Algoritmos e estruturas de Dados

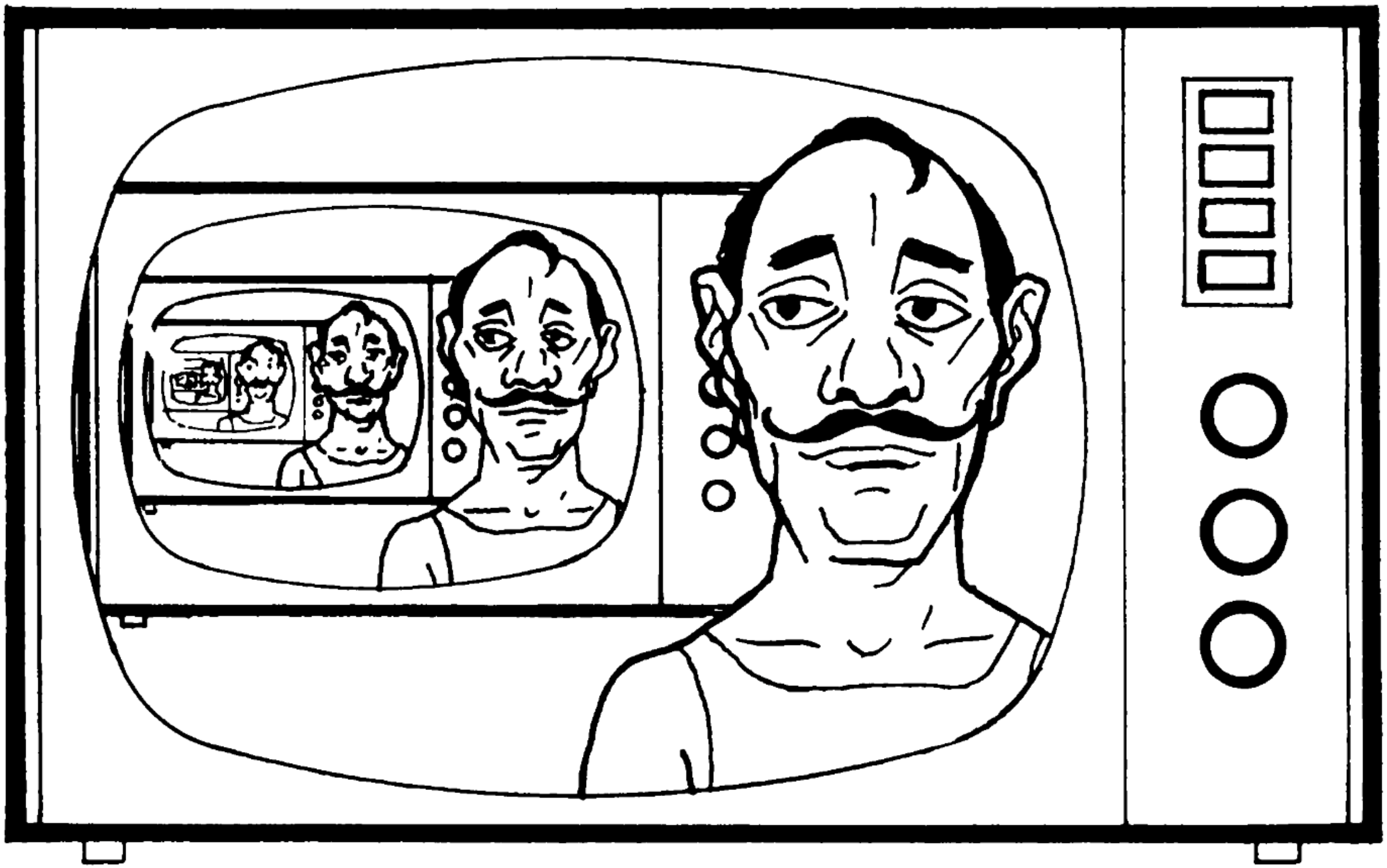
Recursividade

Prof. Maiquel de Brito

Recursividade

- *Para entender recursividade é preciso primeiro entender a recursividade*

Recursividade

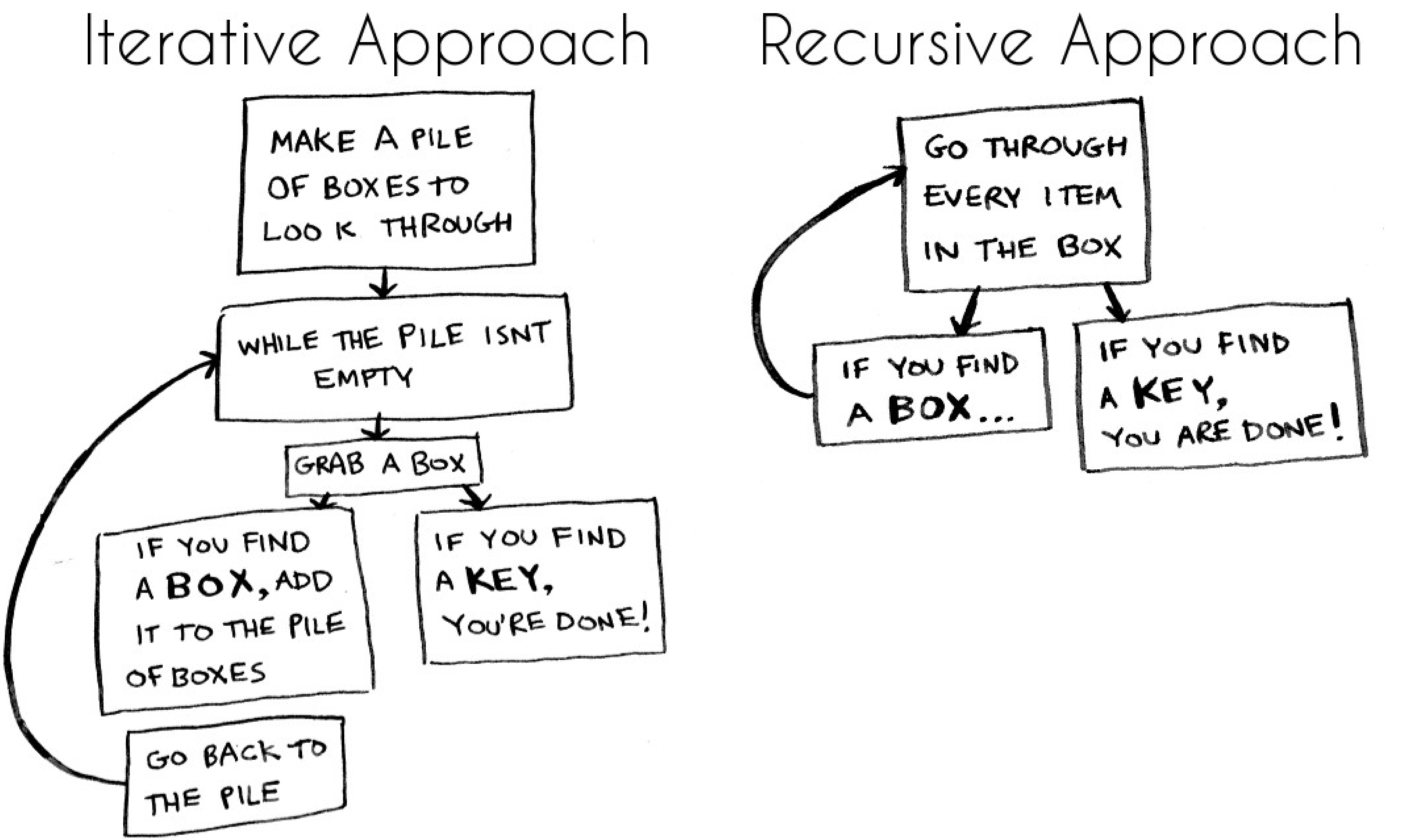


Fonte: WIRTH, Niklaus. Algorithms + Data Structures = Programs

Recursividade

Fonte: www.itcuties.com/java/recursion-and-iteration/

Recursividade



Recursividade

- **Definição**
  - É um princípio poderoso que permite que um problema seja definido em termos de instâncias menores e menores do próprio problema.

- **Computação**
  - Na computação resolvemos problemas recursivos usando funções recursivas que são funções que invocam (chamam) a si próprias.

Recursividade

Fatorial – Solução iterativa

n! = (n) * (n - 1) * (n - 2) ... (1)

Ex.: 5! = 5 x 4 x 3 x 2 x 1 = 120

```
int fatorial(int n){
  int i, result = 1;
  for (i = n; i > 1; i--){
    result = result * i;
  }
  return result;
}
```

Recursividade

Exemplo:

Fatorial

O fatorial de n (n!) é produto de todos os números de n até 1

Ex. 4! = 4 * 3 * 2 * 1

5! = 5 x 4 x 3 x 2 x 1

4! = 4 x 3 x 2 x 1

3! = 3 x 2 x 1

2! = 2 x 1

1! = 1

Recursividade

Exemplo:
Fatorial

O fatorial de n (n!) é produto de todos os números de n até 1
Ex. 4! = 4 * 3 * 2 * 1

$$5! = 5 \times \boxed{4 \times 3 \times 2 \times 1} \rightarrow 4!$$

$$4! = 4 \times \boxed{3 \times 2 \times 1} \rightarrow 3!$$

$$3! = 3 \times \boxed{2 \times 1} \rightarrow 2!$$

$$2! = 2 \times \boxed{1} \rightarrow 1!$$

$$1! = 1$$

Recursividade

Fatorial é um exemplo de problema com solução **recursiva**

$$F(n) = \begin{cases} 1 & \text{se } \; n = 0, n = 1 \\ n \, . \, F(n-1) & \text{se } \; n > 1 \end{cases}$$

$$5! = 5 \text{ x } \boxed{4 \text{ x } 3 \text{ x } 2 \text{ x } 1} \longrightarrow 4!$$

$$4! = 4 \text{ x } \boxed{3 \text{ x } 2 \text{ x } 1} \longrightarrow 3!$$

$$3! = 3 \text{ x } \boxed{2 \text{ x } 1} \longrightarrow 2!$$

$$2! = 2 \text{ x } \boxed{1} \longrightarrow 1!$$

$$1! = 1$$

Recursividade

Fatorial – Solução recursiva

n! = (n) * (n – 1)!

$$F(n)=\begin{cases}1 & \text{se } n=0, n=1\\ n.F(n-1) & \text{se } n>1\end{cases}$$

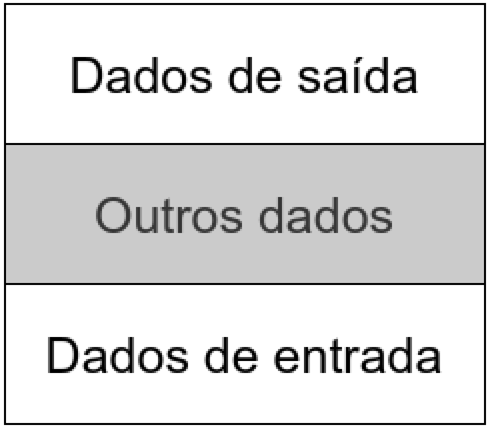
```
int fatorial(int n){
    if (n==0 || n == 1)
        return 1;
    return n * fatorial(n -1);
}
```

5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
2! = 2 x 1!
1! = 1

Estrutura de memória em uma chamada de função

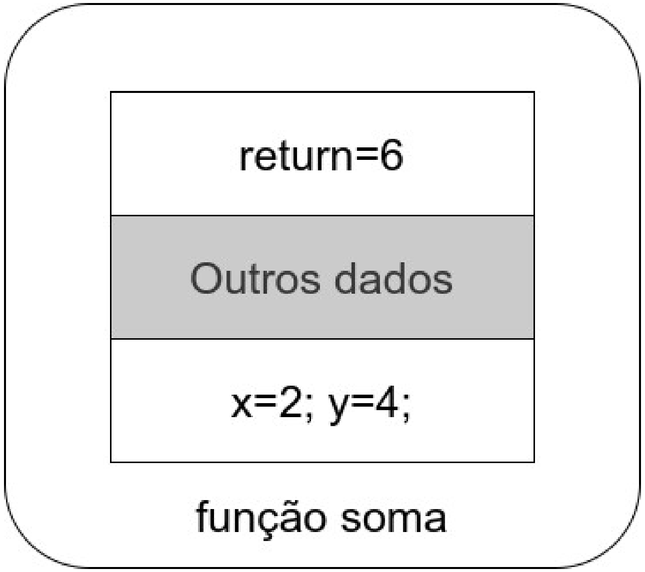
Dados de entrada: parâmetros

Dados de saída: resultado da execução da função



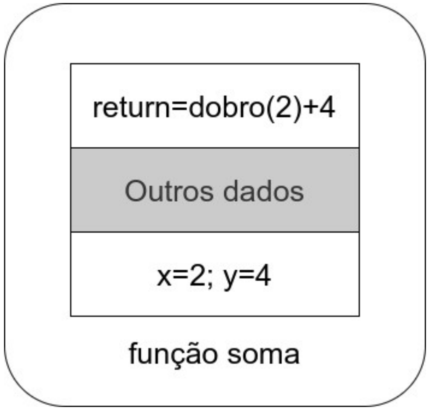
Estrutura de memória em uma chamada de função

```
int soma(int x, int y){

   return x + y;
}



int main(){
   v = soma(2,4)
}
```



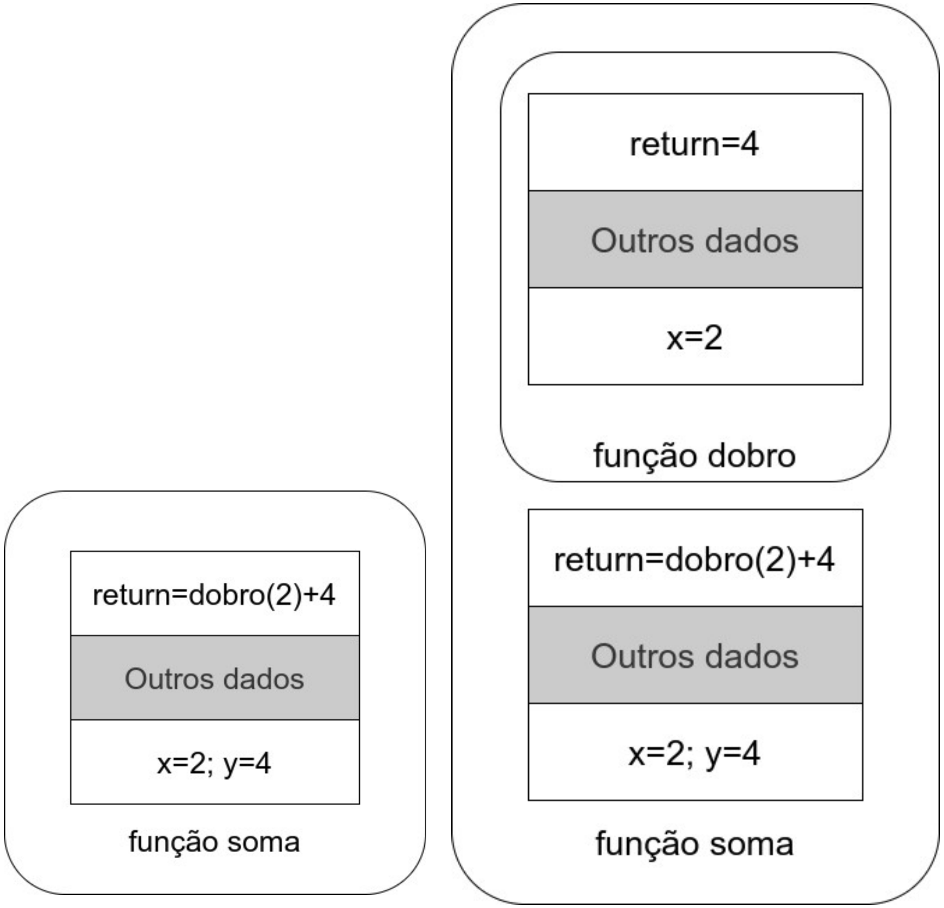
Estrutura de uma chamada de função (memória)

```
int soma(int x, int y){
   return dobro(x) + y;
}

int dobro(int x){
   return x * 2;
}

int main(){
   v = soma(2,4)
}
```



Estrutura de uma chamada de função (memória)

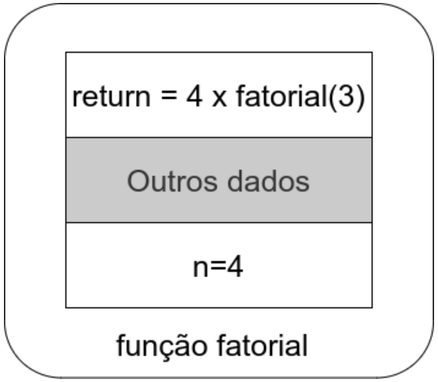
```
int soma(int x, int y){
   return dobro(x) + y;
}

int dobro(int x){
   return x * 2;
}

int main(){
   v = soma(2,4)
}
```



Exemplo - `fatorial(4)`

(slide 1/8)

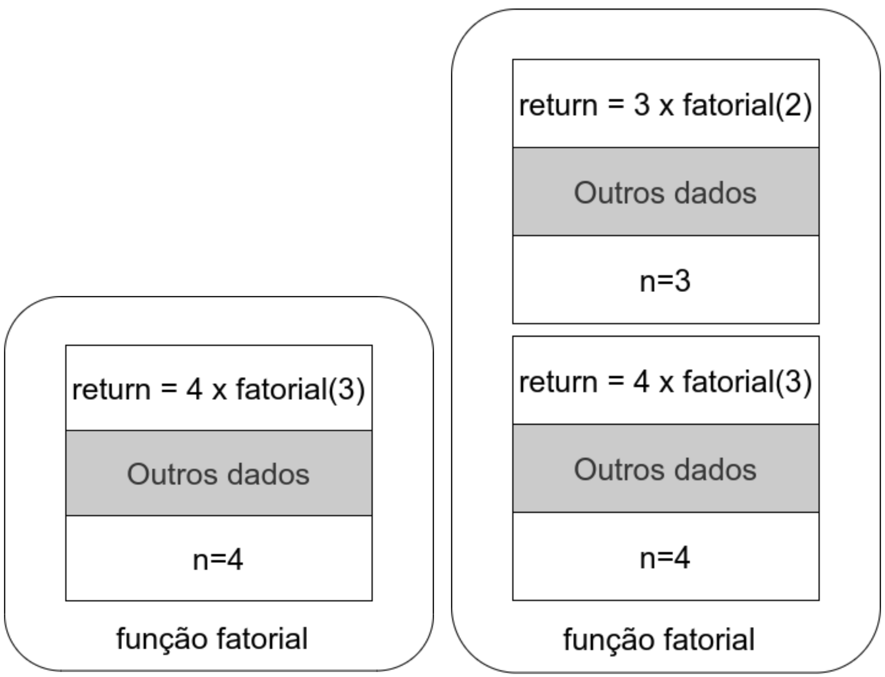
```
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);

}
```



Exemplo - `fatorial(4)`

(slide 2/8)

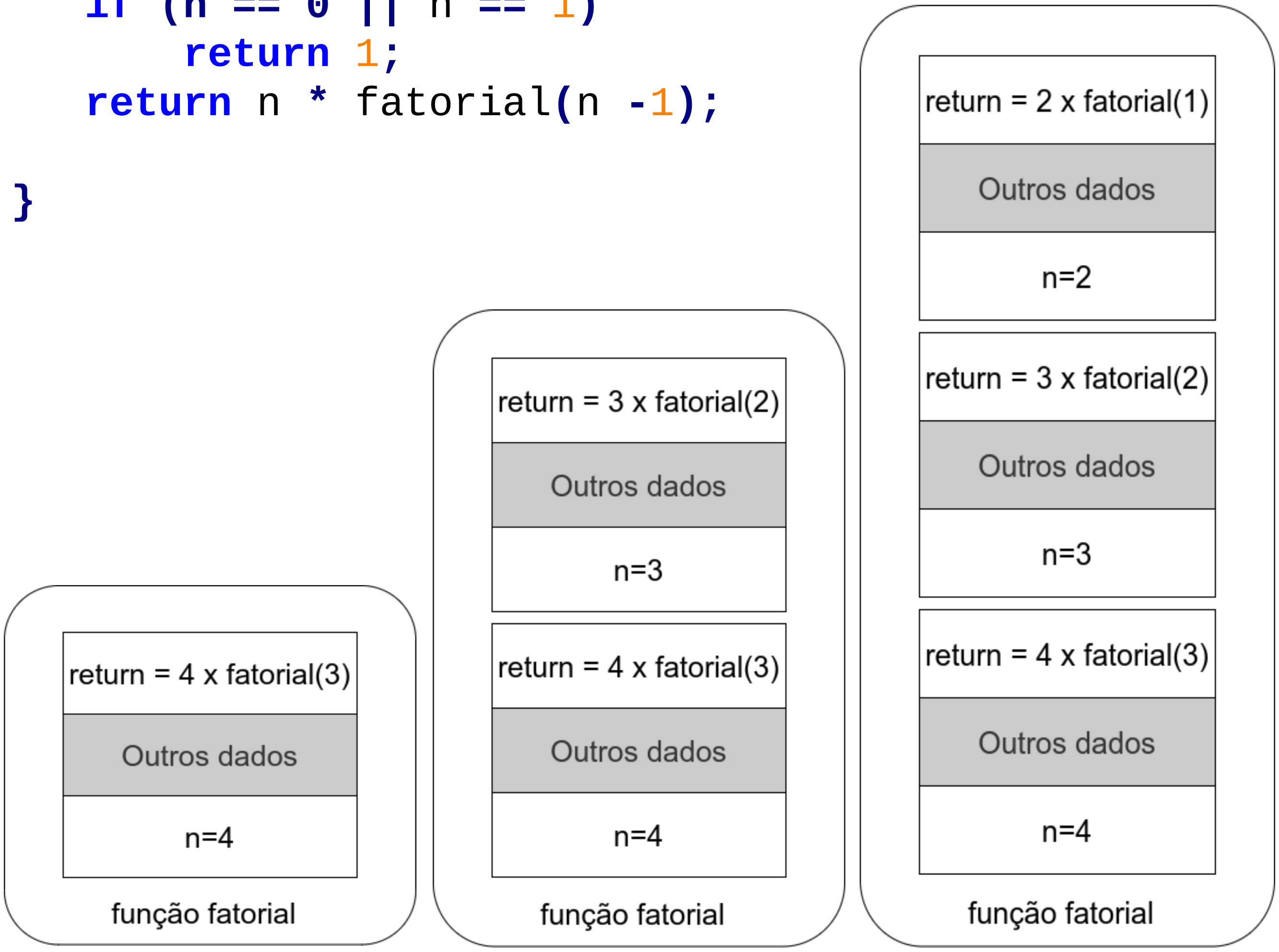
```
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);

}
```



Exemplo - `fatorial(4)`

(slide 3/8)

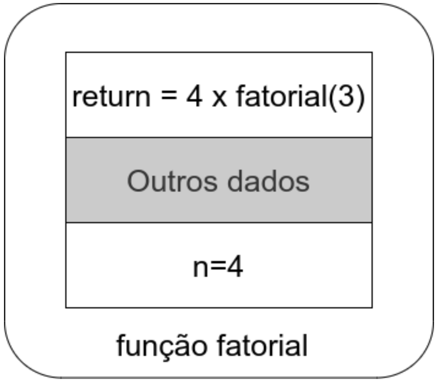
```
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);

}
```

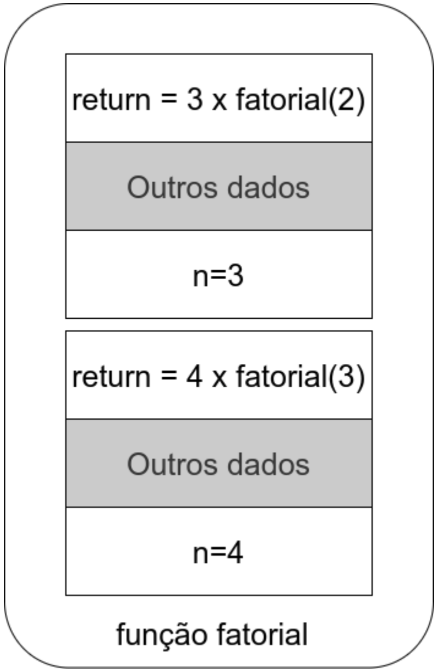


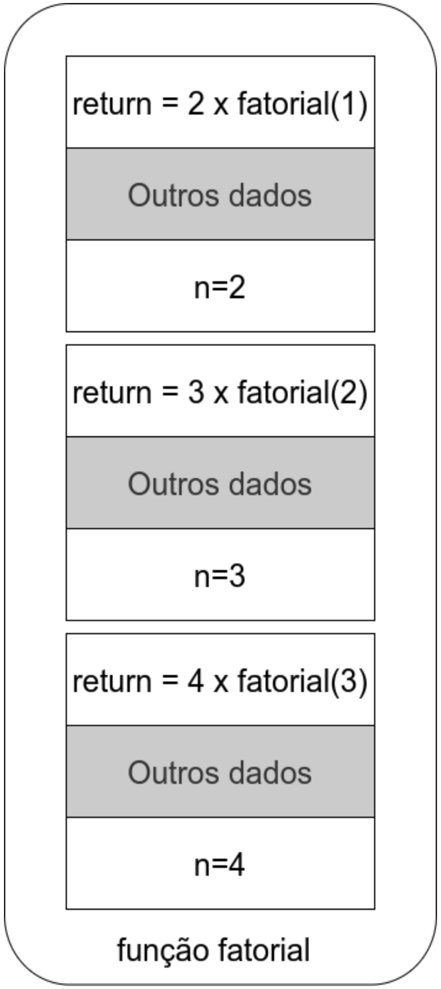
Exemplo - `fatorial(4)`

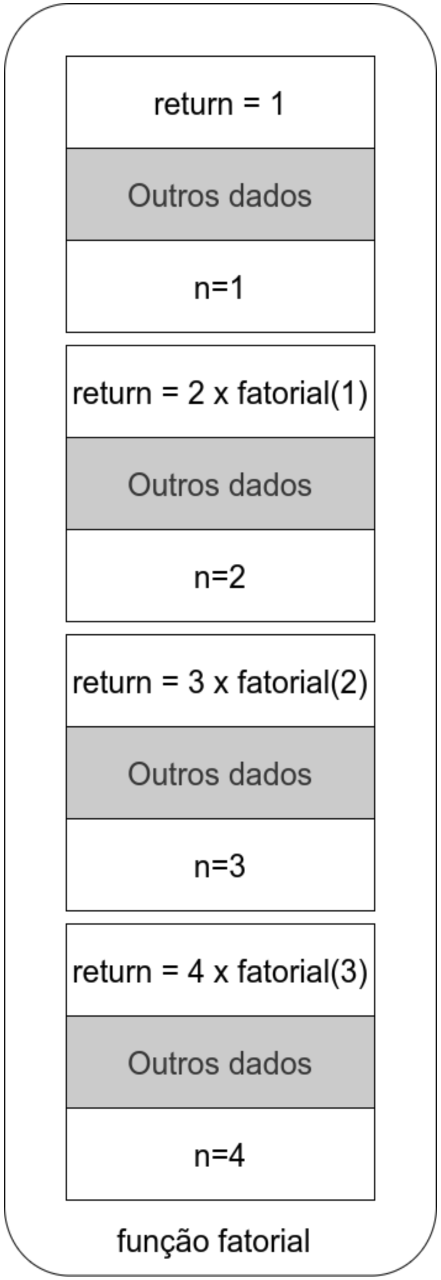
(slide 4/8)

```
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);

}
```



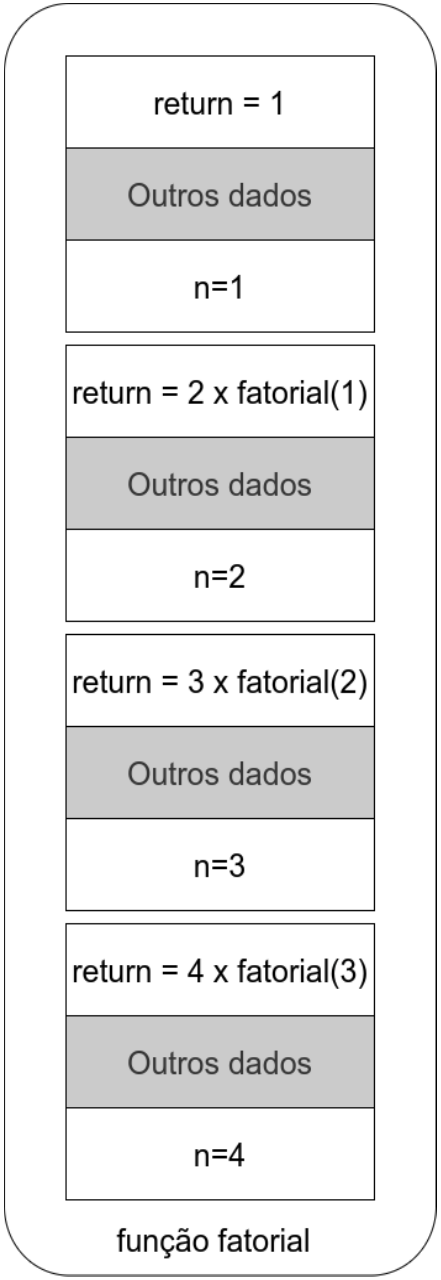






Exemplo - `fatorial(4)`

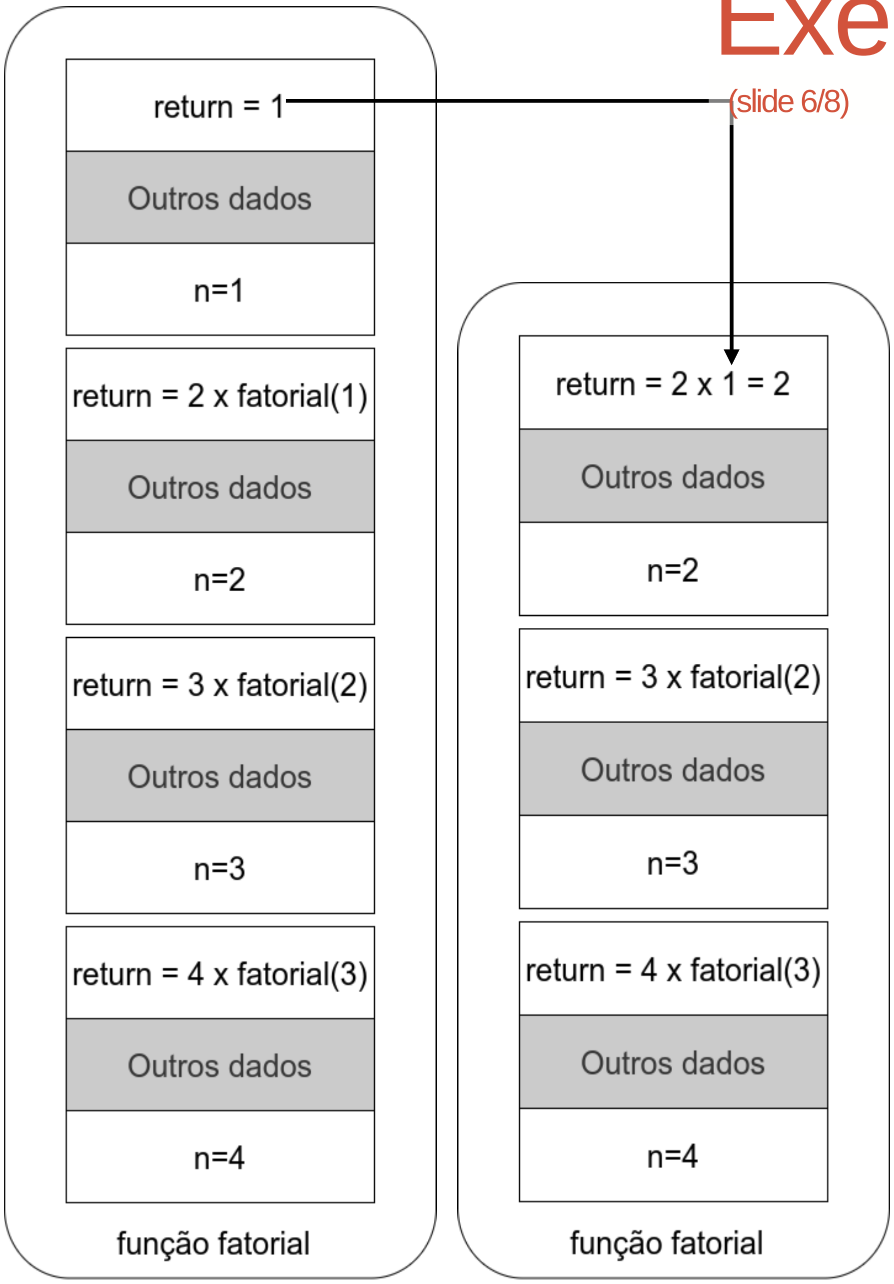
(slide 5/8)




```
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);

}
```

Exemplo - `fatorial(4)`

(slide 6/8)

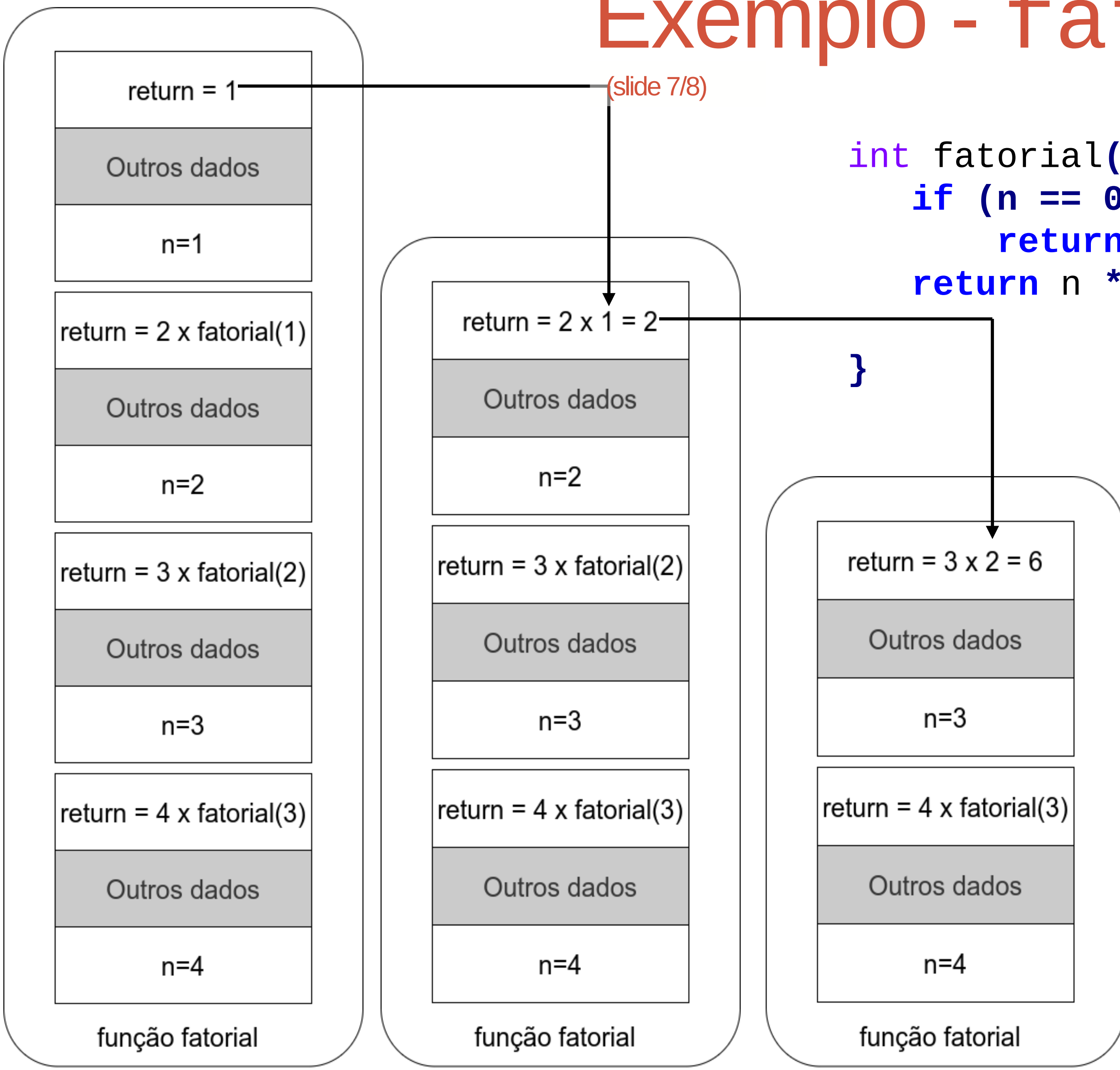



```c
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);

}
```

Exemplo - `fatorial(4)`

(slide 7/8)

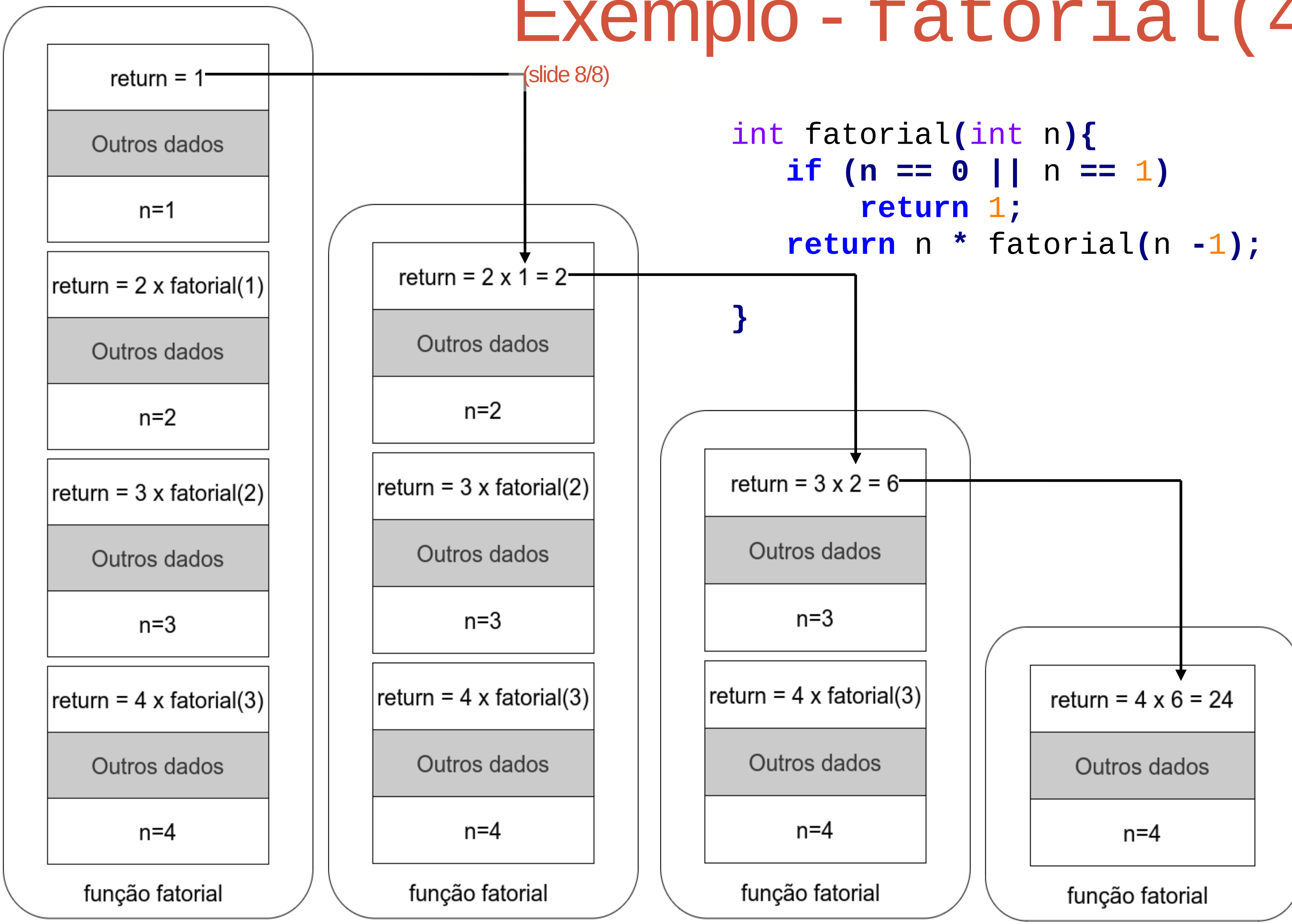
```c
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);

}
```



Exemplo - fatorial(4)

(slide 8/8)

```
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);

}
```



Recursividade – condição de parada

Uma função recursiva sempre terá uma *condição de parada*, que é a condição em que não há nova chamada à mesma função.

Se não houver condição de parada, a função fica eternamente fazendo chamadas recursivas

```
int fatorial(int n){
   if (n==0 || n == 1)
      return 1;
   return n * fatorial(n -1);
}
```

$$F(n)=\begin{cases} 1 & \text{se} \quad n=0, n=1 \\ n.F(n-1) & \text{se} \quad n>1 \end{cases}$$

Quando não usar recursividade?

$$f(n)=\begin{cases}1 & \text{se } n\leqslant 1\\ f(n-1)+f(n-2) & \text{se } n>1\end{cases}$$

```
int f(int n){
   if (n <= 1) return 0;
   return f(n-1)+f(n-2);
}
```

A uma única chamada à função *f(n)* acima pode fazer múltiplas chamadas à própria função para um mesmo *n.*

Estudo avançado (opcional)

Recursividade de cauda (*tail recursion*).

Na recursividade de cauda, a chamada recursiva é a última instrução executada pela função. Isso diminui o crescimento da pilha de execução e permite que compiladores façam otimizações.

Exemplo

```
//sem recursividade de cauda:
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);

}

int main(){
   int x = fatorial(5);
}
```

```
//com recursividade de cauda:
int fatorial(int n, int valor){

   if (n == 0 || n == 1)
       return valor;
   return fatorial(n-1, n*valor);

}

int main(){
   int x = fatorial(5,1);
}
```

Estudo avançado (opcional)

Recursividade de cauda (*tail recursion*).

Exemplo

```
//sem recursividade de cauda:
int fatorial(int n){
   if (n == 0 || n == 1)
       return 1;
   return n * fatorial(n -1);
}

int main(){
   int x = fatorial(5);
}
```

```
//com recursividade de cauda:
int fatorial(int n, int valor){
   if (n == 0 || n == 1)
       return valor;
   return fatorial(n-1, n*valor);
}

int main(){
   int x = fatorial(5,1);
}
```

```
fatorial(4)
4 * fatorial(3)
4 * (3 * fatorial(2))
4 * (3 * (2* fatorial(1))
4 * (3 * (2 * 1))
4 * (3 * 2)
4 * 6
24
```

```
fatorial(4,1)
fatorial(3,4)
fatorial(2,12)
fatorial(1,24)
24
```

Complexidade em recursividade

A complexidade de algoritmos recursivos pode ser descrita através de *equações de recorrência* na forma

$T(n) = aT(n/b) + D(n)$

Onde

  a é a quantidade de subproblemas em que o problema original é dividido
  b é o tamanho de cada subproblema
  D(n) é o custo da execução das instruções em cada chamada recursiva

Exemplo:
Um problema recursivo é resolvido de forma tal que, a cada chamada, lê-se todos os *n* elementos de elementos de um vetor e, em seguida, divide-se o vetor em três partes iguais, aplicando o mesmo processo a cada uma destas partes. Nesse caso, tem-se:

$a = 3$
$b = n/3$
$D(n) = n$

$T(n) = 3(n/3) + n$

Complexidade em recursividade

Há várias formas de resolver recorrências. Uma delas é o **teorema mestre**.

Seja uma equação de recorrência $T(n) = aT(n/b) + D(n)$

Se $D(n) = O(n^d)$ (para $d \geq 0$) então:

$T(n) = O(n^d)$ se $a<b^d$
$T(n) = O(n^d \log n)$ se $a = b^d$
$T(n) = O(n^{\log_b a})$ se $a > b^d$

Exemplo:

$T(n) = 3(n/3) + n$ $a = 3; b = 3, d = 1$ $a = b^d$ $O(n \cdot \log n) =$	$T(n) = 2T(n/4)+n^3$ $a = 2; b = 4; d = 3$ $a < b^d$ $O(n^3)$

Referências

Figuras retiradas do livro:

Loudon, Kyle. Mastering Algorithms with C. O’Reilly Media, Inc. 2009.