Funções II

INF0446 — Introdução à Computação

Prof. Me. Raphael Guedes

raphaelguedes@ufg.br

2024

INF
INSTITUTO DE INFORMÁTICA

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

Sumário

Parâmetros da função

- Devem ser declarados individualmente com o respectivo tipo

```
01  //Declaração CORRETA de parâmetros
02  int soma(int x, int y){
03        return x + y;
04  }
05
06  //Declaração ERRADA de parâmetros
07  int soma(int x, y){
08        return x + y;
09  }
```

fonte: Backes (2018)

Protótipos de funções

Sem protótipo: a definição deve existir **antes do uso**.

Exemplo: função declarada *antes* da cláusula main.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  int Square (int a){
05    return (a*a);
06  }
07
08  int main(){
09    int n1,n2;
10    printf("Entre com um numero: ");
11    scanf("%d", &n1);
12    n2 = Square(n1);
13    printf("O seu quadrado vale: %d\n", n2);
14    system("pause");
15    return 0;
16  }
```

fonte: Backes (2018)

Protótipos de funções

Com protótipo: a definição pode existir **depois do uso**.

Exemplo: função declarada *depois* da cláusula main.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  //protótipo da função
04  int Square (int a);
05
06  int main(){
07    int n1,n2;
08    printf("Entre com um numero: ");
09    scanf("%d", &n1);
10    n2 = Square(n1);
11    printf("O seu quadrado vale: %d\n", n2);
12    system("pause");
13    return 0;
14  }
15
16  int Square (int a){
17    return (a*a);
18  }
```

fonte: Backes (2018)

Parâmetros vs Argumentos

- parâmetros: os campos da **definição** da função
- argumentos: os campos da **chamada** da função

Exemplo: função declarada *antes* da cláusula main.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  int Square (int a){
05    return (a*a);
06  }
07
08  int main(){
09    int n1,n2;
10    printf("Entre com um numero: ");
11    scanf("%d", &n1);
12    n2 = Square(n1);
13    printf("O seu quadrado vale: %d\n", n2);
14    system("pause");
15    return 0;
16  }
```

"a" é parâmetro

"n1" é argumento

fonte: Backes (2018)

Parâmetros da função

- Funções podem ser declaradas sem parâmetros
  - `tipo minhaFuncao()`
  - `tipo minhaFuncao(void)`
- No primeiro caso, qualquer parâmetro passado é ignorado
- No segundo caso, nenhum parâmetro é aceito.

Sem void

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  void imprime(){
05    printf("Teste de funcao\n");
06  }
07
08  int main(){
09    imprime();
10    imprime(5);
11    imprime(5,'a');
12
13    system("pause");
14    return 0;
15  }
```

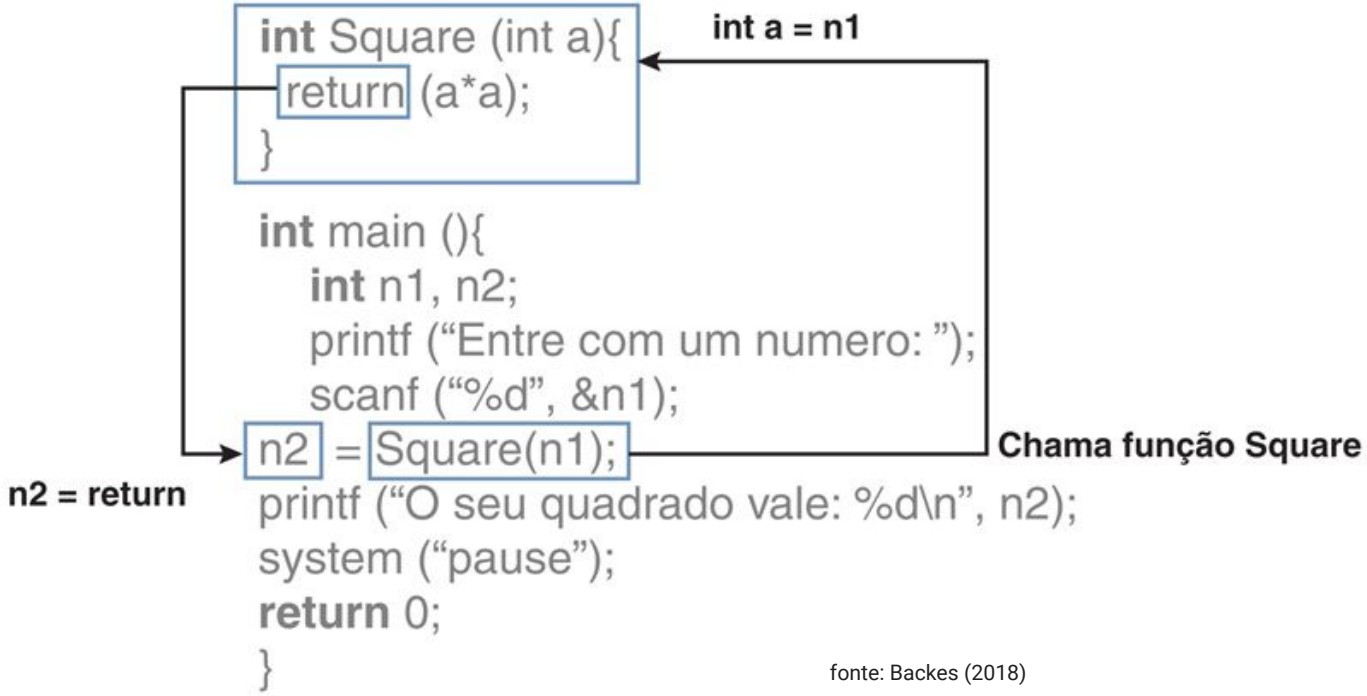
Com void

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

void imprime(void){
  printf("Teste de funcao\n");
}

int main(){
  imprime();
  imprime(5);//ERRO
  imprime(5,'a');//ERRO

  system("pause");
  return 0;
}
```

fonte: Backes (2018)

Fluxo de funções

- O programa principal é executado até encontrar a função.
- A função é executada e o fluxo retorna para o programa principal



fonte: Backes (2018)

Passagem de argumentos: por valor

- Os valores dos argumentos são copiados para os parâmetros.
  - Processo idêntico a uma atribuição.
- Mesmo que tenham nomes iguais, as variáveis da função são locais ao seu escopo, ou seja, não altera o que foi passado.

```c
int soma(int a, int b){
    int sum = a + b;
    return sum;
}
int main(){

    int a, b, sum;

    a = 5;
    b = 10;

    sum = soma(a, b);
    printf("soma: %d\n", sum);

    return 0;
}
```

```
a != a
b != b
sum != sum
```

mesmo nome, mas posições de memória diferentes

Passagem de argumentos: por referência

- Poréééém, é possível modificar as variáveis passadas sem fazer cópia de valor.
- Usamos o endereço de memória da dita-cuja.

Passagem de argumentos: por referência

- Os parâmetros da função usam um "*" antes do nome de cada elemento para permitir a passagem dos endereços das variáveis, quando estas são enviadas para a função.
  - Traduzindo: Na passagem por referência, os parâmetros são ponteiros!

Exemplo

```
tipo nome (tipo *parâmetro1, ..., tipo *parâmetroN) {
      comandos;
}
```

fonte: Coelho (2014)

Passagem de argumentos: por referência

- A partir de agora, o que for modificado dentro da função também é modificado fora dela.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  void soma _ mais _ um(int *n){
05    *n = *n + 1;
06     printf("Dentro da funcao: x = %d\n",*n);
07  }
08
09  int main(){
10    int x = 5;
11    printf("Antes da funcao: x = %d\n",x);
12    soma _ mais _ um(&x);
13    printf("Depois da funcao: x = %d\n",x);
14    system("pause");
15    return 0;
16  }
```

```
Saída   Antes da funcao: x = 5
        Dentro da funcao: x = 6
        Depois da funcao: x = 6
```

fonte: Backes (2018)

por valor vs por referência

Por valor

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  void Troca(int a,int b){
05    int temp;
06    temp = a;
07    a = b;
08    b = temp;
09    printf("Dentro: %d e %d\n",a,b);
10  }
11
12  int main(){
13    int x = 2;
14    int y = 3;
15    printf("Antes: %d e %d\n",x,y);
16    Troca(x,y);
17    printf("Depois: %d e %d\n",x,y);
18    system("pause");
19    return 0;
20  }
```

Saída

```
Antes: 2 e 3
Dentro: 3 e 2
Depois: 2 e 3
```

Por referência

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

void Troca(int*a,int*b){
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
  printf("Dentro: %d e %d\n",*a,*b);
}

int main(){
  int x = 2;
  int y = 3;
  printf("Antes: %d e %d\n",x,y);
  Troca(&x,&y);
  printf("Depois: %d e %d\n",x,y);
  system("pause");
  return 0;
}
```

Saída

```
Antes: 2 e 3
Dentro: 3 e 2
Depois: 3 e 2
```

fonte: Backes (2018)

Vetores em Funções

- Vetores e matrizes são passados para funções por referência.
  - Passa-se o elemento e seu tamanho.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  void imprime (int *n, int m){
05    int i;
06    for (i=0; i<m;i++)
07        printf("%d \n", n[i]);
08  }
09
10  int main(){
11    int v[5] = {1,2,3,4,5};
12    imprime(v,5);
13    system("pause");
14    return 0;
15  }
```

fonte: Backes (2018)

Matrizes em Funções

- Para as matrizes, apenas a primeira dimensão pode ser omitida

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  void imprime_matriz(int m[][2], int n){
05    int i,j;
06    for (i=0; i<n;i++)
07       for (j=0; j< 2;j++)
08            printf("%d \n", m[i][j]);
09  }
10
11  int main(){
12    int mat[3][2] = {{1,2},{3,4},{5,6}};
13    imprime_matriz(mat,3);
14    system("pause");
15    return 0;
16  }
```

fonte: Backes (2018)

Estruturas em Funções

- As estruturas podem ser passadas:
  - Campo a campo
  - Toda a estrutura
- Nos dois casos, a passagem pode ser por valor ou referência.
- Na passagem por referência, usa-se o operador seta (->)

Estruturas por valor

Exemplo: estrutura passada por valor

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  struct ponto {
04    int x, y;
05  };
06  void imprime(struct ponto p){
07    printf("x = %d\n",p.x);
08    printf("y = %d\n",p.y);
09  }
10  int main(){
11    struct ponto p1 = {10,20};
12    imprime(p1);
13
14    system("pause");
15    return 0;
16  }
```

fonte: Backes (2018)

Estruturas por valor

Exemplo: campo da estrutura passado por valor

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  struct ponto {
04    int x, y;
05  };
06  void imprime_valor(int n){
07    printf("Valor = %d\n",n);
08  }
09  int main(){
10    struct ponto p1 = {10,20};
11    imprime_valor(p1.x);
12    imprime_valor(p1.y);
13    system("pause");
14    return 0;
15  }
```

fonte: Backes (2018)

Estruturas por referência

Exemplo: estrutura passada por referência

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  struct ponto {
04    int x, y;
05  };
06  void atribui(struct ponto *p){
07    (*p).x = 10;
08    (*p).y = 20;
09  }
10  int main(){
11    struct ponto p1;
12    atribui(&p1);
13    printf("x = %d\n",p1.x);
14    printf("y = %d\n",p1.y);
15    system("pause");
16    return 0;
17  }
```

	Sem operador seta	Com operador seta
01	`struct ponto {`	`struct ponto {`
02	`  int x, y;`	`  int x, y;`
03	`};`	`};`
04		
05	`void atribui(struct ponto *p){`	`void atribui(struct ponto *p){`
06	`  (*p).x = 10;`	`  p->x = 10;`
07	`  (*p).y = 20;`	`  p->y = 20;`
08	`}`	`}`

fonte: Backes (2018)

Estruturas por referência

Exemplo: campo da estrutura passada por referênca

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  struct ponto {
04    int x, y;
05  };
06  void soma_imprime_valor(int *n){
07    *n = *n + 1;
08    printf("Valor = %d\n",*n);
09  }
10  int main(){
11    struct ponto p1 = {10,20};
12    soma_imprime_valor(&p1.x);
13    soma_imprime_valor(&p1.y);
14    system("pause");
15    return 0;
16  }
```

fonte: Backes (2018)

Retorno de funções: "simples"

- Uma função deve realizar uma tarefa específica e bem-definida
- Se sua função faz várias coisas, algo está errado.
- *Função não é inspetor bugiganga*

- Os retornos podem ser:
  - básicos: **`chat, int, float, double, void, ponteiros`**
  - definidos: **`função`**, **`struct, array`** (indiretamente)

- Tudo depois do retorno é ignorado

```
01  int maior(int x, int y){
02    if(x > y)
03        return x;
04    else
05        return y;
06    printf("Fim da funcao\n");
07  }
```

fonte: Backes (2018)

Inspector Gadget, 1982

Retorno de funções: "simples"

- Se a função tem retorno, a variável precisa ser do mesmo tipo do retorno

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  int Square (int a){
05    return (a*a);
06  }
07
08  int main(){
09    int n1,n2;
10    printf("Entre com um numero: ");
11    scanf("%d", &n1);
12    n2 = Square(n1);
13    printf("O seu quadrado vale: %d\n", n2);
14    system("pause");
15    return 0;
16  }
```

Um retorno inteiro atribuído a um valor inteiro.

fonte: Backes (2018)

Retorno de funções: "simples"

- Uma função, no entanto, pode possuir vários retornos, baseados em condições.
- Retornos demais podem dificultar o entendimento do código
  - Quando saber se preciso de um ou mais retorno?
    - Problema a ser resolvido
    - Bom-senso do programador

```
01  int maior(int x, int y){
02    if(x > y)
03        return x;
04    else
05        return y;
06  }
```

```
01  int maior(int x, int y){
02    int z;
03    if(x > y)
04        z = x;
05    else
06        z = y;
07    return z;
08  }
```

fonte: Backes (2018)

Retorno de funções: múltiplo c/ ponteiros

- É possível retornar mais de um valor usando ponteiros

```
int calculos(int a, int b, int *prod){
    int sum = a + b;
    *prod = a*b;
    return sum;
}
int main(){

    int a, b, sum, prod;

    a = 5;
    b = 10;

    sum = calculos(a, b, &prod);
    printf("soma: %d\n", sum);
    printf("produto: %d\n", prod);


    return 0;
}
```

Retorno de funções: múltiplo c/ struct

- E também usando estruturas

```
typedef struct resultado {
    int soma;
    int produto;
} Resultado;

Resultado calculo(int a, int b) {
    Resultado result;
    result.soma = a + b;
    result.produto = a * b;
    return result;
}

int main() {
    Resultado res;

    res = calculo(5, 10);
    printf("soma: %d\n", res.soma);
    printf("produto: %d\n", res.produto);

    return 0;
}
```

Funções que retornam ponteiros

- Funções que retornam ponteiros precisam ter um tipo associado.
  - O compilador precisa saber para que tipo de dado o ponteiro aponta.
- A função precisa indicar que retorna um ponteiro

```
char *getString() {  // retorna a string
    return "Hello, world!";
}


int main() {
    char *greeting = getString();
    printf("%s\n", greeting);

    return 0;
}
```

```c
int *getMaximo(int vet[], int tam) { // Retorna o maximo via ponteiro
    int *max = &vet[0];

    for (int i = 1; i < tam; i++) {
        if (vet[i] > *max) {
            max = &vet[i];
        }
    }
    return max;
}

int main() {

    int myVet[] = {50, 20, 80, 1};
    int *vetPtr = getMaximo(myVet, 4);
    printf("O maximo eh: %d\n", *vetPtr);
    return 0;
}
```

Recursividade

- Funções recursivas são funções que chamam a si mesmas durante a execução de programa.
- Baseada na ideia de dividir para conquistar.
- É preciso encontrar a lei de formação.
  - 4! = 4 * 3!
  - 3! = 3 * 2!
  - 2! = 2 * 1!
  - 1! = 1 * 0!
  - 0! = 1
  - n! = n * (n-1)!
- Precisa de uma condição de parada para executar a fase de “conquistar”

Recursividade

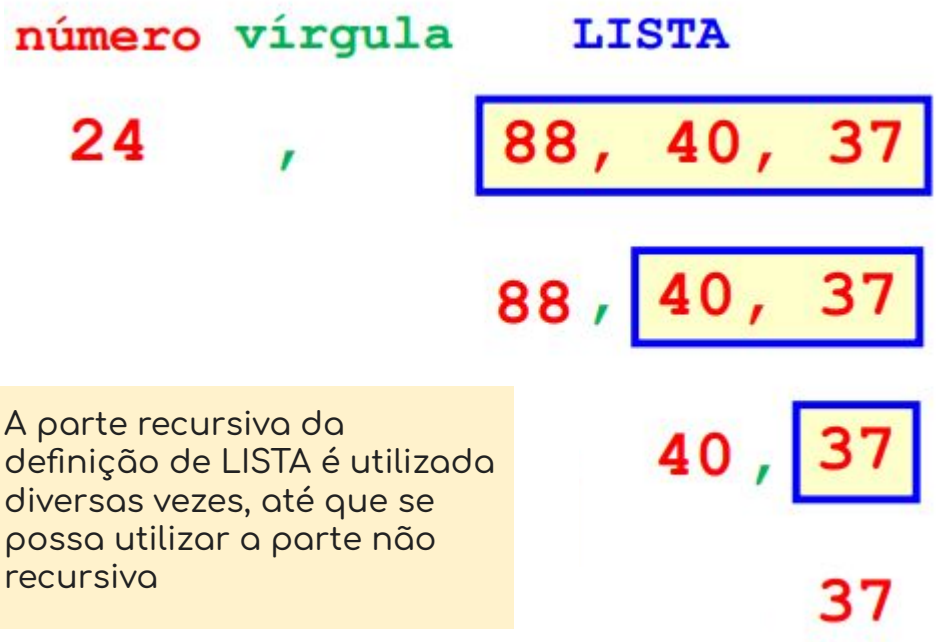
- Uma definição recursiva de um conceito consiste em utilizar o próprio conceito na definição
  - `fat(n) = n * fat(n-1)`
- Definições recursivas em linguagem natural não são, geralmente, muito úteis
- Contudo, em outras situações, uma definição recursiva pode ser a mais apropriada para explicar um conceito
- Recursão é uma técnica de programação que pode fornecer soluções elegantes para determinados problemas

Recursividade: Definição recursiva

- Considere a seguinte lista de números
  - **24, 88, 40, 37**
- Podemos definir uma lista de números como:
- Uma **LISTA** é um: **número**
  - ou um: **número vírgula LISTA**

- Uma LISTA é definida ou como **um único número**, ou como **um número seguido de uma vírgula seguida de uma LISTA**

Recursividade: Definição recursiva



Parte recursiva da definição da LISTA

A parte recursiva da definição de LISTA é utilizada diversas vezes, até que se possa utilizar a parte não recursiva

parte não recursiva

Caso Base e Caso Geral

- Toda definição recursiva deve ter uma parte recursiva e outra não recursiva
- Se não houver a parte não recursiva, o caminho recursivo nunca termina
  - Gera uma recursão infinita
  - Similar a um laço infinito
- Caso base é a parte não recursiva da definição, enquanto o caso geral é a parte recursiva

Programação Recursiva

- Uma função em C pode chamar a si mesma
  - Função recursiva
- O código de uma função recursiva **deve** tratar o caso base e o caso geral
- Cada chamada da função cria novos parâmetros e variáveis locais
  - Cria-se uma nova instância da função
- Como em qualquer chamada de função, assim que a função termina sua execução, o controle retorna para a função que a chamou (que neste caso, é outra instância da mesma função)

Programação Recursiva: passos

- Definir o problema em termos recursivos
  - Definir o caso geral
- Determinar o caso base
- Definir uma solução de modo que a cada chamada recursiva, podemos nos aproximar do caso base

Recursão vs Laços de Repetição

- Não é porque existe uma solução recursiva, que sempre deve ser usada
- Soluções iterativas (com laço) são geralmente mais eficientes
  - Soluções recursivas geralmente demandam mais poder de processamento e memória (*pois envolve a criação de mais variáveis e chamadas de função*)
- Porém, soluções com recursão podem ser mais elegantes e compreensíveis que soluções iterativas.
- Quando saber quando preciso usar recursão?
  - Problema a ser resolvido
  - Bom-senso do programador

Programação Recursiva

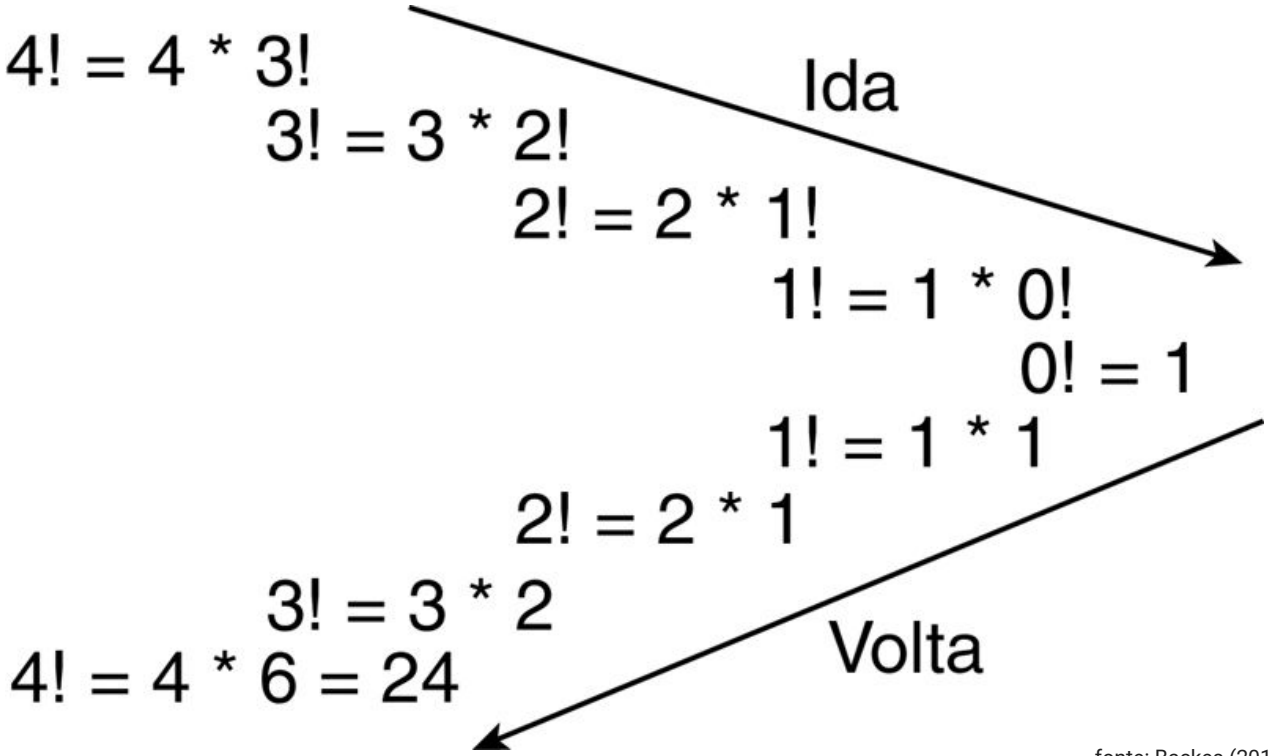
Exemplo: calculando o fatorial

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int fatorial (int n){
04    if(n == 0)
05        return 1;
06    else
07        return n*fatorial(n-1);
08  }
09  int main(){
10    int x;
11    x = fatorial(4);
12    printf("4! = %d\n",x);
13    system("pause");
14    return 0;
15  }
```

Caso base

Caso geral

Recursividade: fatorial



fonte: Backes (2018)

Recursividade: fatorial

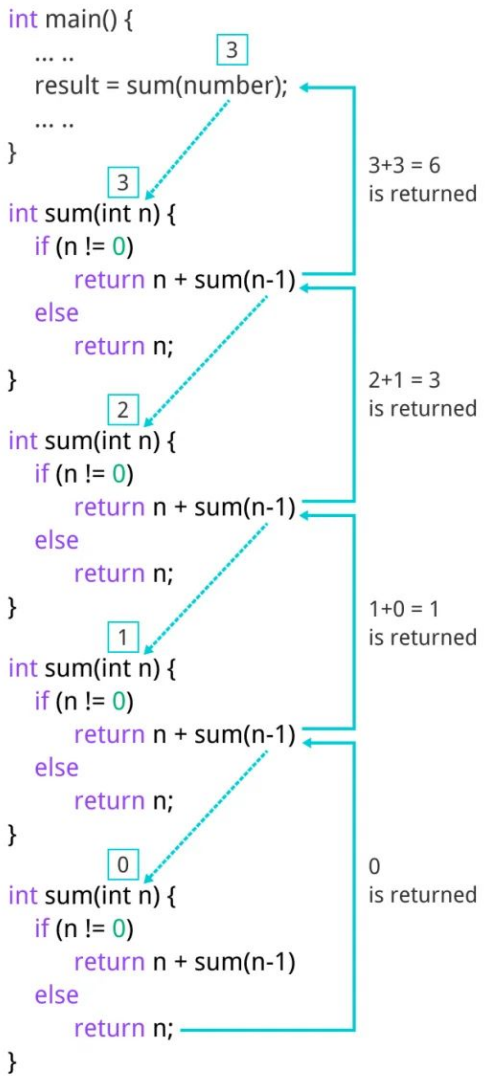
Exemplo: calculando o fatorial

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int fatorial (int n){
04    if(n == 0)
05        return 1;
06    else
07        return n*fatorial(n-1);
08  }
09  int main(){
10    int x;
11    x = fatorial(4);
12    printf("4! = %d\n",x);
13    system("pause");
14    return 0;
15  }
```

fonte: Backes (2018)

Soma recursiva

```
int main() {
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

    printf("sum = %d", result);
    return 0;
}

int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}
```



Referências

- COELHO, Hebert. **Funções - Parte II**. Goiânia: Instituto de Informática, 2014. 60 slides, color.
- BACKES, André. **Linguagem C - Completa e Descomplicada**. Disponível em: Grupo GEN, (2 edição). Grupo GEN, 2018.

Até mais!

raphaelguedes@ufg.br

INF
INSTITUTO DE INFORMÁTICA

UFG
UNIVERSIDADE FEDERAL DE GOIÁS