



UNIVERSIDADE FEDERAL  
DO ESPÍRITO SANTO

Centro Tecnológico  
Departamento de Informática

Prof. Veruska Zamborlini

[veruska.zamborlini@inf.ufes.br](mailto:veruska.zamborlini@inf.ufes.br)

<http://www.inf.ufes.br/~veruska.zamborlini>

# Aula 3/4

## Nomes, vinculações e escopos 2021/2



Esta obra está licenciada com uma licença Creative Commons Atribuição-  
Compartilha Igual 4.0 Internacional: <http://creativecommons.org/licenses/by-sa/4.0/>.

Material adaptado  
Prof.s Vitor Souza e Eduardo Zambon

# Introdução

- LPs => manipulação de dados
  - Manipulação:
    - Procedimentos ou funções, nomeados ou não
  - Dados:
    - **Valores** de um certo **tipo**, armazenados na **memória** do computador, **nomeados** ou não.
- LPs imperativas (how / como)
  - Variáveis = abstrações para manipulação de células de memória;
- LPs funcionais (what / o que)
  - Parâmetros = abstrações para manipulação de valores

# Vinculações/ligações/amarrações

- Associação entre **entidades de programação**:
  - *Uma **variável** e seu **nome** e/ou **valor**;*
  - *Um **identificador** e seu **tipo**;*
  - *Um **subprograma** e seu **código em memória**; etc.*

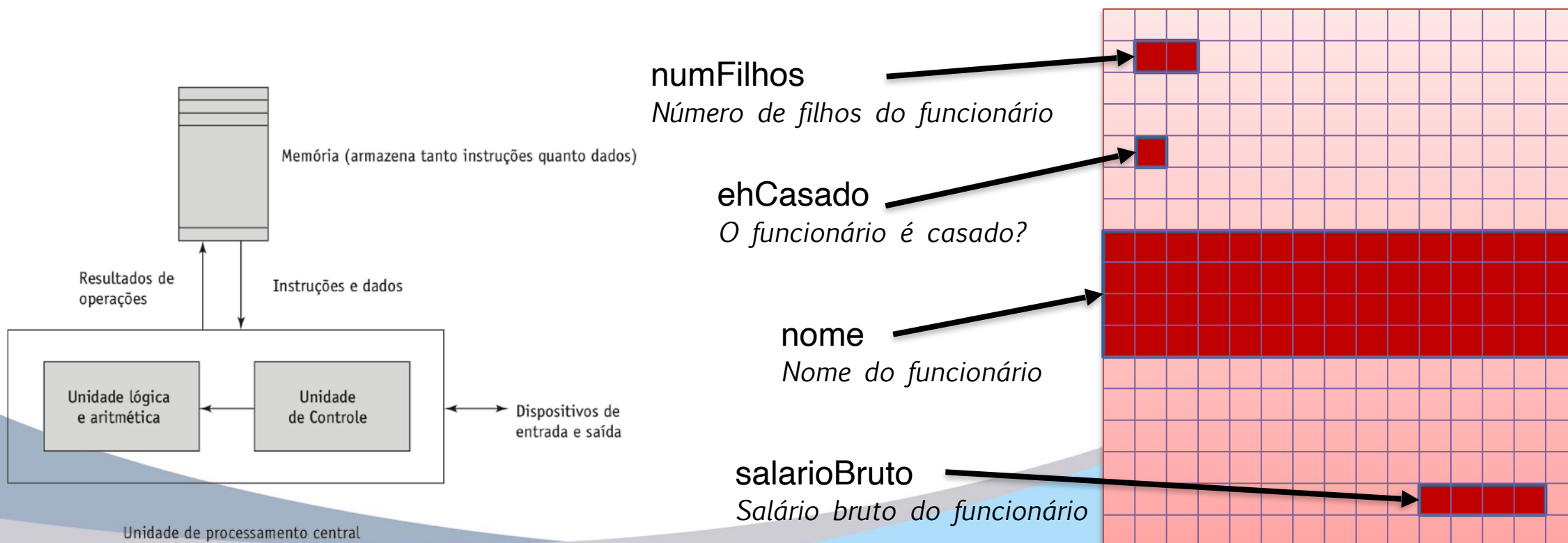
Sebesta:  
“vinculação”

Tucker:  
“ligação”

Varejão:  
“amarração”

# Variáveis

- Abstração para uma ou mais células de memória responsáveis por armazenar o estado de uma entidade de computação;



# Variáveis

- Principal evolução das linguagens de montagem (*assembly*) em relação às linguagens de máquina: endereçamento local (o tradutor converte);
- Importância do conceito:

“Uma vez que o programador tenha entendido o uso de variáveis, ele entendeu a essência da programação”. – Dijkstra

Em linguagens imperativas...



# Vinculações - exemplos

a variável ocupa tamanho(int) bits a partir do endereço de memória 0x... contendo o valor binário correspondente ao valor 10 tipo int.

#1: a variável se chama var.

#4: a variável ocupa o endereço de memória 0x...

// Código em C:  
`int var = 100;`

#2: a variável é do tipo int.

#3: a variável possui valor 100.



Qual é o tempo de cada vinculação?

# Tempos de vinculação

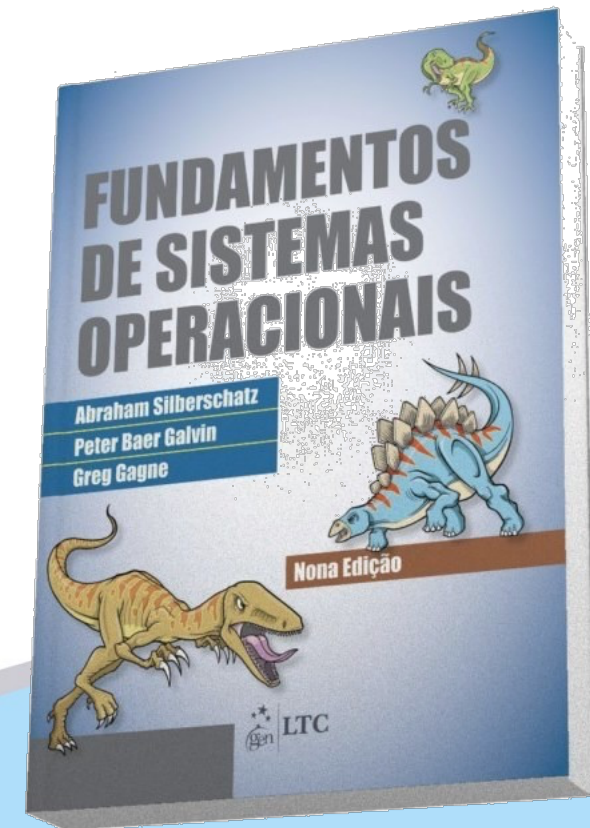
Tempo de vinculação	Identificador	Entidade
Projeto da LP	*	Operação de multiplicação (C, C++, Java).
Projeto da LP	int	Intervalo do tipo inteiro (Java).
Implementação do compilador	int	Intervalo do tipo inteiro (C).
Compilação	Variável	Tipo da variável (C).
Execução	Variável	Tipo do objeto em polimorfismo (C++, Java).
Ligação	Função	Código correspondente à função.
Carga do programa	Variável global	Posição de memória ocupada.
Execução	Variável local	Posição de memória ocupada.

A vinculação pode ser estática (feita antes da execução e não muda) ou dinâmica (muda durante execução).



# Vinculação física

- Paginação, segmentação, swap, etc.
- Complexo de se analisar;
- Vamos abstrair...
  - *“célula abstrata” -  
endereço “relativo”  
ao invés de “físico/absoluto”*





# Características das variáveis

- *Nome*
- *Endereço*
- *Valor*
- *Tipo*
- *Escopo*
- *Tempo de vida*

# Nomes

- **Variáveis** (geralmente) possuem **nomes**;
  - *+ legibilidade, + redigibilidade, + modificabilidade*
  
- Podem ser **anônimas**
  
- Podem ter **apelidos** (aliases):
  - Dois nomes para a mesma “entidade” (endereço);
  - Ex.: ponteiros (C, C++), referências (C++, Java).

# Exemplos de escolhas

## ■ Tamanho:

- *Fortran I – Fortran 77: máximo de 6 caracteres;*
- *Fortran 95: máximo de 31 caracteres;*
- *C99: não limita, mas só considera os 63 primeiros;*
- *Java e C#: sem limites;*

## ■ Caracteres aceitos:

- *Fortran < 90: apenas letras maiúsculas, mas podiam ter espaços em branco (ignorados);*
- *Scala: ?+-<>:|!&%#\^@~\*\_ \_ é válido;*

# Exemplos de escolhas

- Sensibilidade à capitalização:
  - *Sim: C, C++, C#, Dart, Go, Groovy, Java, Java/TypeScript, Julia, Kotlin, Lua, Perl, Python, Ruby, Scala, Swift;*
  - *Não: Fortran, Pascal, PHP, (Visual) Basic;*
  - *Polêmica sobre legibilidade.*
- Outras regras:
  - *PHP: variáveis começam com \$;*
  - *Perl: \$scalars, @arrays, %hashes;*
  - *Ruby: @variavelInstancia, @@variavelClasse;*

# Palavras especiais

- Conceitos ortogonais:
  - *Palavras-chave: possui significado especial em alguns contextos;*
  - *Palavras reservadas: não pode ser usada como nome.*
- Palavras pré-definidas: palavras-chave cujo significado pode ser modificado.

! Código válido em FORTRAN.

! São palavras-chave, mas não reservadas.

INTEGER REAL

REAL INTEGER

goto é reservado em Java.  
true/false podem ser  
redefinidos em Pascal.

# Nomes

## ■ Projeto de LP's

- ⦿ Os nomes são sensíveis à capitalização (*case sensitive*)?
- ⦿ Quais os caracteres aceitos em um nome?
- ⦿ Qual o tamanho máximo de um nome (e o que acontece se passar)?
- ⦿ As palavras especiais são reservadas ou palavras-chave?

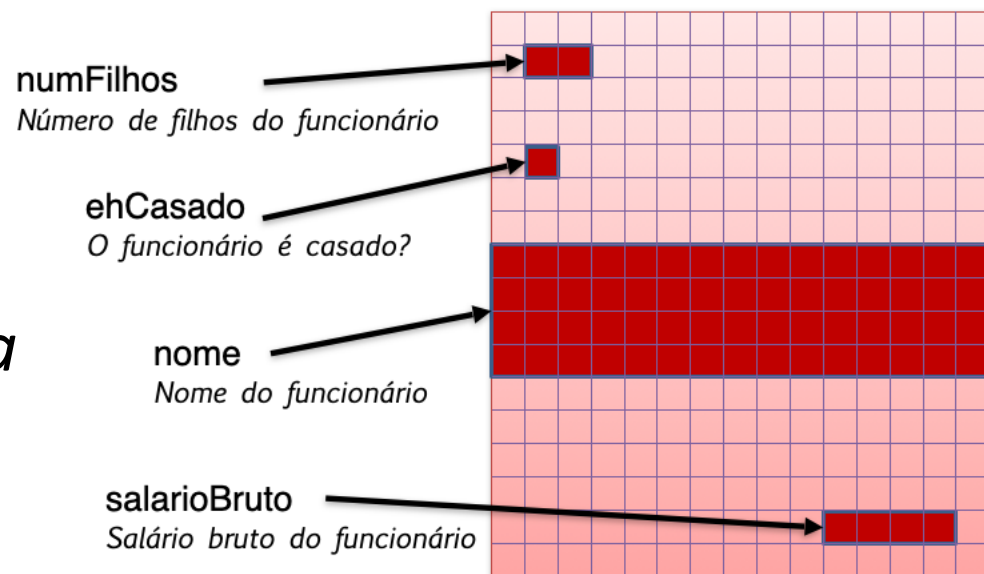
# Características das variáveis

- *Nome*
- *Endereço*
- *Valor*
- *Tipo*
- *Escopo*
- *Tempo de vida*



# Tipos

- *Determina os valores e operações possíveis;*
- *(a ser discutido no cap. 6)*
- *Vinculação de variável/endereço a seu tipo*



# Tipos - Vinculações (tipagem)

## ■ Estática:

- *Especificação explícita*  
– **Obj** o = new Obj();
- *Especificação implícita (sintática ou inferência)*  
– **var** o = new Obj();
- *Fortran: se começa com I, J, ..., N é Integer, se não, é Real;*
- *C, C++, C#, Java, Fortran, Kotlin, Pascal, Scala, Swift, etc.*

## ■ Dinâmica: **Interpretadores!**

- *Flexibilidade, questões de confiabilidade e custo;*
- *Groovy, JavaScript, Julia, Lua, Perl, PHP, Python, Ruby, etc.*

**Deteção de erros no tempo de vinculação!**

# Características das variáveis

- *Nome*
- *Endereço*
- *Valor*
- *Tipo*
- *Escopo*
- *Tempo de vida*

# Endereço e Valor

- Endereço (*l-value*):
  - célula(s) (ou “célula abstrata”) de memória
- Valor (*r-value*):
  - Conteúdo da(s) célula(s) de memória,
  - Interpretado pelo seu tipo.

// Exemplo em C:

unsigned x;

x = 7;

x = x + 3;

Endereço da  
variável (*l-value*)

Valor da variável  
(*r-value*)

x	FF00	00000000
	FF01	00000000
	FF02	00000000
	FF03	00000000

x	FF00	00000000
	FF01	00000111
	FF02	00000000
	FF03	00000000

x	FF00	00000000
	FF01	00001010
	FF02	00000000
	FF03	00000000

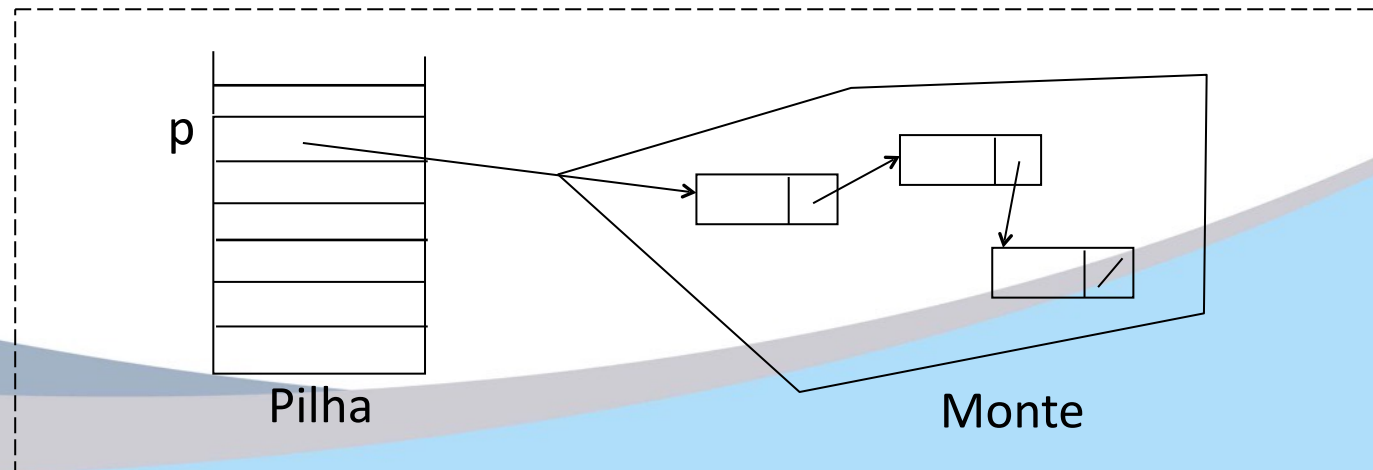
# Endereço - Vinculação

- Análise de 4 categorias de variáveis:
  - *Estáticas;*
  - *Dinâmicas na pilha;*
  - *Dinâmicas no monte explícitas;*
  - *Dinâmicas no monte implícitas.*

**Diferente de tipagem estática x dinâmica!**

# Recapitulando: pilha e monte

- Pilha: alocação organizada, mas incompatível com alocação dinâmica (alocação em tempo de carga ou alocação dinâmica contígua);
- Monte: alocação desorganizada porém dinâmica. Custo de manutenção dos espaços livres e ocupados;
- Linguagens *Algol-like*: pilha + monte (ponteiros).



# Variáveis estáticas

- Vinculadas a endereços antes do início da execução;
- Diz-se que são armazenadas na base (**nem pilha, nem monte**, junto com o código do programa);
- C/C++: palavra-chave static;
- Eficientes, globais, não permitem recursão.



# Memória

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
    int local1, local2;
    - - -
}

void func_B ()
{
    int localA, localB;
    localA = func_A();
    localB = func_A();
}

main ()
{
    a = "Essa aula é legal";
    b = "Será mesmo?";
    func_B();
}
```



\*\* Material da Prof.a Patricia

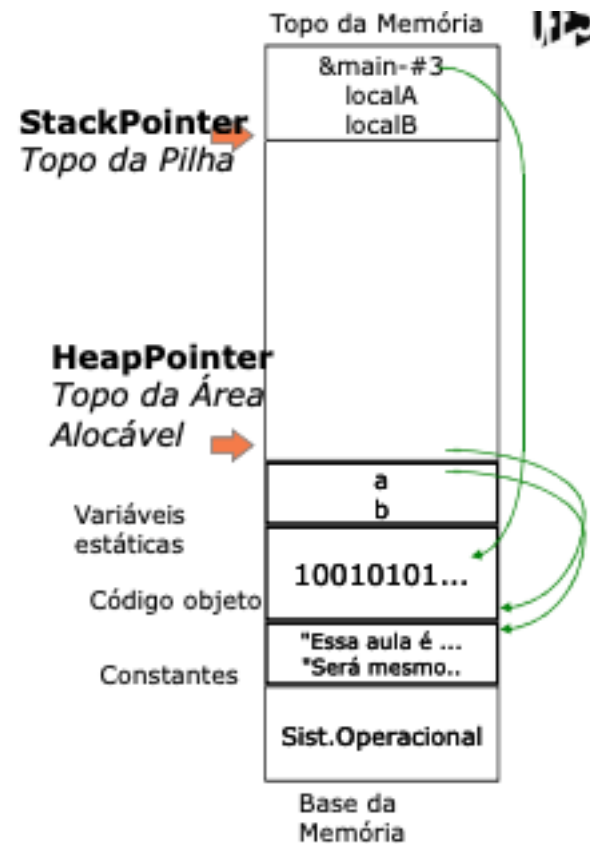
# Variáveis dinâmicas na pilha

- Tipos estaticamente vinculados;
- Endereços vinculados quando suas sentenças de declaração são elaboradas (execução);
- Ex.: variáveis locais de uma função/método;
- Relativamente eficientes, escopo restrito, permitem recursão.

# Memória

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
    int local1, local2;
    - - -
}
void func_B ()
{
    int localA, localB;
    localA = func_A();
    localB = func_A();
}
main ()
{
    a = "Essa aula é legal";
    b = "Será mesmo?";
    func_B();
}
```



\*\* Material da Prof.a Patricia

# Variáveis dinâmicas do monte explícitas

- Células de memória não nomeadas, alocadas e liberadas por instruções explícitas (ex.: malloc/free);
- Alocação no monte, uso de ponteiros;
- Tipos estaticamente vinculados, endereço dinâmico;
- Usadas para construir estruturas dinâmicas;
- Custo e complexidade (armazenamento, ponteiros).

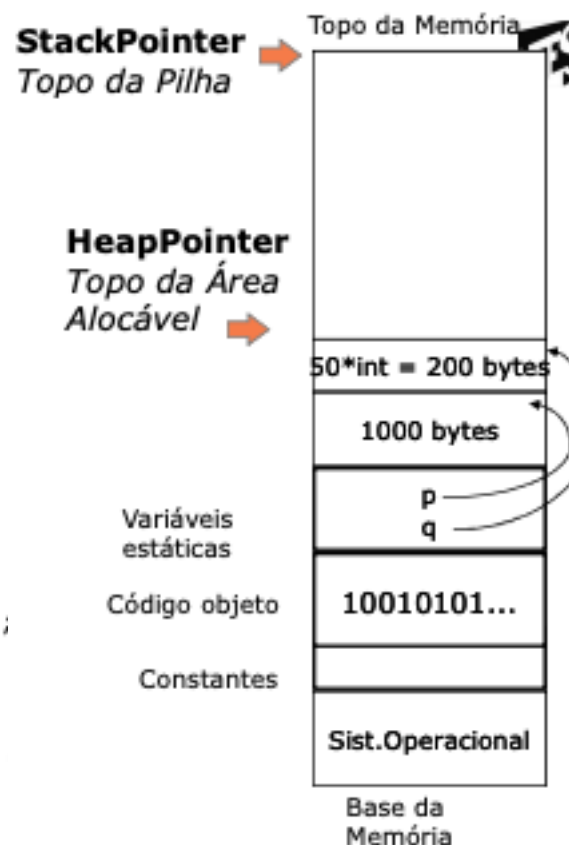
# Memória

```
#include <stdlib.h>
#include <stdio.h>

char *p;
int *q;

main ()
{
    p = (char *) malloc(1000);
        // Aloca 1000
        // bytes de RAM

    q = (int *) malloc(50*sizeof(int));
        // Aloca espaço
        // para 50 inteiros.
}
```



Alocação no monte não é desocupada “automaticamente” ao término de uma função

\*\* Material da Prof.a Patricia

# Variáveis dinâmicas do monte implícitas

- Similar à categoria anterior;
- Porém vinculadas a armazenamento apenas quando recebem valor;
- Tipos dinamicamente vinculados;
- Mais alto grau de flexibilidade, perda de confiabilidade e eficiência.

```
// Ex.: atribuição dinâmica de um vetor a  
// uma variável em JavaScript:  
highs = [74, 84, 86, 90, 71];
```

# Características das variáveis

- *Nome*
- *Endereço*
- *Valor*
- *Tipo*
- *Escopo*
- *Tempo de vida*



# Escopo

- ***Faixa de sentenças na qual uma variável/vinculação é visível (ou seja, pode ser referenciada);***

# Escopo - Vinculação

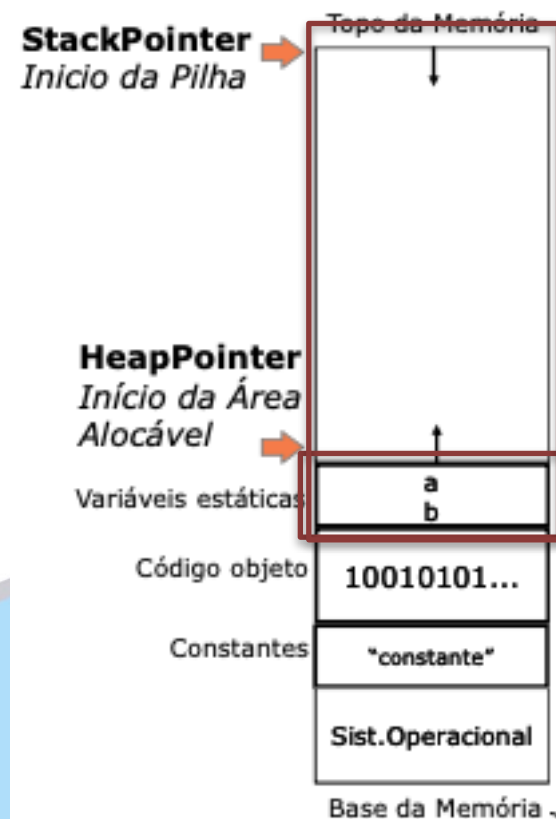
- **Escopo global:**
  - Variáveis globais são potencialmente visíveis em todo o código.
  - Pode requerer uma palavra reservada para tornar-se visível.
- **Escopo local:**
  - Variáveis locais são visíveis apenas na função ou bloco que as declaram.

```
#include <stdio.h>
char *a, *b;

int func_A ()
{
    int local1, local2;
    - - -
}

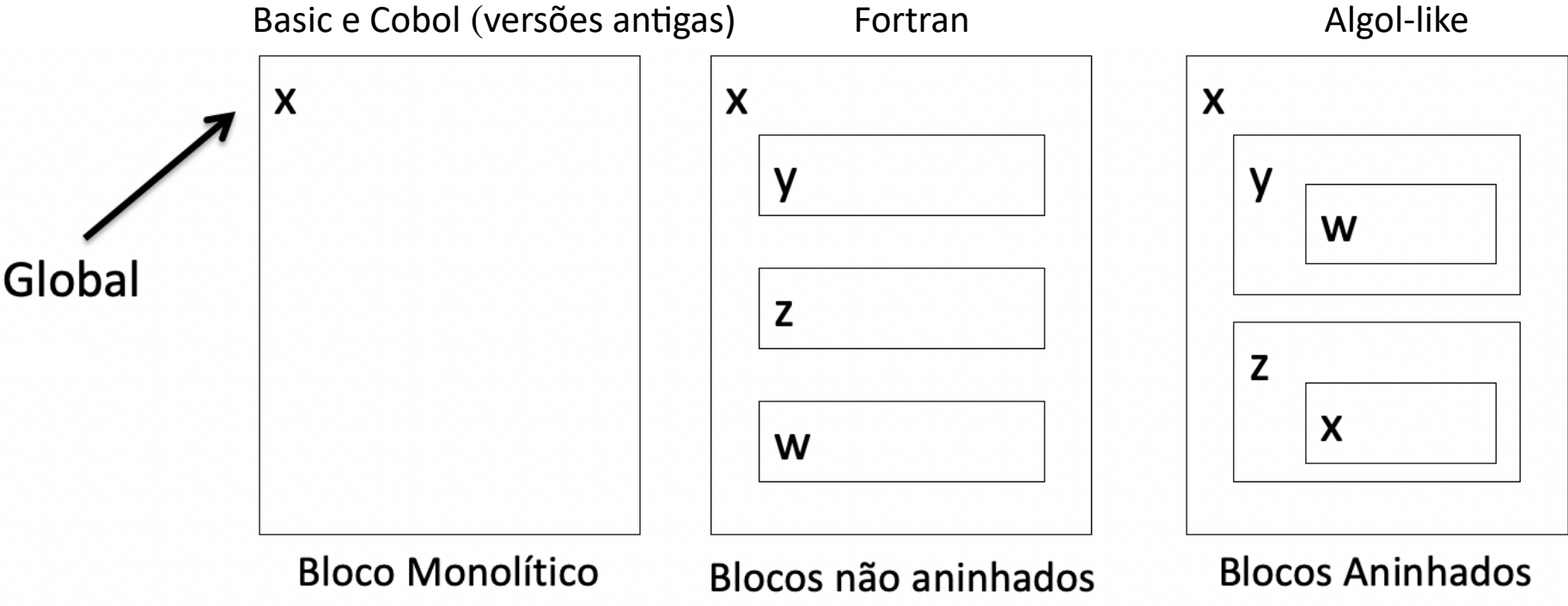
void func_B ()
{
    int localA, localB;
    localA = func_A();
    localB = func_A();
}

main ()
{
    a = "Essa aula é legal";
    b = "Será mesmo?"
    func_B();
}
```



# Escopo - Blocos e Funções

	Algol	C	Java	Ada
Função	Aninhado	Não-aninhado	Não-aninhado	Aninhado
Bloco	Aninhado	Aninhado	Aninhado	Aninhado



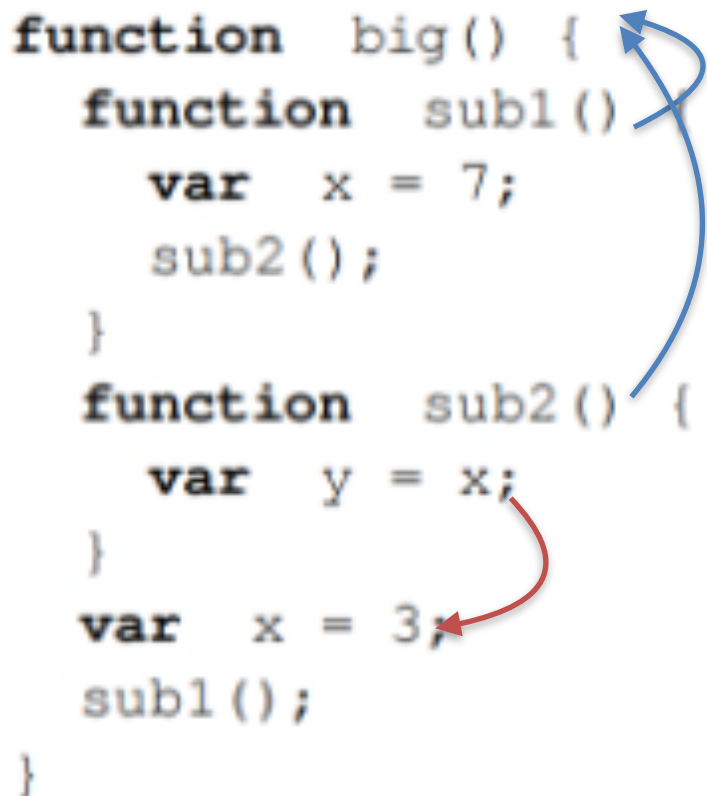
# Escopo - Vinculação

- Escopo **estático** (ou léxico):
  - *Tempo de compilação;*
  - *Relacionado à estrutura (texto) do programa*
  
- Escopo **dinâmico** (fluxo de execução):
  - *Tempo de execução;*
  - *Relacionado ao fluxo de execução do programa*

# Escopo estático x dinâmico

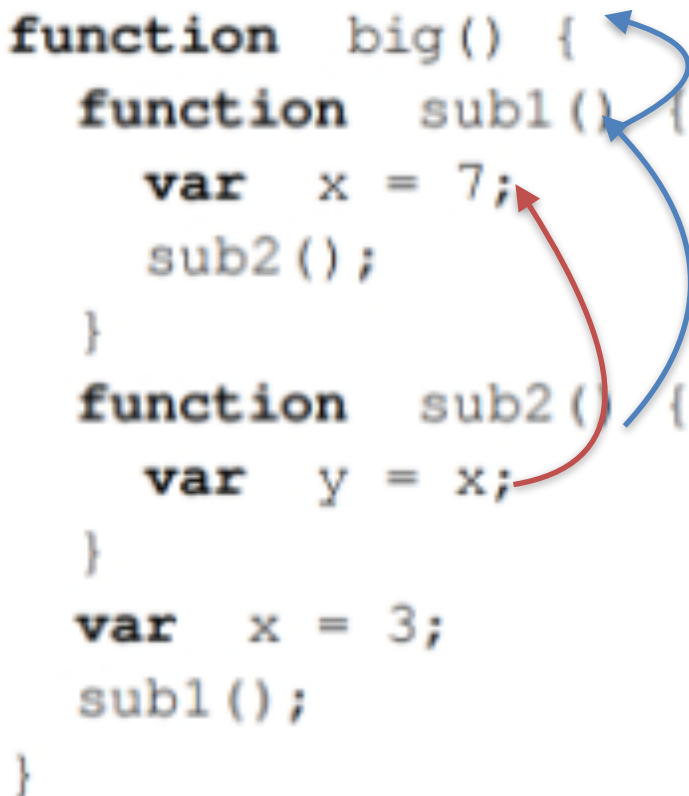
## Funções aninhadas

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```



- *Pai **estático**, que **declarou** o subprograma*

```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```



- *Pai **dinâmico**, que **chamou** o subprograma*

# Escopo estático

## Blocos Aninhados

```
void sub() { Bloco 1
    int count;
    . . .
    while (. . .) { Bloco 2
        int count;
        count++;
        . . .
    }
    . . .
}
```

O uso do mesmo nome em C faz com que o “*count*” externo não seja visível no Bloco 2

Válido em C e C++, mas inválido em Java e C#.

# Escopo estático - Ordem de declaração

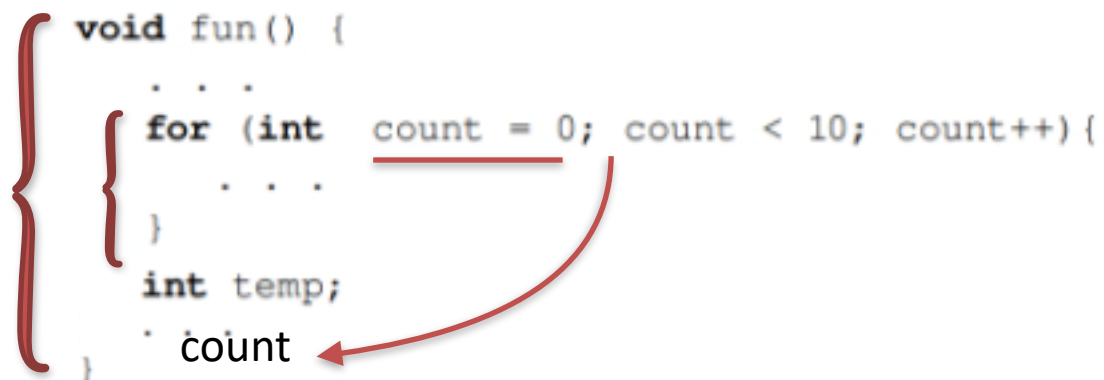
Dependendo da linguagens/versões de linguagem:

escopo de **count**:

fun() x for loop

```

{ void fun() {
  . . .
  { for (int count = 0; count < 10; count++) {
    . . .
  }
  int temp;
  count
}
    
```





# Escopo estático - Ordem de declaração

Dependendo da linguagens/versões de linguagem:

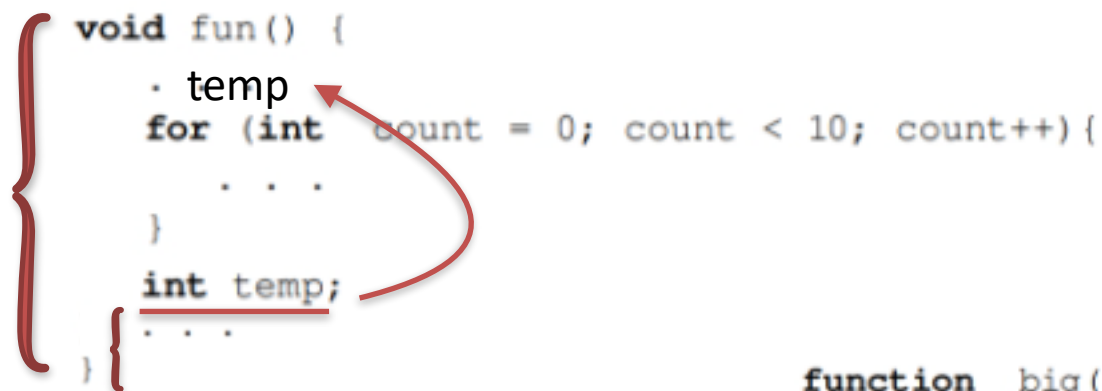
escopo de **count**:

fun() x for loop

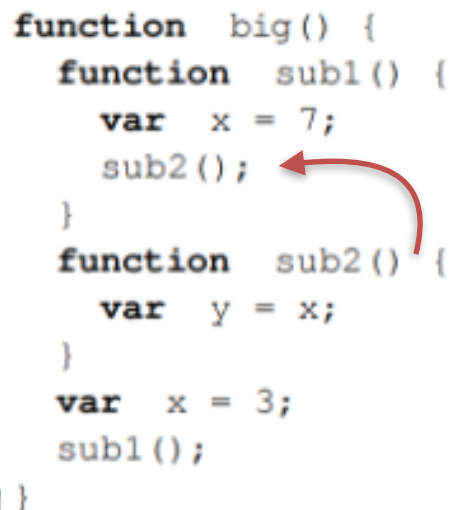
escopo de **temp**:

fun() x sentenças após "int temp"

```
void fun() {
    . temp
    for (int count = 0; count < 10; count++) {
        . . .
    }
    int temp;
    . . .
}
```



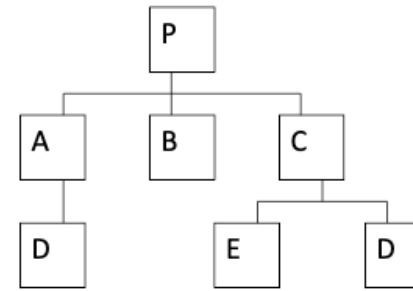
```
function big() {
    function sub1() {
        var x = 7;
        sub2();
    }
    function sub2() {
        var y = x;
    }
    var x = 3;
    sub1();
}
```



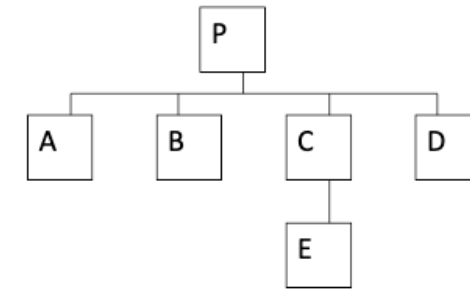
# Escopo estático - Avaliação

- Possíveis problemas:

- Manutenção



(a)



(b)

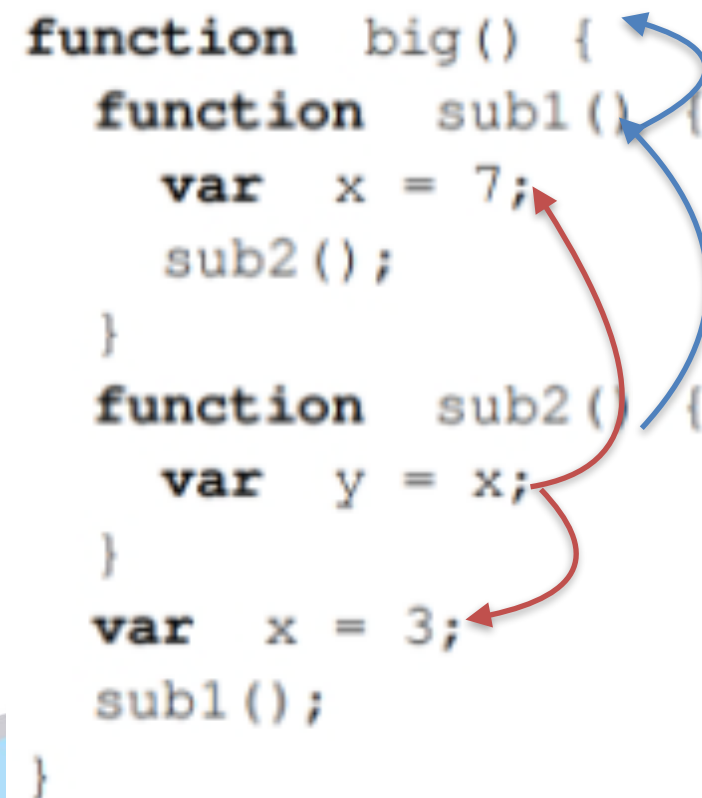
- D precisa ser usado dentro de A e C:
    - *Em (a) há repetição;*
    - *Em (b), D fica visível para P e B também.*

- Alternativa:

- Encapsulamento - capítulo 11

# Escopo dinâmico - Avaliação

- Problemas:
  - **Eficiência:** checagem de tipos durante execução, acesso deve seguir sequência de chamadas;
  - **Legibilidade:** deve-se seguir a sequência de chamadas para entender a amarração – propenso a erros do programador;
  - **Confiabilidade:** subprograma pode acessar variáveis locais do bloco que o chama;
- Pouquíssimo usado por LPs:
  - APL, Snobol4 e versões iniciais de Lisp e Perl;
  - Common Lisp e Perl suportam os dois tipos.
- Alternativa: passagem de parâmetros.



```
function big() {
  function sub1() {
    var x = 7;
    sub2();
  }
  function sub2() {
    var y = x;
  }
  var x = 3;
  sub1();
}
```

# Características das variáveis

- *Nome*
- *Endereço*
- *Valor*
- *Tipo*
- *Escopo*
- *Tempo de vida*

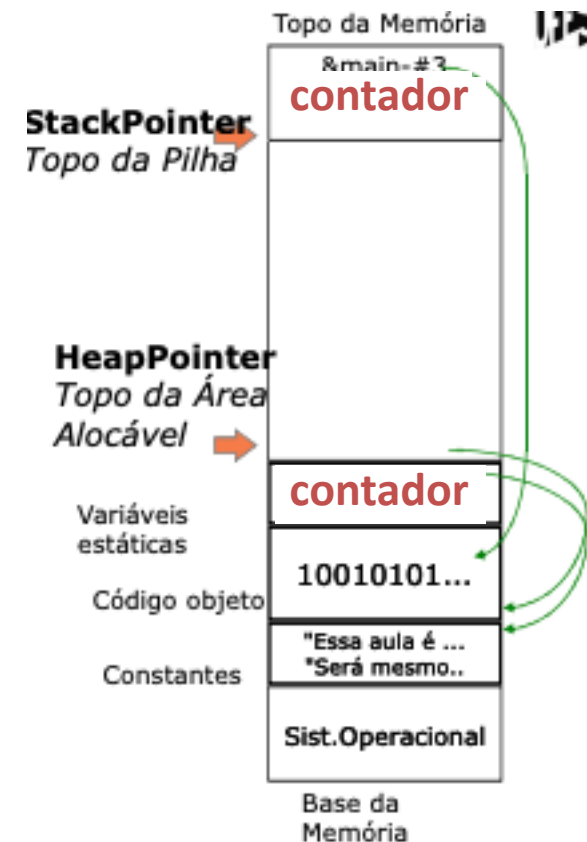
# Tempo de vida

- *Tempo durante o qual uma variável está vinculada a um endereço de memória – da **alocação** à **liberação**;*

# Tempo de vida

```
void conta() {
    int contador = 0; // Vinculação (endereço) dinâmica na pilha
                      // Tempo de vida = execução da função
}
```

```
void conta() {
    static int contador = 0; // Vinculação (endereço) estática
                             // Tempo de vida = execução do programa
}
```

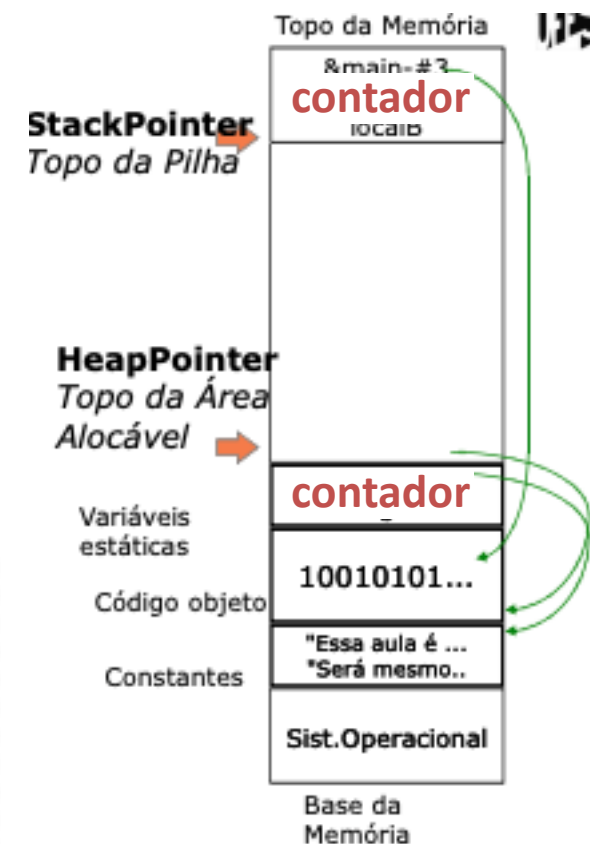


# Escopo (espacial) vs. tempo de vida (temporal)

- Geralmente estão relacionados (ex.: variável local);
- Em alguns casos, são diferentes:
  - *Ocultamento (já vimos);*
  - *Alocação estática de variáveis locais:*

```
void conta() {           // Escopo local
    int contador = 0;    // Vinculação (endereço) dinâmica na pilha
                        // Tempo de vida = execução da função
}
```

```
void conta() {           // Escopo local
    static int contador = 0; // Vinculação (endereço) estática
                        // Tempo de vida = execução do programa
}
```



# Ambientes de referenciamento

- Varejão: “ambiente de amarração”;
- Coleção de todas as variáveis visíveis numa sentença.

**Ambiente:** do ponto de vista de sentenças, todas as vinculações visíveis.

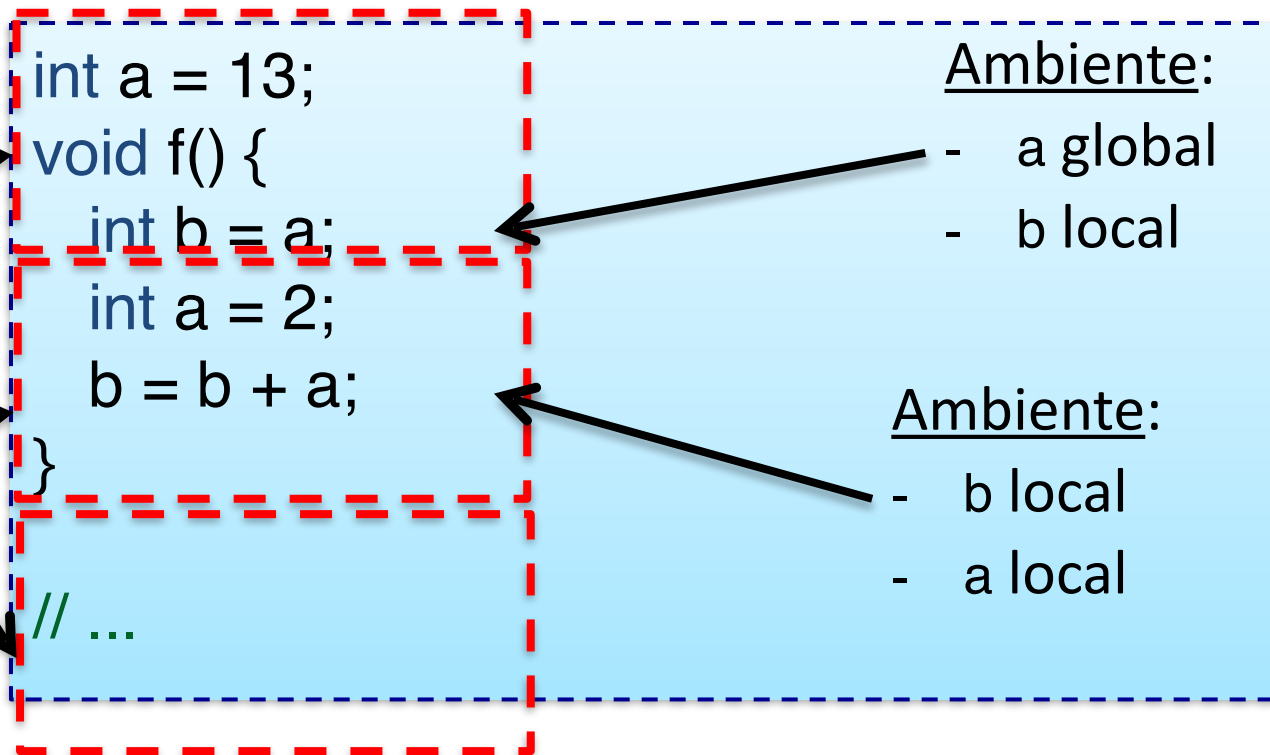
**Escopo:** do ponto de vista de vinculações, todas as sentenças a que são visíveis.



# Ambiente vs. escopo

Escopo de  
a global

Escopo de  
a local



# Definição vs. declaração

## ■ Definição:

- *Produz vinculações entre identificadores e entidades criadas na própria definição;*
- `typedef struct TCoor { double x, y; } *Coor;`
- `int sum; // global neste arquivo.`

## ■ Declaração:

- *Produz vinculações entre identificadores e entidades já criadas ou que ainda o serão (não causa alocação);*
- `typedef struct TCoor *Coor;`
- `extern int sum; // global em outro arquivo.`

# Constantes nomeadas

- Como uma variável, mas vinculada a um valor apenas uma vez
  - *ex.: static final double PI = 3.14159;*  
*final int len = 100;*  
*const int result = 2 \* width + 1;*
  - *+ legibilidade, + confiabilidade, + modificabilidade;*
- Podem ter:
  - *Vinculação estática, ex.: **const** em C#;*
  - *Vinculação dinâmica, ex.: **readonly** em C#;*
  - *Constantes com vinculação estática podem apenas receber valores estáticos em sua inicialização.*