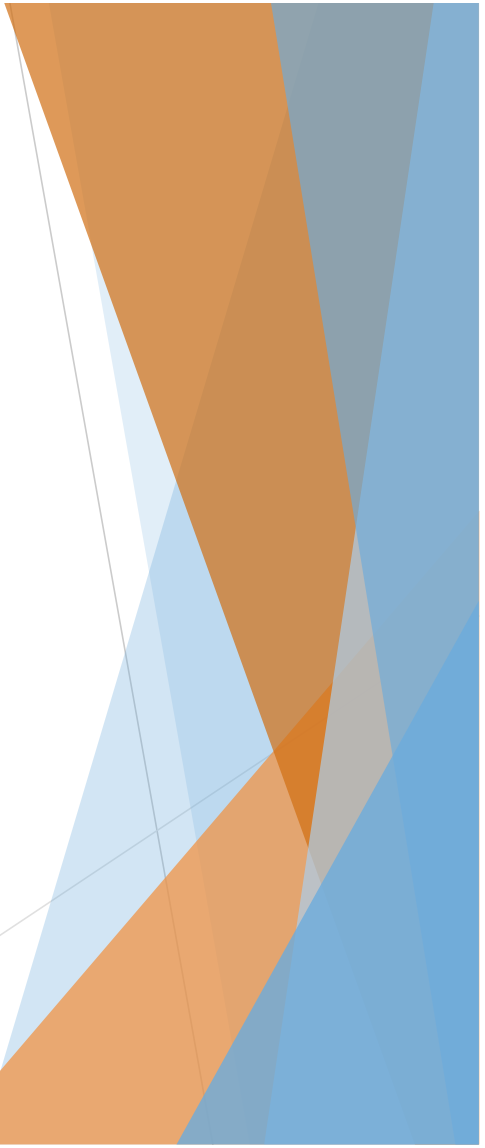
ALGORITMOS E PROGRAMAÇÃO DE COMPUTADORES

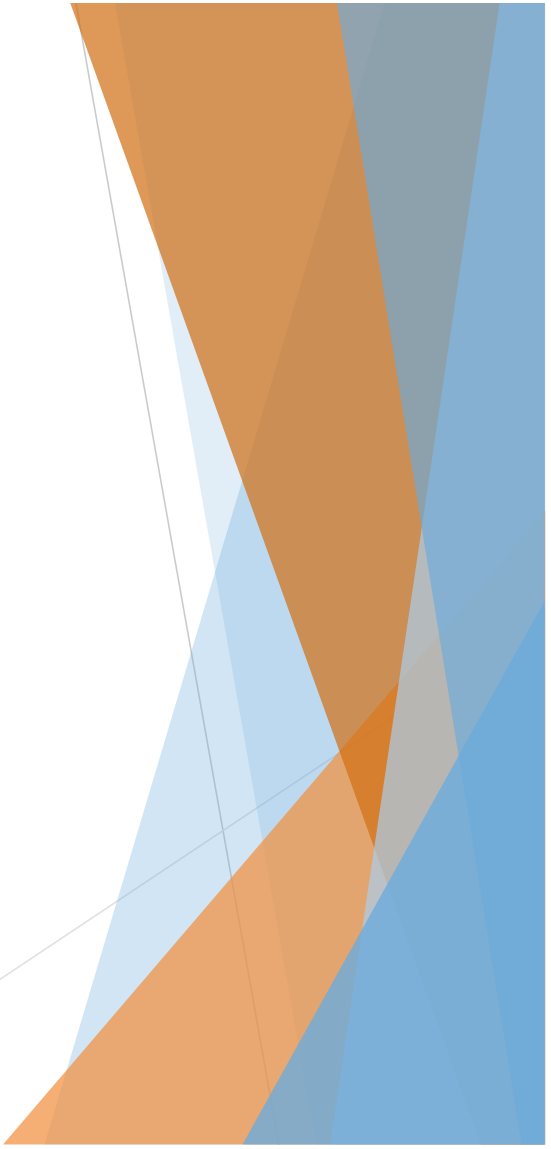
Disciplina: 113476

Profa. Carla Denise Castanho

Universidade de Brasília – UnB
Instituto de Ciências Exatas – IE
Departamento de Ciência da Computação – CIC

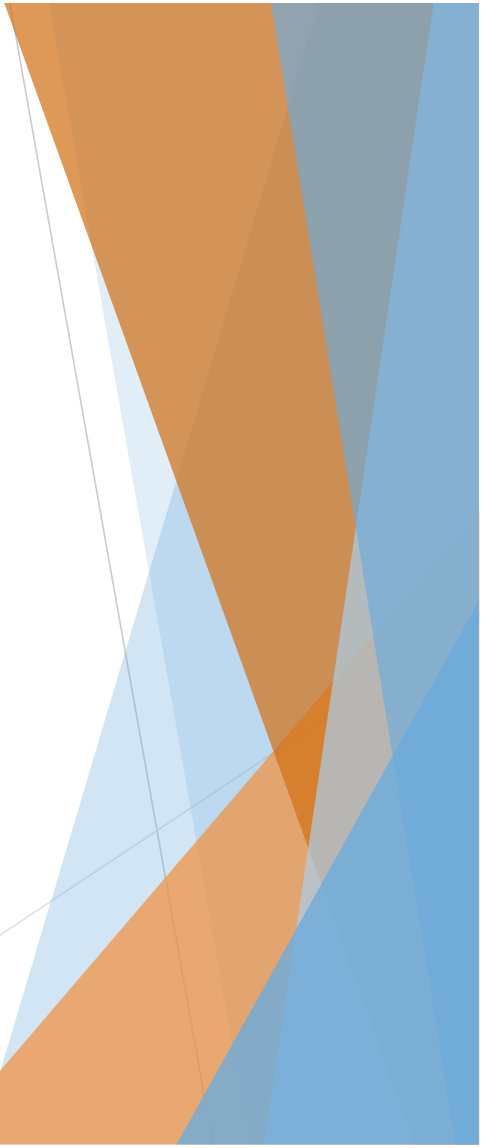


13. RECURSIVIDADE



Recursividade

- Depois de ter trabalhado com funções, você já deve ter se perguntado: uma função pode chamar a si mesma?
- Sim! Essa técnica chama-se **recursividade**, e muitas vezes facilita a resolução de certos tipos de problemas.



Recursividade

- Um problema *recursivo* é aquele em que **uma instância** do problema pode ser **definida em termos de instâncias menores** ou seja, sub-instâncias do mesmo problema. Chamamos esse tipo de definição de **recorrência**.
- Por exemplo, o fatorial de *N* pode ser definido tanto de maneira **iterativa**:

$$Fat(N) = N * (N-1) * ... * 1.$$

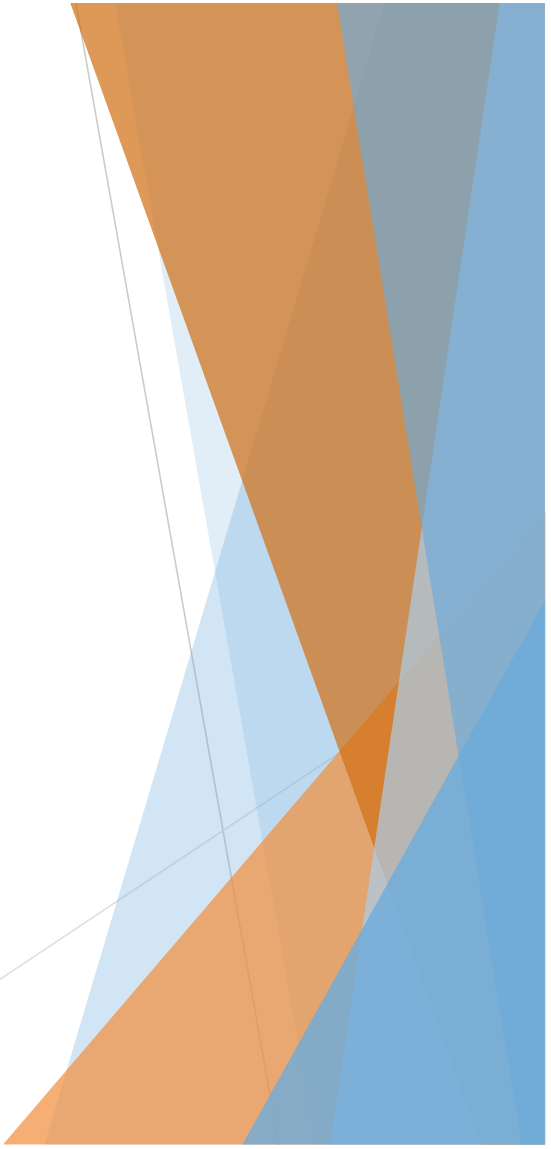
- Como de maneira **recursiva**, em termos do fatorial de *N-1*:

$$Fat(N) = N * Fat(N-1),$$

$$\text{dado } Fat(0) = 1.$$

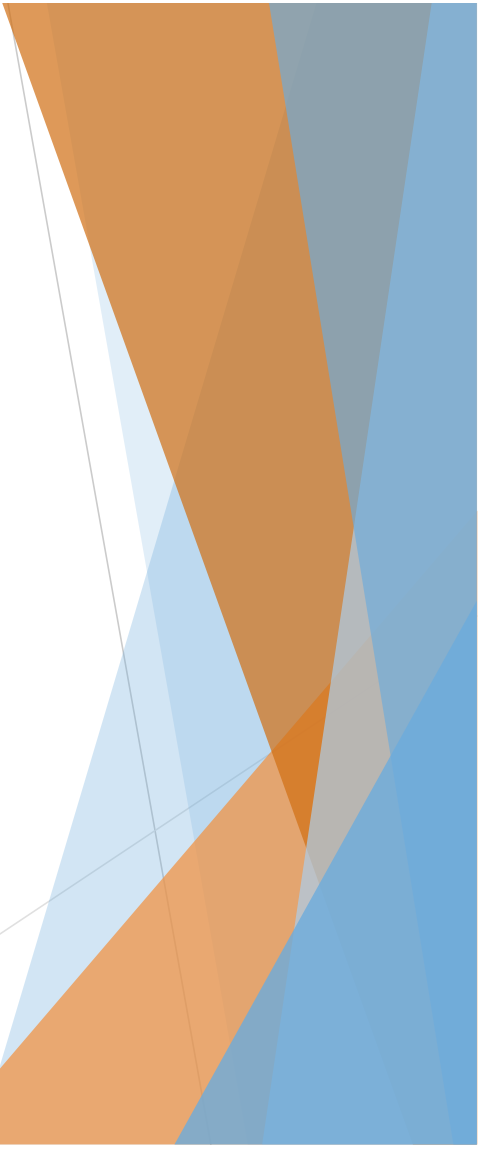
Recursividade

- Mas a pergunta é: **quando parar?**
- Se a função chama a si mesma sempre, ela entra em loop: faz **infinitas chamadas recursivas** e nunca para de computar!
- Por isso é necessário definir um **caso base**! No nosso exemplo, Fat(0) = 1.
- Vamos ver como funciona...

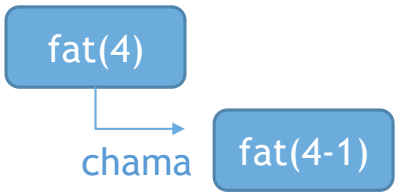


Recursividade

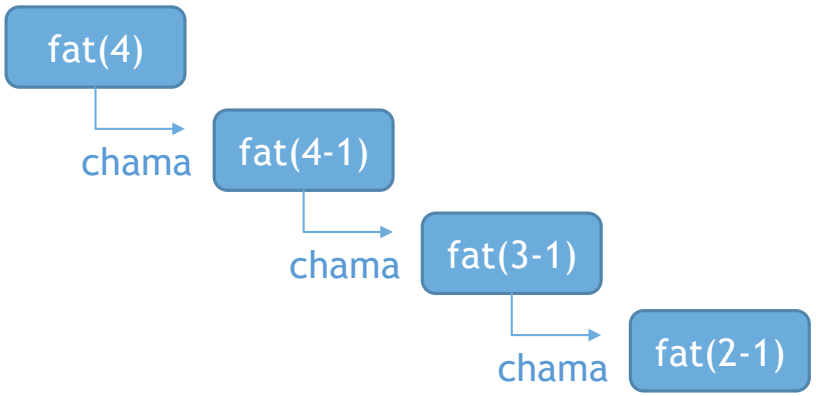
fat(4)



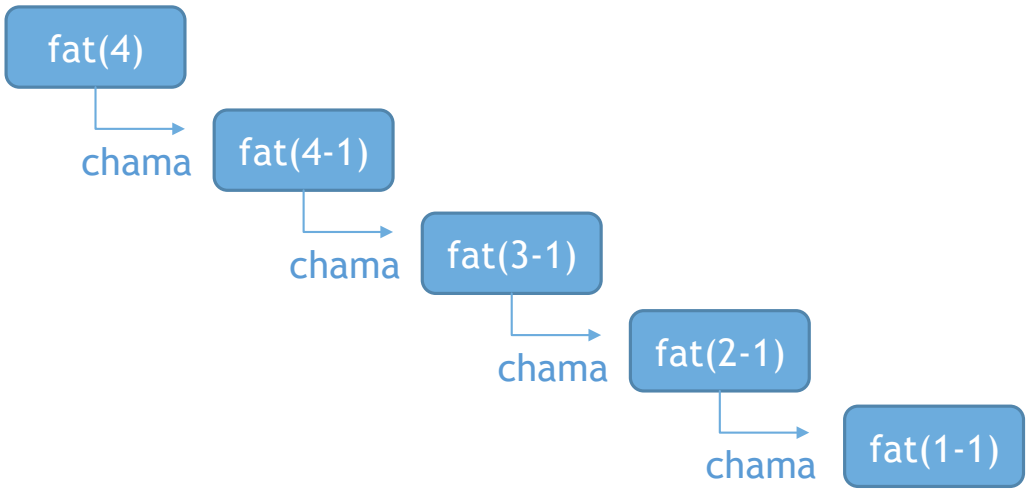
Recursividade



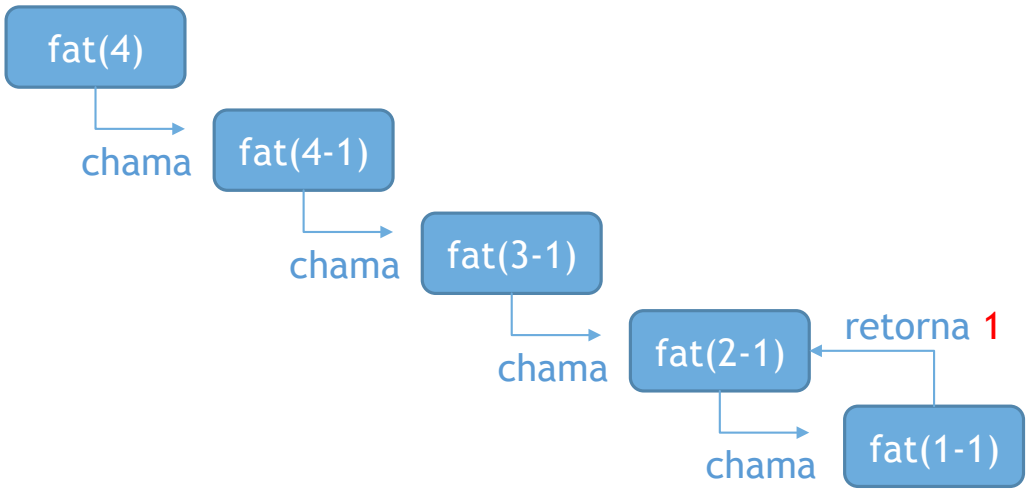
Recursividade



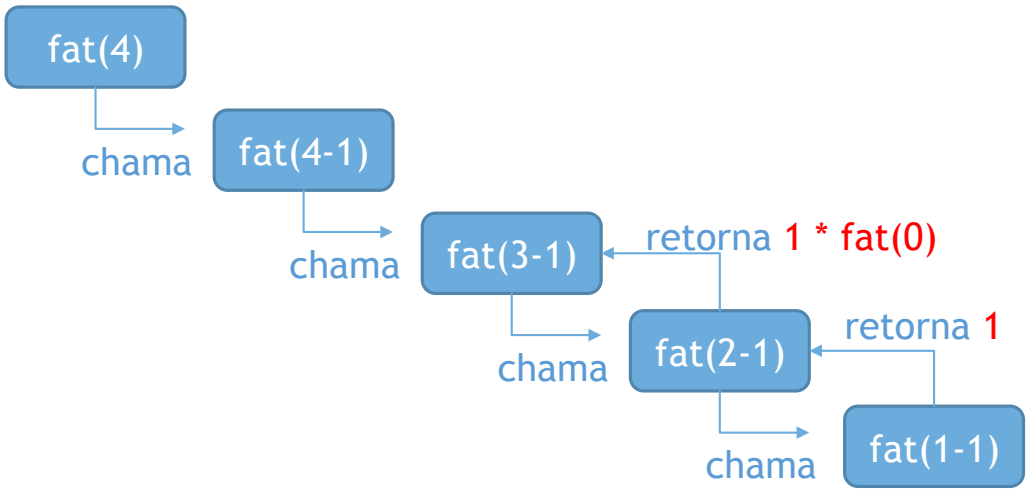
Recursividade



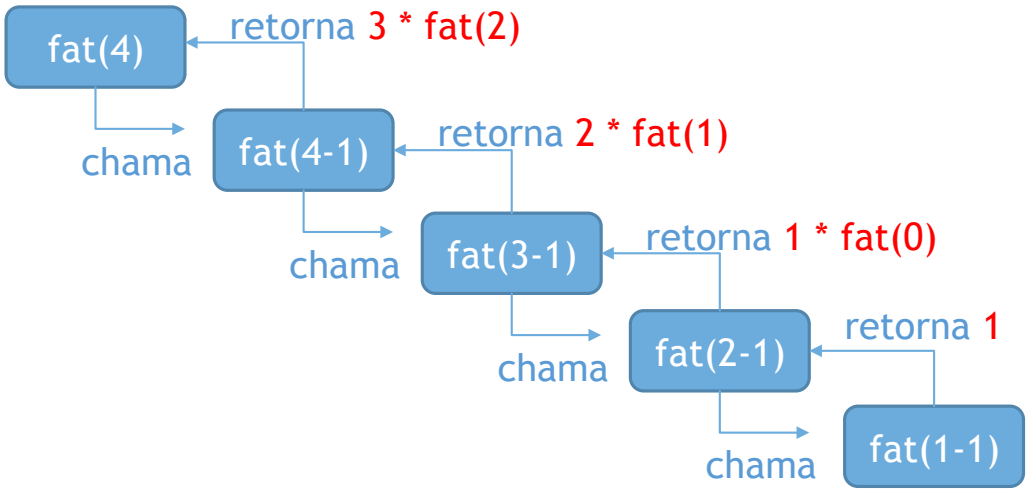
Recursividade



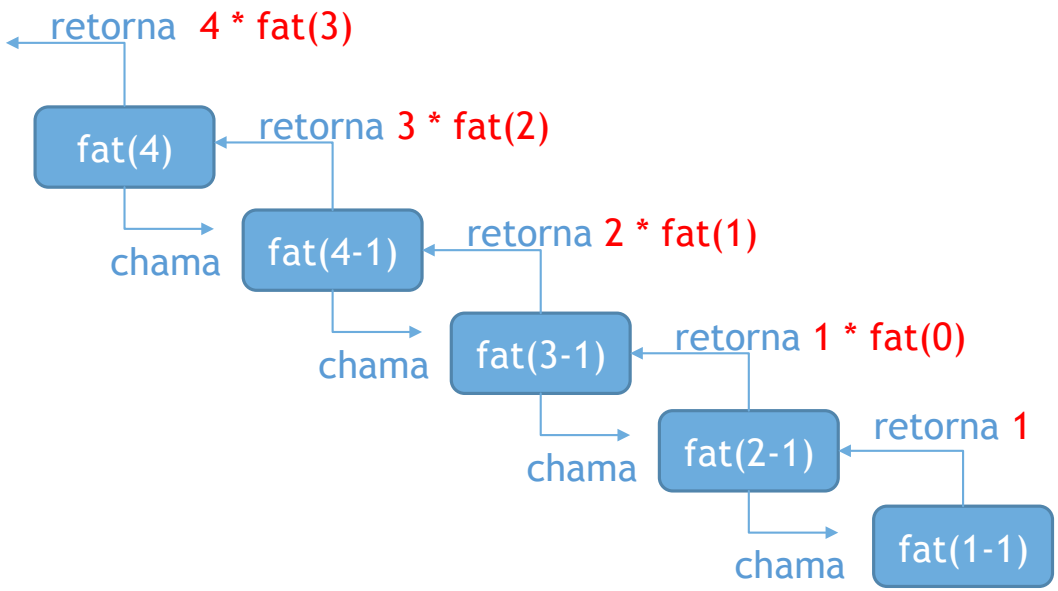
Recursividade



Recursividade



Recursividade



Recursividade

Exemplo - Fatorial Recursivo

```
Algoritmo Fatorial
Função Fat (n : inteiro)
Início
    Se (n = 0)
        retorne 1
    Senão
        retorne n * Fat(n-1)
Fim
Variáveis
    n : inteiro
Início
    Escreva ("Informe um inteiro não negativo:")
    Leia (n)
    Escreva ("O fatorial de ", n, " é: ", Fat(n))
Fim
```

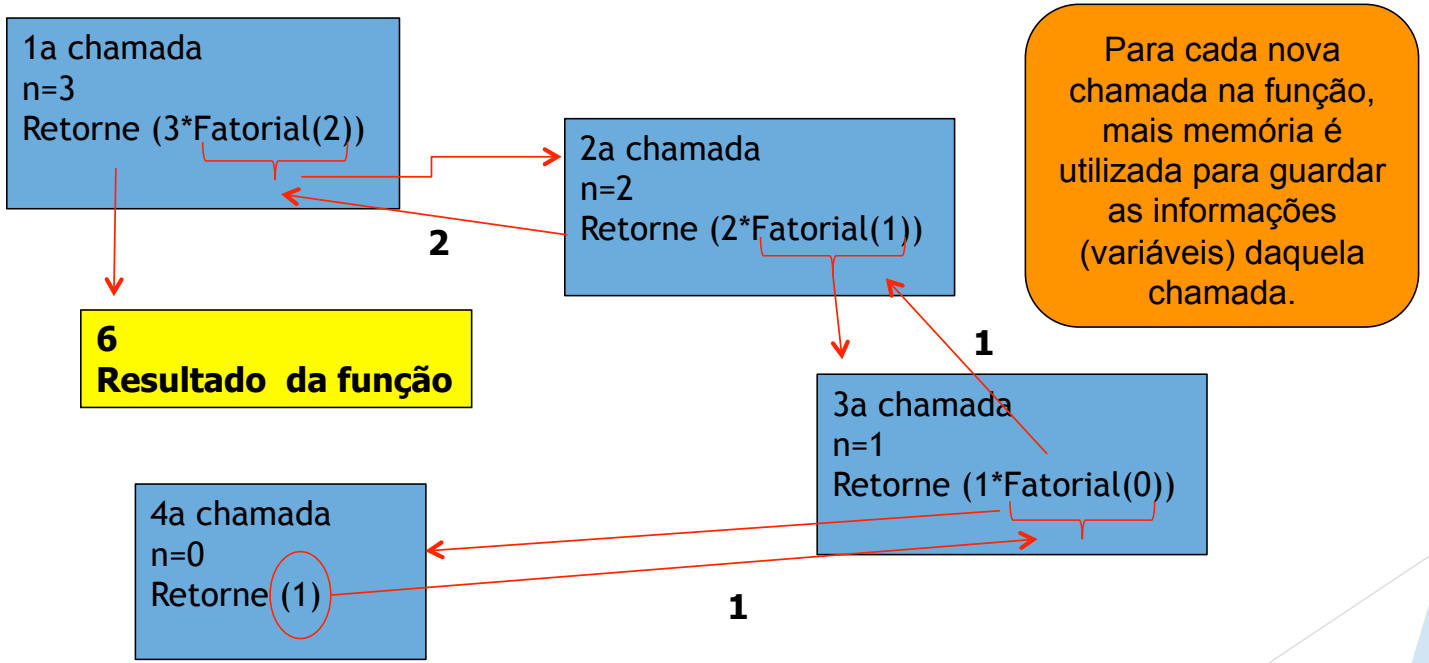
Recursividade

Programa C do Exemplo Anterior

```c
#include <stdio.h>

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

int main () {
    int n;

    printf("Informe um inteiro positivo: ");
    scanf("%d", &n);
    printf("O fatorial de %d eh %d.\n", n, fatorial(n));

    getchar();
    return 0;
}
```

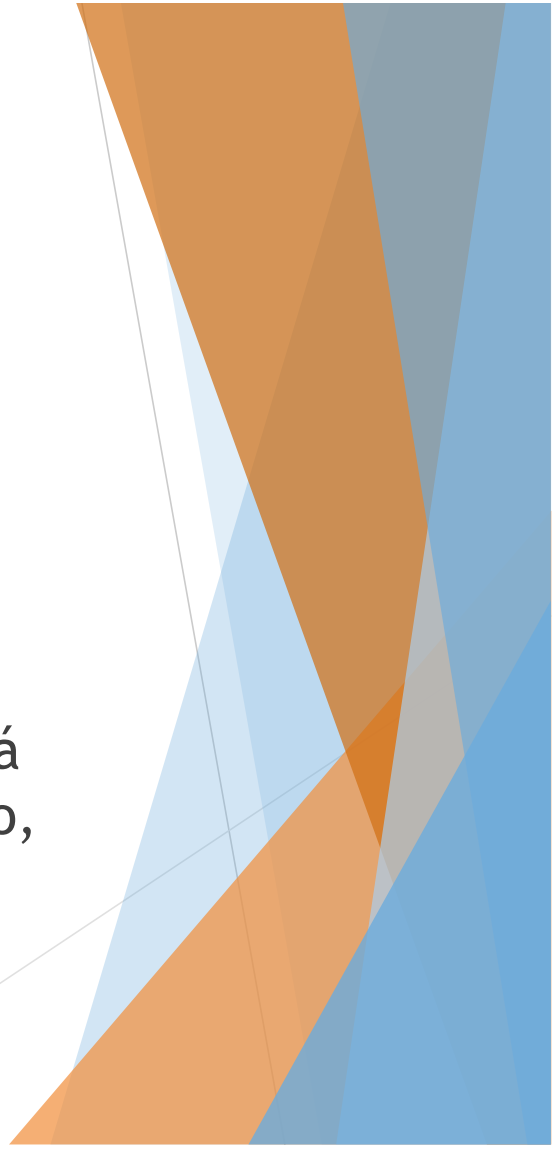
Recursividade

- Visualmente podemos analisar o fatorial de 3, fat(3), da seguinte forma:



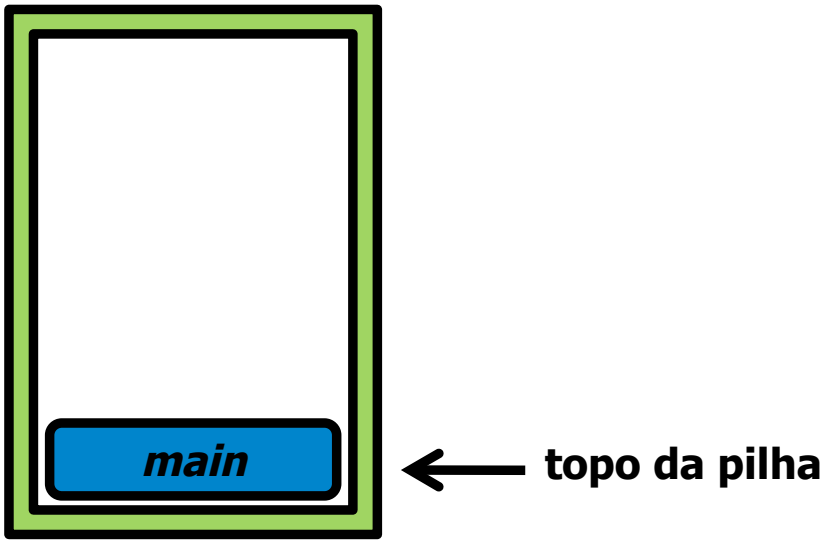
Pilha de Execução

- Para entender como funciona a recursividade, precisamos entender o conceito de **pilha de execução**.
- Toda vez que nós chamamos uma função, o computador **reserva memória** para as variáveis e parâmetros daquela chamada específica.
- A função que está sendo executada no momento está no **topo** da pilha, se ela chamar alguma outra função, esta será **empilhada** e passa a ficar no topo.
- Vamos ver um esquema...



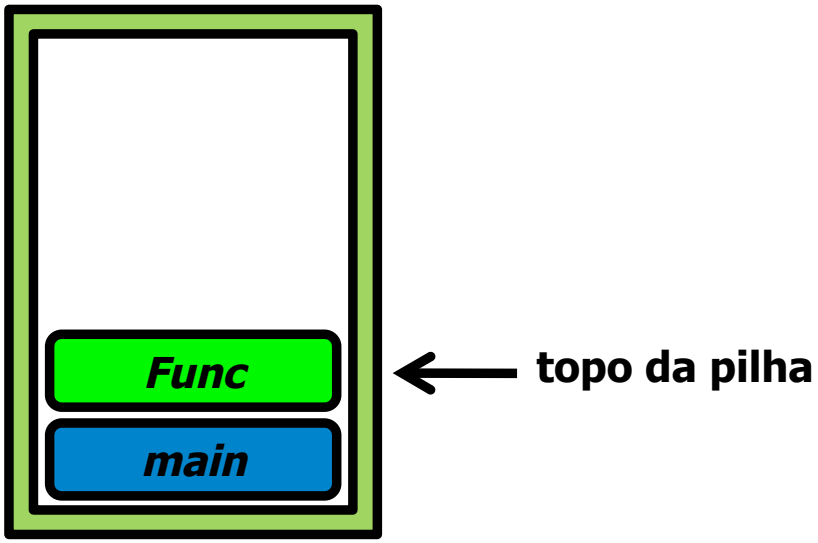
Pilha de Execução

- No início do programa, a pilha de execução parece mais ou menos assim:



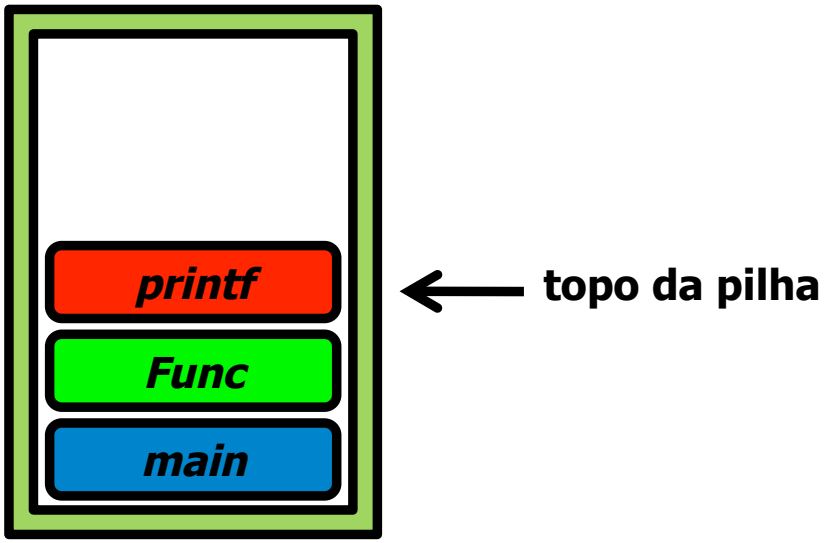
Pilha de Execução

- Por exemplo, se *main* chamar a função *Func*, a pilha ficará assim:



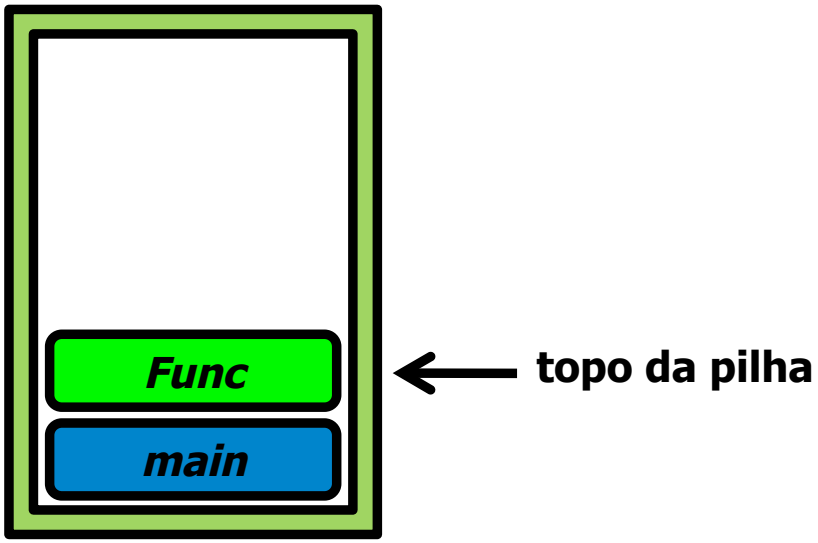
Pilha de Execução

- E se, dentro de *Func*, for chamada *printf*, então teremos:



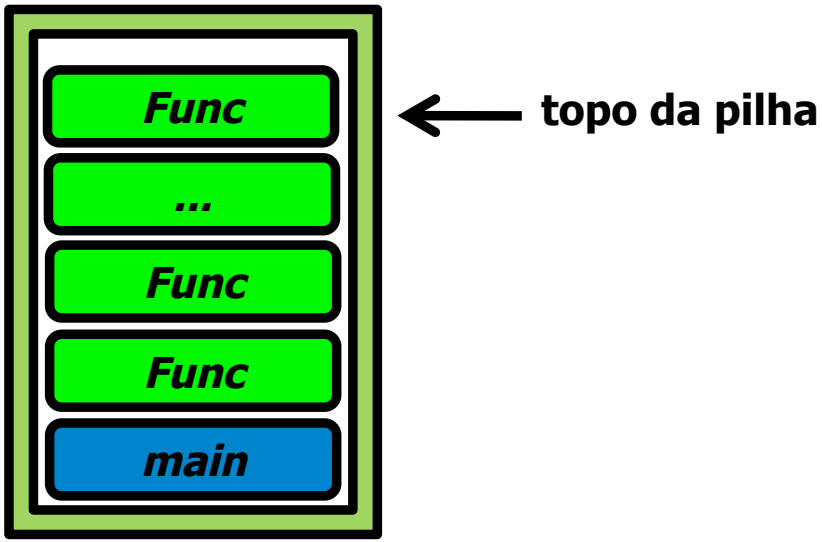
Pilha de Execução

- Quando uma chamada de função termina, ela é **desempilhada**, e a função anterior continua. Por exemplo, depois *printf* retorna, voltamos a:



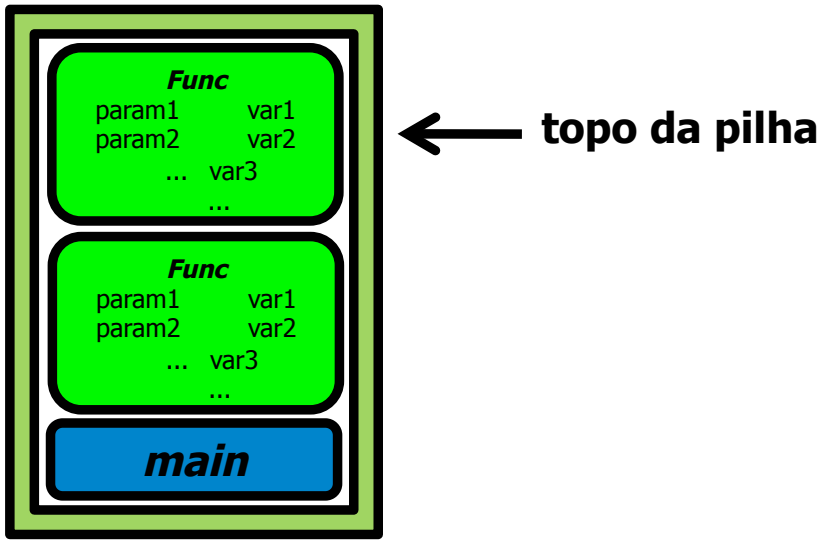
Pilha de Execução

- Logo, uma chamada recursiva nada mais é que uma simples chamada de função. Tudo que acontece é que várias instâncias da mesma função ficam empilhadas mais de uma vez...



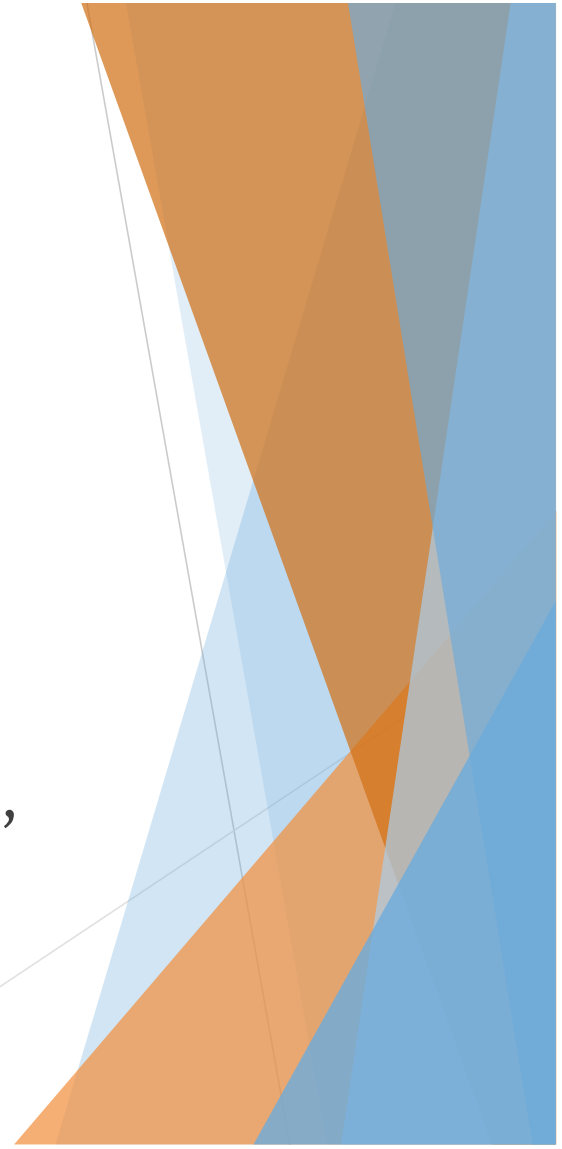
Pilha de Execução

- Cada vez que *Func* é empilhada, são reservados **novos** espaços para todas as **variáveis locais** de *Func*, inclusive seus **parâmetros**, que são e então usados pela **nova instância**:



Pilha de Execução

- Portanto, o que acontece em **uma instância** de uma função recursiva **não influencia o que acontece em outras instâncias** porque as variáveis, de fato, estão em locais diferentes da memória.
- Mas note que **chamar funções ocupa memória**!
- Por isso, se um algoritmo recursivo faz muitas chamadas, ele pode causar um **estouro de pilha**, ou seja, ficar **sem memória** suficiente para continuar!



Cuidados com a Recursividade!

- Se um problema é inerentemente recursivo, o algoritmo recursivo normalmente é **mais fácil de escrever e mais simples de entender**.
- No entanto, é preciso **analisar se vale a pena a solução recursiva**: ela pode **ocupar muita memória** e é sempre **mais lenta** que a solução iterativa, pois gasta-se tempo empilhando e desempilhando as várias chamadas.

Conclusões

- Uma solução recursiva potencialmente ocupa mais memória e pode ser mais lenta que a solução iterativa para um mesmo problema.
- Em cada instância, é sempre alocada memória para todos os parâmetros e variáveis locais, independentemente dos que já existiam antes.
- A cada nova chamada, o sistema deve guardar a posição de memória de onde a chamada foi feita, para que posteriormente possa voltar ao lugar certo.

Conclusões

- Um programa recursivo é normalmente mais elegante e menor que a sua versão iterativa, além de exibir com maior clareza o processo utilizado, desde que o problema ou dados sejam naturalmente definidos através da recorrência.
- É fácil criar funções que chamem a elas mesmas.
- É difícil reconhecer situações apropriadas para a utilização de recursividade.
- Há certos problemas cuja natureza permite uma solução recursiva bem mais simples e intuitiva do que a solução iterativa.