

INF01202

Algoritmos e Programação

Modalidade Ead – Turma H

Material de apoio: Recursão em C
(funções recursivas)

Uma definição ou problema é dito **recursiva** se é definível em termos de si mesmo.

Exemplo de um definição recursiva - fatorial.
Vejam os:

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

é equivalente a

$$n! = n * (n - 1)!$$

isto é, fatorial de n é igual a n vezes o fatorial de n-1.

Definição da fatorial recorre a si mesma para se definir!!

Em uma pseudo-linguagem para se escrever a lógica do fatorial de um número, teremos:

se $n = 0$

fatorial de $n = 1$

senão

fatorial de $n = n * \text{fatorial de } n-1$

Um exemplo numérico:

fatorial de 5 = 5 * fatorial de (4 * fatorial de (3 * fatorial de (2 * fatorial de (1 * fatorial (0))

Implementação de recursividade em C

Através do cálculo do fatorial de um valor , veremos como se implementa a recursão em C:

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

Chamada de fatorial ,
novamente!!

Observar que se o valor usado para ativar a função for diferente de 0, pelo menos uma vez o *else* do *if* será ativado e a função será novamente chamada, antes que a execução anterior tenha sido concluída.



Implementação de recursividade em C

Desde que o escopo de um identificador de um subprograma (função) estende-se do seu cabeçalho até o final do bloco em que é declarado,

um subprograma (função) pode ser chamado de dentro de outro subprograma (função) ou até mesmo de dentro de si mesmo.

Funções recursivas

Criação e destruição de elementos locais às funções e parâmetros por valor

Ao ser ativada uma função, é alocada memória para os elementos **locais** à mesma (variáveis, etc.) e seus parâmetros por valor, ou seja, eles **são criados**,

e, tão logo a função conclui, essas áreas de memória são **liberadas**, ou seja, os elementos que a elas correspondem **deixam de existir**.

Implementação de uma função recursiva

Problema

A tentativa de execução de uma função recursiva implica tipicamente na ocorrência de **execuções interrompidas** que em um momento futuro devem ser retomadas e concluídas.

PROBLEMA:

Como garantir que uma execução interrompida possa ser retomada do ponto em que ocorreu a interrupção, com garantia de que ela seguirá sendo executada com os recursos locais idênticos àqueles do momento da interrupção?

Implementação de uma função recursiva

Solução:

empilhamento de locais e endereço de retorno

- A cada interrupção de uma execução a ser retomada no futuro, salva-se em uma pilha (stack):
 - os elementos locais (variáveis, etc.) e os parâmetros por valor; e
 - o endereço de retorno.
- Quando uma execução interrompida tiver que ser retomada:
 - restaura-se os valores dos elementos locais (variáveis, etc.) e dos parâmetros por valor a partir da pilha (stack); e
 - a execução segue a partir do ponto indicado pelo endereço de retorno.

Stack (pilha)

Espaço de memória onde são guardados os endereços de retorno associados às chamadas a funções, bem como os valores dos elementos locais às funções, em caso de chamadas recursivas em suspenso.

Execução de uma função recursiva

- Início da execução:

criação (alocação de memória) de locais:

- variáveis, constantes, etc. locais à função
e
- parâmetros por valor.

Execução de uma função recursiva (cont.)

- Processamento:

Tentativa de execução do código até o final da função.

Se a execução for interrompida por nova chamada recursiva, ocorre o empilhamento das variáveis, constantes, etc. locais à função e de seus parâmetros por valor, bem como do endereço de retorno da função.

Execução de uma função recursiva (cont.)

- Retomada de uma execução interrompida:

Retomada da execução da função **a partir do ponto onde foi interrompida**, usando o endereço de retorno que estava também empilhado.

Desempilhamento dos elementos locais à função com **restauração de seus conteúdos** conforme encontravam-se ao dar-se a interrupção da execução.

Execução de uma função recursiva (cont.)

- Fim de uma execução:

liberação das áreas de memórias dos **elementos locais à função e dos parâmetros por valor** (= eliminação dos mesmos), e retorno ao ponto de chamada da função recursiva.

ATENÇÃO

Parâmetros por endereço
NÃO
sofrem empilhamento!!!

Eles
NÃO SÃO LOCAIS
às funções!!!

Desenvolvimento de uma função recursiva

CUIDADO BÁSICO

Definir, tipicamente no início, a condição de interrupção das sucessivas chamadas recursivas que, uma vez ocorrendo, conclua finalmente uma execução e permita a conclusão de todas as outras execuções pendentes.

Desenvolvimento de uma função recursiva

CAUIDADO BÁSICO

Ou seja, iniciar sempre com a(s) condição(ões) que determina(m) as condições de término da função recursiva.

No caso da função recursiva fatorial, o **critério de término** se dá quando o **valor** passado à função é **zero**:

```
int fatorial(int valor)
{
    if (valor == 0) // término
        return 1;
    else
        return valor * fatorial (valor -
1);
}
```

Simulação da execução da função FATORIAL recursiva para o valor 5

Atenção:

os endereços de retorno não estão indicados
na pilha, embora sejam também nela
armazenados.

Código da função fatorial e programa que a chama

```
#define <stdio.h>
#define <stdlib.h>
int fatorial(int);
int main ( )
{
    int valfat;
    system("color 70");
    printf("\nValor para calcular fatorial: ");
    scanf("%d", &valfat);
    printf("\nFatorial de %d: %d\n", valfat, fatorial(valfat));
    system("pause");
    return 0;
}


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

Valor é um parâmetro por valor: é criado quando fatorial é ativada e deixa de existir quando fatorial encerra sua execução.

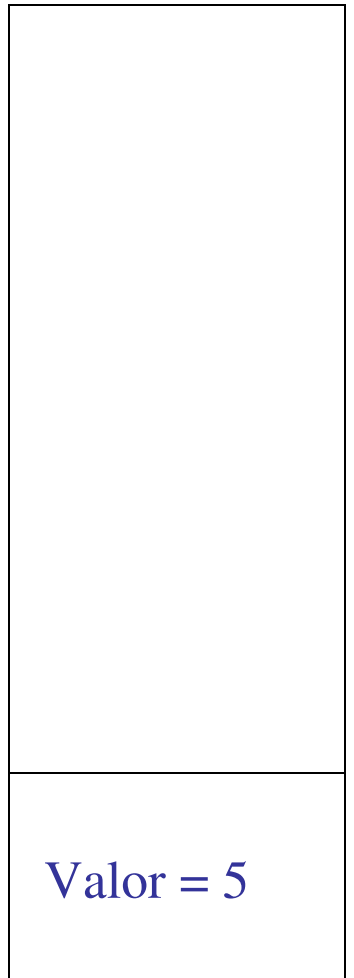
Chamada 1 (Valor = 5) - INTERROMPIDA

A Pilha
(Stack)
na saída

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




FATORIAL: Indefinida

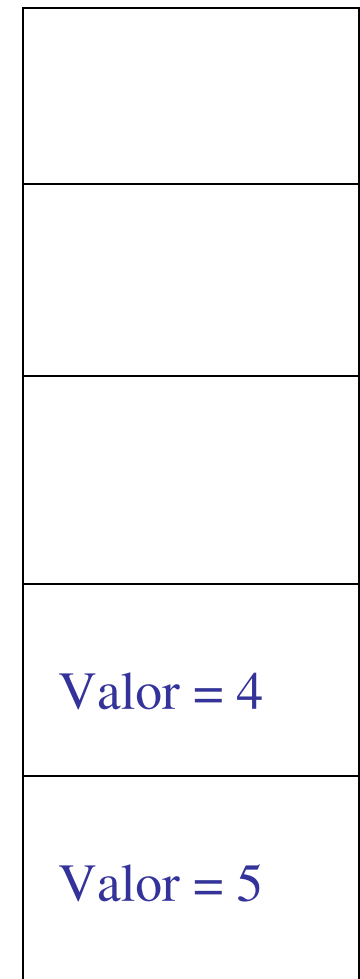


Chamada 2 (Valor = 4) - INTERROMPIDA

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




**A Pilha
(Stack)
na saída**



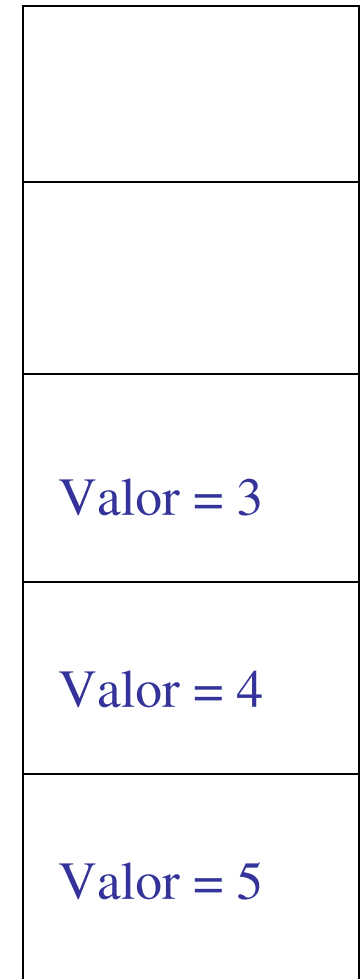
FATORIAL: Indefinida

Chamada 3 (Valor = 3) - INTERROMPIDA

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




**A Pilha
(Stack)
na saída**



FATORIAL: Indefinida

Chamada 4 (Valor = 2) - INTERROMPIDA

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




**A Pilha
(Stack)
na saída**

Valor = 2
Valor = 3
Valor = 4
Valor = 5

FATORIAL: Indefinida

Chamada 5 (Valor = 1) - INTERROMPIDA

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



**A Pilha
(Stack)
na saída**

Valor = 1
Valor = 2
Valor = 3
Valor = 4
Valor = 5

FATORIAL: Indefinida

Chamada 6 (Valor = 0) - CONCLUÍDA

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

FATORIAL = 1

**A Pilha
(Stack)
na saída**

Valor = 1
Valor = 2
Valor = 3
Valor = 4
Valor = 5

Chamada 5 (Valor = 1) - RETOMADA E CONCLUÍDA

A Pilha
(Stack)
na saída

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

Valor = 2

Valor = 3

Valor = 4

Valor = 5

FATORIAL = 1 (1 * 1)



Chamada 4 (Valor = 2) - RETOMADA E CONCLUÍDA

A Pilha
(Stack)
na saída

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

Valor = 3

Valor = 4

Valor = 5

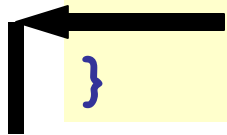
FATORIAL = 2 (2 * 1)



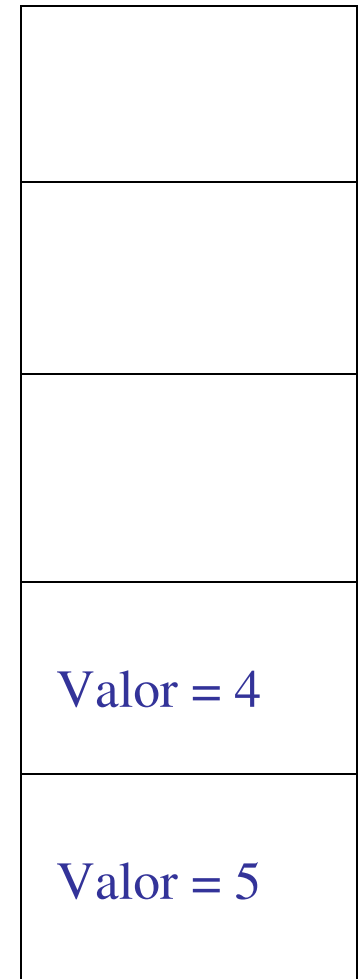
Chamada 3 (Valor = 3) - RETOMADA E CONCLUÍDA

A Pilha
(Stack)
na saída

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



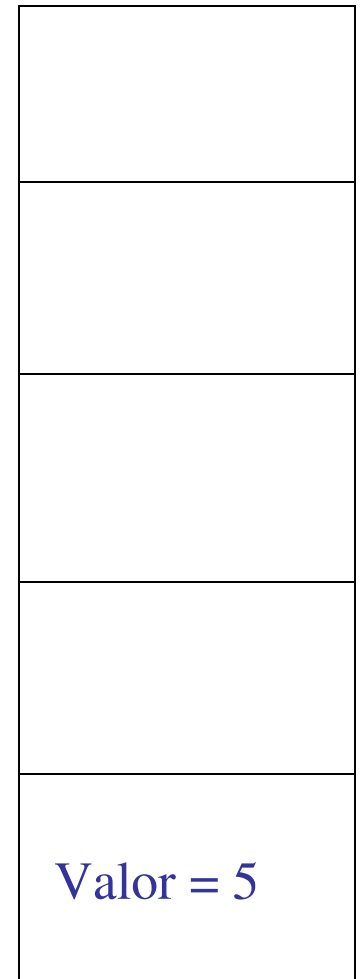
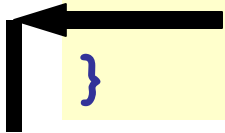
FATORIAL = 6 (3 * 2)



Chamada 2 (Valor = 4) - RETOMADA E CONCLUÍDA

A Pilha
(Stack)
na saída

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

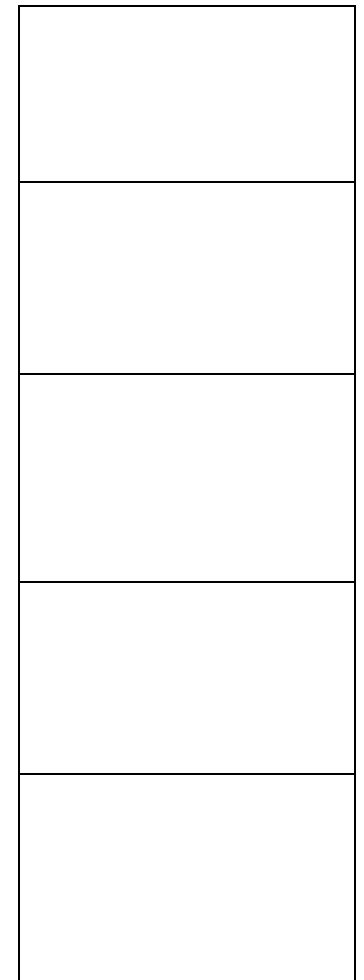
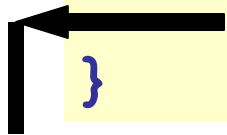


FATORIAL = 24 (4 * 6)

Chamada 1 (Valor = 5) - RETOMADA E CONCLUÍDA

A Pilha
(Stack)
na saída

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



FATORIAL = 120 (5 * 24)

EXEMPLOS

O que muda se for invertida a ordem dos dois comandos em destaque?

```
void nome( int i, int n)
{
    if (i < n)
    {
        nome( i+2,  n );
        printf("\n%d\n", i );
    }
    else
        printf("Primeira execucao concluida %d\n", i );
}
```



```

#include <stdio.h>
#include <stdlib.h>
void nome( int , int);
int main ( )
{
    int i_main, n_main;
    system("color 70");
    i_main = 2;
    n_main = 8;
    printf("\nPrimeiro chama nome, depois apresenta i\n\n");
    nome(i_main,n_main);
    printf("\n\n");
    system("pause");
    return 0;
}
void nome( int i, int n)
{
    if (i < n)
    {
        nome( i+2,  n );
        printf("\n%d\n", i );
    }
    else
        printf("\nPrimeira execucao concluida: %d\n", i );
}

```

Versão 1:
Primeiro chama nome
depois apresenta i

C:\backupcida\LinguagemCPagina20081\RECURSIVIDADE

Primeiro chama nome, depois apresenta i

Primeira execucao concluida: 8

6

4

2

Pressione qualquer tecla para continuar. . .

i é apresentado na fase do desempilhamento,
por isso os valores aparecem em ordem decrescente.

```

#include <stdio.h>
#include <stdlib.h>
void nome( int , int);
int main ( )
{
    int i_main, n_main;
    system("color 70");
    i_main = 2;
    n_main = 8;
    printf("\nPrimeiro apresenta i, depois chama nome\n");
    nome(i_main,n_main);
    printf("\n\n");
    system("pause");
    return 0;
}
void nome( int i, int n)
{
    if (i < n)
    {
        printf("\n%d\n", i );
        nome( i+2,  n );
    }
    else
        printf("\nPrimeira execucao concluida: %d\n", i );
}

```

Versão 2:
Primeiro apresenta i
depois chama nome

C:\backupcida\LinguagemCPagina20081\RECURSIVIDADE\Re

Primeiro apresenta i, depois chama nome

2

4

6

Primeira execucao concluida: 8

Pressione qualquer tecla para continuar. . . _

i é apresentado na fase do empilhamento,
por isso os valores aparecem em ordem crescente.

//Qual o resultado da execução recursiva de funcx?

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

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

```
void funcx( int , int *);
```

```
int main ( )
```

```
{
```

```
    int val1;
```

```
    int val2;
```

```
    system("color 70");
```

```
    val1 = 4;
```

```
    val2 = 12;
```

```
    funcx(val1, &val2);
```

```
    printf("\nval1 = %d val2 = %d\n", val1, val2);
```

```
    printf("\n\n");
```

```
    system("pause");
```

```
    return 0;
```

```
}
```

```
void funcx( int valor1, int *valor2)
```

```
{
```

```
    valor1++;
```

```
    *(valor2) = *(valor2) - 1;
```

```
    if (valor1 < *(valor2))
```

```
        funcx(valor1, valor2);
```

```
    printf("\nvalor1 = %d valor2 = %d\n", valor1, *(valor2));
```

```
}
```

```
C:\backupcida\LinguagemCPagina20081\RECURSIVIDADE\
```

```
valor1 = 8 valor2 = 8
```

```
valor1 = 7 valor2 = 8
```

```
valor1 = 6 valor2 = 8
```

```
valor1 = 5 valor2 = 8
```

```
val1 = 4 val2 = 8
```

```
Pressione qualquer tecla para continuar. . . _
```

Parâmetros por endereço não empilham.
Com exceção da última linha (val1...) todas as demais linhas são escritas na fase de desempilhamento, quando valor2 já deixou de ser alterado. Os valores de valor1 que estão sendo apresentados estão sendo retirados da pilha.

Prós e contras da Recursividade

PRÓS

- Código mais compacto.
- Especialmente conveniente para estruturas de dados definidas recursivamente tais como árvores.
- Eventualmente código pode ser mais fácil de entender.

Prós e contras da Recursividade

CONTRAS

- Em geral não oferece economia de memória. Os valores locais sendo processados têm que ser empilhados, o quê consome espaço.
- Uma solução recursiva nem sempre será mais rápida que uma correspondente iterativa, por exemplo.