



Estrutura de Dados

```
int factorialRecursivoCauda(int n, int resultado)
{
    if(n > 1) { //caso recursivo - chama nova
        // função com o mesmo nome
        // só muda o valor da variável resultado
        return factorialRecursivoCauda(n-1, n * resultado);
    } else{ //caso base - finaliza a recursão
        return resultado;
    }
}
```

RECURSIVIDADE

APRESENTAÇÃO

O lá aluno(a), você já conhece funções, já sabe que um código pode, e deve, ser modularizado para ser mais ágil e aproveitar melhor seu conteúdo.

Mas você já se perguntou se é possível uma função chamar ela mesma? E se for, em que situação isso poderia ser feito? Pois já antecipo que sim, e digo que isso se chama recursividade.

Neste módulo falaremos sobre esse assunto. Vamos começar?

OBJETIVOS DE APRENDIZAGEM

Ao final desse módulo você deverá ser capaz de:

- Compreender o que é recursividade;
- Diferenciar função recursiva de iterativa;
- Aplicar técnicas para criar algoritmos recursivos.

RECURSIVIDADE

Introdução

Primeiro vamos compreender qual a definição de recursividade.

Um algoritmo é considerado recursivo quando sua execução chama a si mesmo, ou chama uma sequência de outros, onde um deles volta a chamar o primeiro. E esse é exatamente o conceito de recursividade, na literatura também é conhecido como recursão ou recorrência.

Pois bem, mas o que significa tudo isso? Códigos recursivos possuem, entre suas linhas, chamadas para uma execução dele mesmo, ou seja, re-executa a mesma função, mas atenção, essa re-execução é feita com parâmetros e em escopo diferente.

Calma, agora explicarei isso com mais detalhes...



A ideia principal é dividir um problema complexo em problemas menores, e cada pequena solução será agregada para a solução completa do original. Essa estratégia faz parte de um tipo de algoritmo conhecido como: DIVIDIR PARA CONQUISTAR.

Primeiro entenderemos como um código deste é escrito, depois aplicaremos um exemplo para que você possa compreender melhor.



Montagem do código

IMPORTANTE

A primeira regra, importante para criar um algoritmo recursivo, é saber que ele não possui estrutura de repetição (*for*, *do* ou *while*), sua repetição é feita através das chamadas de si mesmo.



Se você pensar no conceito, “*um algoritmo que chama a si mesmo*” pode chegar a conclusão que esta chamada será feita infinitamente. Na verdade, para que isto não aconteça, teremos a segunda regra, onde o caso de parada será essencial. Seu código sempre deverá ter uma opção onde deixará de chamar a si mesmo, gerando a parada, também conhecido como caso base, ou âncora.

Sua solução deve ser montada utilizando o comando de decisão *if* e pensando nas duas partes importantes. Vamos relembrar:

1ª PARTE: CASO RECURSIVO

Neste item você deverá re-executar o algoritmo com uma porção menor do que a proposta original. Já pensando na condução para a proposta final, ou seja, em algum momento este item não será mais executado, passando a chamar o caso base.

2ª PARTE: CONHECIDO COMO ÂNCORA, CASO BASE OU CASO DE PARADA

Neste item você deverá incluir a proposta final do algoritmo, ou seja, a situação em que ele não chamará a si mesmo, que será seu «ponto de parada».



Aplicação de um caso recursivo

Para que você compreenda melhor vamos utilizar um clássico da recursividade que é o cálculo do fatorial de um número.

Só para relembrarmos, o fatorial de um número (n), que pode ser representado por $n!$, corresponde ao cálculo dos produtos de todos os números inteiros e positivos, menores ou iguais a n .

Sendo que:

Fatorial de 0 é igual a 1.

Portanto,

$$\text{fatorial}(n) = \begin{cases} 1, & n = 0 \\ n * \text{fatorial}(n - 1), & n > 0 \end{cases}$$

REFLITA

Vamos aplicar em um exemplo prático para que você compreenda melhor.

Imagine que calcularemos o fatorial de 5, portanto $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

Dizemos que o fatorial de 5 é 120.



Sabendo disto, vamos pensar na solução desta problema:

$$\text{fatorial}(5) = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$\text{fatorial}(4) = 4 \times 3 \times 2 \times 1 = 24$$

$$\text{fatorial}(3) = 3 \times 2 \times 1 = 6$$

$$\text{fatorial}(2) = 2 \times 1 = 2$$

$$\text{fatorial}(1) = 1 \text{ (olha o nosso caso base!!)}$$

Observe que, ao analisarmos os produtos percebemos que:

$$\text{fatorial}(5) = 5 \times \text{fatorial}(4)$$

$$\text{fatorial}(4) = 4 \times \text{fatorial}(3)$$

$$\text{fatorial}(3) = 3 \times \text{fatorial}(2)$$

$$\text{fatorial}(2) = 2 \times \text{fatorial}(1)$$

$$\text{fatorial}(1) = 1$$

Antes de conhecer o algoritmo recursivo, só fazíamos nosso código de forma iterativa, ou seja, utilizando as estruturas de repetição disponíveis, portanto, vamos criar um código iterativo para calcularmos o fatorial e depois criaremos um código recursivo, assim será possível avaliar a diferença entre eles.



Algoritmo factorial - iterativo

Para calcularmos iterativamente o factorial, utilizaremos uma estrutura de repetição. Essa é a forma clássica, sem recursão.

```
1 int factorialIterativo(int n) {
2     int fat = 1;
3     for (; n > 1; n--) {
4         fat *= n; //fat *= fat * n;
5     }
6     return fat;
7 }
```

OBSERVAÇÃO

Observe que a repetição iniciou no número solicitado (n), foi decrementando a variável n , até que ela alcance o valor 2, afinal, qualquer número multiplicado por 1 é igual a ele mesmo, não é? Por isso, a multiplicação por 1 foi dispensada.



Podemos concluir que a parada do loop foi qualquer número ≤ 1 , caso contrário, haverá repetição. Veja a simulação da execução:

	Execução 01	Execução 02	Execução 03	Execução 04
Valor de n	5	4	3	2
Cálculo efetuado	$5 * 1$	$4 * 5$	$3 * 20$	$2 * 60$
Valor de fat	5	20	60	120

ATENÇÃO

Uma informação importante é que, ao executar esse código, tudo será realizado na mesma instância, ou seja, no mesmo espaço de memória, afinal, é a mesma função executando um *loop*.



Agora “transformaremos” esse código iterativo em um código recursivo.



Algoritmo factorial - recursivo

Para calcularmos recursivamente o factorial, utilizaremos as partes necessárias, onde:

Se ($n > 1$) então retorne número multiplicado pelo próximo factorial caso contrário, retorne 1.

```
1 int factorialRecursoivo(int n){  
2     if(n > 1) { //caso recursivo - chama nova instância da função  
3         return n * factorialRecursoivo(n-1);  
4     }else{ //caso base - finaliza a repetição  
5         return 1;  
6     }  
7 }
```

Instância executada	Instância 01	Instância 02	Instância 03	Instância 04	Instância 05
Valor de n	5	4	3	2	1
Retorno (return)	$5 * \text{fatorial}(5-1)$	$4 * \text{fatorial}(4-1)$	$3 * \text{fatorial}(3-1)$	$2 * \text{fatorial}(2-1)$	1 (caso base)

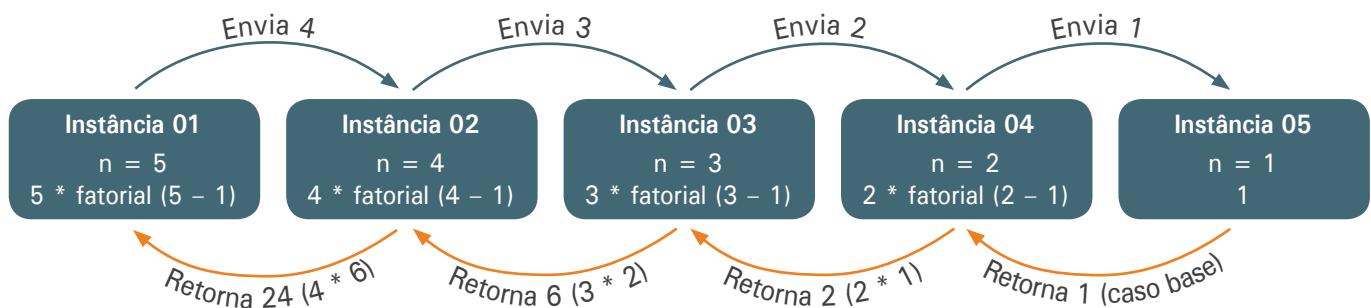
IMPORTANTE

Importante lembrar que, cada instância representa uma área de memória diferente, portanto, para esta execução utilizamos 5 áreas de memórias.



Observe que cada instância esperará a resposta da instância que abriu. As respostas começarão a ser retornadas a partir do caso base (última instância), todas retornarão até que a primeira obtenha o valor final do cálculo.

Vamos representar melhor:



Veja que, a medida em que as instâncias vão retornando, os cálculos serão executados, afinal, haverá resposta para cada chamada de função.



POR QUE UTILIZAR RECURSIVIDADE?

Uma dúvida muito comum é quanto a utilização da recursividade.

Existem vantagens e desvantagens para aplicação desta técnica.

RECURSIVIDADE	ITERAÇÃO
Utiliza estrutura de seleção	Utiliza estrutura de repetição
funções repetidas	repetições via loop
termina no caso base	termina na condição do loop
lento	rápido
simples e fácil manutenção	complicada e difícil manutenção
utiliza muita memória	utiliza muito processamento

Uma das maiores desvantagens da técnica é o uso excessivo de memória, podendo gerar estouro de pilha (*stack overflow*). Mas saiba que existem linguagens que são funcionais, ou seja, são executadas a partir de funções. Para estas, não existem estruturas de repetição disponíveis, apenas recursão.



Recursividade de cauda

Sabendo que a recursividade tem como sua principal desvantagem o uso de memória, em algumas situações, mesmo com uma implementação aparentemente simples, este uso torna-se excessivo, causando um estouro de pilha, que significa que a memória não “suportou” tantas chamadas recursivas.

Neste caso, a aplicação ideal seria a recursividade de cauda, que minimiza o problema de estouro de pilha.

```
1 int factorialRecursoCauda(int n, int resultado){  
2     if(n > 1) { //caso recursivo - chama nova instância da função  
3         return factorialRecursoCauda(n-1, n * resultado);  
4     }else{ //caso base - finaliza a repetição  
5         return resultado;  
6     }  
7 }
```

Uma regra para escrever este tipo de recursividade é que a última coisa a ser feita é a chamada recursiva, facilitando a utilização da memória durante o empilhamento. O código não fica “pendente” esperando um resultado ser retornado, tudo acontece na próxima instância.

Cada chamada recursiva carrega consigo um resultado parcial, que será agregado ao resultado final, e este será a informação enviada para a função que efetuou a primeira chamada, não sendo necessário efetuar processamento ao desempilhar todas as chamadas anteriores.

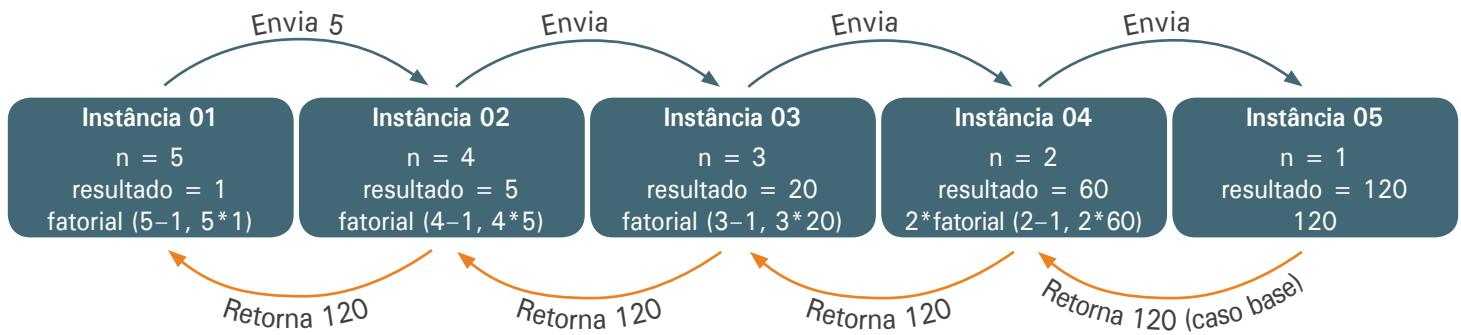
Instância executada	Instância 01	Instância 02	Instância 03	Instância 04	Instância 05
Valor de n	5	4	3	2	1
Valor de resultado	1	5	20	60	120
Retorno (return)	factorial(5-1, 5 * 1)	factorial(4-1, 4 * 5)	factorial(3-1, 3 * 20)	factorial(2-1, 2 * 60)	120 (caso base)



A chamada ideal para este código seria:

```
fatorialRecursivoCauda(n, 1);
```

Sendo n um número inteiro, positivo ou nulo e o número 1 seria passado como inicialização da variável resultado.



Empilhar funções na Stack

Todas as funções são colocadas em uma área de memória conhecida como *Stack* (pilha), e neste caso, serão retiradas à medida em que forem finalizadas, sempre começando das que foram colocadas mais recentes para as mais antigas.

▼ Call Stack		
#	Function	File:Line
0	fatorialRecursivo	main.c:21
1	fatorialRecursivo	main.c:23
2	fatorialRecursivo	main.c:23
3	fatorialRecursivo	main.c:23
4	fatorialRecursivo	main.c:23
5	main	main.c:14

Na figura podemos perceber o funcionamento da *Stack*, e, neste caso, a função *main* só poderá ser acessada após finalizar todas as cinco instâncias abertas da função *fatorialRecursivo*, e sempre finalizando as que foram colocadas por último até alcançar as mais antigas e, por fim, a função *main*, que foi a primeira a ser colocada na *Stack*.



Síntese

Finalizamos o conteúdo recursividade.

Você descobriu que um código recursivo chama a si mesmo, abrindo novas instâncias da mesma função, porém com parâmetros diferentes.

Percebeu também que o intuito é que cada instância resolva parte do problema principal, e, assim, cada pequeno resultado será agregado ao resultado final.

É importante se lembrar que códigos recursivos não possuem estruturas de repetição e precisam ser montados com estrutura de decisão. Para isto considere que, em seu código sempre existirá um caso base, que finaliza a repetição e um caso recursivo, que chama uma nova instância para execução.

Um dos maiores problemas gerados pela recursividade é o estouro de pilha, por isso, uma solução viável é criar seu código como recursividade de cauda, assim, um resultado parcial será enviado de instância em instância até chegar ao caso base com o resultado final.

Por fim, você descobriu também que quem controla as áreas de memórias utilizadas para execução de funções, recursivas ou não, é a *Stack*, também conhecida como pilha, e seu funcionamento vai das funções mais atuais para as mais antigas.

Até mais,

Referências

BACKES, André. **Linguagem C: completa e descomplicada**. 2. ed. Rio de Janeiro: Elsevier, 2019.

DAMAS, Luis. **Linguagem C**. 10. ed. Rio de Janeiro: Ltc, 2016.

MIZRAHI, Victorine Viviane. **Treinamento em linguagem C**. São Paulo: Pearson Prentice Hall, 2008. 407 p.

REESE, Richard. **Understanding and using C pointers**. Sebastopol: O'Reilly, 2008. 407 p.