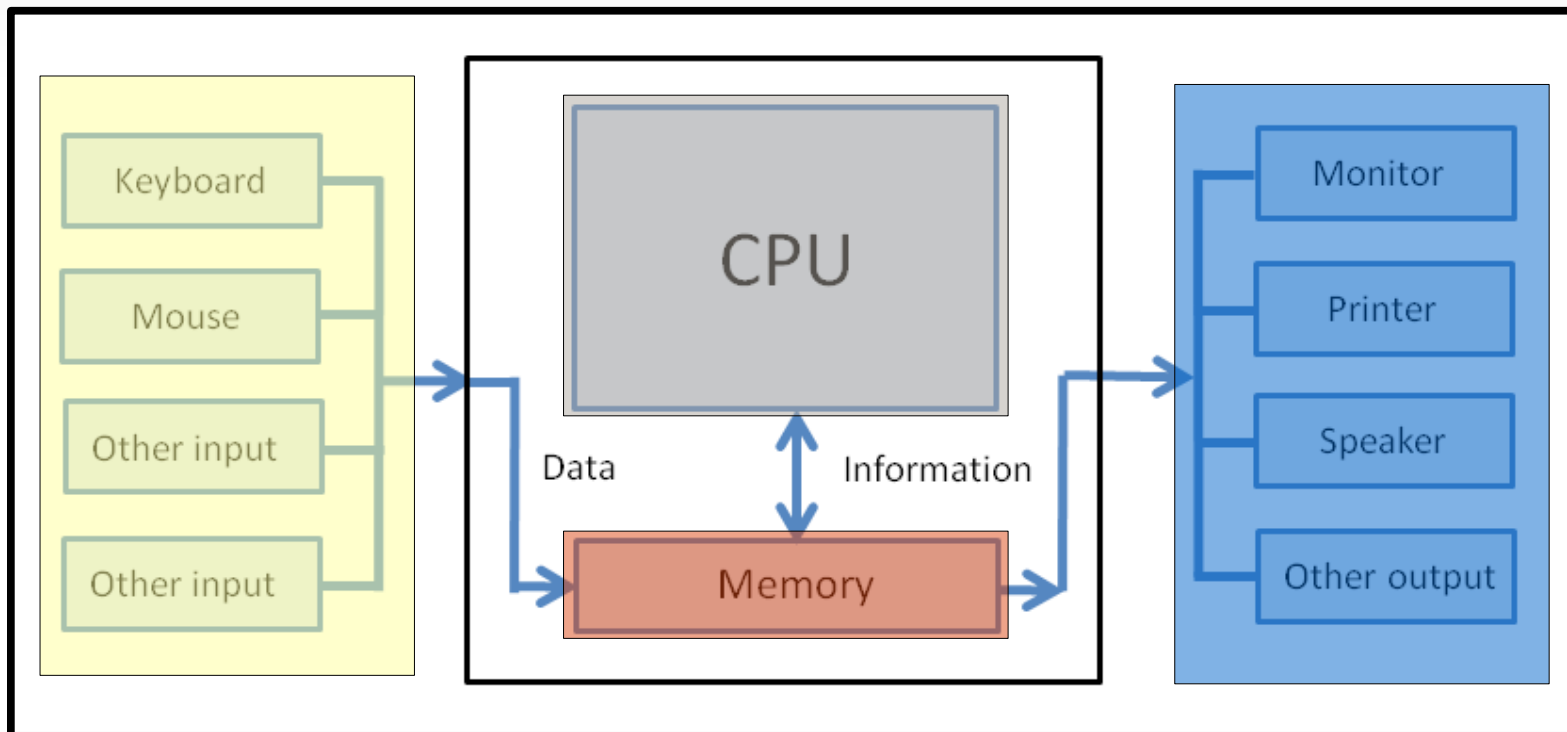
# Linguagem como Abstração

```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```



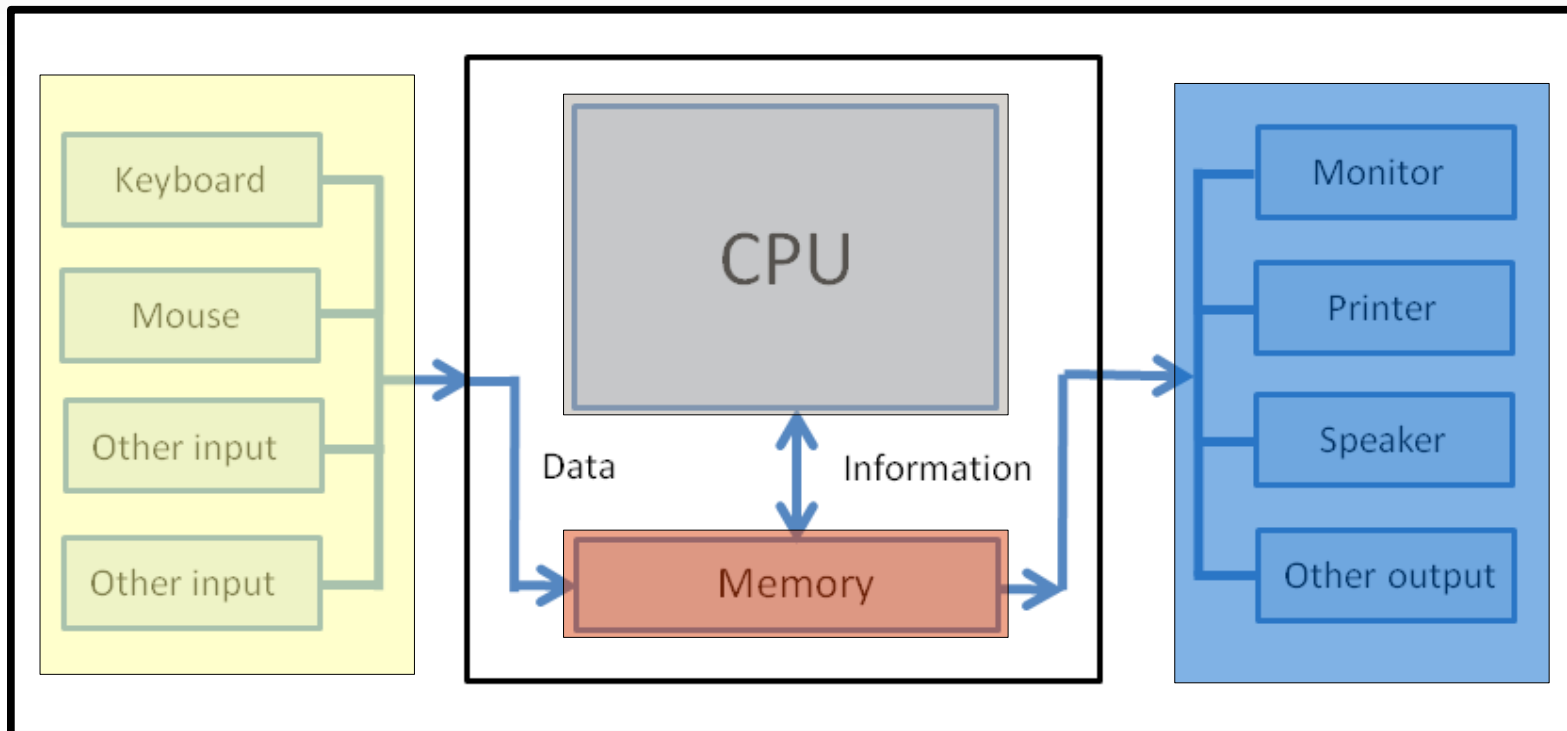
# Linguagem como Abstração

```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```



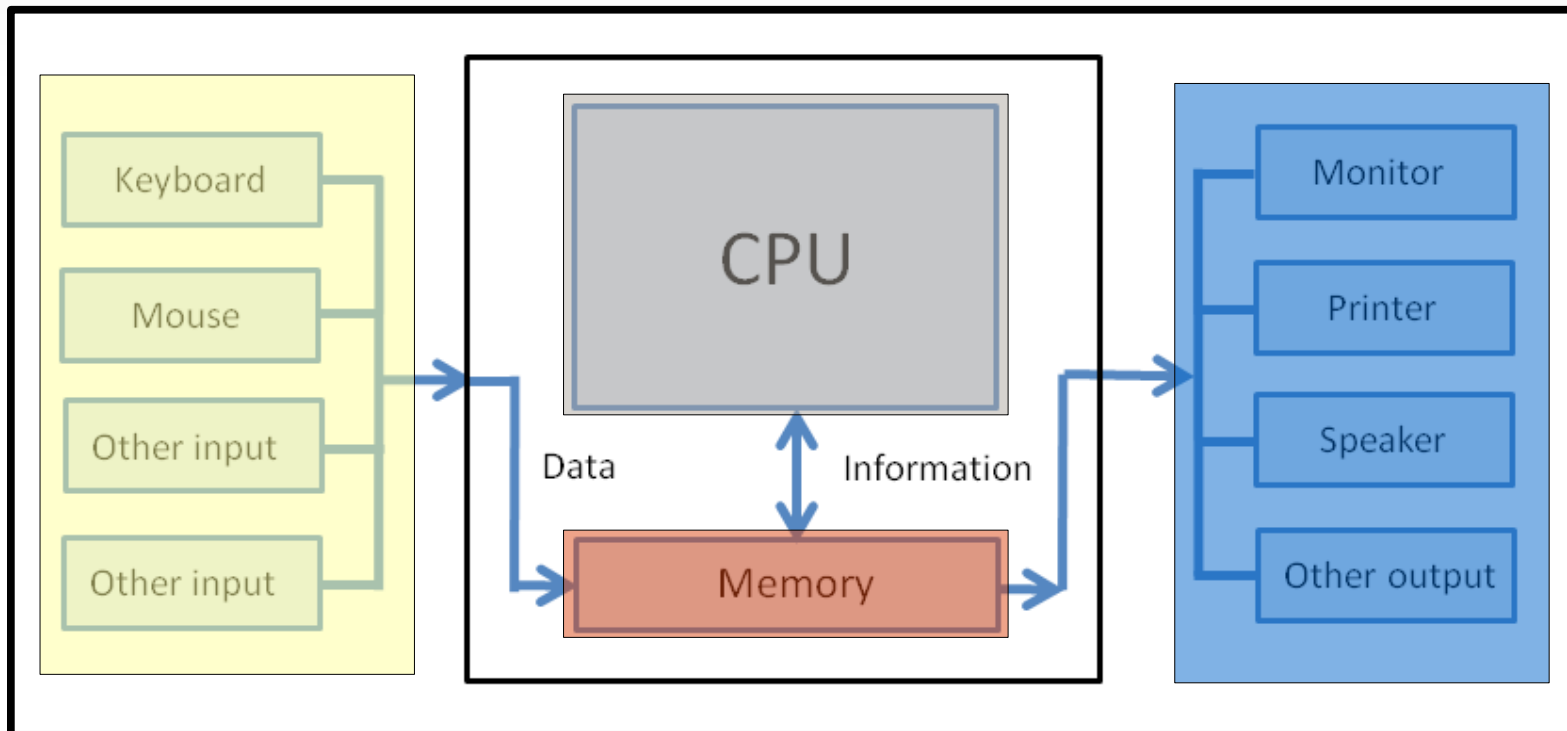
# Linguagem como Abstração

```
frase = input()  
print("----")  
for i in range(1,5):  
    print(i, frase)
```



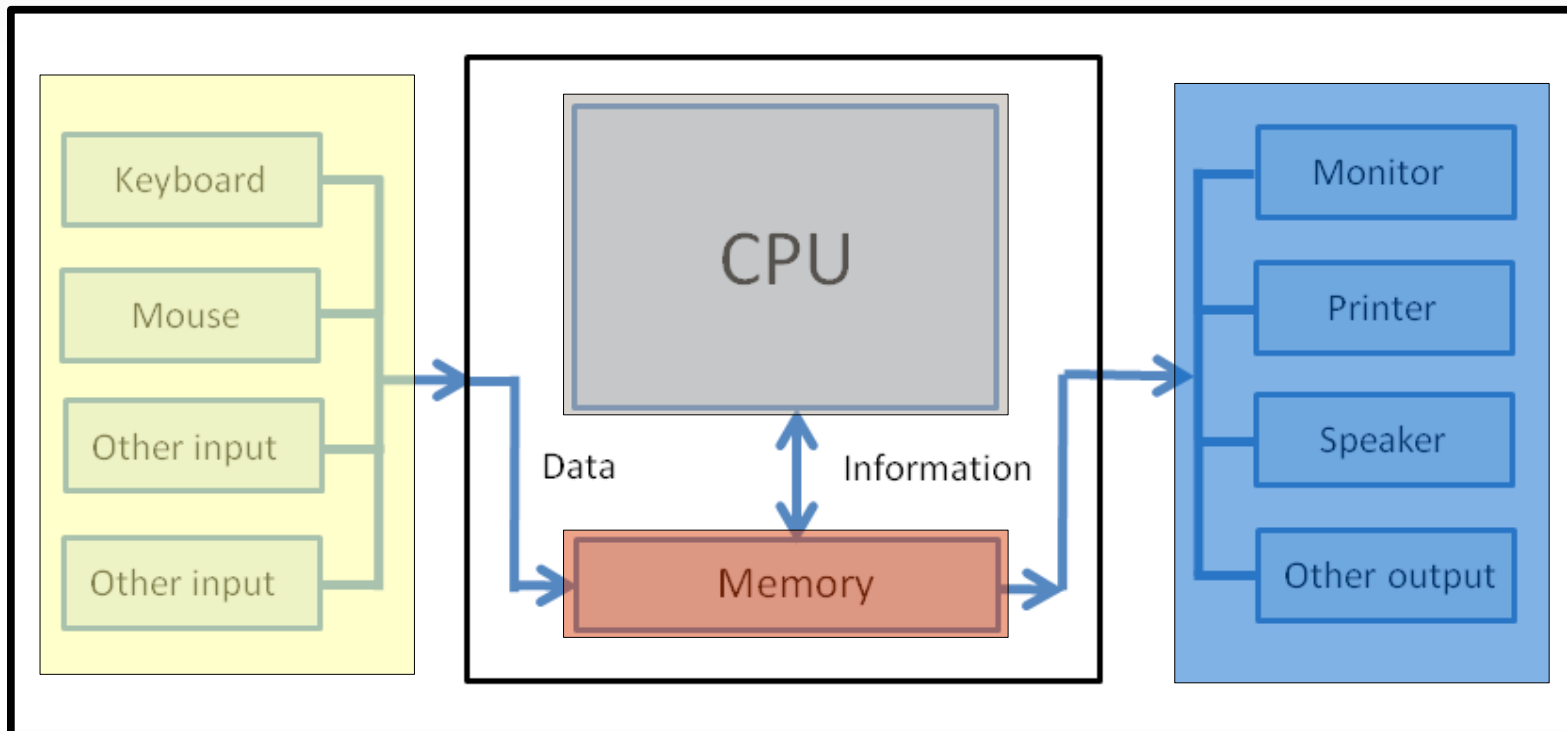
# Linguagem como Abstração

```
frase = input()  
print("----")  
for i in range(1,5):  
    print(i, frase)
```



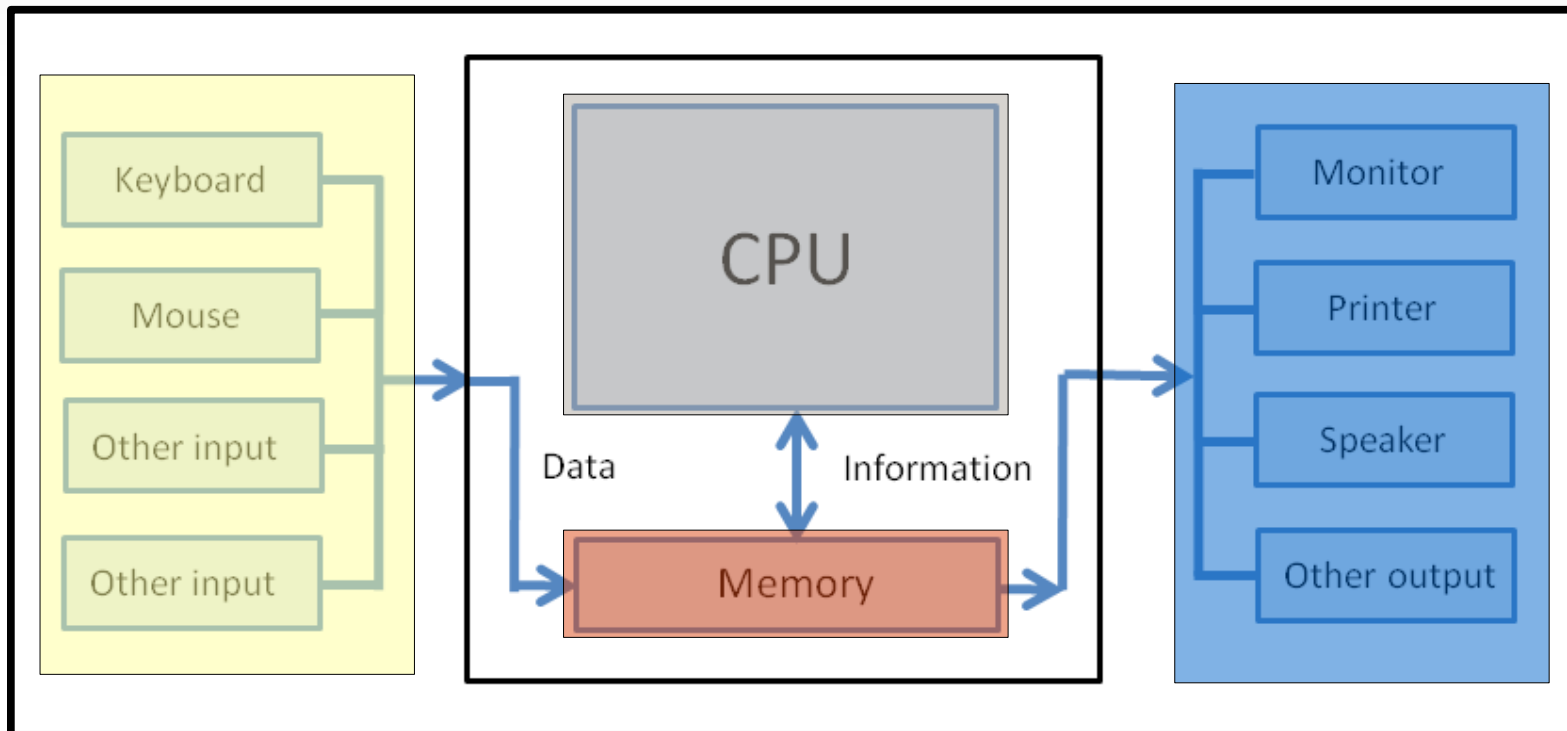
# Linguagem como Abstração

```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```



# Linguagem como Abstração

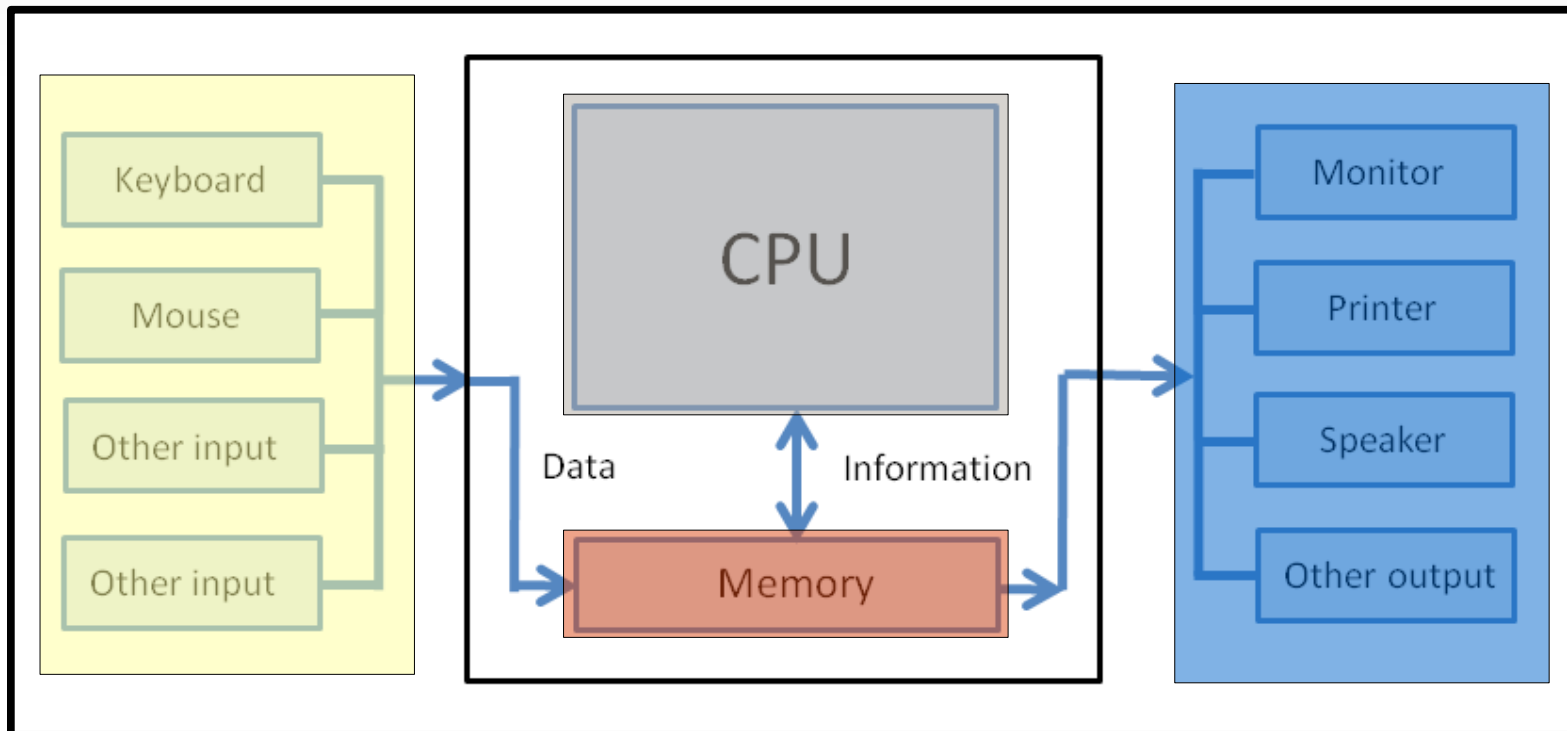
```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```





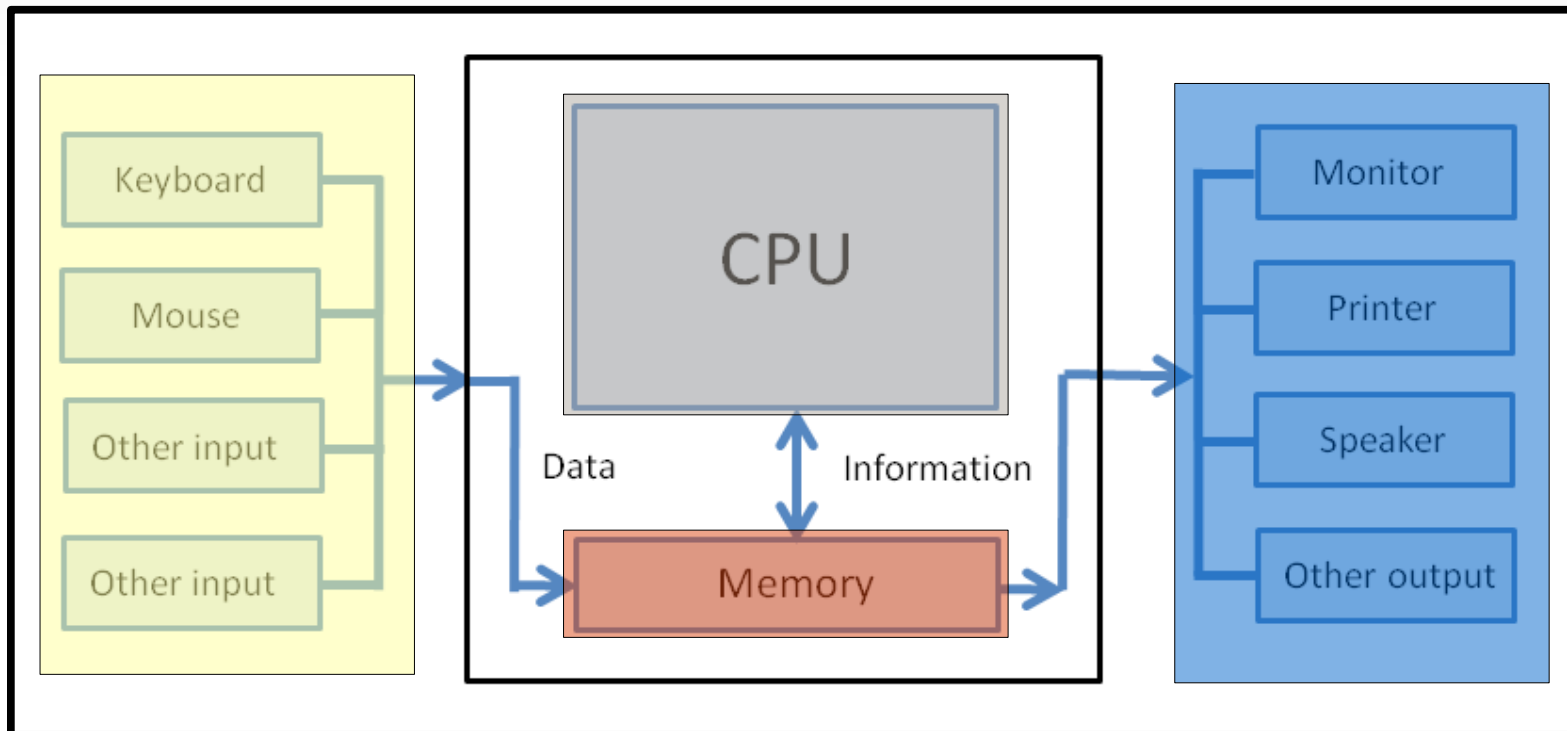
# Linguagem como Abstração

```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```



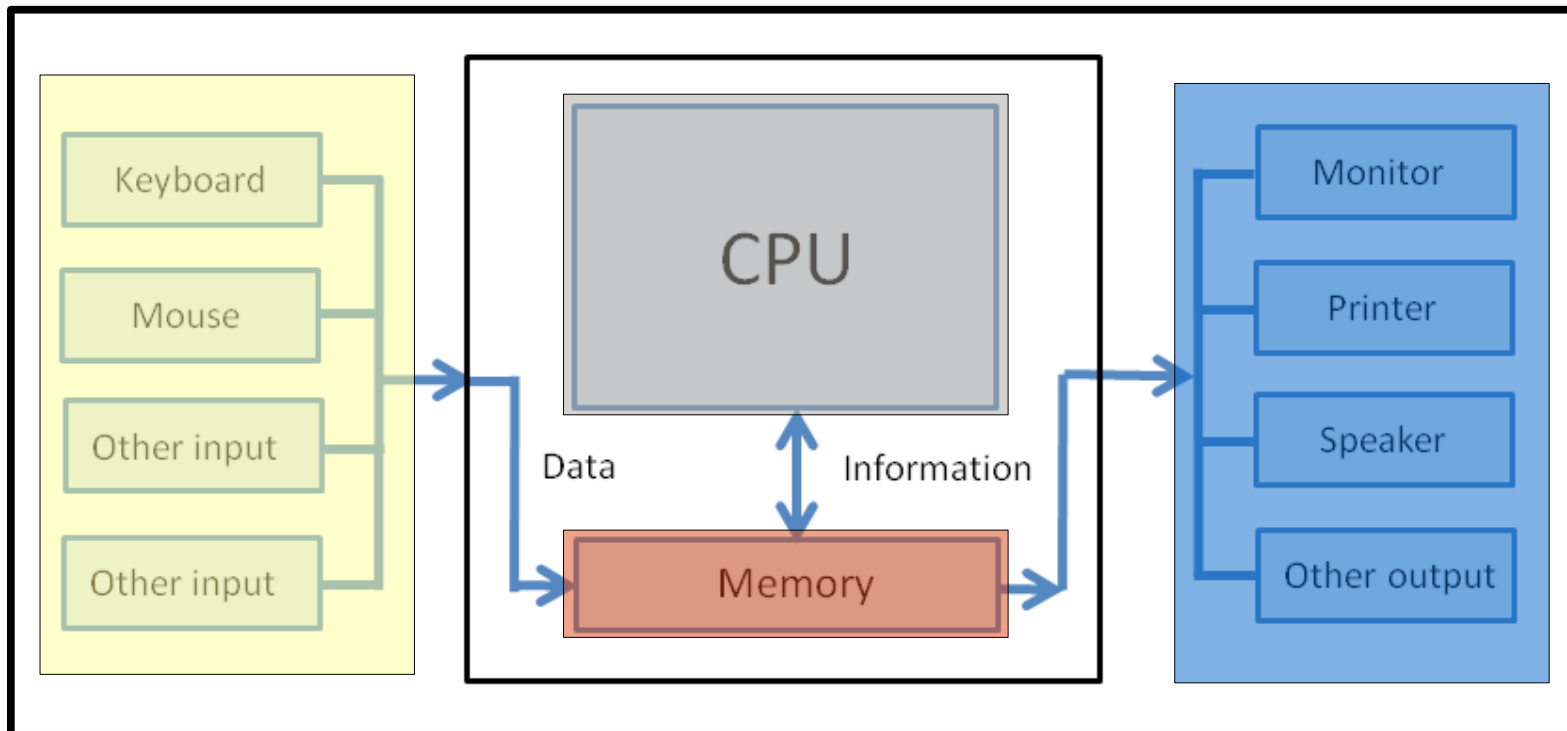
# Linguagem como Abstração

```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```



# Linguagem como Abstração

```
frase = input()
print("----")
for i in range(1,5):
    print(i, frase)
```



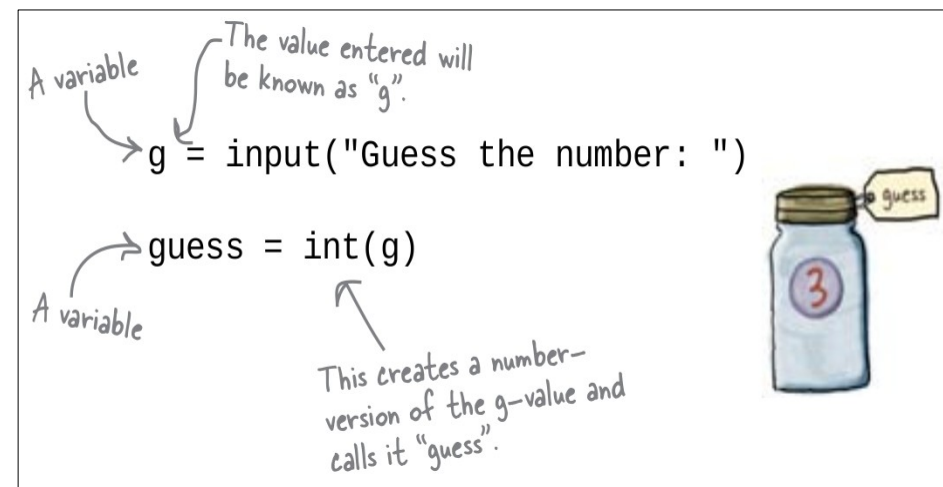
# Variáveis

# Variáveis

- Uma “etiqueta” (ou nome) que representa uma região de memória

# Variáveis

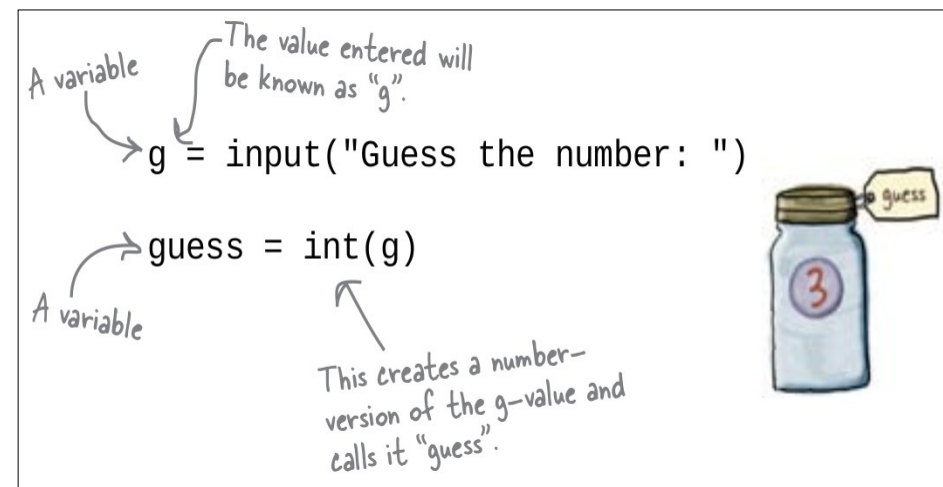
- Uma “etiqueta” (ou nome) que representa uma região de memória



Créditos: "Head First Programming"

# Variáveis

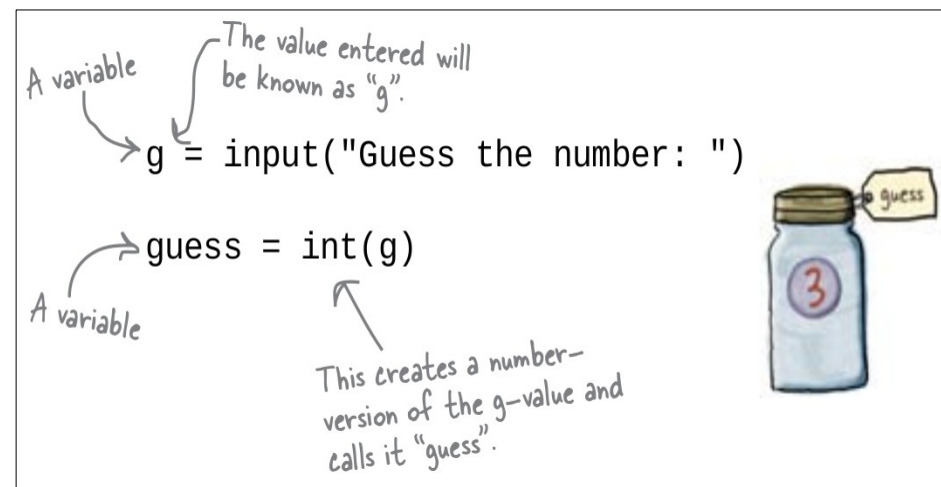
- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador



Créditos: "Head First Programming"

# Variáveis

- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador
  - endereço

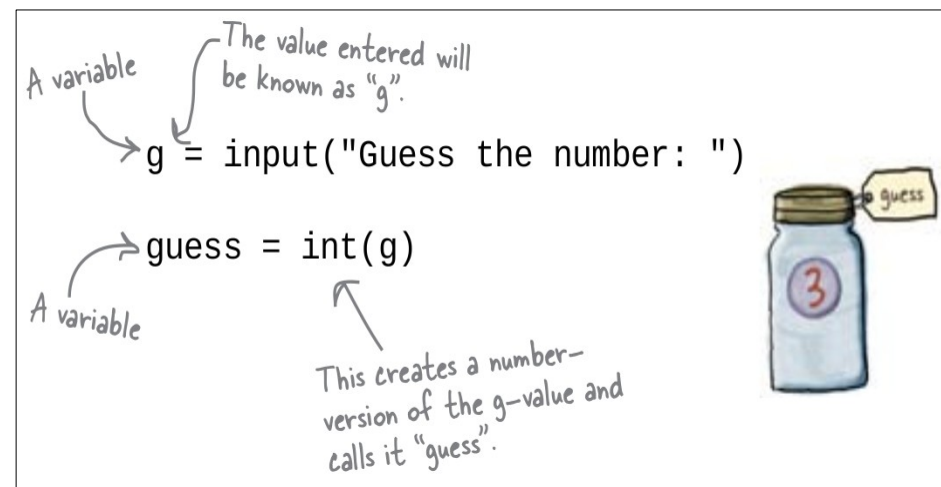


Créditos: "Head First Programming"



# Variáveis

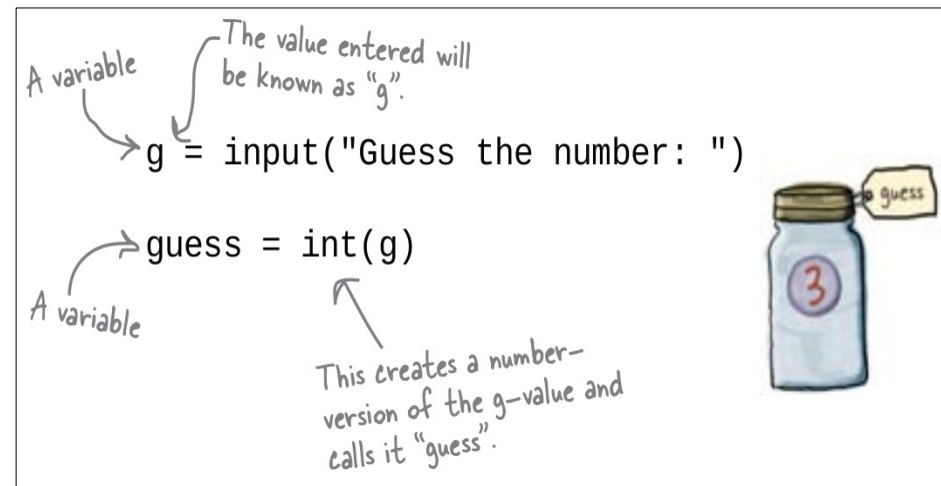
- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador
  - endereço
  - valor



Créditos: "Head First Programming"

# Variáveis

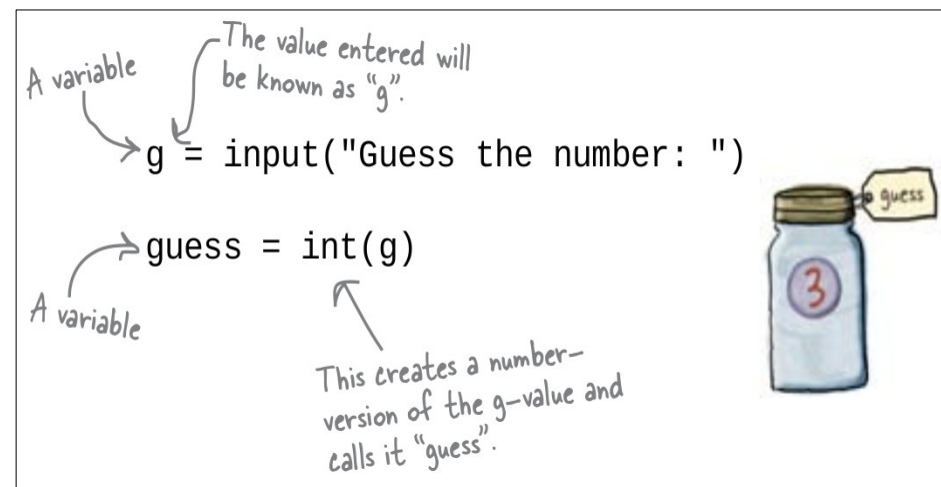
- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador
  - endereço
  - valor
  - tipo



Créditos: "Head First Programming"

# Variáveis

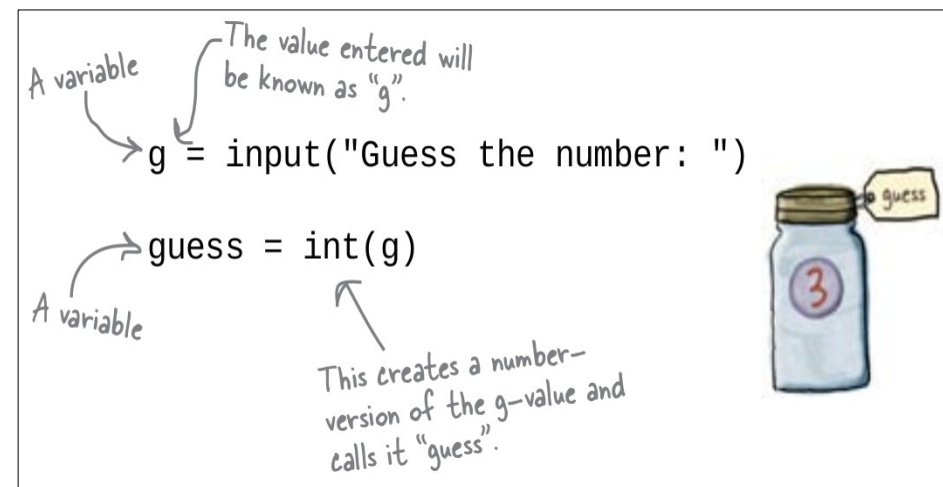
- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador
  - endereço
  - valor
  - tipo
  - escopo



Créditos: "Head First Programming"

# Variáveis

- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador
  - endereço
  - valor
  - tipo
  - escopo
  - tempo de vida



Créditos: "Head First Programming"

# Variáveis - Sintaxe

# Variáveis - Sintaxe

- string de caracteres
  - `i, minha_variavel`

# Variáveis - Sintaxe

- string de caracteres
  - `i, minha_variavel`
  - `10i, $i, variável, if`

# Variáveis - Sintaxe

- string de caracteres
  - `i, minha_variavel`
  - `10i, $i, variável, if`
- Palavras reservadas?



# Variáveis - Sintaxe

- string de caracteres
  - `i, minha_variavel`
  - `10i, $i, variável, if`
- Palavras reservadas?
- “Case sensitive”?

# Variáveis - Sintaxe

- string de caracteres
  - `i, minha_variavel`
  - `10i, $i, variável, if`
- Palavras reservadas?
- “Case sensitive”?
- Caracteres especiais?

# Variáveis - Sintaxe

- string de caracteres
  - `i, minha_variavel`
  - `10i, $i, variável, if`
- Palavras reservadas?
- “Case sensitive”?
- Caracteres especiais?

```
print("Olá $nome")
```

# Variáveis - Sintaxe

- string de caracteres
  - `i`, `minha_variavel`
  - `10i`, `$i`, `variável`, `if`
- Palavras reservadas?
- “Case sensitive”?
- Caracteres especiais?

```
print("Olá $nome")
```

Names in most programming languages have the same form: a letter followed by a string consisting of letters, digits, and underscore characters ( `_` ).

# Variáveis - Sintaxe

## Instance variable: self vs @

Here is some code:

120



43

```
class Person
  def initialize(age)
    @age = age
  end

  def age
    @age
  end

  def age_difference_with(other_person)
    (self.age - other_person.age).abs
  end

  protected :age
end
```

What I want to know is the difference between using `@age` and `self.age` in `age_difference_with` method.

# Sintaxe - Forma

```
@numeros = (0,1,2);  
$numeros = @numeros;  
print "$numeros: @numeros\n";
```

# Sintaxe - Forma

```
@numeros = (0,1,2);  
$numeros = @numeros;  
print "$numeros: @numeros\n";
```

```
$ perl numeros.pl  
3: 0 1 2  
$
```

# Exercícios



# Exercícios

## 1. Sobre identificadores em LISP (e/ou derivados)...

- quais caracteres podem ser usados?
- por quê não são permitidos em outras linguagens?
- que usos incomuns e interessantes você encontrou?

# Exercícios

## 1. Sobre identificadores em LISP (e/ou derivados)...

- quais caracteres podem ser usados?
- por quê não são permitidos em outras linguagens?
- que usos incomuns e interessantes você encontrou?

## 2. Sobre *átomos* ou *símbolos* em LISP, Erlang e Ruby...

- o que são?
- para que servem?
- que usos interessantes você encontrou?

# Exercícios

1. Sobre identificadores em LISP (e/ou derivados)...
  - quais caracteres podem ser usados?
  - por quê não são permitidos em outras linguagens?
  - que usos incomuns e interessantes você encontrou?
2. Sobre *átomos* ou *símbolos* em LISP, Erlang e Ruby...
  - o que são?
  - para que servem?
  - que usos interessantes você encontrou?
3. Sobre os prefixos em variáveis Perl...
  - quais são e o que representam?
  - quais são as regras de conversão existentes?

# *Amarração de Nomes*

(continuação...)

Estrutura de Linguagens

<https://github.com/fsantanna-uerj/EDL/>

Francisco Sant'Anna

[francisco@ime.uerj.br](mailto:francisco@ime.uerj.br)



# Amarração (Binding)

# Amarração (Binding)

- Associação entre “nome” e “entidade”

# Amarração (Binding)

- Associação entre “nome” e “entidade”
  - variáveis → memória (endereço, tipo, valor)

# Amarração (Binding)

- Associação entre “nome” e “entidade”
  - variáveis → memória (endereço, tipo, valor)
  - funções e operadores → trecho de código
    - `print` → exibe texto na tela
    - `+` → soma números



# Amarração (Binding)

- Associação entre “nome” e “entidade”
  - variáveis → memória (endereço, tipo, valor)
  - funções e operadores → trecho de código
    - `print` → exibe texto na tela
    - `+` → soma números
  - comandos → semântica da linguagem
    - `if` → testa uma condição e executa um ou outro caminho
    - `while` → repete comandos até uma condição falhar

# Amarração (Binding)

- Associação entre “nome” e “entidade”

# Amarração (Binding)

- Associação entre “nome” e “entidade”
  - *tempo de amarração (binding time)*

# Amarração (Binding)

- Associação entre “nome” e “entidade”
  - *tempo de amarração (binding time)*
    - *language design time (especificação)*
    - *language implementation time (implementação)*
    - *preprocess time (pré-processamento)*
    - *compile time (compilação)*
    - *link time (ligação)*
    - *load time (carregamento)*
    - *run time (execução)*

# Amarração (Binding)

- Associação entre “nome” e “entidade”
  - *tempo de amarração (binding time)*
    - *language design time (especificação)*
    - *language implementation time (implementação)*
  - *preprocess time (pré-processamento)*
  - *compile time (compilação)*
  - *link time (ligação)*
  - *load time (carregamento)*
  - *run time (execução)*

# Amarração (Binding)

- Associação entre “nome” e “entidade”
  - *tempo de amarração (binding time)*
    - *language design time (especificação)*
    - *language implementation time (implementação)*
    - *preprocess time (pré-processamento)*
    - *compile time (compilação)*
    - *link time (ligação)*
    - *load time (carregamento)*
    - *run time (execução)*

# Amarração (Binding)

- *language design time*
  - especificação da linguagem

# Amarração (Binding)

- *language design time*
  - especificação da linguagem

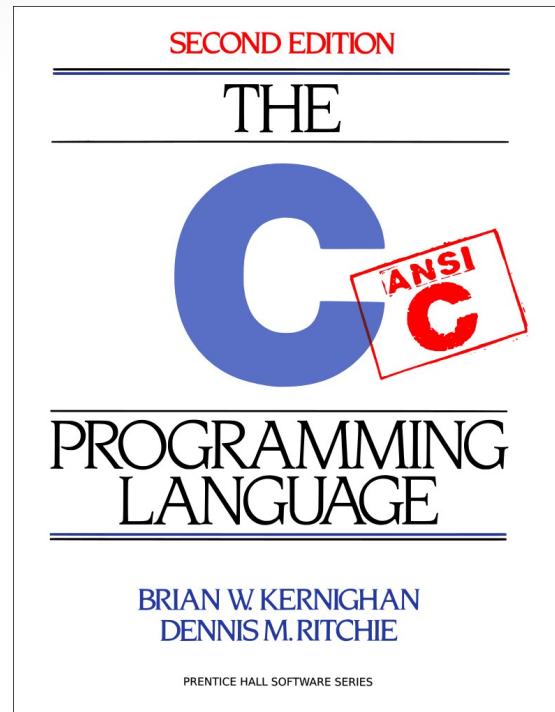
```
if  
char  
printf
```



# Amarração (Binding)

- *language design time*
  - especificação da linguagem

```
if  
char  
printf
```



# Amarração (Binding)

- *language design time*
- *language implementation time*

# Amarração (Binding)

- *language design time*
- *language implementation time*



# Amarração (Binding)

- *language design time*
- *language implementation time*

`int -> ?`



# Amarração (Binding)

- *language design time*
- *language implementation time*

# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*

# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*

```
#include ...  
#define PI 3.14  
#ifdef ...  
#endif
```

# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*

```
#include ...  
#define PI 3.14  
#ifdef ...  
#endif
```

```
gcc -E
```

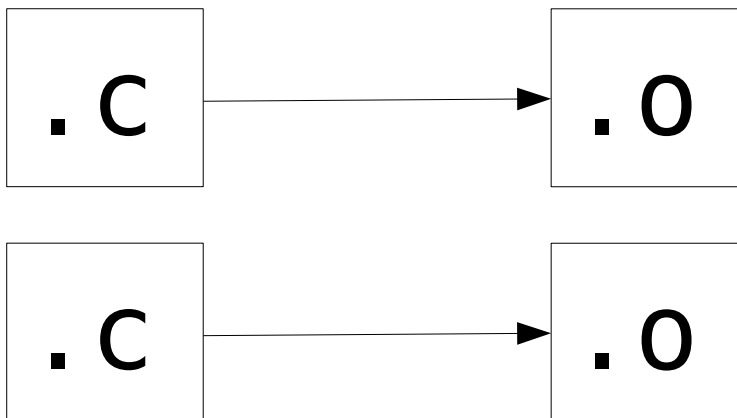


# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*

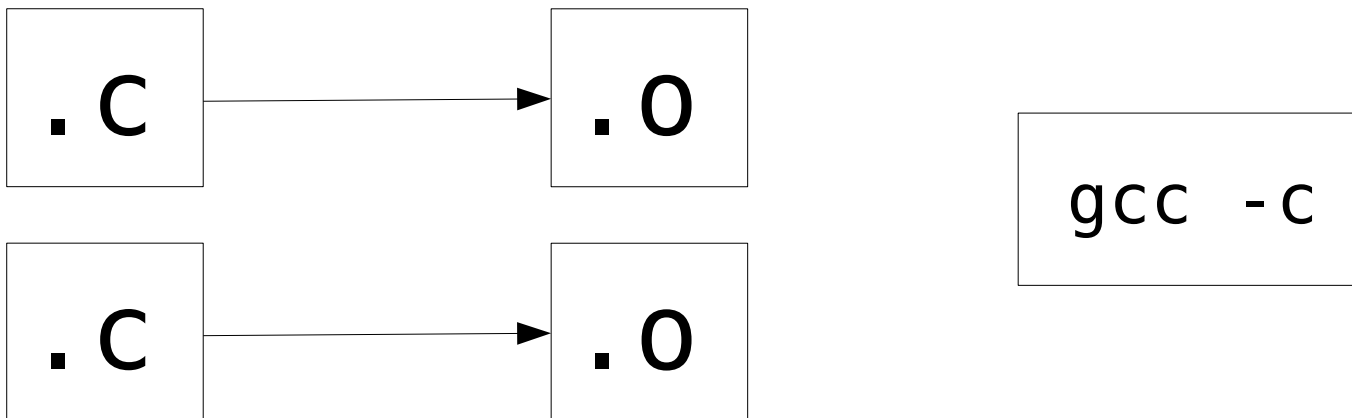
# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*



# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*



# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*
- *link time*

# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*
- *link time*

```
$ gcc -lpthread . . .
```

# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*
- *link time*

```
$ gcc -lpthread . . .
```

# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*
- *link time*
- *load time*
- *run time*

# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*
- *link time*
- *load time*
- *run time*



# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*
- *link time*
- *load time*
- *run time*

```
$ ./prog.exe
```

# Amarração (Binding)

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*
- *link time*
- *load time*
- *run time*

```
$ ./prog.exe
```

0101



CPU/MEM

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
- Endereço de **v**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
- Endereço de **v**
- Tamanho de **int**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
- Endereço de **v**
- Tamanho de **int**
- Implementação de **f**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
- Endereço de **v**
- Tamanho de **int**
- Implementação de **f**
- Tipo de retorno de **f**



# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
- Endereço de **v**
- Tamanho de **int**
- Implementação de **f**
- Tipo de retorno de **f**
- Tamanho de **uint8\_t**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
- Endereço de **v**
- Tamanho de **int**
- Implementação de **f**
- Tipo de retorno de **f**
- Tamanho de **uint8\_t**
- Endereço de **x**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
- Endereço de **v**
- Tamanho de **int**
- Implementação de **f**
- Tipo de retorno de **f**
- Tamanho de **uint8\_t**
- Endereço de **x**
- Semântica de “+”

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
- Endereço de **v**
- Tamanho de **int**
- Implementação de **f**
- Tipo de retorno de **f**
- Tamanho de **uint8\_t**
- Endereço de **x**
- Semântica de “+”
- Implementação de **sin**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
  - Endereço de **v**
  - Tamanho de **int**
  - Implementação de **f**
  - Tipo de retorno de **f**
- pré-processamento*
- Tamanho de **uint8\_t**
  - Endereço de **x**
  - Semântica de “+”
  - Implementação de **sin**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- Valor de **PI**
  - Endereço de **v**
  - Tamanho de **int**
  - Implementação de **f**
  - Tipo de retorno de **f**
- |                          |
|--------------------------|
| <i>pré-processamento</i> |
| <i>carregamento</i>      |
- Tamanho de **uint8\_t**
  - Endereço de **x**
  - Semântica de “+”
  - Implementação de **sin**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- |                               |                          |                               |
|-------------------------------|--------------------------|-------------------------------|
| ■ Valor de <b>PI</b>          | <i>pré-processamento</i> | ■ Tamanho de <b>uint8_t</b>   |
| ■ Endereço de <b>v</b>        | <i>carregamento</i>      | ■ Endereço de <b>x</b>        |
| ■ Tamanho de <b>int</b>       | <i>implementação</i>     | ■ Semântica de “+”            |
| ■ Implementação de <b>f</b>   |                          | ■ Implementação de <b>sin</b> |
| ■ Tipo de retorno de <b>f</b> |                          |                               |

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- |                               |                          |                               |
|-------------------------------|--------------------------|-------------------------------|
| ■ Valor de <b>PI</b>          | <i>pré-processamento</i> | ■ Tamanho de <b>uint8_t</b>   |
| ■ Endereço de <b>v</b>        | <i>carregamento</i>      | ■ Endereço de <b>x</b>        |
| ■ Tamanho de <b>int</b>       | <i>implementação</i>     | ■ Semântica de “+”            |
| ■ Implementação de <b>f</b>   | <i>ligação</i>           | ■ Implementação de <b>sin</b> |
| ■ Tipo de retorno de <b>f</b> |                          |                               |



# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- |                               |                          |                               |
|-------------------------------|--------------------------|-------------------------------|
| ■ Valor de <b>PI</b>          | <i>pré-processamento</i> | ■ Tamanho de <b>uint8_t</b>   |
| ■ Endereço de <b>v</b>        | <i>carregamento</i>      | ■ Endereço de <b>x</b>        |
| ■ Tamanho de <b>int</b>       | <i>implementação</i>     | ■ Semântica de “+”            |
| ■ Implementação de <b>f</b>   | <i>ligação</i>           | ■ Implementação de <b>sin</b> |
| ■ Tipo de retorno de <b>f</b> | <i>compilação</i>        |                               |

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- |                               |                          |                               |               |
|-------------------------------|--------------------------|-------------------------------|---------------|
| ■ Valor de <b>PI</b>          | <i>pré-processamento</i> | ■ Tamanho de <b>uint8_t</b>   | <i>design</i> |
| ■ Endereço de <b>v</b>        | <i>carregamento</i>      | ■ Endereço de <b>x</b>        |               |
| ■ Tamanho de <b>int</b>       | <i>implementação</i>     | ■ Semântica de “+”            |               |
| ■ Implementação de <b>f</b>   | <i>ligação</i>           | ■ Implementação de <b>sin</b> |               |
| ■ Tipo de retorno de <b>f</b> | <i>compilação</i>        |                               |               |

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

■ Valor de **PI**

*pré-processamento*

■ Endereço de **v**

*carregamento*

■ Tamanho de **int**

*implementação*

■ Implementação de **f**

*ligação*

■ Tipo de retorno de **f**

*compilação*

■ Tamanho de **uint8\_t**

*design*

■ Endereço de **x**

*execução*

■ Semântica de “+”

■ Implementação de **sin**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

■ Valor de **PI**

*pré-processamento*

■ Endereço de **v**

*carregamento*

■ Tamanho de **int**

*implementação*

■ Implementação de **f**

*ligação*

■ Tipo de retorno de **f**

*compilação*

■ Tamanho de **uint8\_t**

*design*

■ Endereço de **x**

*execução*

■ Semântica de “+”

*compilação*

■ Implementação de **sin**

# Binding - Exemplo

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
static int v = 10;
int f (void);
int main (void) {
    uint8_t x = sin(PI) + v + f();
    return x;
}
```

- |                               |                          |                               |                   |
|-------------------------------|--------------------------|-------------------------------|-------------------|
| ■ Valor de <b>PI</b>          | <i>pré-processamento</i> | ■ Tamanho de <b>uint8_t</b>   | <i>design</i>     |
| ■ Endereço de <b>v</b>        | <i>carregamento</i>      | ■ Endereço de <b>x</b>        | <i>execução</i>   |
| ■ Tamanho de <b>int</b>       | <i>implementação</i>     | ■ Semântica de “+”            | <i>compilação</i> |
| ■ Implementação de <b>f</b>   | <i>ligação</i>           | ■ Implementação de <b>sin</b> | <i>ligação</i>    |
| ■ Tipo de retorno de <b>f</b> | <i>compilação</i>        |                               |                   |

# Binding - Estático vs Dinâmico

# Binding - Estático vs Dinâmico

- Estático
  - binding ocorre antes da execução (e não é mais alterado)
  - (*design, implementação, pré-processamento, compilação, ligação*)

# Binding - Estático vs Dinâmico

- Estático

- binding ocorre antes da execução (e não é mais alterado)
- (*design, implementação, pré-processamento, compilação, ligação*)

- Dinâmico

- binding ocorre durante a execução
- (*carregamento, execução*)



# Lua: Binding Times

- *language design time*
- *language implementation time*
- *preprocess time*
- *compile time*
- *link time*
- *load time*
- *run time*

# Lua: Binding Times

- *language design time*
- ~~*language implementation time*~~
- ~~*preprocess time*~~
- *compile time?*
- ~~*link time*~~
- ~~*load time*~~
- *run time*

# Exercícios

1. Escolha um programa escrito em uma linguagem estática.
  - dê dois exemplos de *nome/entidade* para cada tempo de amarração
2. Escolha um programa escrito em uma linguagem dinâmica.
  - dê dois exemplos de *nome/entidade* para cada tempo de amarração

# *Amarração de Nomes*

(continuação...)

Estrutura de Linguagens

<https://github.com/fsantanna-uerj/EDL/>

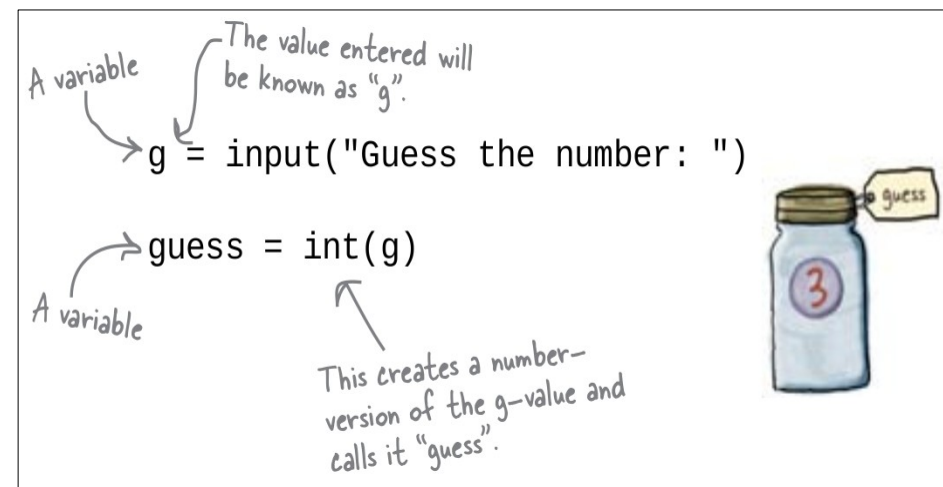
Francisco Sant'Anna

[francisco@ime.uerj.br](mailto:francisco@ime.uerj.br)



# Variáveis

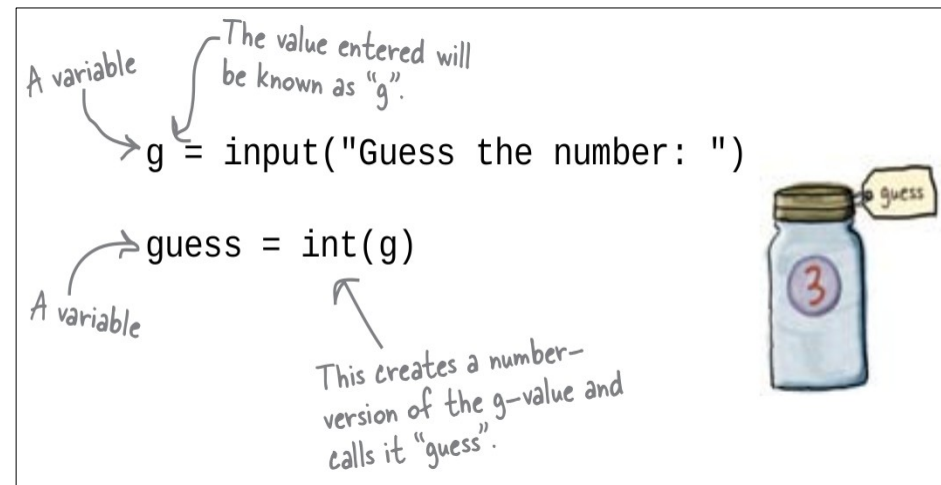
- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador



Créditos: "Head First Programming"

# Variáveis

- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador
  - endereço
  - valor
  - tipo
  - **escopo**
  - **tempo de vida**



Créditos: "Head First Programming"

# Binding de Variável

# Binding de Variável

- Escopo
  - trecho de visibilidade da variável no código



# Binding de Variável

- Escopo
  - trecho de visibilidade da variável no código

```
{  
    int a;  
    ...  
}
```

# Binding de Variável

- Escopo
  - trecho de visibilidade da variável no código
- Tempo de vida
  - período entre alocação e desalocação da variável

```
{  
    int a;  
    ...  
}
```

# Binding de Variável

- Escopo
  - trecho de visibilidade da variável no código
- Tempo de vida
  - período entre alocação e desalocação da variável
- Escopo = Tempo de vida ?

```
{  
    int a;  
    ...  
}
```

# Tempo de Vida vs Escopo

```
#include <stdio.h>

int f (void) {
    static int v = 0;
    v = !v;
    return v;
}

int main (void) {
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    return 0;
}
```

# Tempo de Vida vs Escopo

```
#include <stdio.h>

int f (void) {
    int v = 0;
    v = !v;
    return v;
}

int main (void) {
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    printf("f = %d\n", f());
    return 0;
}
```

# Tempo de Vida

# Tempo de Vida

- Estático

# Tempo de Vida

- Estático
  - globais



# Tempo de Vida

- Estático
  - globais
  - locais estáticas
- Dinâmico

# Tempo de Vida

- Estático
  - globais
  - locais estáticas
- Dinâmico
  - locais na pilha

# Tempo de Vida

- Estático
  - globais
  - locais estáticas
- Dinâmico
  - locais na pilha
  - heap

# Tempo de Vida

- Estático
  - globais
  - locais estáticas
- Dinâmico
  - locais na pilha
  - heap
    - explícito (e.g., *malloc/free*)
    - implícito (e.g., *new/coletor*, construtores primitivos)

# Tempo de Vida

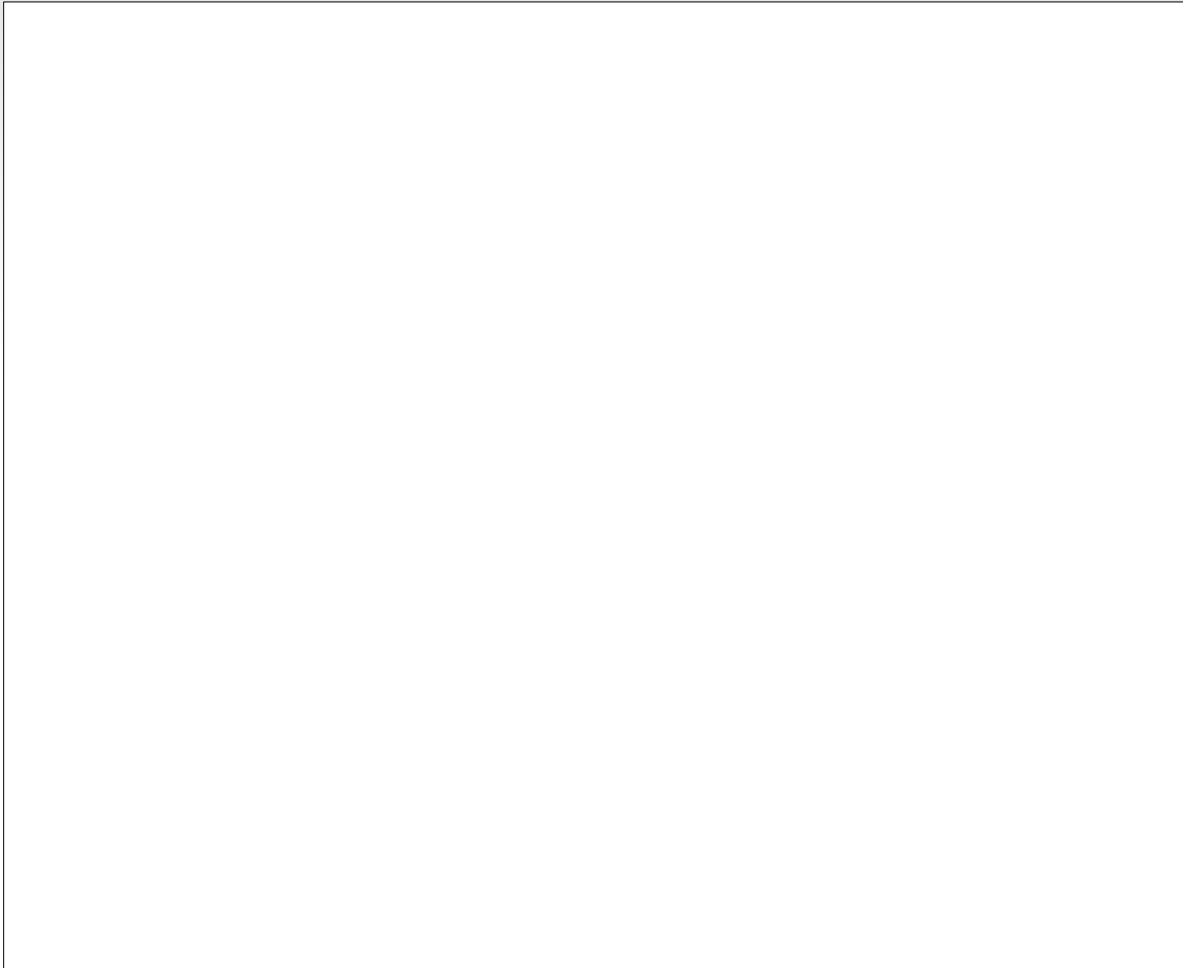
- Estático
  - globais
  - locais estáticas
- Dinâmico
  - locais na pilha
  - heap
    - explícito (e.g., *malloc/free*)
    - implícito (e.g., *new/coletor*, construtores primitivos)

$l = [1, 2, 3]$

# Exemplo - Pilha



# Exemplo - Pilha



# Exemplo - Pilha





# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```

# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```

# Exemplo - Pilha

---

0xFFFF...

```
int fat (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fat(n-1);
    }
}

int vec[4];

void main (void) {
    for (int i=0; i<4; i++) {
        vec[i] = fat(i);
    }
}
```

---

0x0000...

# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```

▲ 0xFFFF...

0x0000...

# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```

▲ 0xFFFF...

0x0000...

(estática)

# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```

(pilha)

▲ 0xFFFF...

0x0000...

(estática)

# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```

(pilha)

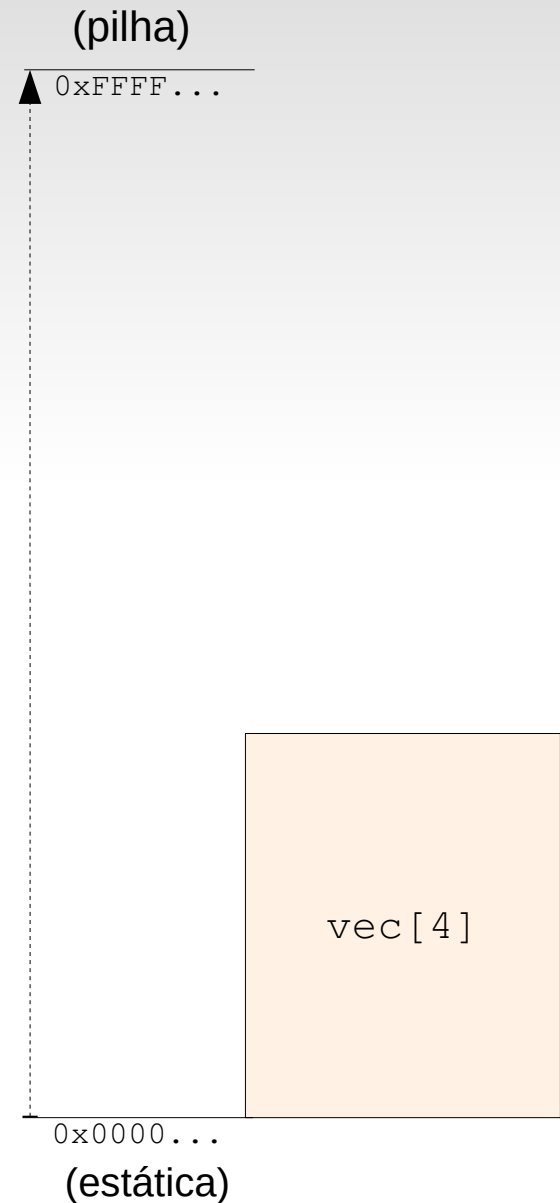
▲ 0xFFFF...

0x0000...

(estática)

# Exemplo - Pilha

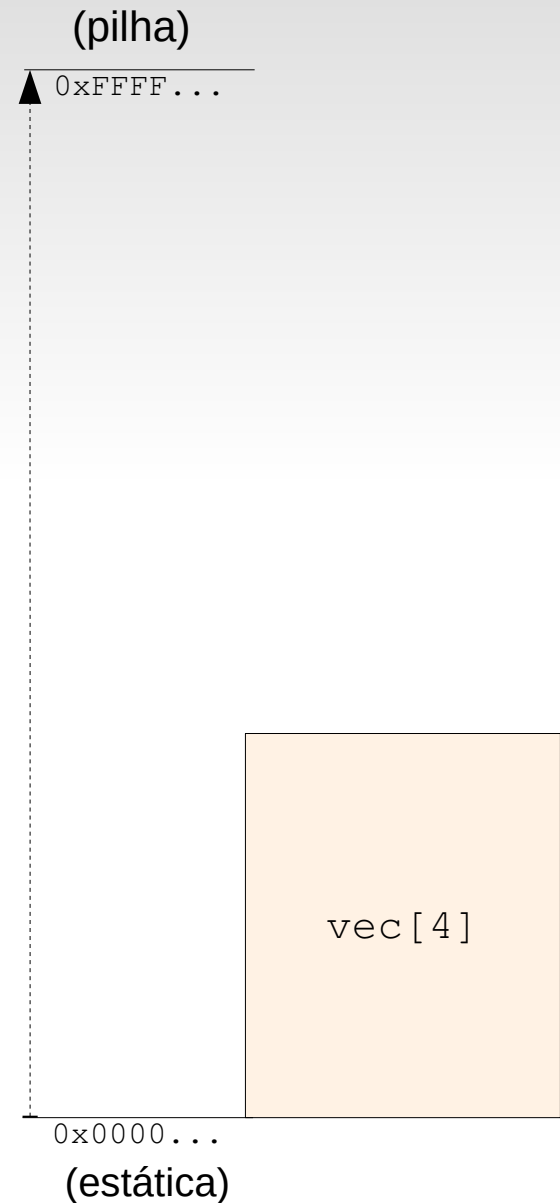
```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```





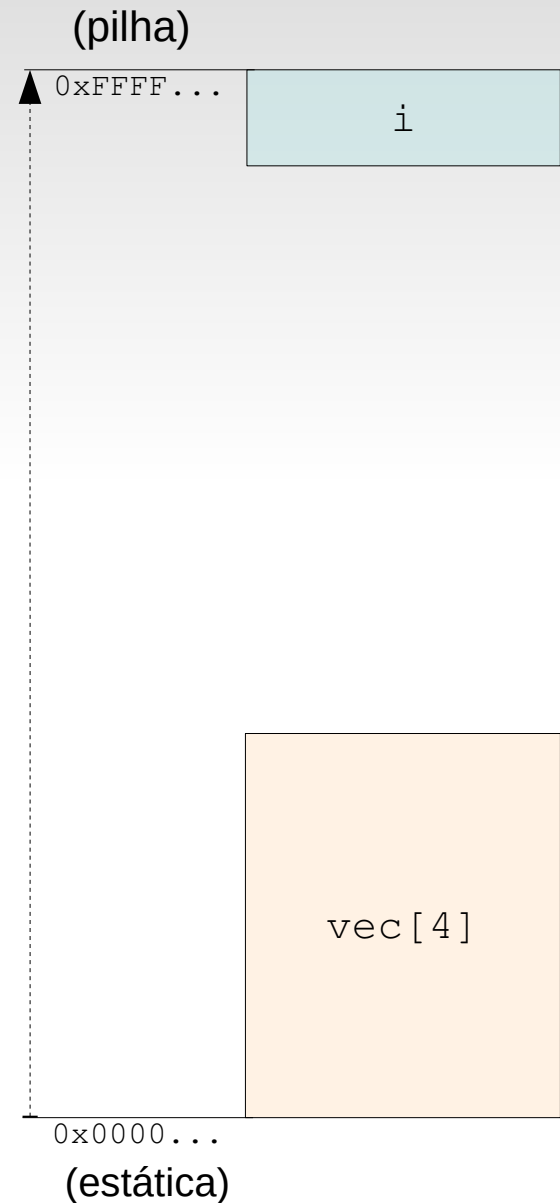
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



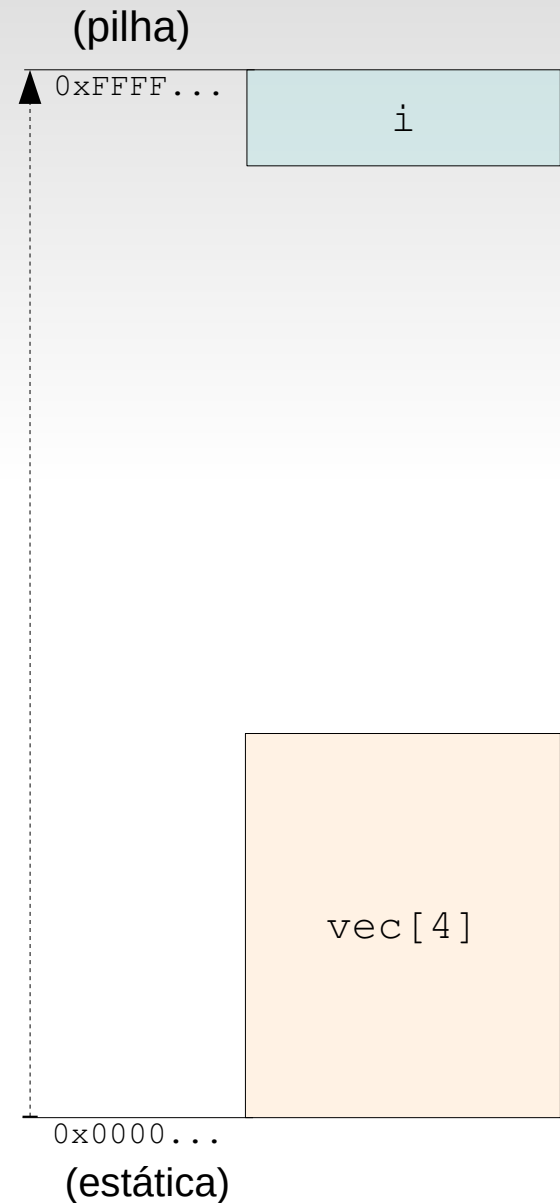
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



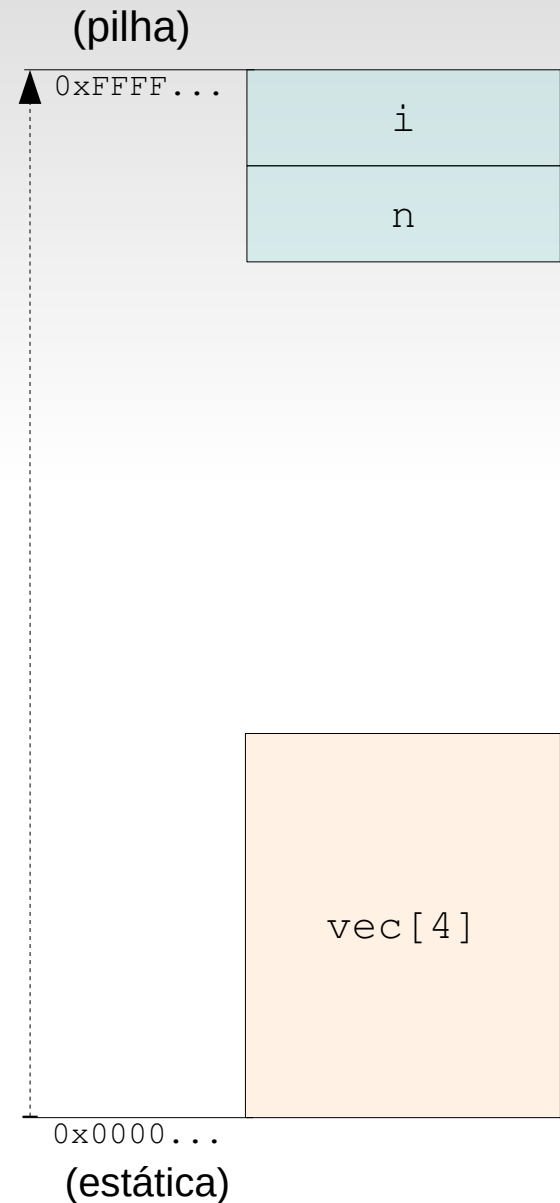
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



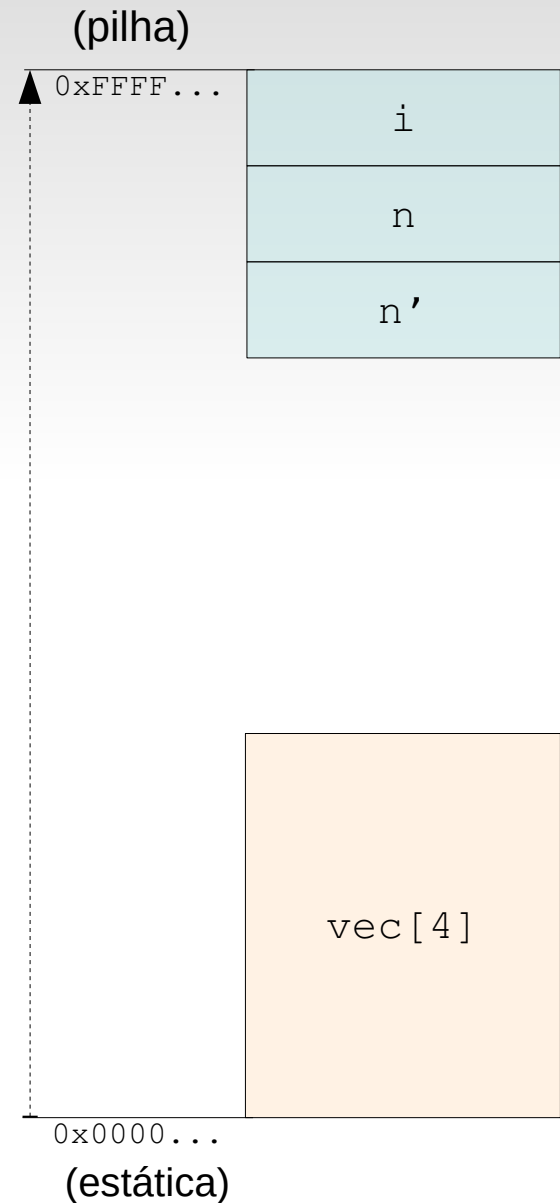
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



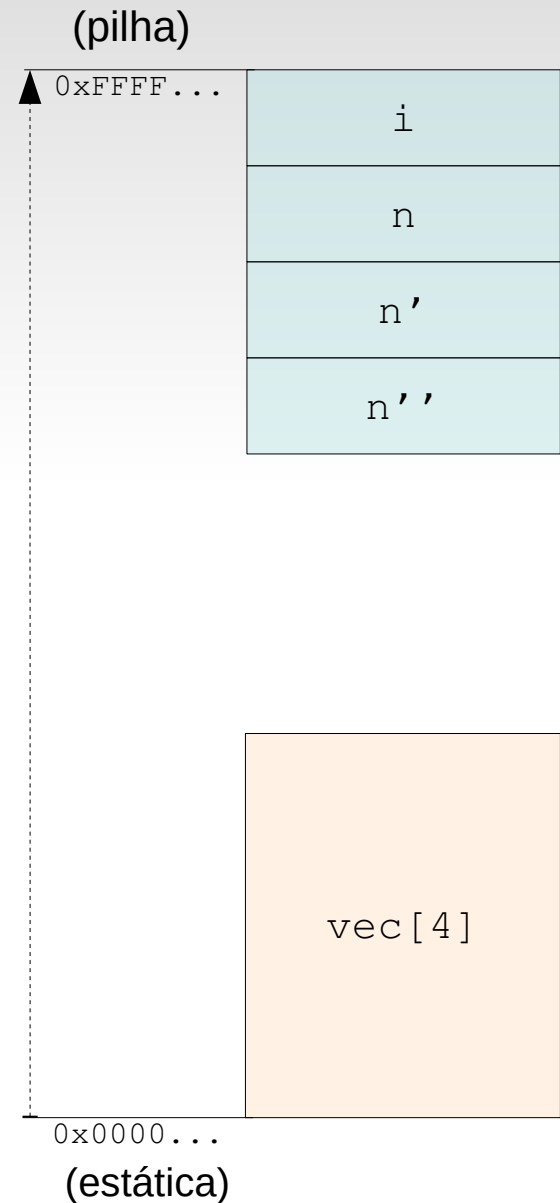
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



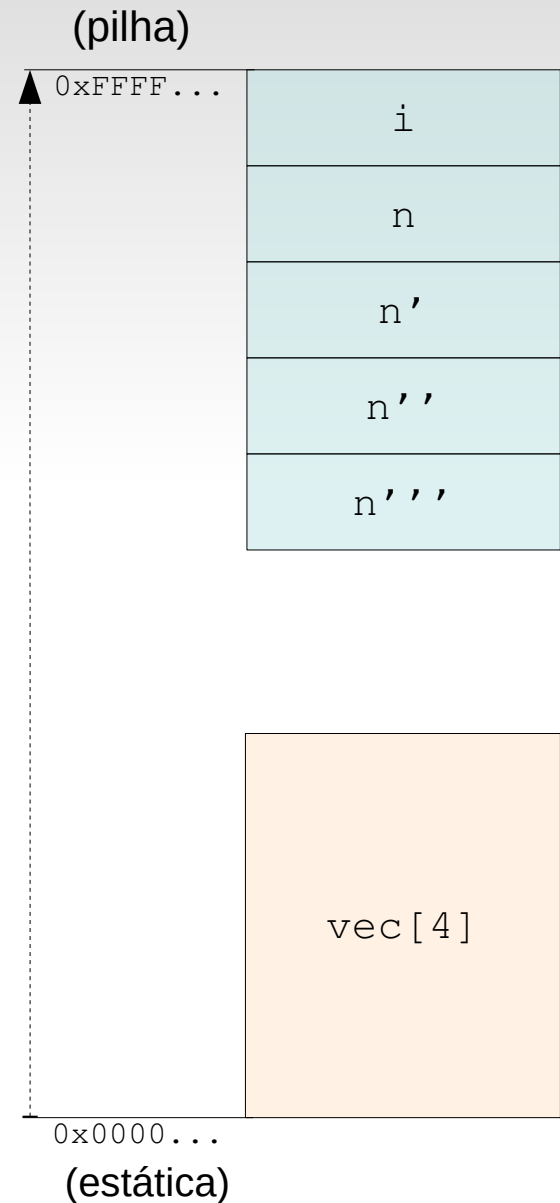
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



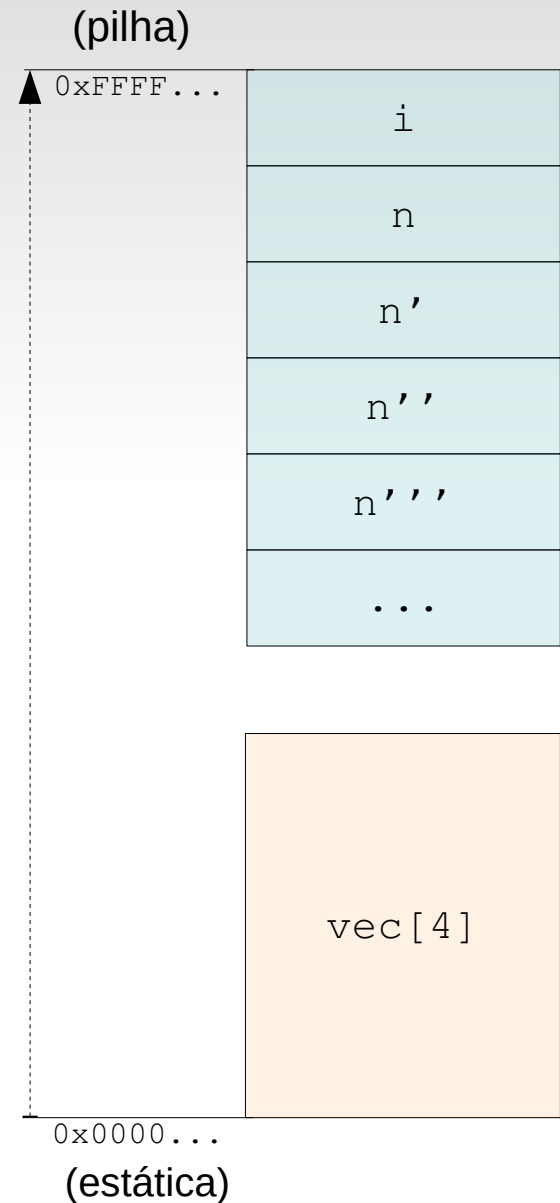
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



# Exemplo - Pilha

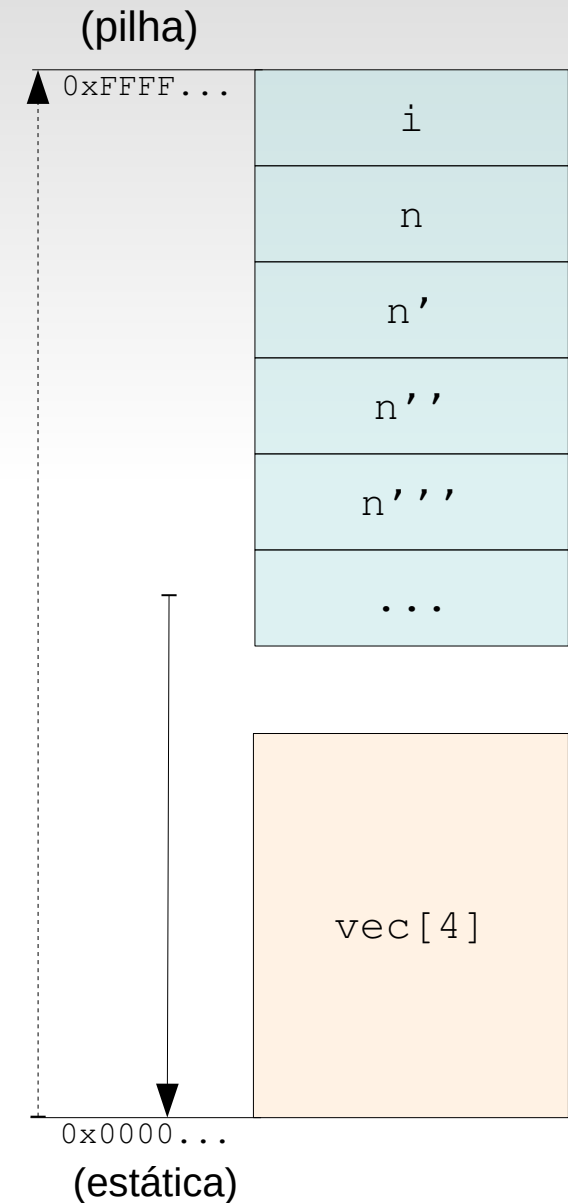
```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```





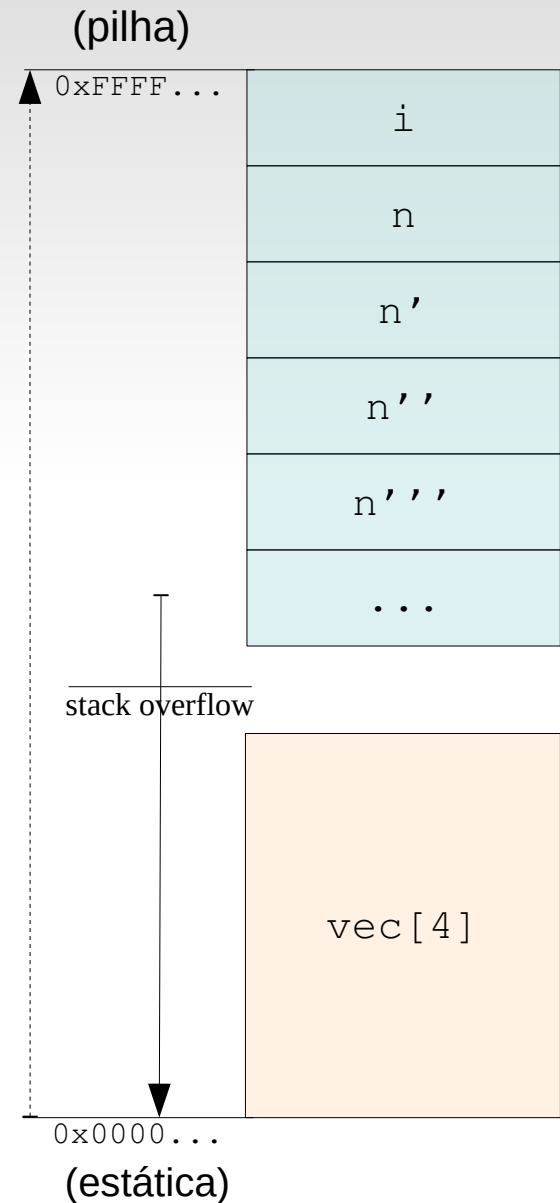
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



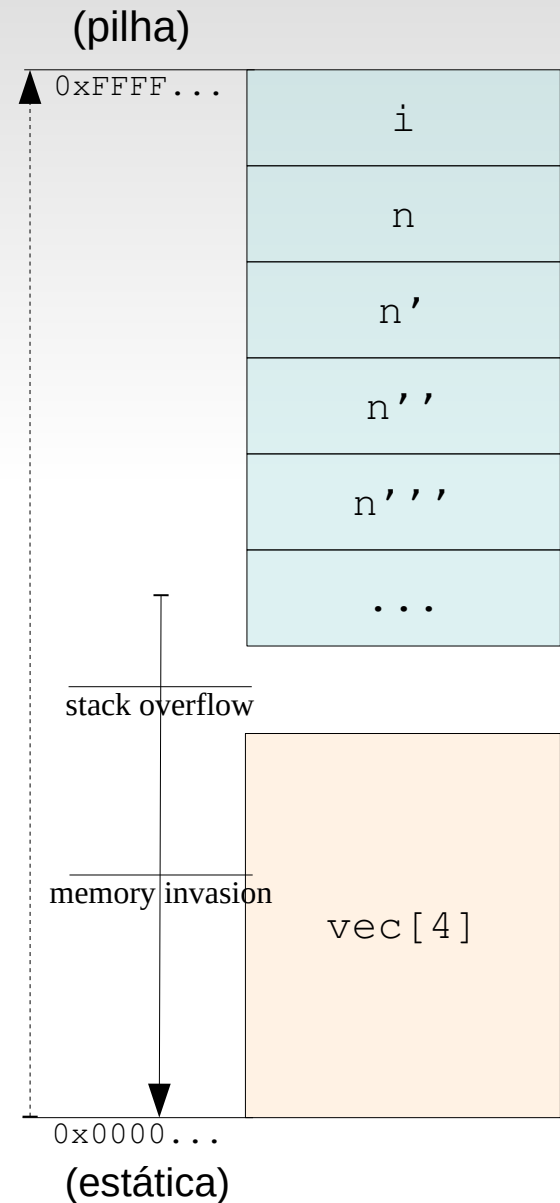
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



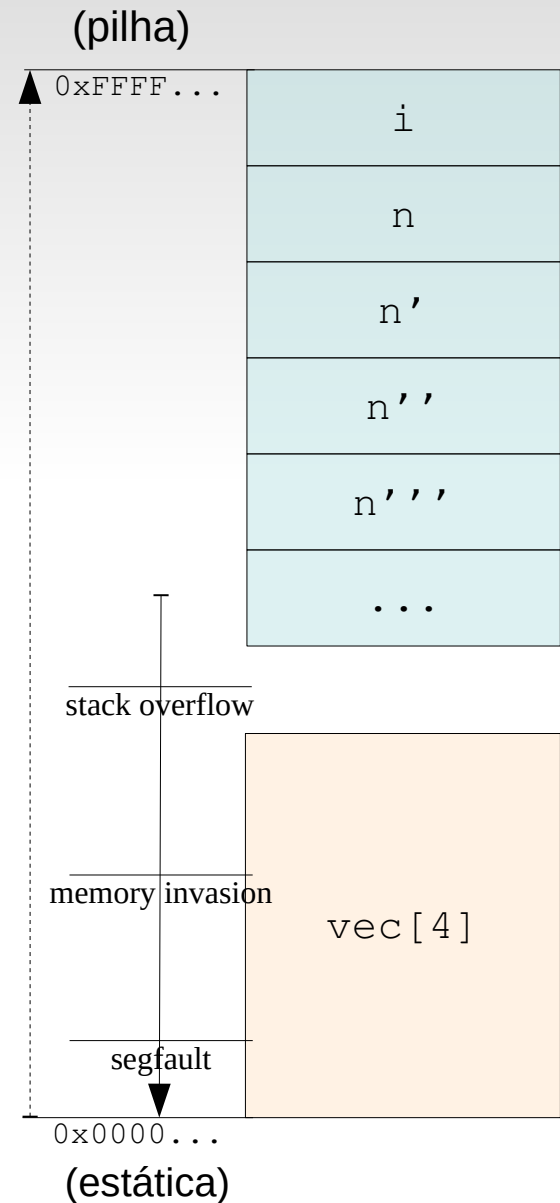
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



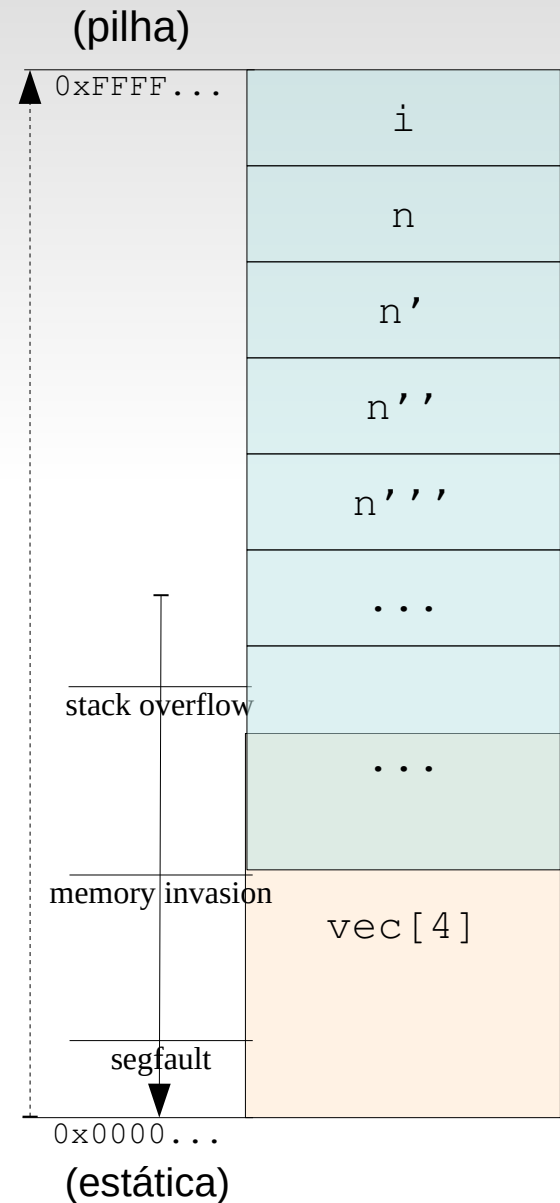
# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



# Exemplo - Pilha

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[4];  
  
void main (void) {  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
}
```



# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```

# Exemplo - Heap

(pilha)

---

0xFFFF...

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```

---

0x0000...

(estática)

# Exemplo - Heap

(pilha)

0xFFFF...

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```

0x0000...

(estática)



# Exemplo - Heap

(pilha)

0xFFFF...

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```

0x0000...

(estática)

# Exemplo - Heap

```
int fat (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fat(n-1);
    }
}

void main (void) {
    int* vec = malloc(sizeof(int) * 4);
    for (int i=0; i<4; i++) {
        vec[i] = fat(i);
    }
    free(vec);
}
```

(pilha)

0xFFFF...

vec

0x0000...

(estática)

# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```

(pilha)

0xFFFF...

vec

0x0000...

(estática)

# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```

(pilha)

0xFFFF...

vec

sizeof(int)\*4

0x0000...

(estática)

# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```

(pilha)

0xFFFF...

vec

(heap)

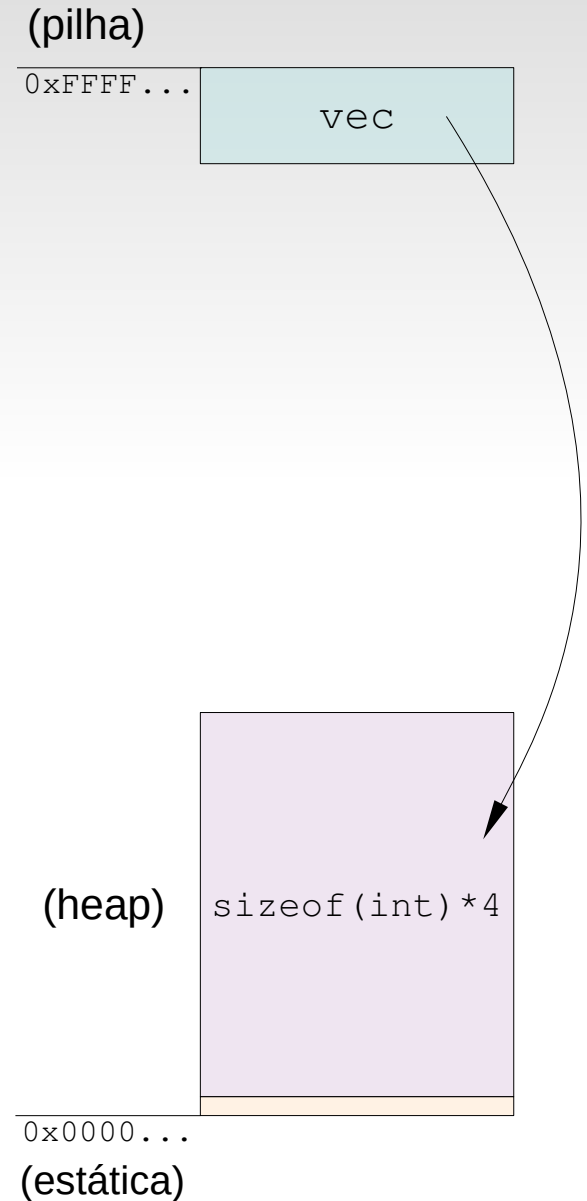
sizeof(int)\*4

0x0000...

(estática)

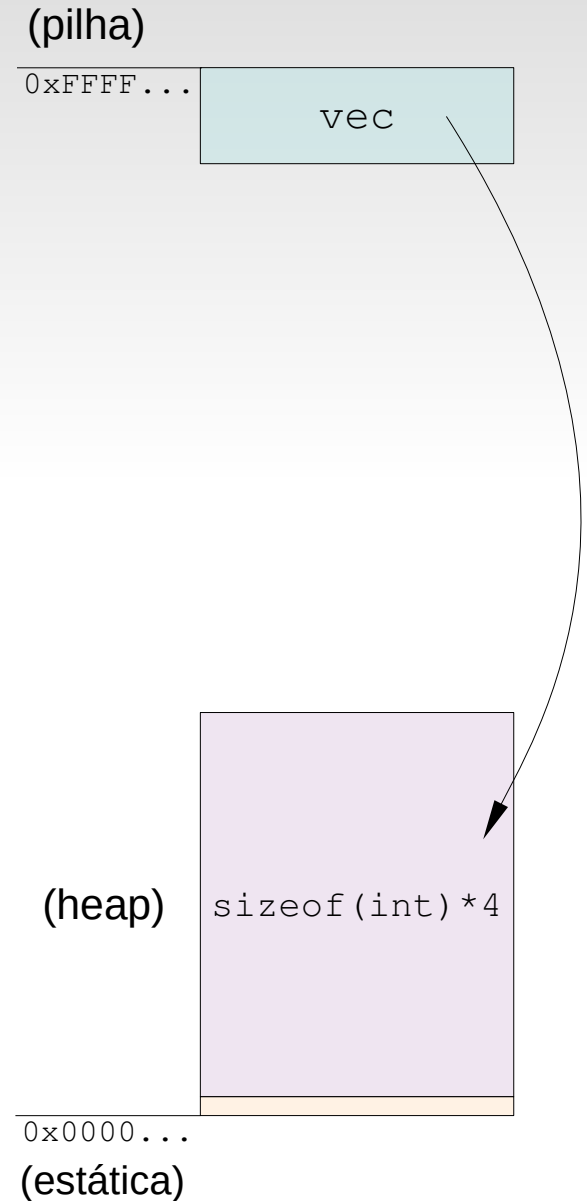
# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```



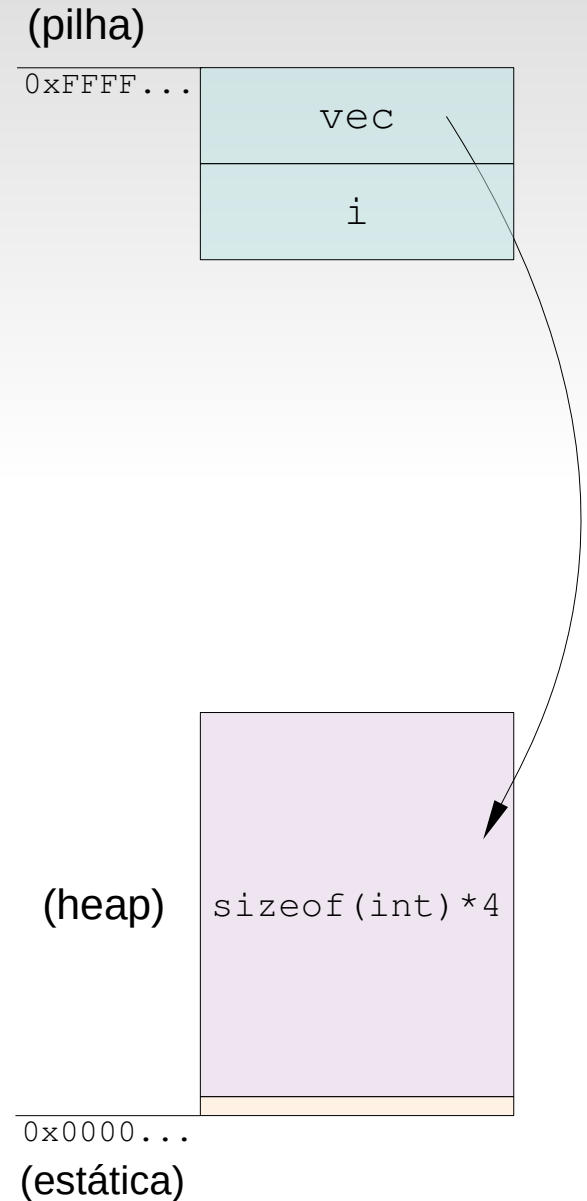
# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```



# Exemplo - Heap

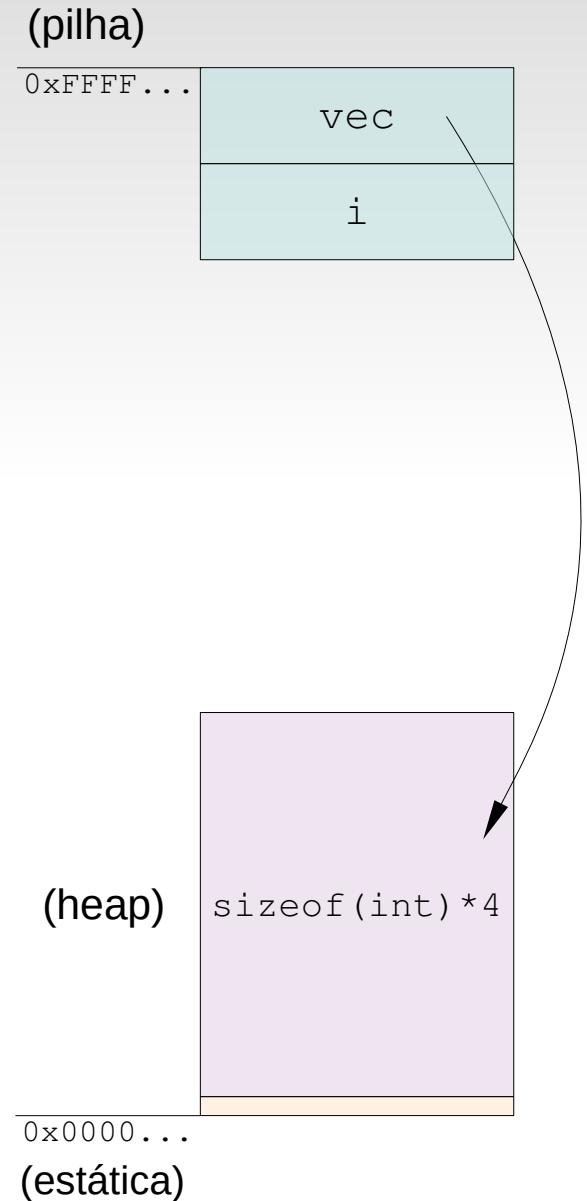
```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```





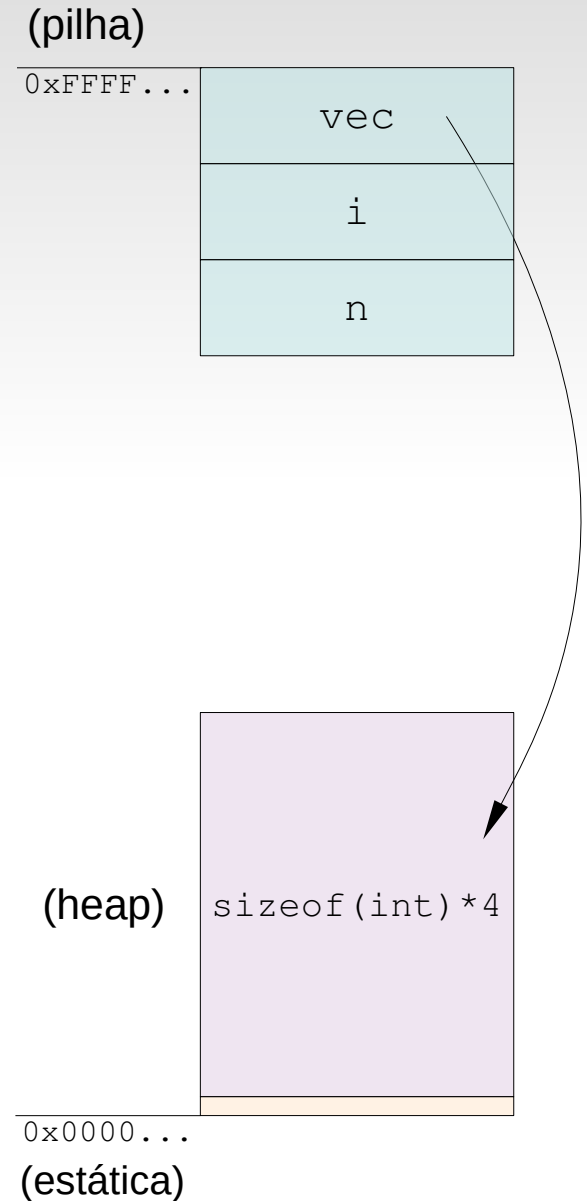
# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```



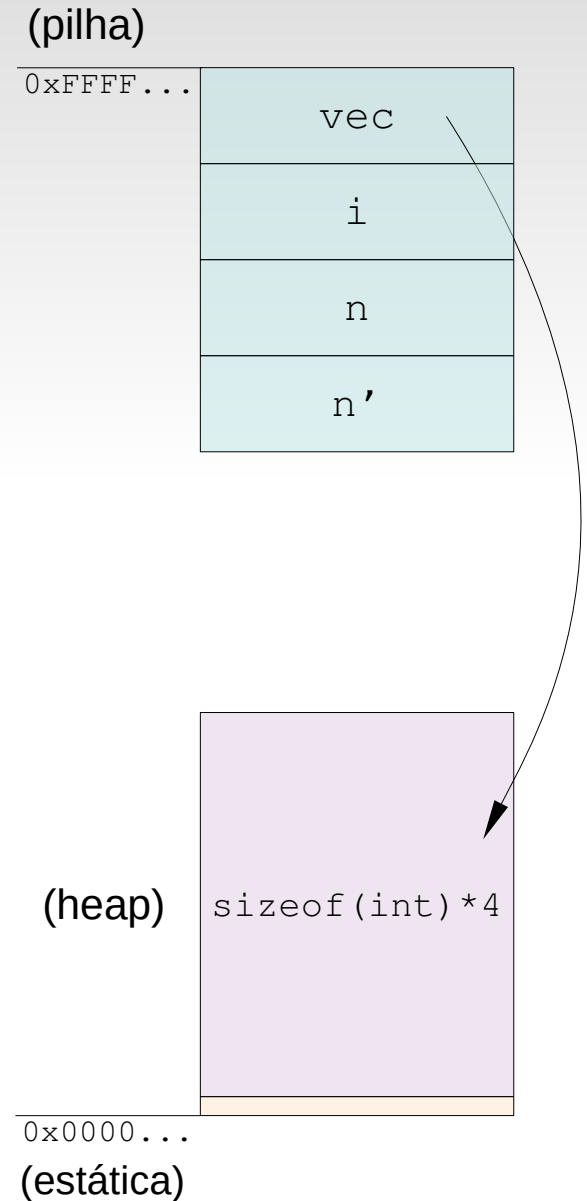
# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```



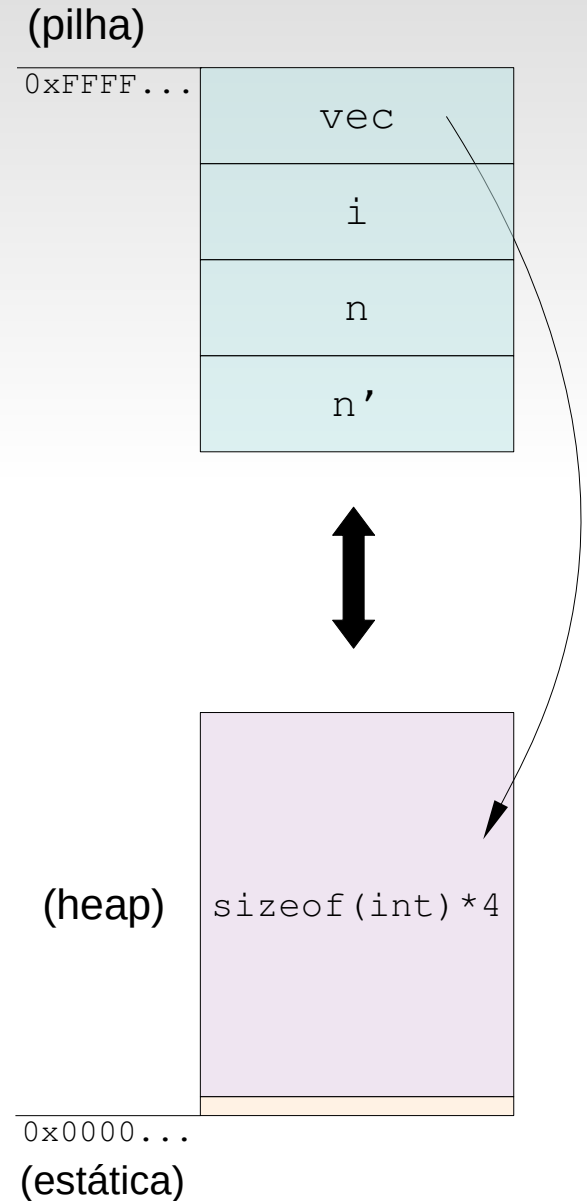
# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```



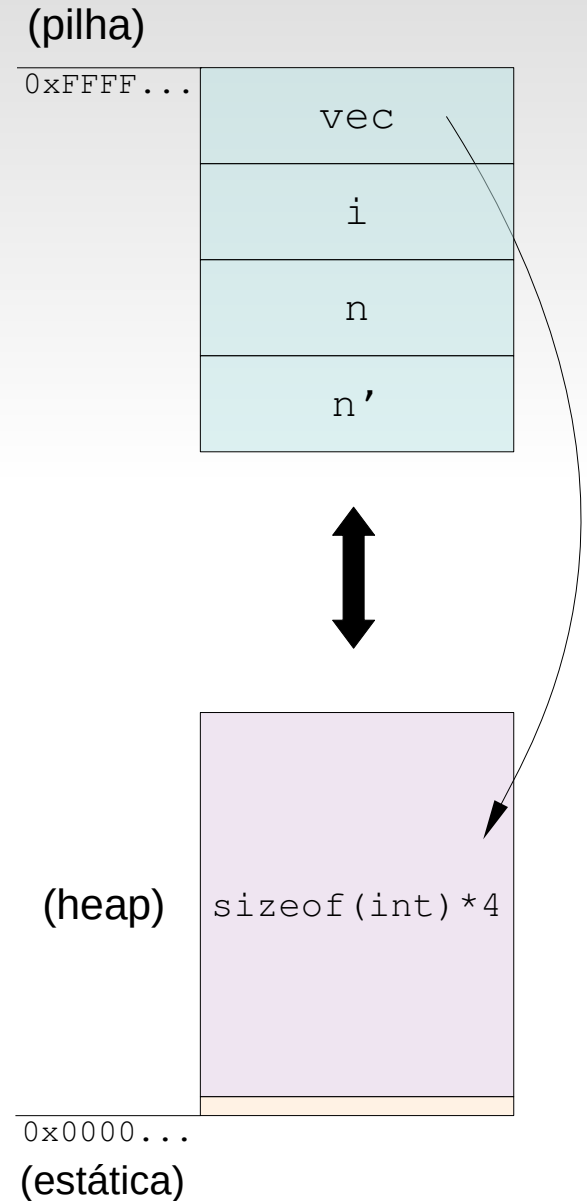
# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```



# Exemplo - Heap

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
void main (void) {  
    int* vec = malloc(sizeof(int) * 4);  
    for (int i=0; i<4; i++) {  
        vec[i] = fat(i);  
    }  
    free(vec);  
}
```



# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

(pilha)

---

0xFFFF...

---

0x0000...

(estática)

# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

(pilha)

0xFFFF...

0x0000...

(estática)

# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

(pilha)

0xFFFF...

(heap)

...

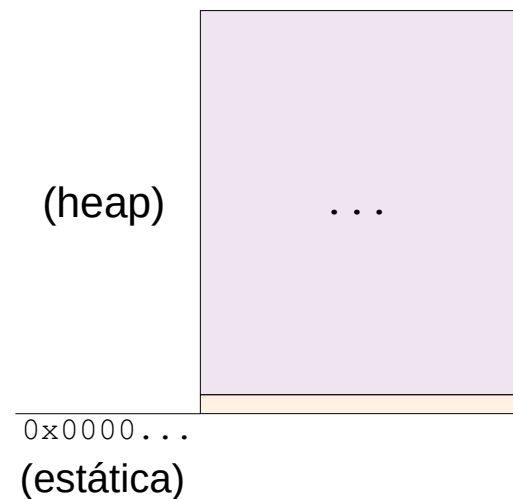
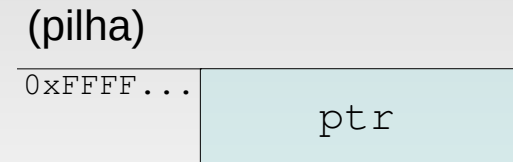
0x0000...

(estática)



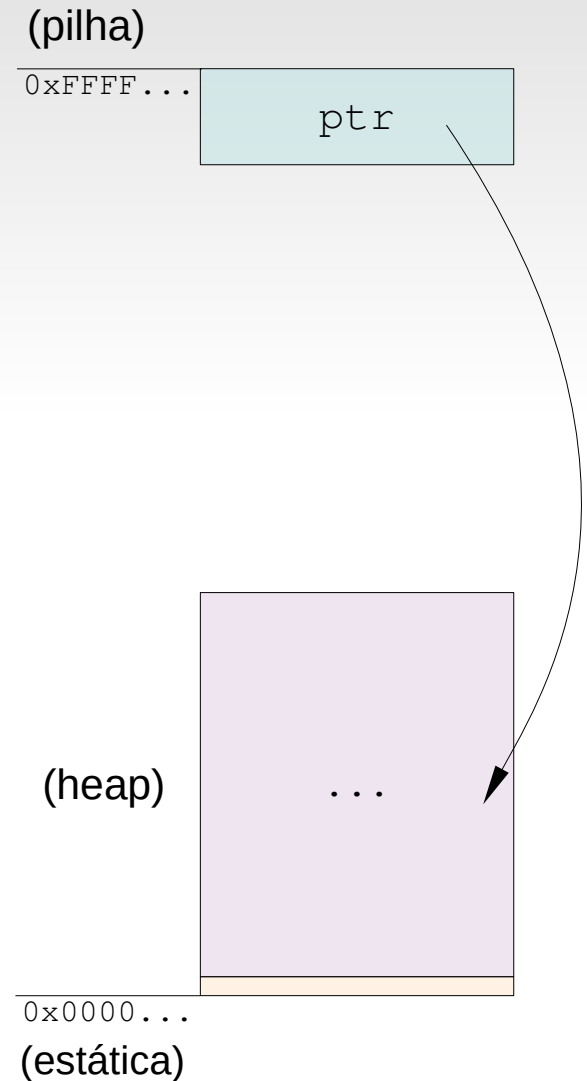
# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```



# Heap: Escopo vs Tempo de Vida

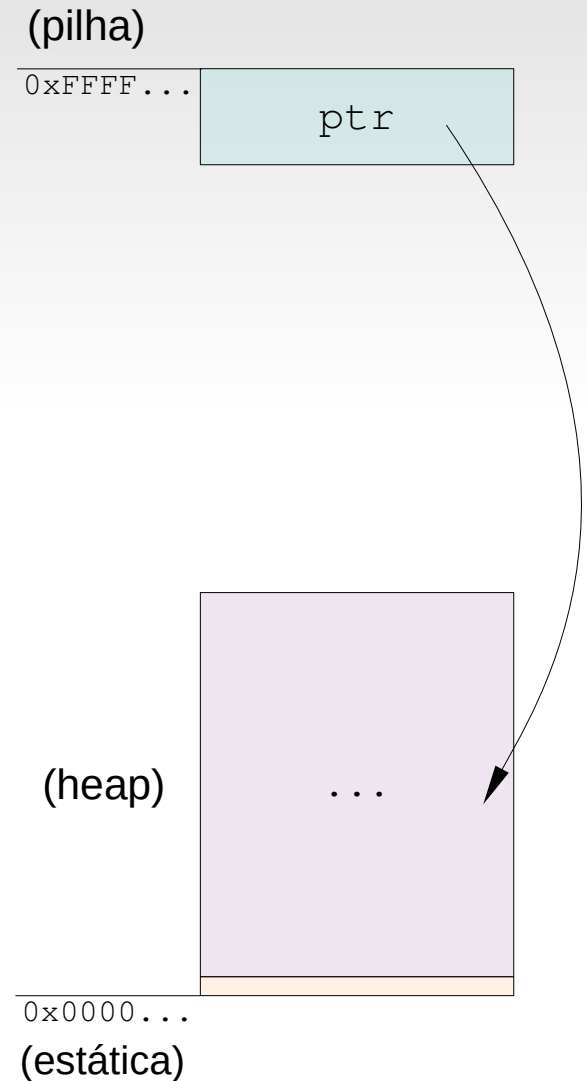
```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```



# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo = tempo de vida



# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo = tempo de vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

(pilha)

0xFFFF...

ptr

(heap)

...

0x0000...

(estática)



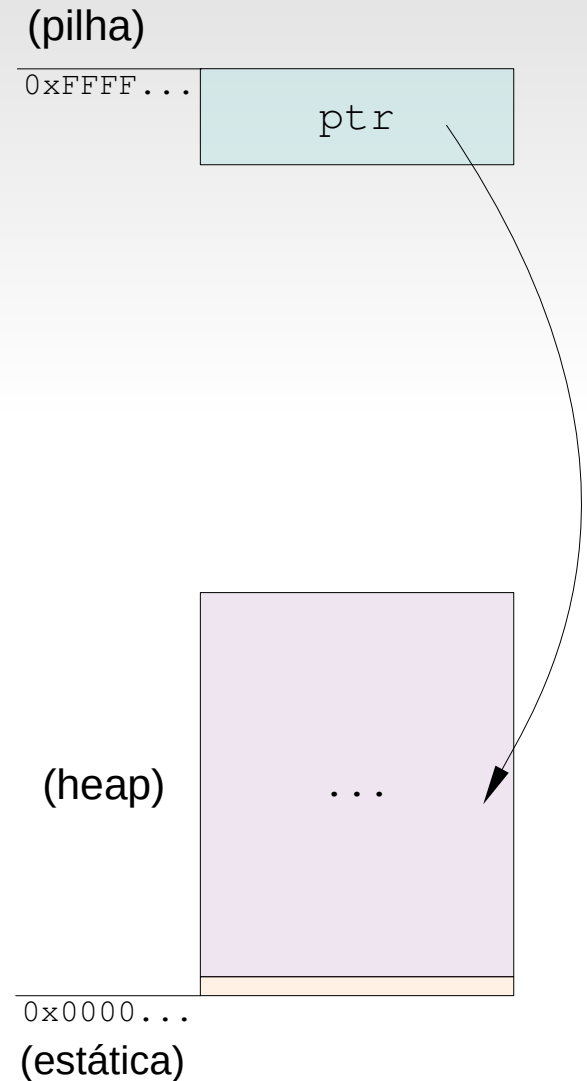
# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo = tempo de vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo < tempo de vida  
(*memory leak / vazamento de memória*)



# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo = tempo de vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo < tempo de vida  
(*memory leak /*  
*vazamento de memória*)

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
    ...  
    *ptr = ...;  
    ...  
}
```

(pilha)

0xFFFF...

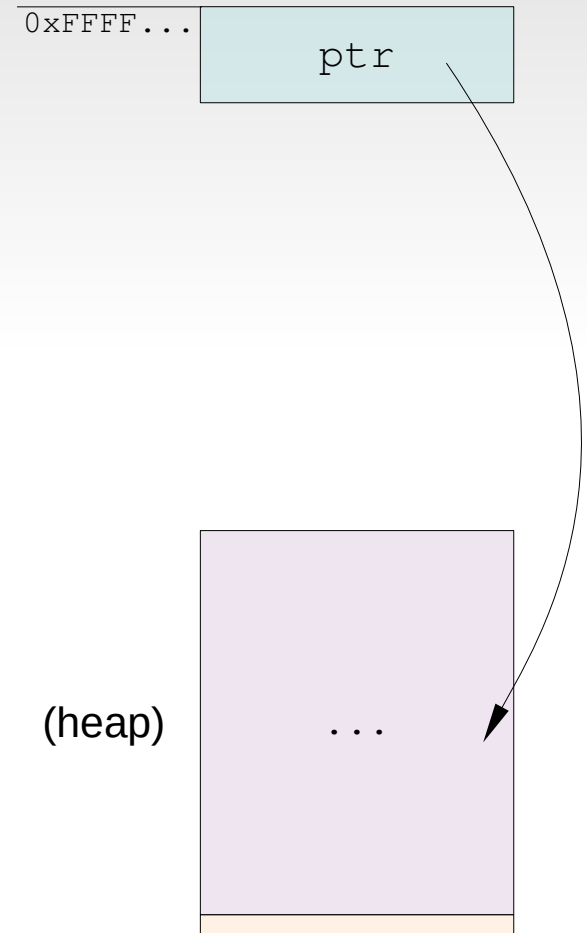
ptr

(heap)

...

0x0000...

(estática)



# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo = tempo de vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo < tempo de vida  
(memory leak /  
vazamento de memória)

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
    ...  
    *ptr = ...;  
    ...  
}
```

escopo > tempo de vida  
(dangling pointer /  
ponteiro pendente)

(pilha)

0xFFFF...

ptr

(heap)

...

0x0000...

(estática)



# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo = tempo de vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo < tempo de vida  
(memory leak /  
vazamento de memória)

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
    ...  
    *ptr = ...;  
    ...  
}
```

escopo > tempo de vida  
(dangling pointer /  
ponteiro pendente)

(pilha)

0xFFFF...

ptr

(heap)

...

0x0000...

(estática)





# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

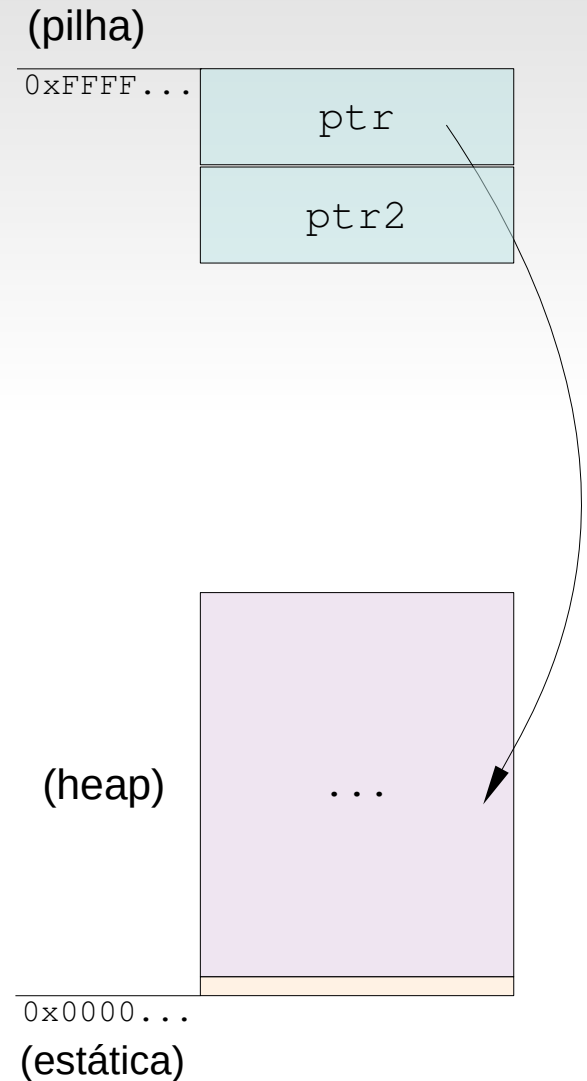
escopo = tempo de vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo < tempo de vida  
(memory leak /  
vazamento de memória)

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
    ...  
    *ptr = ...;  
    ...  
}
```

escopo > tempo de vida  
(dangling pointer /  
ponteiro pendente)



# Heap: Escopo vs Tempo de Vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo = tempo de vida

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
}
```

escopo < tempo de vida  
(memory leak /  
vazamento de memória)

```
{  
    int* ptr = malloc(...);  
    ...  
    free(ptr);  
    ...  
    *ptr = ...;  
    ...  
}
```

escopo > tempo de vida  
(dangling pointer /  
ponteiro pendente)

(pilha)

0xFFFF...

ptr

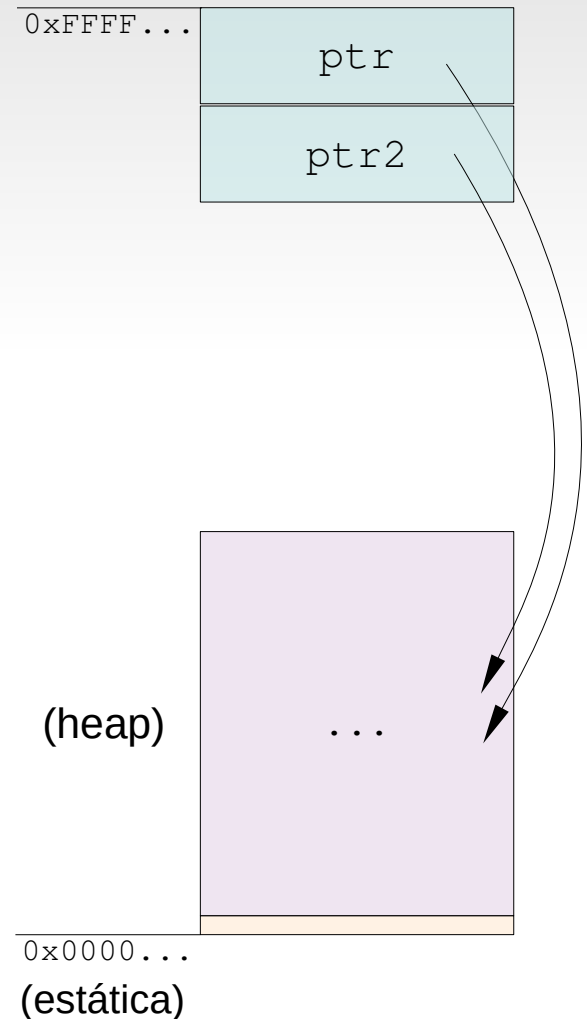
ptr2

(heap)

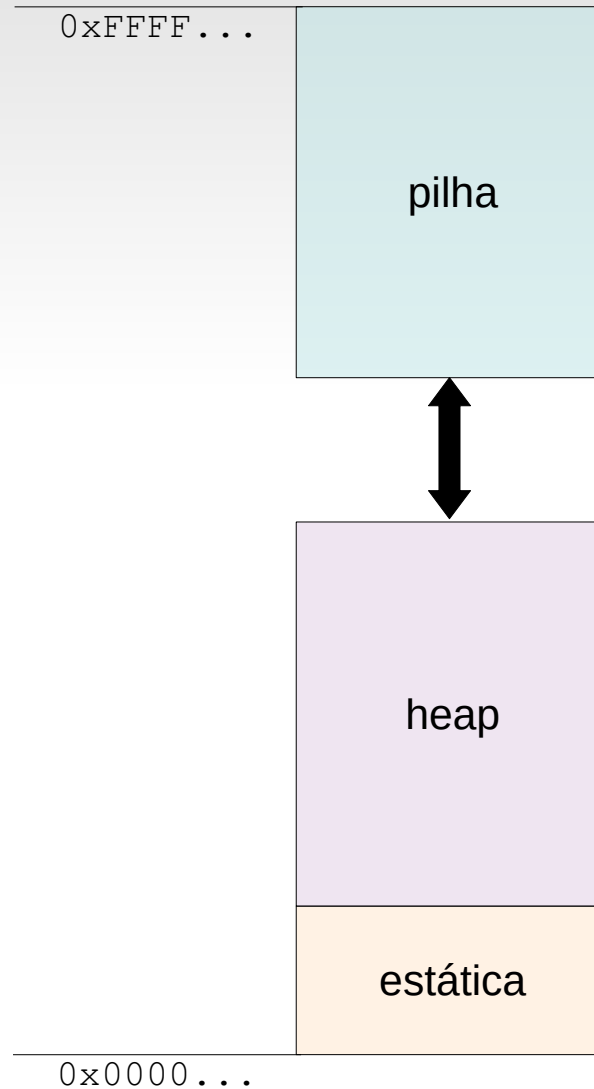
...

0x0000...

(estática)



# Organização da Memória



# Exercícios

1. Pegue um programa seu razoavelmente grande e enumere todos os usos de memória estática, pilha e heap.
  - O programa deve ter pelo menos 5 usos da pilha e da heap.
2. Pesquise sobre a pilha em alguma linguagem ou arquitetura.
  - Qual é o tamanho da pilha em bytes?
  - Quantas chamadas recursivas são suportadas?
  - Quantas locais eu posso guardar em cada chamada?

# *Amarração de Nomes*

(continuação...)

Estrutura de Linguagens

<https://github.com/fsantanna-uerj/EDL/>

Francisco Sant'Anna

[francisco@ime.uerj.br](mailto:francisco@ime.uerj.br)



# Estática

# Estática

- Amarração nome/endereço ocorre antes da execução

# Estática

- Amarração nome/endereço ocorre antes da execução
- Globais



# Estática

- Amarração nome/endereço ocorre antes da execução
- Globais
- Trade-offs:

# Estática

- Amarração nome/endereço ocorre antes da execução
- Globais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente

# Estática

- Amarração nome/endereço ocorre antes da execução
- Globais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente
  - Desempenho (+)
    - Acesso direto
    - Sem overhead de alocação/desalocação

# Estática

- Amarração nome/endereço ocorre antes da execução
- Globais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente
  - Desempenho (+)
    - Acesso direto
    - Sem overhead de alocação/desalocação
  - Expressividade (-)
    - Recursão

# Estática

- Amarração nome/endereço ocorre antes da execução
- Globais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente
  - Desempenho (+)
    - Acesso direto
    - Sem overhead de alocação/desalocação
  - Expressividade (-)
    - Recursão
  - Uso de Espaço (-)
    - Compartilhamento de memória

# Estática

- Amarração nome/endereço ocorre antes da execução
- Globais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente
  - Desempenho (+)
    - Acesso direto
    - Sem overhead de alocação/desalocação
  - Expressividade (-)
    - Recursão
  - Uso de Espaço (-)
    - Compartilhamento de memória

```
{  
    int soma = ...  
    ...  
}  
  
{  
    int media = ...  
    ...  
}
```

# Pilha (dinâmico)

# Pilha (dinâmico)

- Amarração nome/endereço ocorre durante a execução



# Pilha (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Locais

# Pilha (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Locais
- Trade-offs:

# Pilha (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Locais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente

# Pilha (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Locais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente
  - Desempenho (+)
    - Acesso indireto (via `%ebp`)
    - Pouco overhead de alocação/desalocação (em bloco)

# Pilha (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Locais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente
  - Desempenho (+)
    - Acesso indireto (via `%ebp`)
    - Pouco overhead de alocação/desalocação (em bloco)

(pilha)

0xFFFF...

i

n

n'

n''

ebp'''

n''''

# Pilha (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Locais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente
  - Desempenho (+)
    - Acesso indireto (via `%ebp`)
    - Pouco overhead de alocação/desalocação (em bloco)
  - Expressividade (+/-)
    - Recursão

(pilha)

0xFFFF...

i

n

n'

n''

ebp'''

n'''

# Pilha (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Locais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente
  - Desempenho (+)
    - Acesso indireto (via `%ebp`)
    - Pouco overhead de alocação/desalocação (em bloco)
  - Expressividade (+/-)
    - Recursão
  - Uso de Espaço (+)
    - Compartilhamento de memória

(pilha)

0xFFFF...

i

n

n'

n''

ebp'''

n'''

# Pilha (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Locais
- Trade-offs:
  - Segurança (+)
    - Escopo / Tempo de vida gerenciados automaticamente
  - Desempenho (+)
    - Acesso indireto (via `%ebp`)
    - Pouco overhead de alocação/desalocação (em bloco)
  - Expressividade (+/-)
    - Recursão
  - Uso de Espaço (+)
    - Compartilhamento de memória

(pilha)

0xFFFF...

i

n

n'

n''

ebp'''

n''''

```
{  
    int soma = ...  
    ...  
}  
  
{  
    int media = ...  
    ...  
}
```



# Pilha (dinâmico)

```
function funcC ()  
    local g,h,i  
end
```

```
function funcB ()  
    local d,e,f  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```

# Pilha (dinâmico)

```
function funcC ()  
    local g,h,i  
end
```

```
function funcB ()  
    local d,e,f  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```

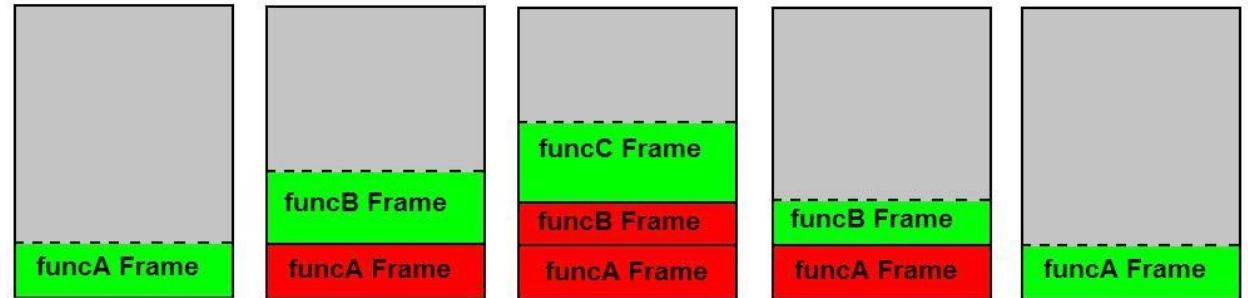
# Pilha (dinâmico)

```
function funcC ()  
    local g,h,i  
end
```

```
function funcB ()  
    local d,e,f  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```



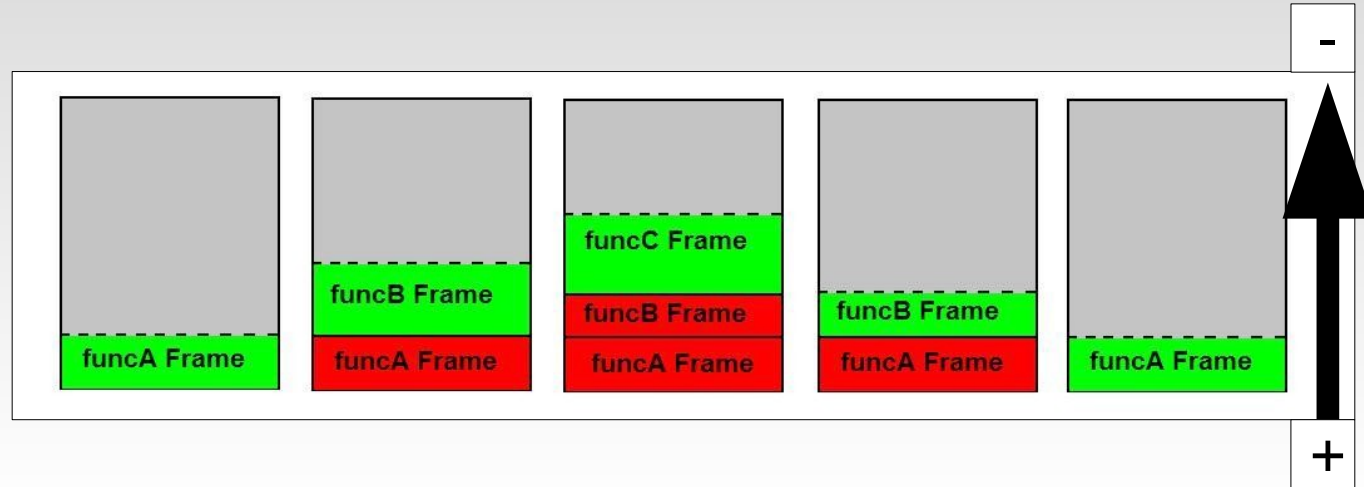
# Pilha (dinâmico)

```
function funcC ()  
    local g,h,i  
end
```

```
function funcB ()  
    local d,e,f  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```



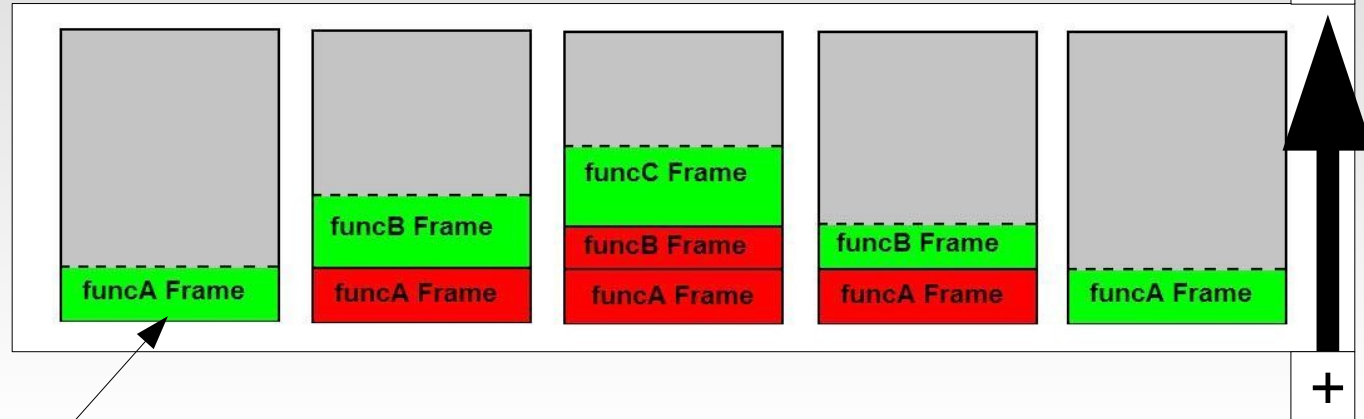
# Pilha (dinâmico)

```
function funcC ()  
    local g,h,i  
end
```

```
function funcB ()  
    local d,e,f  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```



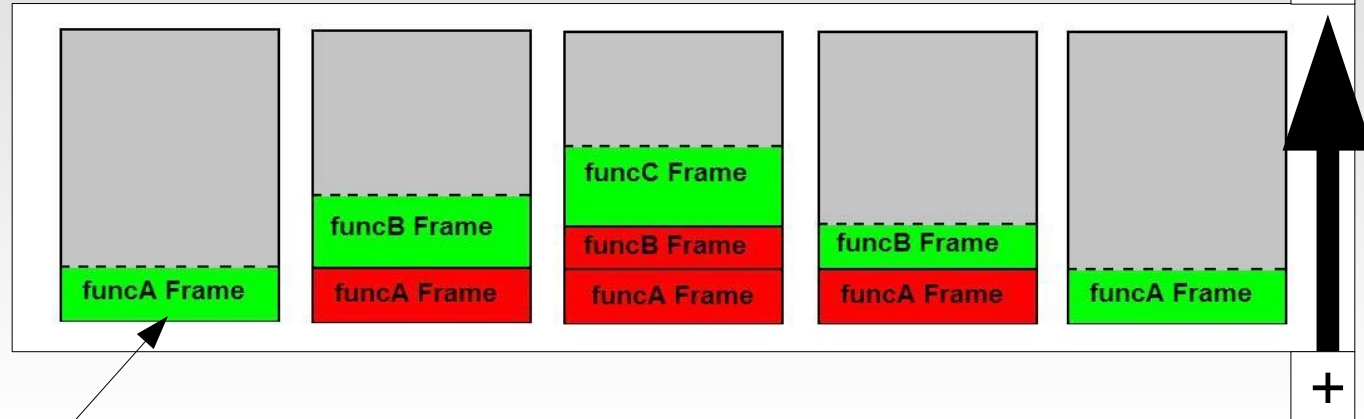
# Pilha (dinâmico)

```
function funcC ()  
  local g,h,i  
end
```

```
function funcB ()  
  local d,e,f  
  funcC()  
end
```

```
function funcA ()  
  local a,b,c  
  funcB()  
end
```

```
funcA()
```



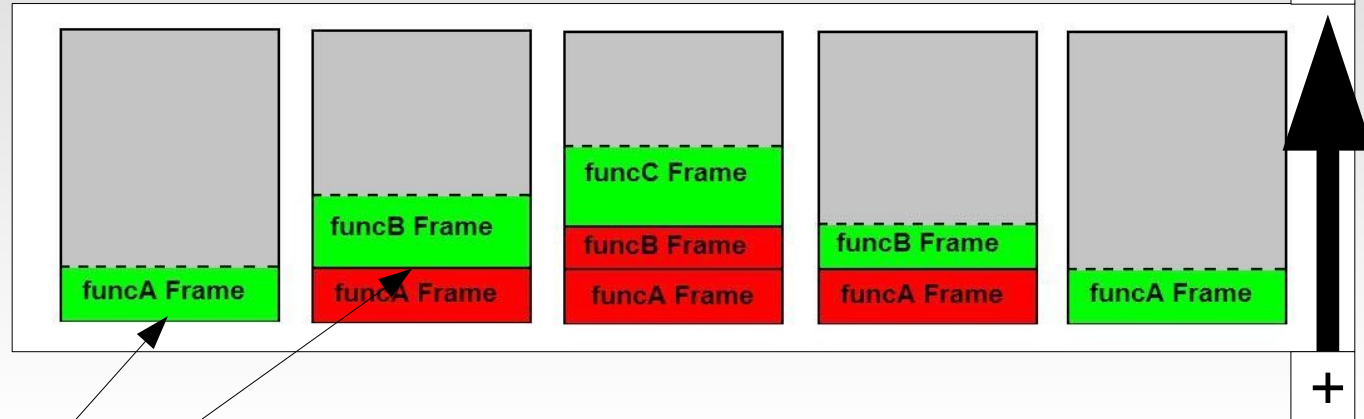
# Pilha (dinâmico)

```
function funcC ()  
    local g,h,i  
end
```

```
function funcB ()  
    local d,e,f  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```



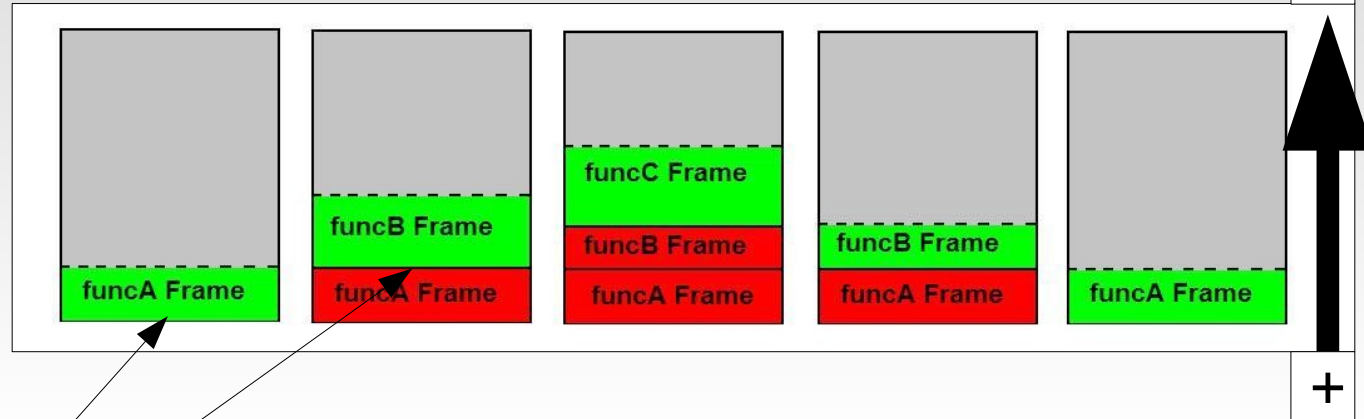
# Pilha (dinâmico)

```
function funcC ()  
    local g,h,i  
end
```

```
function funcB ()  
    local d,e,f  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```





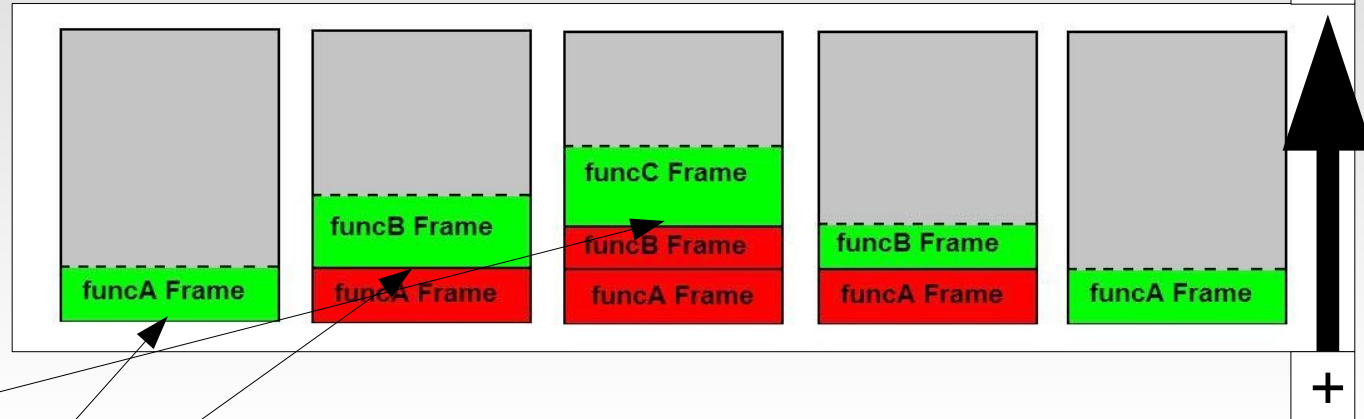
# Pilha (dinâmico)

```
function funcC ()  
    local g,h,i  
end
```

```
function funcB ()  
    local d,e,f  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```



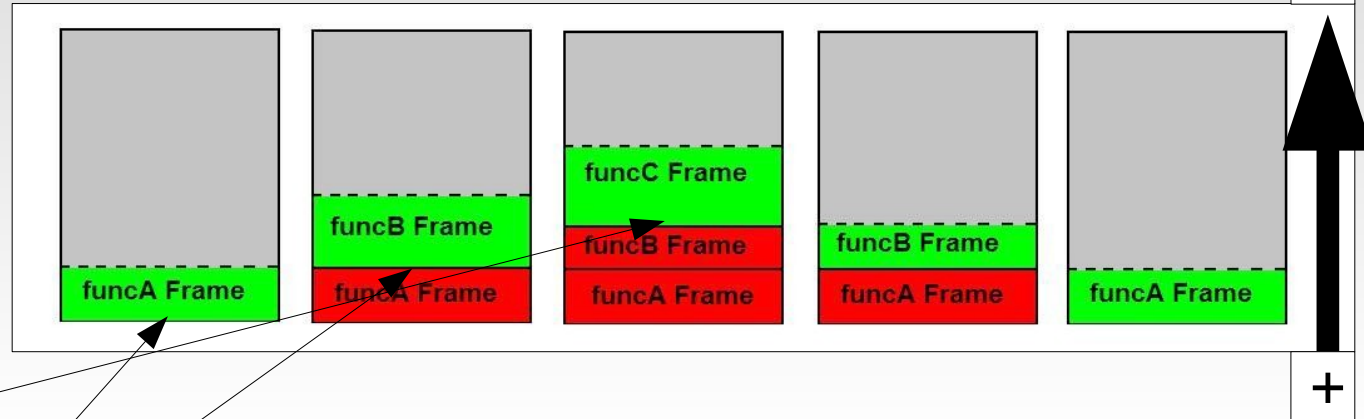
# Pilha (dinâmico)

```
function funcC ()  
  local g,h,i  
end
```

```
function funcB ()  
  local d,e,f  
  funcC()  
end
```

```
function funcA ()  
  local a,b,c  
  funcB()  
end
```

```
funcA()
```



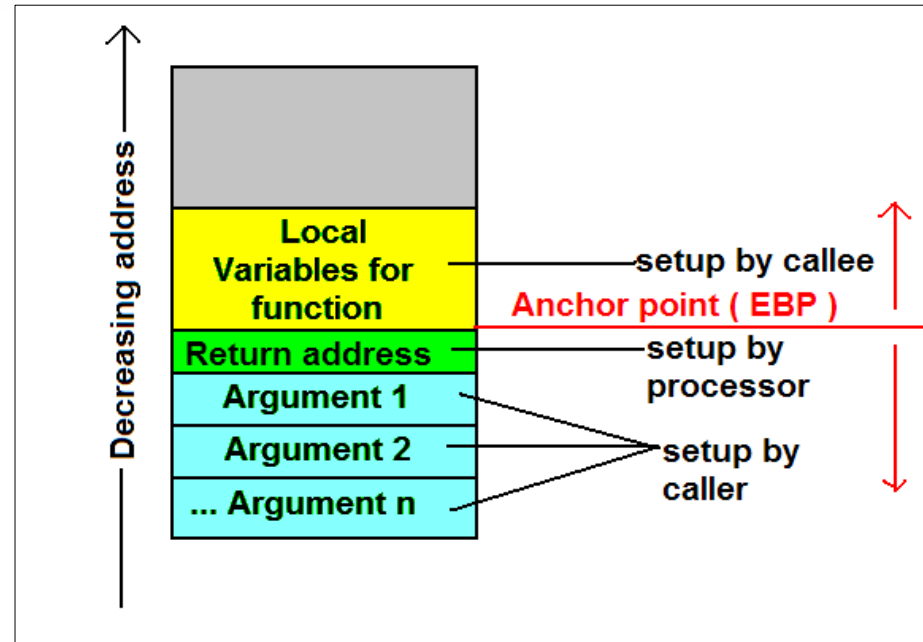
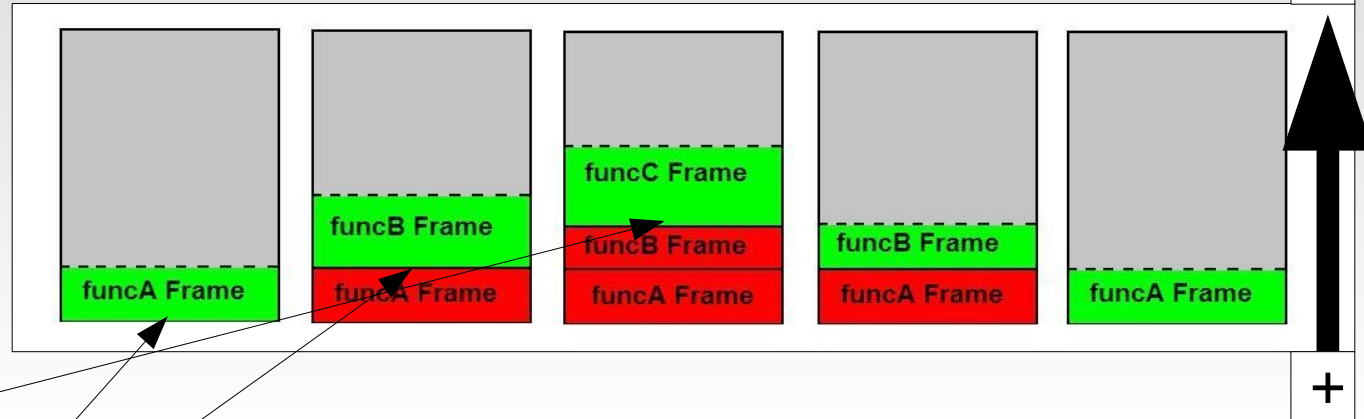
# Pilha (dinâmico)

```
function funcC ()  
    local g,h,i  
end
```

```
function funcB ()  
    local d,e,f  
    funcC()  
end
```

```
function funcA ()  
    local a,b,c  
    funcB()  
end
```

```
funcA()
```



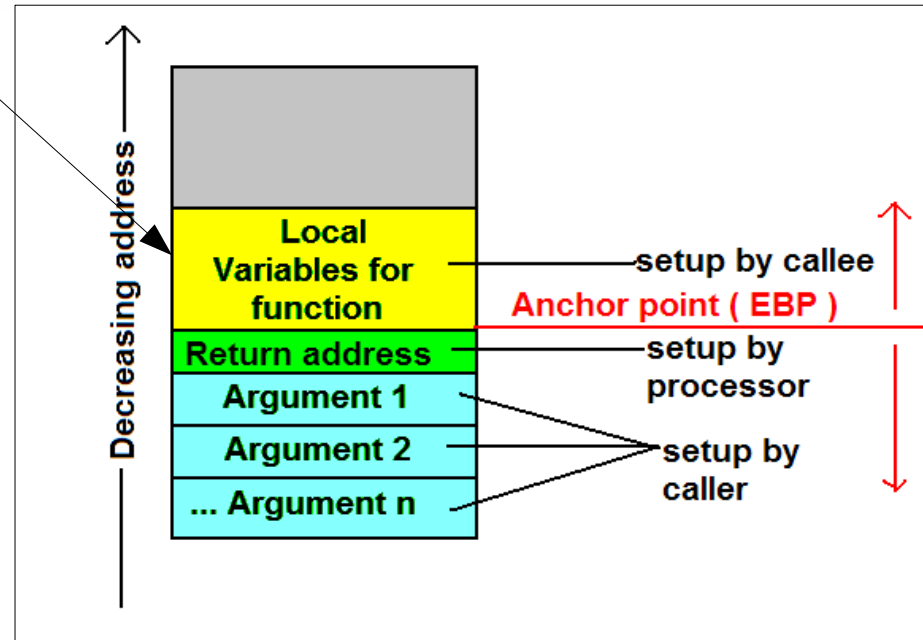
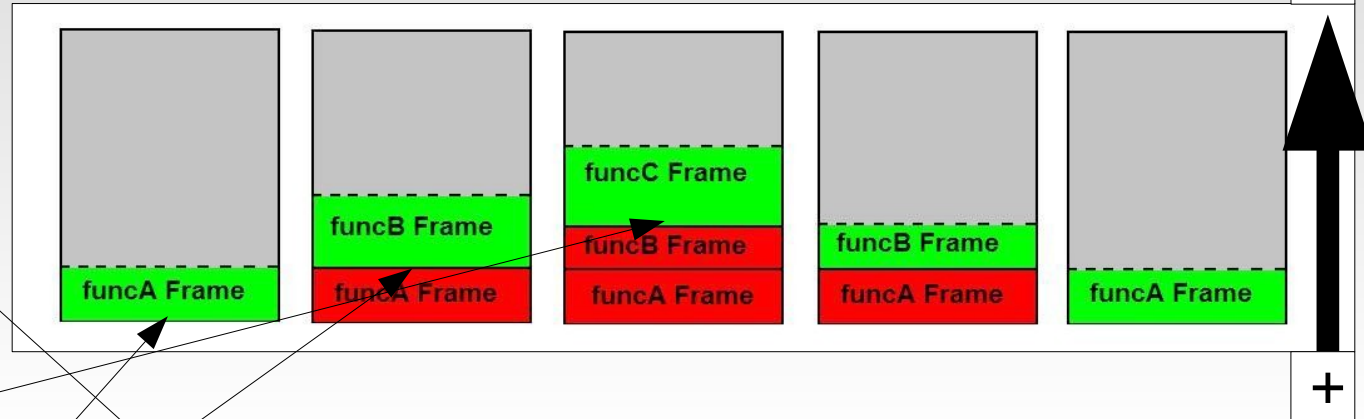
# Pilha (dinâmico)

```
function funcC ()  
  local g,h,i  
end
```

```
function funcB ()  
  local d,e,f  
  funcC()  
end
```

```
function funcA ()  
  local a,b,c  
  funcB()  
end
```

```
funcA()
```



# Pilha (dinâmico)



Wikipedia says:

15

Early languages like Fortran did not initially support recursion because variables were statically allocated, as well as the location for the return address.



[http://en.wikipedia.org/wiki/Subroutine#Local\\_variables.2C\\_recursion\\_and\\_re-entrancy](http://en.wikipedia.org/wiki/Subroutine#Local_variables.2C_recursion_and_re-entrancy)

FORTRAN 77 does not allow recursion, Fortran 90 does, (recursive routines must be explicitly declared so).

Most FORTRAN 77 compilers allow recursion, some (e.g. DEC) require using a compiler option (see compiler options chapter). The GNU g77, which conforms strictly to the Fortran 77 standard, doesn't allow recursion at all.

<http://www.ibiblio.org/pub/languages/fortran/ch1-12.html>

share improve this answer

answered Nov 24 '10 at 21:31



**Robert Harvey**

124k ● 30 ● 276 ● 458

# Heap (dinâmico)

# Heap (dinâmico)

- Amarração nome/endereço ocorre durante a execução

# Heap (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Estruturas dinâmicas
  - Alocação e desalocação explícita (e.g., `malloc/free/new/GC`)



# Heap (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Estruturas dinâmicas
  - Alocação e desalocação explícita (e.g., `malloc/free/new/GC`)
- Trade-offs:

# Heap (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Estruturas dinâmicas
  - Alocação e desalocação explícita (e.g., `malloc/free/new/GC`)
- Trade-offs:
  - Segurança (-)
    - Escopo / Tempo de vida gerenciados manualmente (GC)

# Heap (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Estruturas dinâmicas
  - Alocação e desalocação explícita (e.g., `malloc/free/new/GC`)
- Trade-offs:
  - Segurança (-)
    - Escopo / Tempo de vida gerenciados manualmente (GC)
  - Desempenho (-)
    - Acesso indireto (via ponteiro/alias)
    - Overhead de alocação/desalocação (por objeto) (GC)

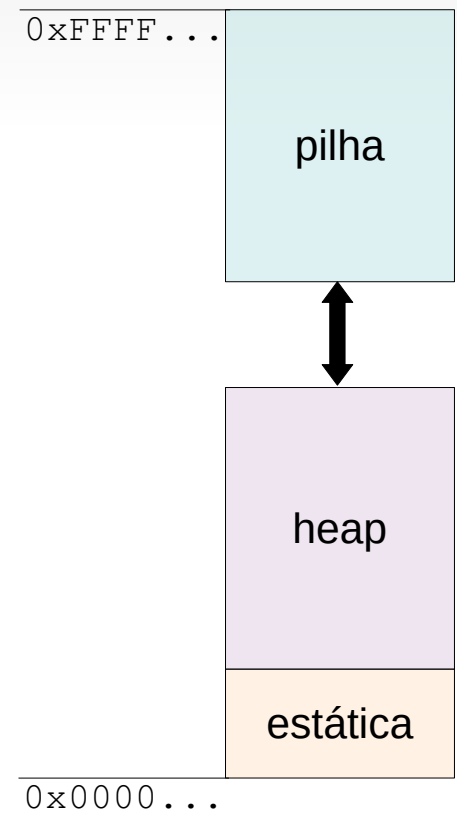
# Heap (dinâmico)

- Amarração nome/endereço ocorre durante a execução
- Estruturas dinâmicas
  - Alocação e desalocação explícita (e.g., `malloc/free/new/GC`)
- Trade-offs:
  - Segurança (-)
    - Escopo / Tempo de vida gerenciados manualmente (GC)
  - Desempenho (-)
    - Acesso indireto (via ponteiro/alias)
    - Overhead de alocação/desalocação (por objeto) (GC)
  - Expressividade (+)
    - Estruturas dinâmicas ajustáveis (vetores, listas, árvores)

# Heap (dinâmico)

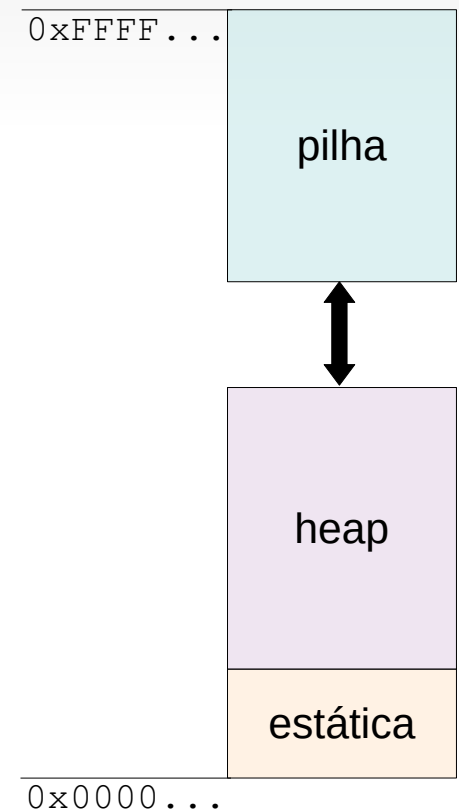
- Amarração nome/endereço ocorre durante a execução
- Estruturas dinâmicas
  - Alocação e desalocação explícita (e.g., `malloc/free/new/GC`)
- Trade-offs:
  - Segurança (-)
    - Escopo / Tempo de vida gerenciados manualmente (GC)
  - Desempenho (-)
    - Acesso indireto (via ponteiro/alias)
    - Overhead de alocação/desalocação (por objeto) (GC)
  - Expressividade (+)
    - Estruturas dinâmicas ajustáveis (vetores, listas, árvores)
  - Uso de espaço (+/-)
    - Controle total de alocação e desalocação
    - Depende do alocador

# Heap (dinâmico)



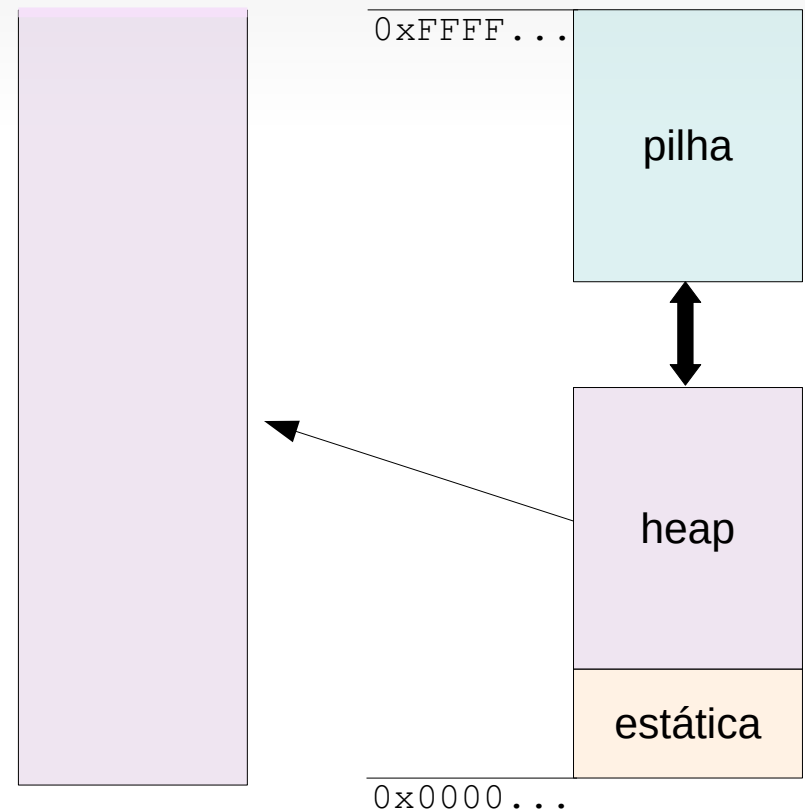
# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente



# Heap (dinâmico)

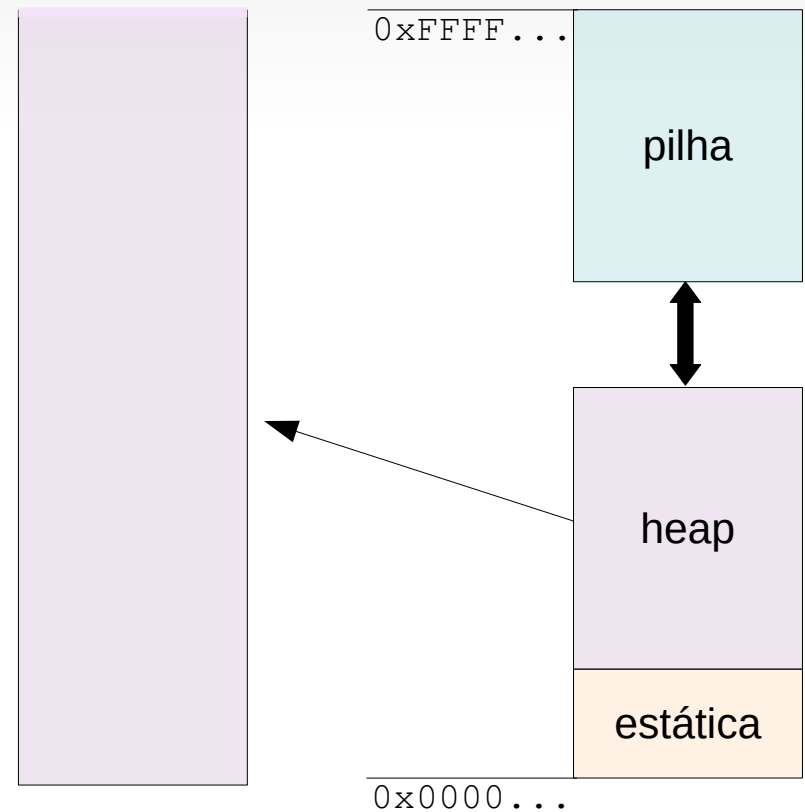
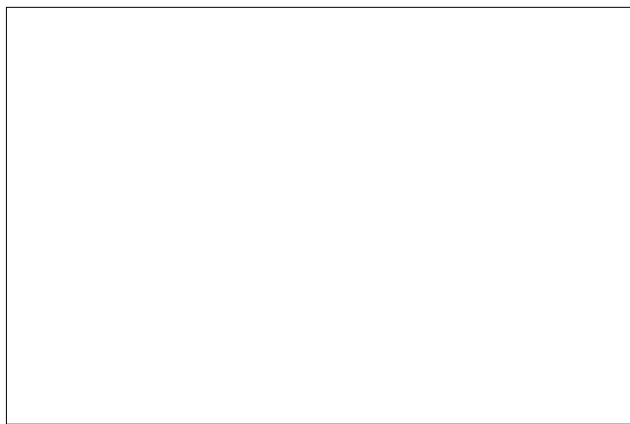
- Alocação “aleatória”
  - impossível determinar padrão estaticamente





# Heap (dinâmico)

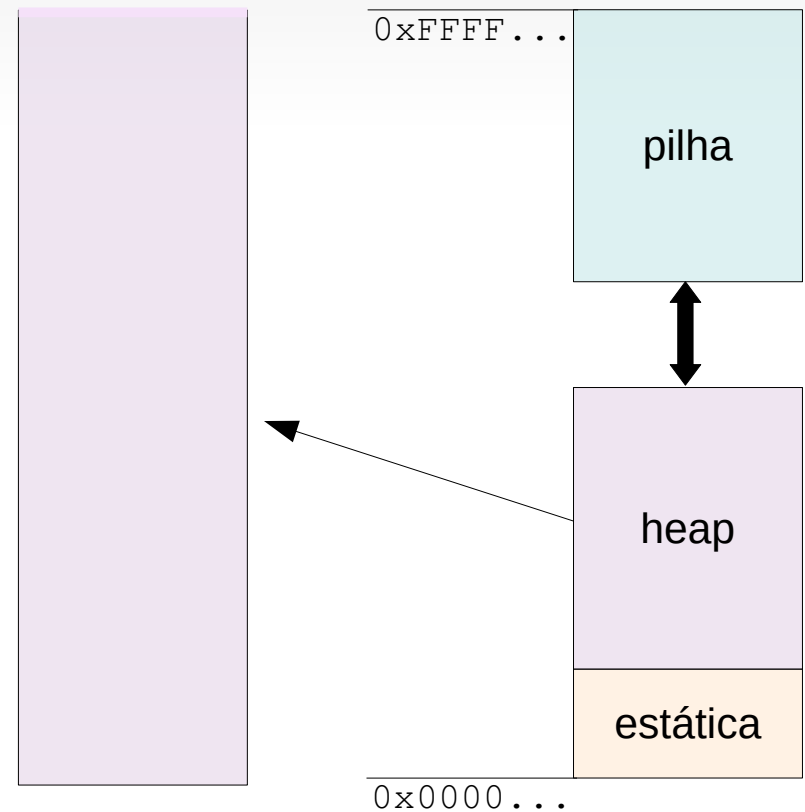
- Alocação “aleatória”
  - impossível determinar padrão estaticamente



# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

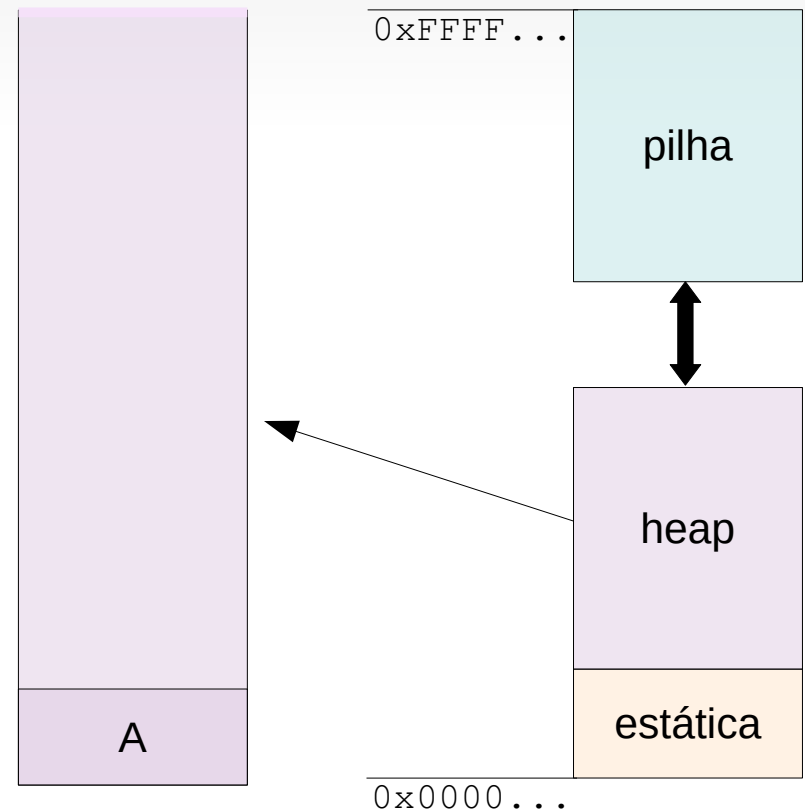
```
A = malloc(10);
```



# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

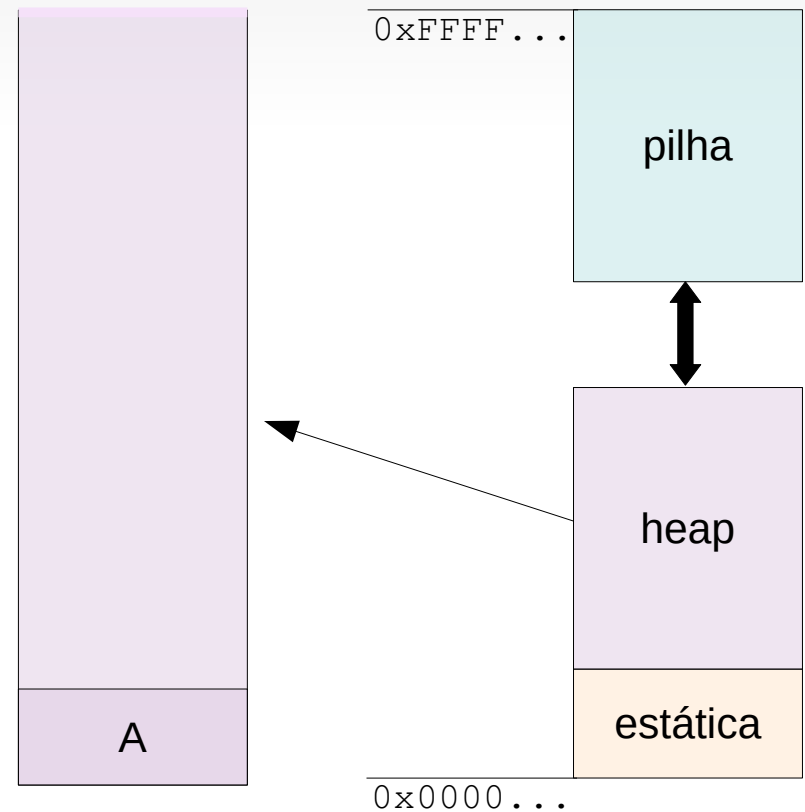
```
A = malloc(10);
```



# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

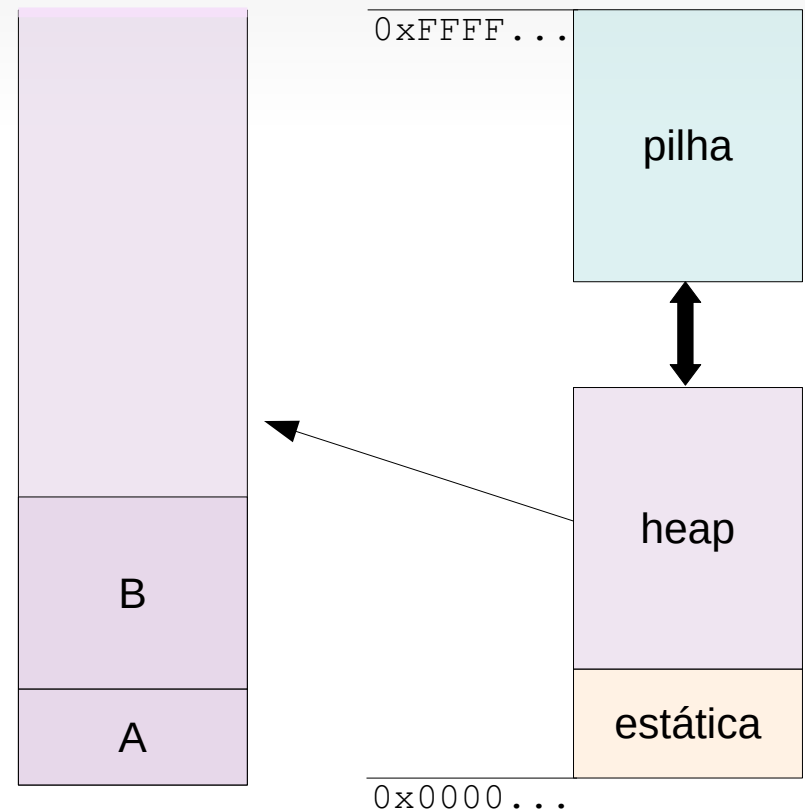
```
A = malloc(10);  
B = malloc(20);
```



# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

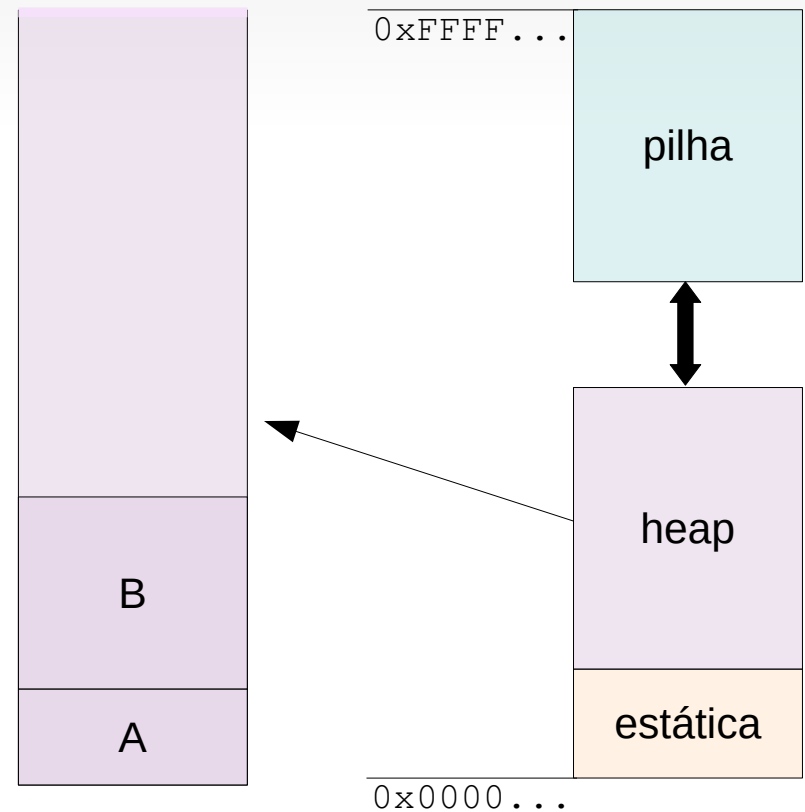
```
A = malloc(10);  
B = malloc(20);
```



# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

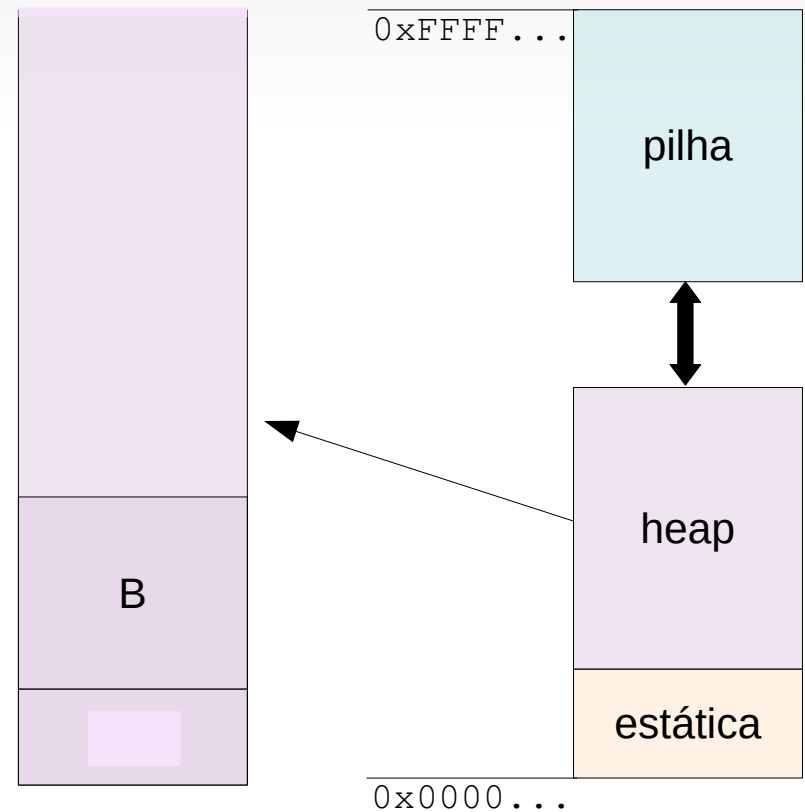
```
A = malloc(10);  
B = malloc(20);  
free(A);
```



# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

```
A = malloc(10);  
B = malloc(20);  
free(A);
```

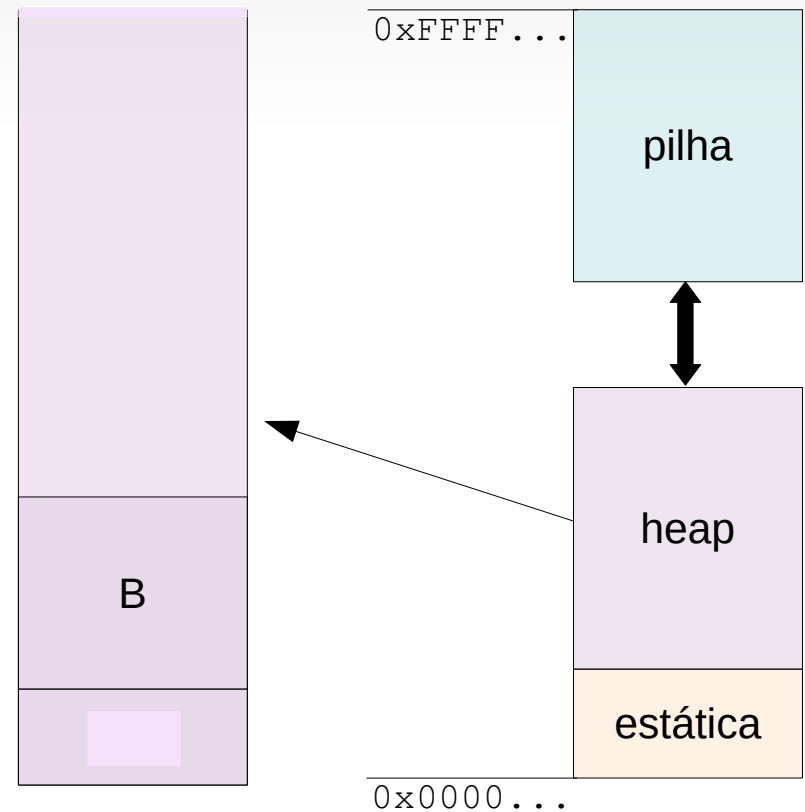


# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

```
A = malloc(10);  
B = malloc(20);  
free(A);
```

fragmentação



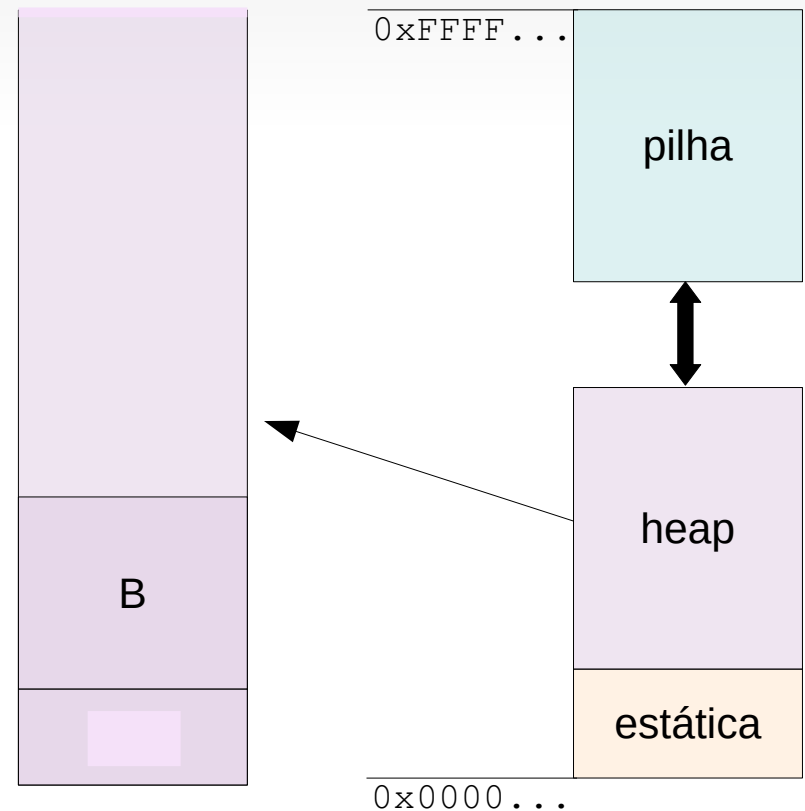


# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

```
A = malloc(10);  
B = malloc(20);  
free(A);  
C = malloc(15);
```

fragmentação

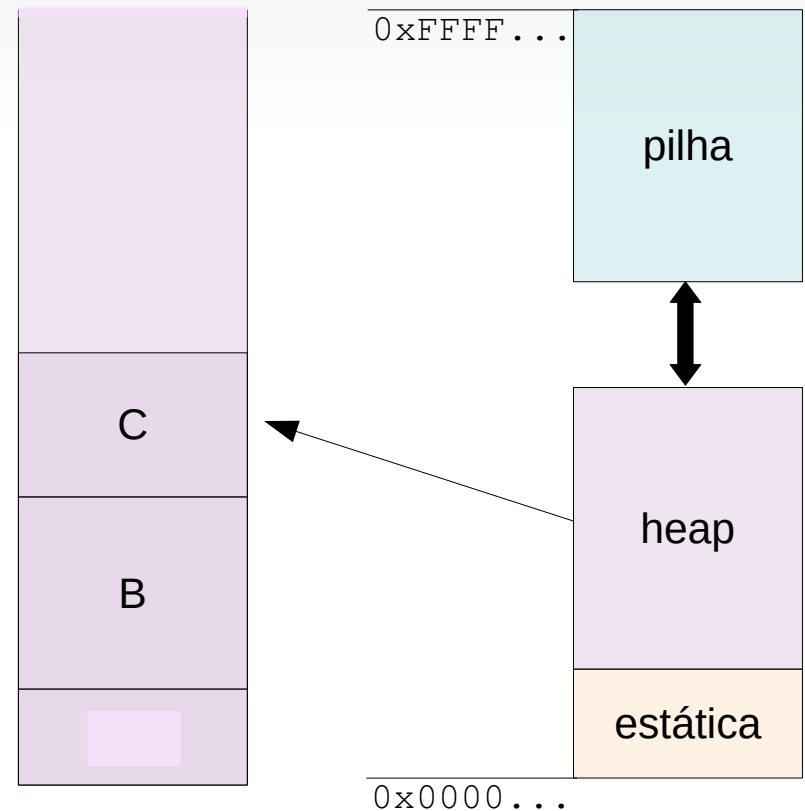


# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

```
A = malloc(10);  
B = malloc(20);  
free(A);  
C = malloc(15);
```

fragmentação

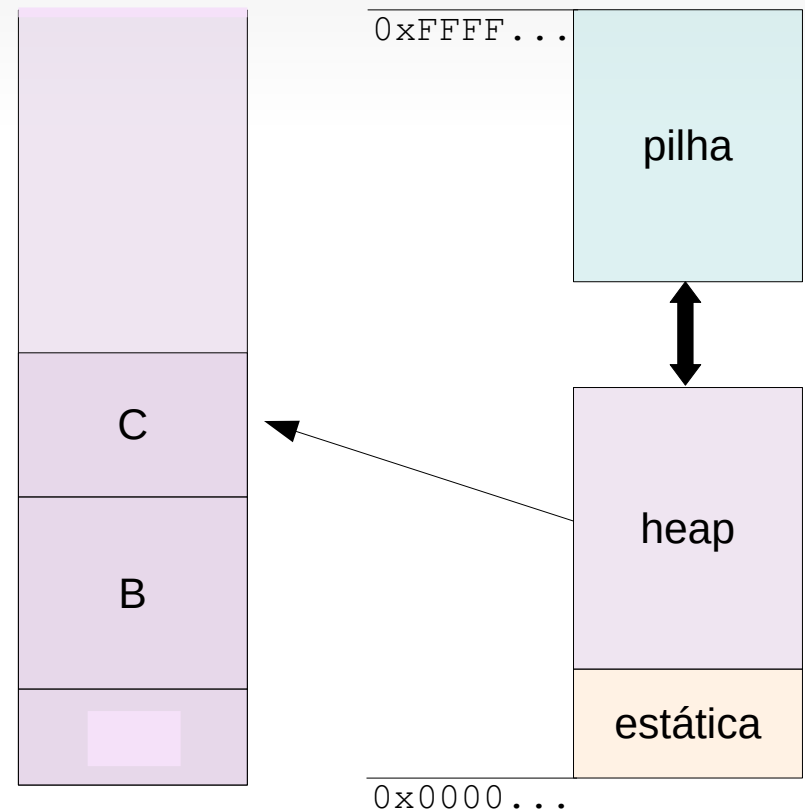


# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

```
A = malloc(10);  
B = malloc(20);  
free(A);  
C = malloc(15);  
free(B);
```

fragmentação

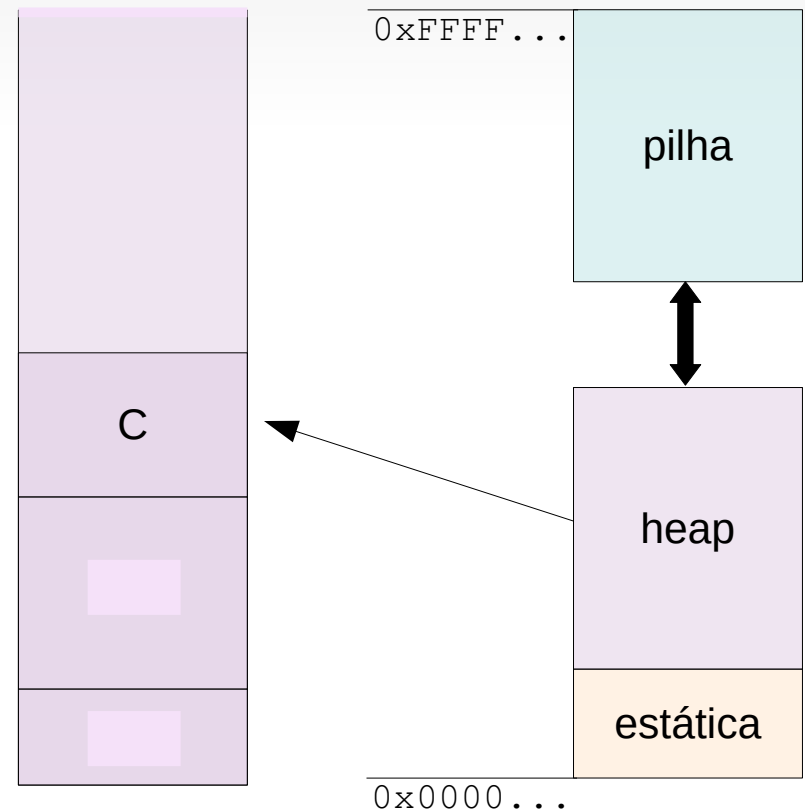


# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

```
A = malloc(10);  
B = malloc(20);  
free(A);  
C = malloc(15);  
free(B);
```

fragmentação

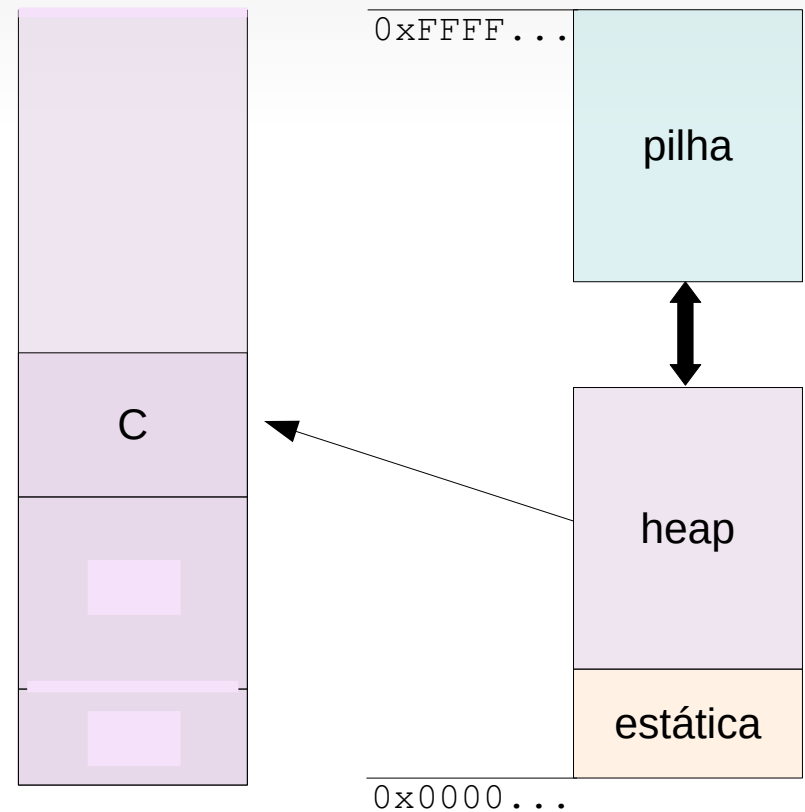


# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

```
A = malloc(10);  
B = malloc(20);  
free(A);  
C = malloc(15);  
free(B);
```

fragmentação

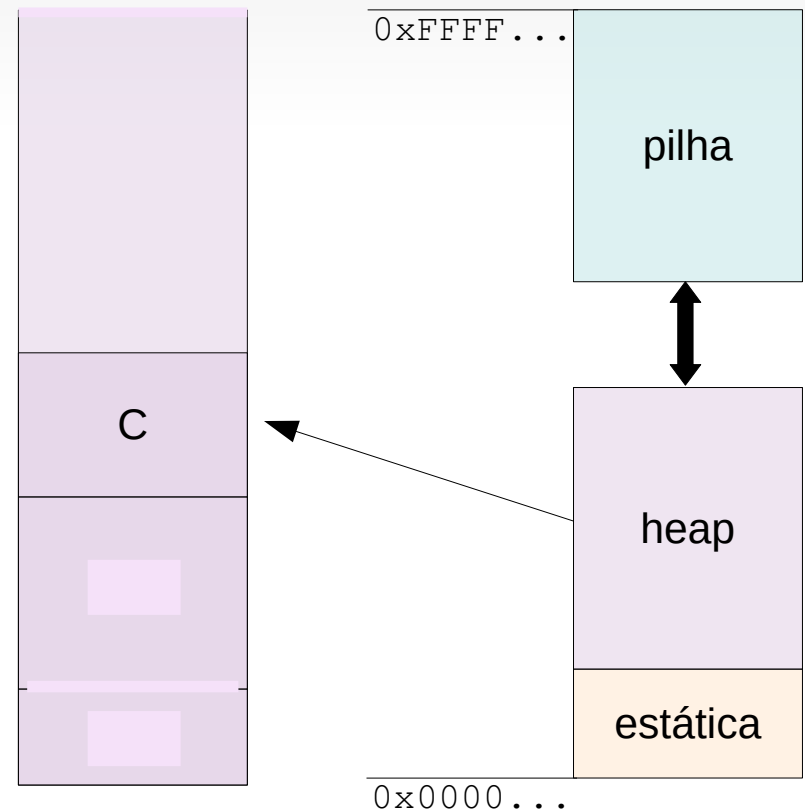


# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

```
A = malloc(10);  
B = malloc(20);  
free(A);  
C = malloc(15);  
free(B);  
D = malloc(40);
```

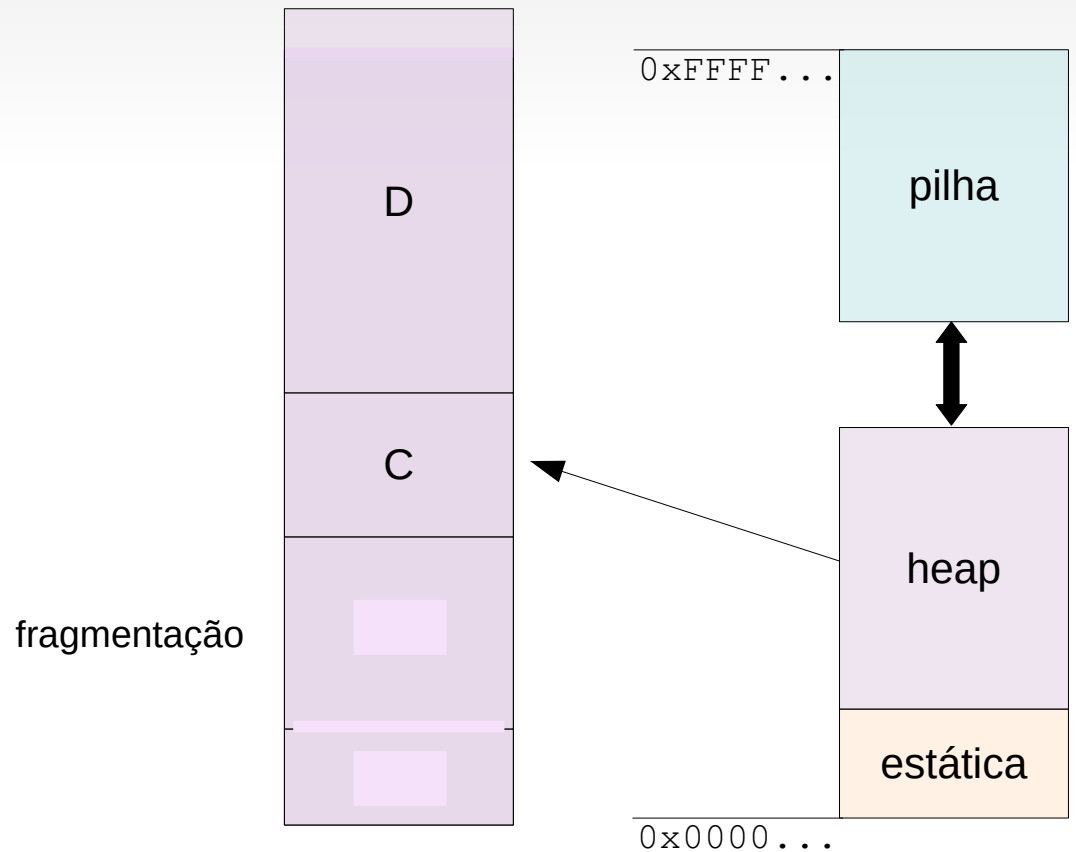
fragmentação



# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

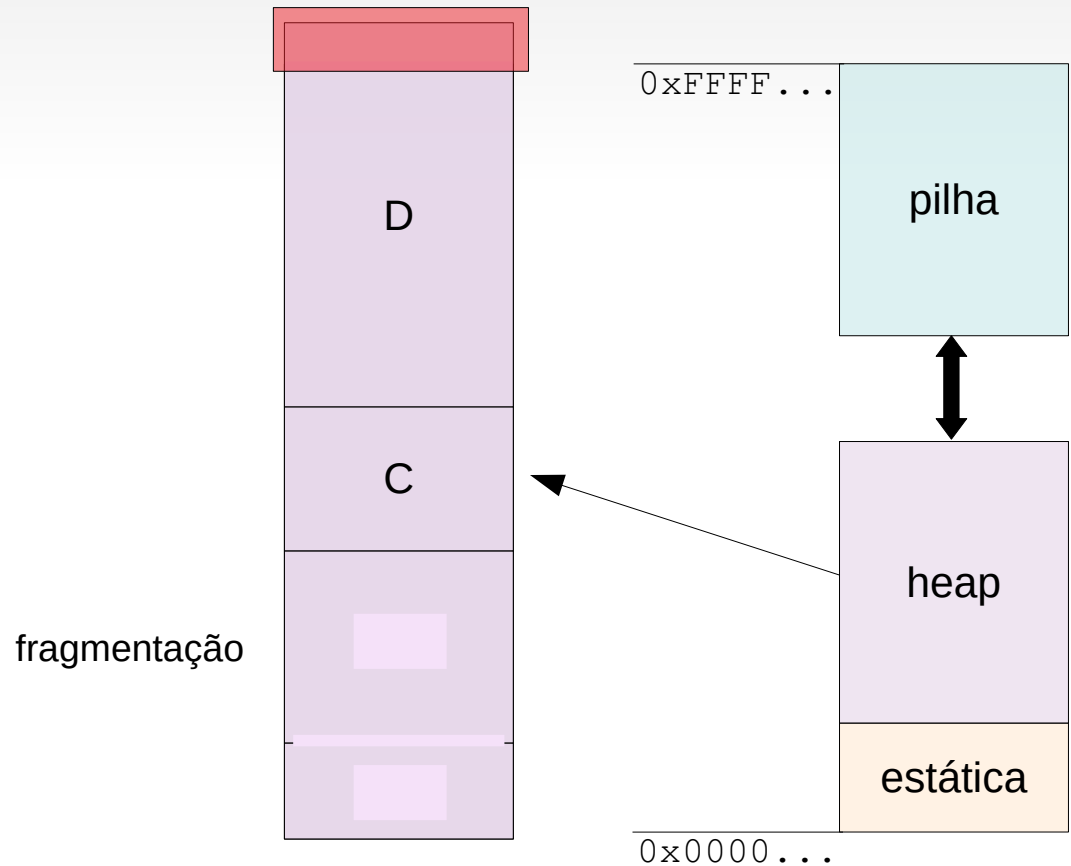
```
A = malloc(10);  
B = malloc(20);  
free(A);  
C = malloc(15);  
free(B);  
D = malloc(40);
```



# Heap (dinâmico)

- Alocação “aleatória”
  - impossível determinar padrão estaticamente

```
A = malloc(10);  
B = malloc(20);  
free(A);  
C = malloc(15);  
free(B);  
D = malloc(40);
```





# Exercícios

```
int fat (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fat(n-1);
    }
}

int vec[100];

void main (void) {
    for (int i=0; i<100; i++) {
        vec[i] = fat(i);
    }
}
```

# Exercícios

1. Qual é o máximo de memória em bytes que o programa abaixo usa em um determinado momento? Explique como você fez o cálculo.

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[100];  
  
void main (void) {  
    for (int i=0; i<100; i++) {  
        vec[i] = fat(i);  
    }  
}
```

# Exercícios

1. Qual é o máximo de memória em bytes que o programa abaixo usa em um determinado momento? Explique como você fez o cálculo.
2. Implementar a função `fat` do exercício anterior sem usar variáveis locais. Dica: você vai continuar precisando de uma pilha.

```
int fat (int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fat(n-1);
    }
}

int vec[100];

void main (void) {
    for (int i=0; i<100; i++) {
        vec[i] = fat(i);
    }
}
```

# Exercícios

1. Qual é o máximo de memória em bytes que o programa abaixo usa em um determinado momento? Explique como você fez o cálculo.
2. Implementar a função `fat` do exercício anterior sem usar variáveis locais. Dica: você vai continuar precisando de uma pilha.
3. Quais são as regiões de memória do Java e como elas são utilizadas pela JVM?

```
int fat (int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fat(n-1);  
    }  
}  
  
int vec[100];  
  
void main (void) {  
    for (int i=0; i<100; i++) {  
        vec[i] = fat(i);  
    }  
}
```

# *Amarração de Nomes*

(continuação...)

Estrutura de Linguagens

<https://github.com/fsantanna-uerj/EDL/>

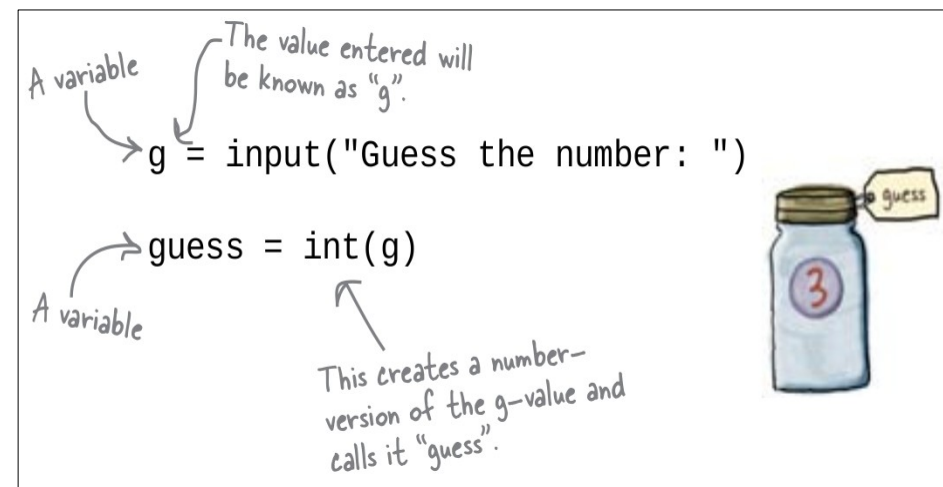
Francisco Sant'Anna

[francisco@ime.uerj.br](mailto:francisco@ime.uerj.br)



# Variáveis

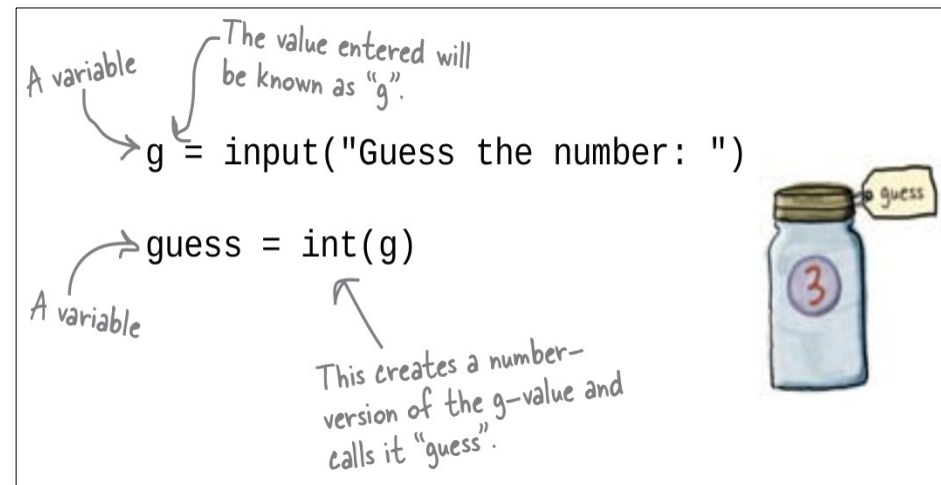
- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador



Créditos: "Head First Programming"

# Variáveis

- Uma “etiqueta” (ou nome) que representa uma região de memória
- Uma abstração da memória do computador
  - endereço
  - valor
  - tipo
  - **escopo**
  - **tempo de vida**



Créditos: "Head First Programming"

# Binding de Variável



# Binding de Variável

- Escopo
  - trecho de visibilidade da variável no código

# Binding de Variável

- Escopo
  - trecho de visibilidade da variável no código

```
{  
    int a;  
    ...  
}
```

# Binding de Variável

- Escopo
  - trecho de visibilidade da variável no código
- Tempo de vida
  - período entre alocação e desalocação da variável

```
{  
    int a;  
    ...  
}
```

# Binding de Variável

- Escopo
  - trecho de visibilidade da variável no código
- Tempo de vida
  - período entre alocação e desalocação da variável
- Escopo = Tempo de vida ?

```
{  
    int a;  
    ...  
}
```

# Escopo

# Escopo

- Estático / Léxico

# Escopo

- Estático / Léxico
  - determinado em tempo de compilação
- Dinâmico

# Escopo

- Estático / Léxico
  - determinado em tempo de compilação
- Dinâmico

```
namespace A {  
    ...  
    class B {  
        void C () {  
            ... X  
        }  
    }  
}  
  
int main (void) {  
    ... X  
}
```



# Escopo

- Estático / Léxico
  - determinado em tempo de compilação
- Dinâmico

```
namespace A {  
    int x;  
    class B {  
        void C () {  
            ... X  
        }  
    }  
}  
  
int main (void) {  
    ... X  
}
```

# Escopo

- Estático / Léxico
  - determinado em tempo de compilação
- Dinâmico

```
namespace A {  
    int x;  
    class B {  
        void C () {  
            ... X  
        }  
    }  
}  
  
int main (void) {  
    ... X  
}
```

# Escopo

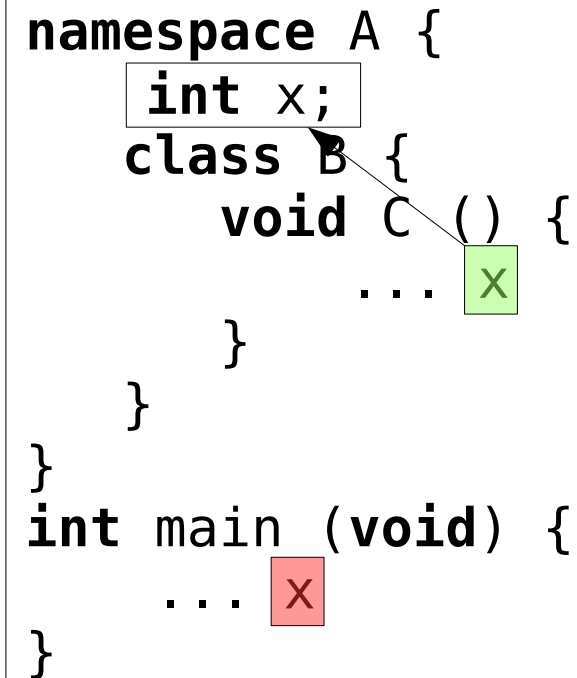
- Estático / Léxico
  - determinado em tempo de compilação
- Dinâmico

```
namespace A {  
    int x;  
    class B {  
        void C () {  
            ... X  
        }  
    }  
}  
  
int main (void) {  
    ... X  
}
```

# Escopo

- Estático / Léxico
  - determinado em tempo de compilação
- Dinâmico

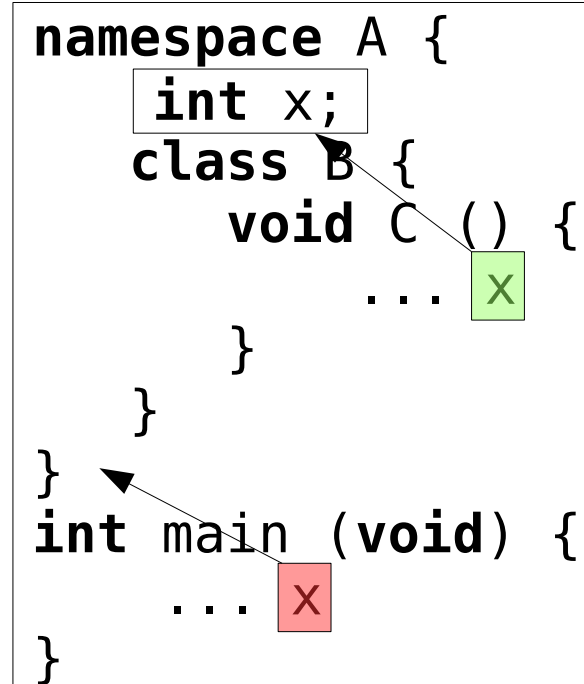
```
namespace A {  
    int x;  
    class B {  
        void C () {  
            ... X  
        }  
    }  
}  
  
int main (void) {  
    ... X  
}
```



# Escopo

- Estático / Léxico
  - determinado em tempo de compilação
- Dinâmico

```
namespace A {  
    int x;  
    class B {  
        void C () {  
            ... X  
        }  
    }  
}  
int main (void) {  
    ... X  
}
```



# Escopo

- Estático / Léxico
  - determinado em tempo de compilação
- Dinâmico
  - depende da execução do programa

```
namespace A {  
    int x;  
    class B {  
        void C () {  
            ... X  
        }  
    }  
}  
int main (void) {  
    ... X  
}
```

# Locais e “não locais”

# Locais e “não locais”

```
{  
    int x = ...;  
    _printf(“%d\n”, x + y);  
}
```



# Locais e “não locais”

- Locais

```
{  
    int x = ...;  
    _printf(“%d\n”, x + y);  
}
```

# Locais e “não locais”

- Locais

```
{  
    int x = ...;  
    _printf(“%d\n”, x + y);  
}
```

# Locais e “não locais”

- Locais

```
{  
    int x = ...;  
    _printf("%d\n", x + y);  
}
```

# Locais e “não locais”

- Locais

```
{  
    int x = ...;  
    _printf("%d\n", x + y);  
}
```

# Locais e “não locais”

- Locais
- Não locais

```
{  
    int x = ...;  
    _printf("%d\n", x + y);  
}
```

# Locais e “não locais”

- Locais
- Não locais
  - Globais

```
{  
    int x = ...;  
    _printf("%d\n", x + y);  
}
```

# Locais e “não locais”

- Locais
- Não locais
  - Globais
  - Pacote, Namespace

```
{  
    int x = ...;  
    _printf("%d\n", x + y);  
}
```

# Locais e “não locais”

- Locais
- Não locais
  - Globais
  - Pacote, Namespace
  - OO

```
{  
    int x = ...;  
    _printf("%d\n", x + y);  
}
```



# Locais e “não locais”

- Locais
- Não locais
  - Globais
  - Pacote, Namespace
  - OO
    - Variáveis de classe (Class.y)

```
{  
    int x = ...;  
    _printf("%d\n", x + y);  
}
```

# Locais e “não locais”

- Locais
- Não locais
  - Globais
  - Pacote, Namespace
  - OO
    - Variáveis de classe (Class.y)
    - Variáveis de instância (this.y)

```
{  
    int x = ...;  
    _printf("%d\n", x + y);  
}
```

# Locais e “não locais”

- Locais
- Não locais
  - Globais
  - Pacote, Namespace
  - OO
    - Variáveis de classe (Class.y)
    - Variáveis de instância (this.y)
  - Upvalues (capturadas por closures)

```
{  
    int x = ...;  
    _printf("%d\n", x + y);  
}
```

```

#include <stdio.h>

namespace Geometria {
    int OFF = 0;           // deslocamento padrao

    class Retangulo {
    public:
        static int WIDTH;  // comprimento padrao
        int x, width;
        Retangulo (int x) {
            this->x = x;
            this->width = WIDTH;
        }
        int getX () {
            return OFF + x;
        }
    };
    int Retangulo::WIDTH = 100;
}

int main (void) {
    using namespace Geometria;

    Retangulo r1(10);
    printf(">>> R1: x=%d w=%d\n", r1.getX(), r1.width);

    OFF = 50;
    r1.WIDTH = 10;

    Retangulo r2(20);
    printf(">>> R2: x=%d w=%d\n", r2.getX(), r2.width);

    return 0;
}

```

# Escopo Estático

# Escopo Estático

- Blocos
- Rotinas
- Classes
- Namespaces
- Arquivos
- Globais

# Escopo Estático

- Blocos
- Rotinas
- Classes
- Namespaces
- Arquivos
- Globais



# Escopo Estático

- Blocos
- Rotinas
- Classes
- Namespaces
- Arquivos
- Globais



```
int x;  
void f (int x) {  
    void g (int x) {  
        int x;  
        _printf("%d\n", x);  
    }  
}
```

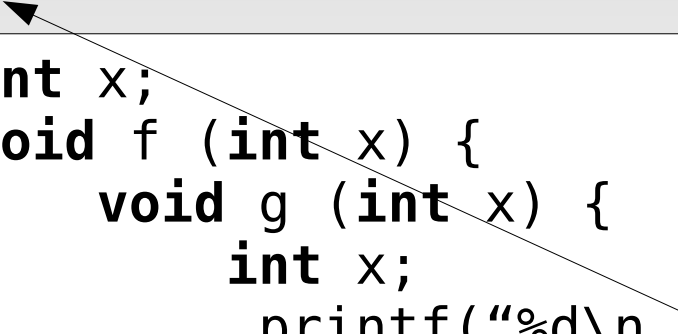


# Escopo Estático

- Blocos
- Rotinas
- Classes
- Namespaces
- Arquivos
- Globais



```
int x;  
void f (int x) {  
    void g (int x) {  
        int x;  
        _printf("%d\n", x);  
    }  
}
```



# Escopo Estático

- Blocos
- Rotinas
- Classes
- Namespaces
- Arquivos
- Globais



```
int x;  
void f (int x) {  
    void g (int x) {  
        int x;  
        _printf("%d\n", x);  
    }  
}
```

*Hiding, Shadowing*

# Hiding, Shadowing

# Hiding, Shadowing

Why do programming languages allow shadowing/hiding of variables and functions?



27

Many of the most popular programming languages (such as C++, Java, Python etc.) have the concept of **hiding** / **shadowing** of variables or functions. When I've encountered hiding or shadowing they have been the cause of hard to find bugs and I've never seen a case where I found it necessary to use these features of the languages.



To me it would seem better to disallow hiding and shadowing.



2

Does anybody know of a good use of these concepts?

**Update:**

I'm not referring to encapsulation of class members (private/protected members).

programming-languages

language-design

share improve this question

edited Oct 22 '13 at 8:05

asked Oct 21 '13 at 12:04



Simon

259 ● 3 ● 6

# Hiding, Shadowing

Why do programming languages allow shadowing/hiding of variables and functions?



27

Many of the most popular programming languages (such as C++, Java, Python etc.) have the concept of **hiding** / **shadowing** of variables or functions. When I've encountered hiding or shadowing they have been the cause of hard to find bugs and I've never seen a case where I found it necessary to use these features of the languages.



To me it would seem better to disallow hiding and shadowing.



2

Does anybody know of a good use of these concepts?

**Update:**

I'm not referring to encapsulation of class members (private/protected members).

programming-languages

share improve this question

**Lambda the Ultimate**  
*The Programming Languages Weblog*

XML

Home

Feedback

Home » forums » LtU Forum

**Disallow shadowing?**

I came across an early post on the topic of **disallowing shadowing** (message quoted below):

# Hiding, Shadowing

```
x = 1

def f():
    x = 2
    print("dentro ", x)

f()

print("fora ", x)
```

# Hiding, Shadowing

```
x = 1

def f():
    x = 2
    print("dentro ", x)

f()

print("fora ", x)
```

- Declaração vs Atribuição
  - binding of scope
  - binding of value

# Hiding, Shadowing

```
x = 1

def f():
    x = 2
    print("dentro ", x)

f()

print("fora ", x)
```

- Declaração vs Atribuição
  - binding of scope
  - binding of value

So, imperative language designers of the future, heed my warning: SHARPLY DISTINGUISH BINDING FROM ASSIGNMENT OR BE FOREVER DAMNED.

By [Matt Hellige](#) at Tue, 2014-02-11 16:55 | [login](#) or [register](#) to post comments



# Escopo Dinâmico

# Escopo Dinâmico

- Se aplica a variáveis não locais

# Escopo Dinâmico

- Se aplica a variáveis não locais

```
{  
    int a;  
    ...  
}
```

# Escopo Dinâmico

- Se aplica a variáveis não locais

```
{  
    int a;  
    ...  
}
```

```
local $x = 'global';  
  
sub print_x {  
    print "x = $x\n";  
}  
  
sub f {  
    local $x = "f";  
    print_x();  
}  
  
print_x();  
f()
```

# Escopo Dinâmico

- Se aplica a variáveis não locais
- Busca declaração ativa mais recente
  - na pilha, em tempo de execução

```
{  
    int a;  
    ...  
}
```

```
local $x = 'global';  
  
sub print_x {  
    print "x = $x\n";  
}  
  
sub f {  
    local $x = "f";  
    print_x();  
}  
  
print_x();  
f()
```

# Escopo Dinâmico

- Se aplica a variáveis não locais
- Busca declaração ativa mais recente
  - na pilha, em tempo de execução
- Perl

```
{  
    int a;  
    ...  
}
```

```
local $x = 'global';  
  
sub print_x {  
    print "x = $x\n";  
}  
  
sub f {  
    local $x = "f";  
    print_x();  
}  
  
print_x();  
f()
```

# Escopo Dinâmico

# Escopo Dinâmico

Quora

What are the advantages of dynamic scoping?



**Yoann Padioleau**, I love to learn new programming languages.

Written Oct 15, 2010

It makes the program more dynamic which in certain contexts is useful. For instance Emacs Lisp can have dynamically scoped variables which makes it easy to tweak function behaviors. It let the caller, or context, influence locally the behavior of what would be otherwise closed and untweakable functions.

321 Views



# Escopo Dinâmico

Quora

What are the advantages of dynamic scoping?



**Yoann Padioleau**, I love to learn new programming languages.

Written Oct 15, 2010

It makes the program more dynamic which in certain contexts is useful. For instance Emacs Lisp can have dynamically scoped variables which makes it easy to tweak function behaviors. It let the caller, or context, influence locally the behavior of what would be otherwise closed and untweakable functions.

321 Views

All sensible languages use lexical scoping. Dynamic scoping is of mostly historical interest --- early implementations of Lisp used dynamic scoping, which Joseph McCarthy (the inventor of Lisp) simply considered a bug. In languages that use dynamic scoping, functions are difficult to use and do not serve as well-behaved abstractions --- it is possible to make a function misbehave in ways that the writer of the function never anticipated, simply by accidentally redefining some name that may be used in the function.

# Escopo Dinâmico

Quora

What are the advantages of dynamic scoping?

McCarthy about that *bug*

In [From LISP 1 to LISP 1.5](#) (1979) McCarthy writes (bold by me):

d. **Free variables.** In all innocence, James R. Slagle programmed the following LISP function definition and complained when it didn't work right:

The object of the function is to find a subexpression of  $x$  satisfying  $p[x]$  and return  $f[x]$ . If the search is unsuccessful, then the continuation function  $u[]$  of no arguments is to be computed and its value returned. The difficulty was that when an inner recursion occurred, the value of  $car[x]$  wanted was the outer value, but the inner value was actually used. In modern terminology, lexical scoping was wanted, and dynamic scoping was obtained.

I must confess that I regarded this difficulty as just a **bug** and expressed confidence that Steve Russell would soon fix it. He did fix it but by inventing the so-called FUNARG device that took the lexical environment along with the functional argument. Similar difficulties later showed up in Algol 60, and Russell's turned out to be one of the more comprehensive solutions to the problem. While it worked well in the interpreter, comprehensiveness and speed seem to be opposed in compiled code, and this led to a succession of compromises. Unfortunately, time did not permit writing an appendix giving the history of the problem, and the interested reader is referred to (Moses 1970) as a place to start. (David Park tells me that Patrick Fischer also had a hand in developing the FUNARG device).

# Escopo Dinâmico

Quora

What are the advantages of dynamic scoping?

McCarthy about that *bug*

In F



30



Like everything else, Dynamic Scoping is merely a tool. Used well it can make certain tasks easier. Used poorly it can introduce bugs and headaches.

I can certainly see some uses for it. One can eliminate the need to pass variables to some functions.

For instance, I might set the display up at the beginning of the program, and every graphic operation just assumes this display.

If I want to set up a window inside that display, then I can 'add' that window to the variable stack that otherwise specifies the display, and any graphic operations performed while in this state will go to the window rather than the display as a whole.

It's a contrived example that can be done equally well by passing parameters to functions, but when you look at some of the code this sort of task generates you realize that global variables are really a much easier way to go, and **dynamic scoping gives you a lot of the sanity of global variables with the flexibility of function parameters.**

[share](#) [improve this answer](#)

edited Nov 16 '09 at 17:18

answered Nov 26 '08 at 15:20



[Adam Davis](#)

58.6k ● 41 ● 208 ● 302

# Exercícios

1. Sobre esconder a declaração de variáveis (hiding)...
  1. Procure uma linguagem que não permita. O que acontece?
  2. Quais são as vantagens e desvantagens de permitir ou não permitir?
2. Quais outras linguagens possuem escopo dinâmico?
  1. Como funciona?
  2. Dê um exemplo interessante do seu uso.
3. Sobre o tratamento de exceções em Java ou Python...
  1. A partir de um erro (*throw/raise*), como determinar o ponto de captura da exceção (*catch/except*).
  2. Tratamento de exceções é mais parecido com escopo estático ou dinâmico. Justifique.