



UNIVERSIDADE
FEDERAL DO CEARÁ



CK0179 – Programação Computacional para Engenharia: Funções

Prof. Maurício Moreira Neto

Objetivo

- Aprender a criar funções e procedimentos na linguagem C
- Utilizar essas funções ao longo dos algoritmos

Função

- São blocos de códigos que podem ser nomeados e chamados de dentro de um programa
- Até o momento, já utilizamos diversas funções em nossos códigos
 - **printf()** : função que escreve na tela
 - **scanf()** : função que lê o teclado

Função

- Facilita a estruturação e reutilização do código
 - **Programação Estruturada:** programas grandes e complexos são construídos bloco a bloco (modularização)
 - **Reutilização:** utilizar funções evita a cópia desnecessária de trechos de códigos que realizam a mesma tarefa, fazendo diminuir o tamanho do programa e a ocorrência de erros

Ordem de Execução

Ao chamar uma função, o programa que a chamou é pausado até que a função termine a sua execução

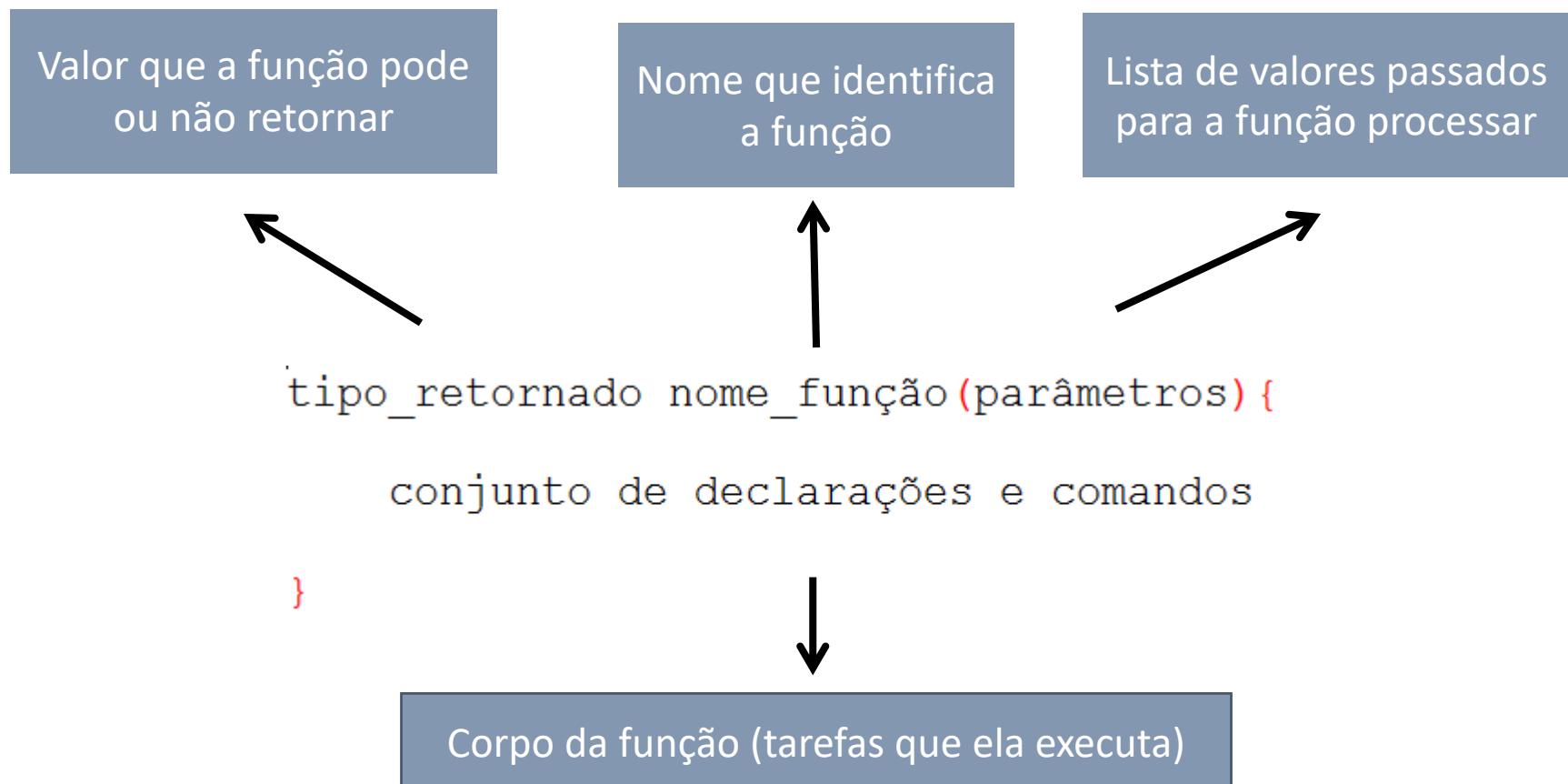
```
float a = v1 e float b = v2
float soma(float a, float b) {
    return a + b;
}

int main() {
    float v1, v2, resultado;
    printf("Primeiro numero: ");
    scanf("%f", &v1);
    printf("Segundo numero: ");
    scanf("%f", &v2);
    resultado = soma(v1, v2);
    printf("O resultado eh: %.2f", resultado);
}
```

Resultado = a + b

Estrutura de uma Função

- Forma geral de uma função



Corpo de uma Função

- O corpo de uma função é a sua parte principal
 - Formada pelos comandos que função deve executar
 - Processa parâmetros (se houver), realizar outras tarefas e gerar saídas (se necessário)
 - Similar a cláusula **main()**

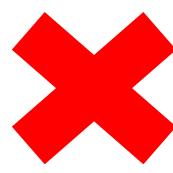
```
int main() {  
    // conjuntos de comandos  
    return 0;  
}
```

Corpo de uma Função

- De modo geral, evita-se fazer operações de leitura e escrita dentro de uma função
 - Uma função é construída com o intuito de realizar uma tarefa específica e bem-definida
 - As operações de entrada e saída de dados (**scanf()** e **printf()**) devem ser feitas em quem chamou a função (por exemplo, na **main()**).
 - Isso assegura que a função construída possa ser utilizada nas mais diversas aplicações, garantindo a sua generalidade.

Parâmetros de uma Função

- A declaração de parâmetros é uma lista de variáveis juntamente com seus tipos:
 - $Tipol\ nome_1, \ tipo_2\ nome_2, \dots, \ tipo_N\ nome_N$
 - Pode-se definir quantos parâmetros achar necessários



```
//Declaração CORRETA de parâmetros
int soma(int x, int y) {
    return x + y;
}

//Declaração ERRADA de parâmetros
int soma(int x, y) {
    return x + y;
}
```

Parâmetros de uma Função

- Os parâmetros são informações que a função recebe do programa principal (ou de quem o chamou)
 - Não é preciso fazer a leitura das variáveis dos parâmetros dentro da função

```

int x = 2;
int y = 3;

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

int main() {
    int z = soma(2, 3);

    return 0;
}
  
```

```

int soma(int x, int y) {
    scanf("%d", &x);
    scanf("%d", &y);

    return x + y;
}
  
```



Parâmetros de uma Função

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

- Podemos criar uma função que não recebe nenhum parâmetro de entrada
- Isso pode ser feito de duas formas:
 - Deixar a lista de parâmetros vazia
 - Colocar void entre os parênteses

Retorno de Função

- Uma função pode ou não retornar um valor
 - Se ela retornar um valor, alguém deverá receber este valor
 - Uma função que retorna nada é definida colocando-se o tipo **void** como valor retornado
- Podemos retornar qualquer valor válido em C
 - tipos pré-definidos: **int**, **char**, **float** e **double**
 - tipos definidos pelo usuário: **struct**

Comando Return

- O valor retornado por uma função pode ser dado pelo comando **return**
- Forma geral:
 - **return** *expressão ou valor;*
 - **return;**
 - Usada para terminar uma função que não retorna um determinado valor
- **Importante:** lembrar que o valor de retorno fornecida tem que ser compatível com o tipo de retorno declarado para a função

Comando Return

Função com o
retorno de valor

```
int soma(int x, int y) {  
    return x + y;  
}  
  
int main() {  
    int z = soma(2, 3);  
  
    return 0;  
}
```

Função sem o
retorno de valor

```
void imprime() {  
    printf("Teste\n");  
}  
  
int main() {  
    imprime();  
  
    return 0;  
}
```

Comando Return

- Uma função pode ter mais de uma declaração **return**
 - Quando o comando **return** é executado, a função termina imediatamente
 - Todos os comandos restantes são **ignoradas**

```
int maior(int x, int y) {
    if(x > y)
        return x;
    else
        return y;
    printf("Esse texto nao sera impresso\n");
}
```

Declaração de Funções

- Funções devem ser declaradas antes de serem utilizadas, ou seja, antes da cláusula **main**
 - Uma função criada pelo programador pode ser utilizada qualquer outra função, inclusive as que foram criadas

```
int quadrado(int a) {
    return a*a;
}

int main() {
    int n1, n2;
    printf("Entre com um numero: ");
    scanf("%d", &n1);

    n2 = quadrado(n1);

    printf("O seu quadrado vale: %d\n", n2);
    return 0;
}
```

Declaração de Funções

- Podemos definir apenas o protótipo da função antes da cláusula **main**
 - O protótipo apenas indica a existência da função
 - Assim, ela pode ser declarada após a cláusula **main()**

tipo_retornado nome_funcao (parametros);

Declaração de Funções

- Exemplo de protótipo

```
int quadrado(int a);

int main(){
    int n1,n2;
    printf("Entre com um numero: ");
    scanf("%d", &n1);

    n2 = quadrado(n1);

    printf("O seu quadrado vale: %d\n", n2);
    return 0;
}

int quadrado(int a) {
    return a*a;
}
```

Escopo

- Função também estão sujeitos ao escopo das variáveis
- O escopo é o conjunto de regras que determinam o uso e validade de variáveis nas diversas partes do programa
 - Variáveis Locais
 - Variáveis Globais
 - Parâmetros Formais

Escopo

- Variáveis locais são aqueles que só tem validade dentro do bloco no qual são declaradas
 - Um bloco começa quando abrimos uma chave e termina quando fechamos a chave
 - Ex: variáveis declaradas dentro da função

```
int factorial (int n) {
    if (n == 0)
        return 1;
    else{
        int i;
        int f = 1;
        for(i = 1; i <= n; i++)
            f = f * i;
        return f;
    }
}
```

Escopo

- Parâmetros formais são declaradas como sendo as entradas de uma função
 - Os parâmetros formais são variáveis locais da função
 - **Exemplo:** 'x' é um parâmetro formal

```
float quadrado(float x);
```

Escopo

- Variáveis globais são declaradas fora de todas as funções do programa
- Estas variáveis são conhecidas e podem ser alteradas por todas as funções do programa
 - Quando uma função tem uma variável local com o mesmo nome de um variável global, a função dará preferência a variável local
- **IMPORTANTE:** é sempre bom evitar variáveis globais

Passagem de Parâmetros

- Na linguagem C, os parâmetros de uma função são sempre passados por **valor**, ou seja, uma cópia do valor do parâmetro é feita e passada para a função
- Mesmo que esse valor mude dentro da função, nada acontece com o valor fora da função
- Mas podemos ter dois tipos de passagens de parâmetros
 - **Por valor**
 - **Por referência**

Passagem por Valor

```
int n = x;
void incrementa(int n) {
    n = n + 1;

    printf("Dentro da funcao: x = %d\n", n);
}

int main() {
    int x = 5;
    printf("Antes da funcao: x = %d\n", x);

    incrementa(x);

    printf("Depois da funcao: x = %d\n", x);
    return 0;
}

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

Passagem por Referência

- Quando se quer que o valor da variável mude dentro da função, usa-se passagem de parâmetros por ***referência***
- Neste tipo de chamada, não se passa a função o valor da variável, mas a sua ***referência*** (seu endereço na memória)

Passagem por Referência

- Utilizando o endereço da variável, qualquer alteração que a variável sofra dentro da função será refletida fora da função
- Ex: função **scanf()**

Passagem por Referência

- Ex: função **scanf()**
 - Sempre que desejamos ler algo do teclado, passamos para a função **scanf()** o nome da variável onde o dado será armazenado
 - Essa variável tem seu valor modificado dentro da função **scanf()**, e seu valor pode ser acessado no programa principal

```
int main() {
    int x = 5;
    printf("Antes do scanf: x = %d\n", x);
    printf("Digite um numero: ");
    scanf("%d", &x);
    printf("Depois do scanf: x = %d\n", x);

    return 0;
}
```

Passagem por Referência

- Para passar um parâmetro por referência, coloca-se um asterisco '*' na frente do nome do parâmetro na declaração da função:

```
//passagem de parâmetro por valor
void incrementa(int n);

//passagem de parâmetro por referência
void incrementa(int *n);
```

- Ao se chamar a função, é necessário agora utilizar o operador '&', assim como feito pela função **scanf()**

```
//passagem de parâmetro por valor
int x = 10;
incrementa(x);

//passagem de parâmetro por referência
int x = 10;
incrementa(&x);
```

Passagem por Referência

- No corpo da função, é necessário colocar um asterisco '*' sempre que se deseja acessar o conteúdo do parâmetro passado por referência

```
//passagem de parâmetro por valor
void incrementa(int n) {
    n = n + 1;
}
//passagem de parâmetro por referência
void incrementa(int *n) {
    *n = *n + 1;
}
```

Passagem por Referência

```
int *n = &x;  
void incrementa(int *n) {  
    *n = *n + 1;  
  
    printf("Dentro da funcao: x = %d\n", n);  
}  
  
int main() {  
    int x = 5;  
    printf("Antes da funcao: x = %d\n", x);  
  
    incrementa(&x);  
  
    printf("Depois da funcao: x = %d\n", x);  
    return 0;  
}
```

Saída:

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

Exercício Proposto

1. Crie uma função que troque o valor de dois números inteiros passados por referência

Dica: lembre-se da variável de auxílio

2. Crie uma função que faça a soma de dois valores inteiros passados por referência.

Resolução

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

void trocar(int *a, int *b){
    int aux;
    aux = *a;
    *a = *b;
    *b = aux;
    printf("Valor de A: %d e valor de B: %d", *a, *b);
}

int somar (int *a, int *b){
    return *a + *b;
}

int main()
{
    int valor_1, valor_2, res;
    printf("Valor de A: ");
    scanf("%d", &valor_1);
    printf("Valor de B: ");
    scanf("%d", &valor_2);
    trocar(&valor_1, &valor_2);
    res = somar(&valor_1, &valor_2);
    printf("\nO valor da soma eh: %d", res);
    return 0;
}
```

Array como Parâmetros

- Para utilizar arrays como parâmetros de funções alguns cuidados simples são necessários
- Arrays são sempre passados por referência para uma função:
 - A passagem de **array por referência** evita cópia desnecessária de grandes quantidades de dados para outras áreas de memória durante a chamada da função, o que afetaria o desempenho do programa.

Array como Parâmetros

- É necessário declarar um segundo parâmetro (em geral uma variável inteira) para passar a função o tamanho do array separadamente
 - Quando passamos um array por parâmetro, independente do seu tipo, o que é de fato passado é o endereço do primeiro elemento do array

Array como Parâmetros

- Na passagem de um array como parâmetro de uma função podemos declarar a função de diferentes maneiras, todas equivalentes:

```
void imprime(int *m, int n);
void imprime(int m[], int n);
void imprime(int m[5], int n);
```

Array como Parâmetros

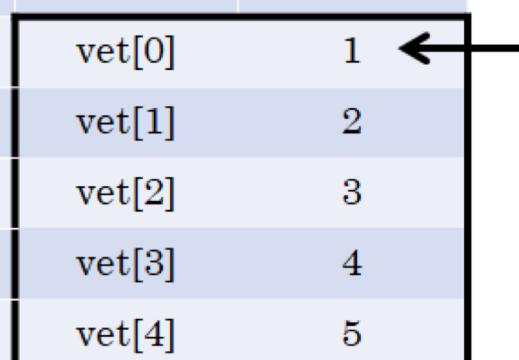
- **Exemplo:**
 - Função que imprime um array

```
void imprime(int *m, int n) {
    int i;
    for (i=0; i< n;i++)
        printf ("%d \n", m[i]);
}

int main () {
    int vet[5] = {1,2,3,4,5};
    imprime(vet,5);

    return 0;
}
```

Memória		
posição	variável	conteúdo
119		
120		
121	int vet[5]	123
122		
123	vet[0]	1
124	vet[1]	2
125	vet[2]	3
126	vet[3]	4
127	vet[4]	5
128		



Array como Parâmetros

- Foi visto que para arrays, não é necessário especificar o número de elementos para a função

```
void imprime (int*m, int n);
void imprime (int m[], int n);
```

- No entanto, para arrays com mais de uma dimensão, é necessário especificar o tamanho de todas as dimensões, exceto a primeiro

```
void imprime (int m[] [5], int n);
```

Array como Parâmetros

- Na passagem de um array para uma função, o compilador precisar saber o tamanho de cada elemento, não o número de elementos
- Uma matriz pode ser interpretada como um array de arrays
 - **int m[4][5]:** array de 4 elementos onde cada elemento é um array de 5 posições inteiros

Array como Parâmetros

- Logo, o compilador precisa saber o tamanho de cada elemento do array

```
int m[4][5]
```

```
void imprime (int m[] [5], int n);
```

- Na notação acima, informamos ao compilador que estamos passando um array, onde cada elemento dele é outro array de 5 posição inteiras

Array como Parâmetros

- Isso é necessário para que o programa saiba que o array possui mais de uma dimensão e mantenha a notação de um conjunto de colchetes por dimensão
- As notações abaixo funcionam para arrays com mais de uma dimensão. Mas o array é tratado como se tivesse apenas uma dimensão dentro da função

```
void imprime (int*m, int n);  
void imprime (int m[], int n);
```

Struct como Parâmetros

- Pode-se passar uma **struct** por parâmetro ou por referência
- Temos duas possibilidades:
 1. Passar por parâmetro toda a struct
 2. Passar por parâmetro apenas um campo específico da struct

Struct como Parâmetros

- Passar por parâmetro apenas um campo específico da **struct**
 - Valem as mesmas regras vistas até o momento
 - Cada campo da struct é como uma variável independente. Pode, portanto, ser passada individualmente por **valor** ou por **referência**

Struct como Parâmetros

- Passar por parâmetro toda a struct
- Passagem por valor
 - Valem as mesmas regras vistas anteriormente
 - A struct é tratada como uma variável qualquer e seu valor é copiado para dentro da função
- Passagem por referência
 - Valem as regras de uso do asterisco '*' e operador de endereço '&'
 - Deve-se acessar o conteúdo da struct para somente depois acessar os seus campos e modificá-los
 - Uma alternativa é usar o **operador seta '->'**

Struct como Parâmetros

Usando '*'

```
struct ponto {  
    int x, y;  
};  
  
void atribui(struct ponto *p){  
    (*p).x = 10;  
    (*p).y = 20;  
}  
  
struct ponto p1;  
  
atribui(&p1);
```

Usando '->'

```
struct ponto {  
    int x, y;  
};  
  
void atribui(struct ponto *p){  
    p->x = 10;  
    p->y = 20;  
}  
  
struct ponto p1;  
  
atribui(&p1);
```

Recursão

- Na linguagem C, uma função pode chamar outra função
 - A função **main()** pode chamar qualquer outra função, seja da biblioteca da linguagem (como a **printf()**) ou definida pelo programador (função **imprime()**)
- Uma função também pode chamar a si própria
 - Chama-se esse tipo de função de **função recursiva**

Recursão

- A recursão também é chamado de definição circular. Ocorre quando algo é definido em termos de si mesmo
- Um exemplo clássico de função que usa recursão é o cálculo do fatorial de um número:
 - $3! = 3 * 2!$
 - $4! = 4 * 3!$
 - $n! = n * (n - 1)!$

Recursão

$$0! = 1$$

$$1! = 1 * 0!$$

$$2! = 2 * 1!$$

$$3! = 3 * 2!$$

$$4! = 4 * 3!$$

$$\begin{aligned}
 4! &= 4 * 3! && \text{Ida} \\
 3! &= 3 * 2! && \\
 2! &= 2 * 1! && \\
 1! &= 1 * 0! && \\
 0! &= 1 && \\
 2! &= 2 * 1 && \\
 3! &= 3 * 2 && \\
 4! &= 4 * 6 = 24 && \text{Volta}
 \end{aligned}$$

$n! = n * (n - 1)!$: **fórmula geral**

$0! = 1$: **caso-base**

Recursão

Com Recursão

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

Sem Recursão

```
int factorial (int n){  
    if (n == 0)  
        return 1;  
    else{  
        int i;  
        int f = 1;  
        for(i = 1; i <= n; i++)  
            f = f * i;  
        return f;  
    }  
}
```

Recursão

- Em geral, formulações recursivas de algoritmos são frequentemente consideradas "mais enxutas" ou "mais elegantes" do que formulações iterativas.
- Porém, algoritmos recursivos tendem a necessitar de mais espaço do que algoritmos iterativos.

Recursão

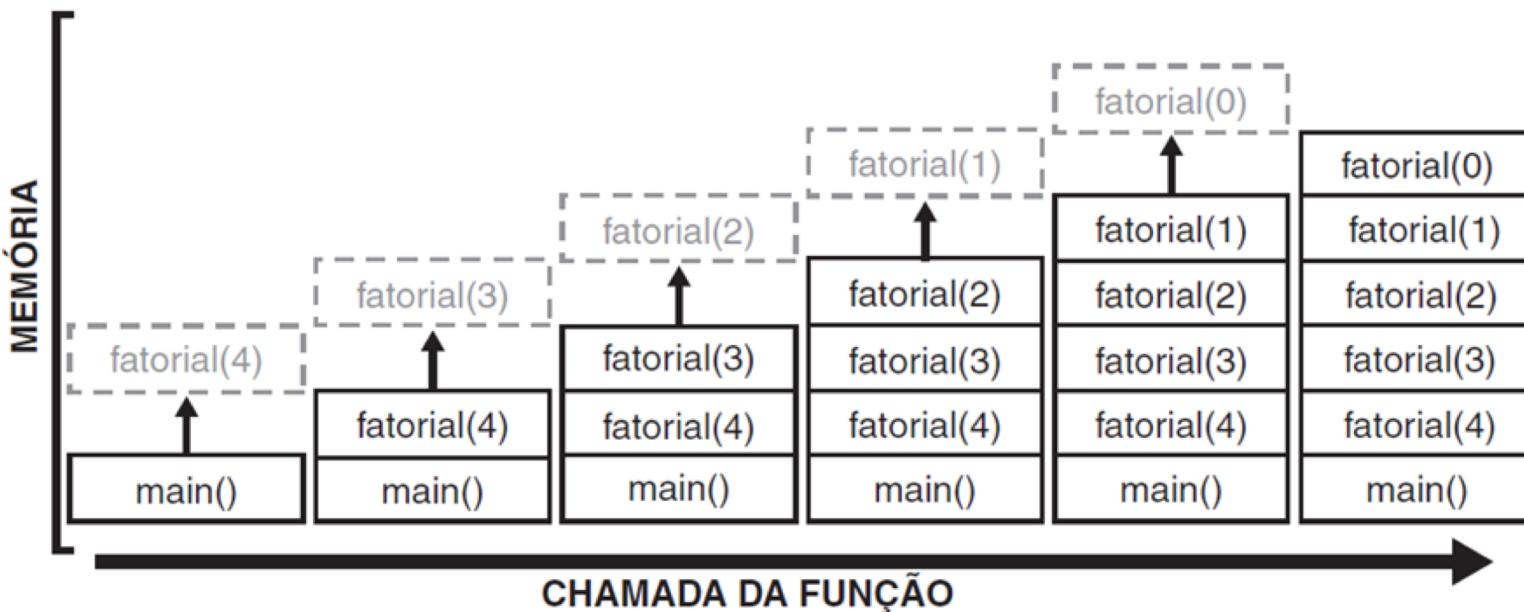
- Todo cuidado é pouco ao se fazer funções recursivas.
 - **Critério de parada:** determina quando a função deverá parar de chamar a si mesma.
 - O parâmetro da chamada recursiva deve ser sempre modificado, de forma que a recursão chegue a um término.

```
int factorial (int n){  
    if (n == 0) //critério de parada  
        return 1;  
    else /*parâmetro de factorial sempre muda*/  
        return n*fatorial(n-1);  
}
```

Recursão

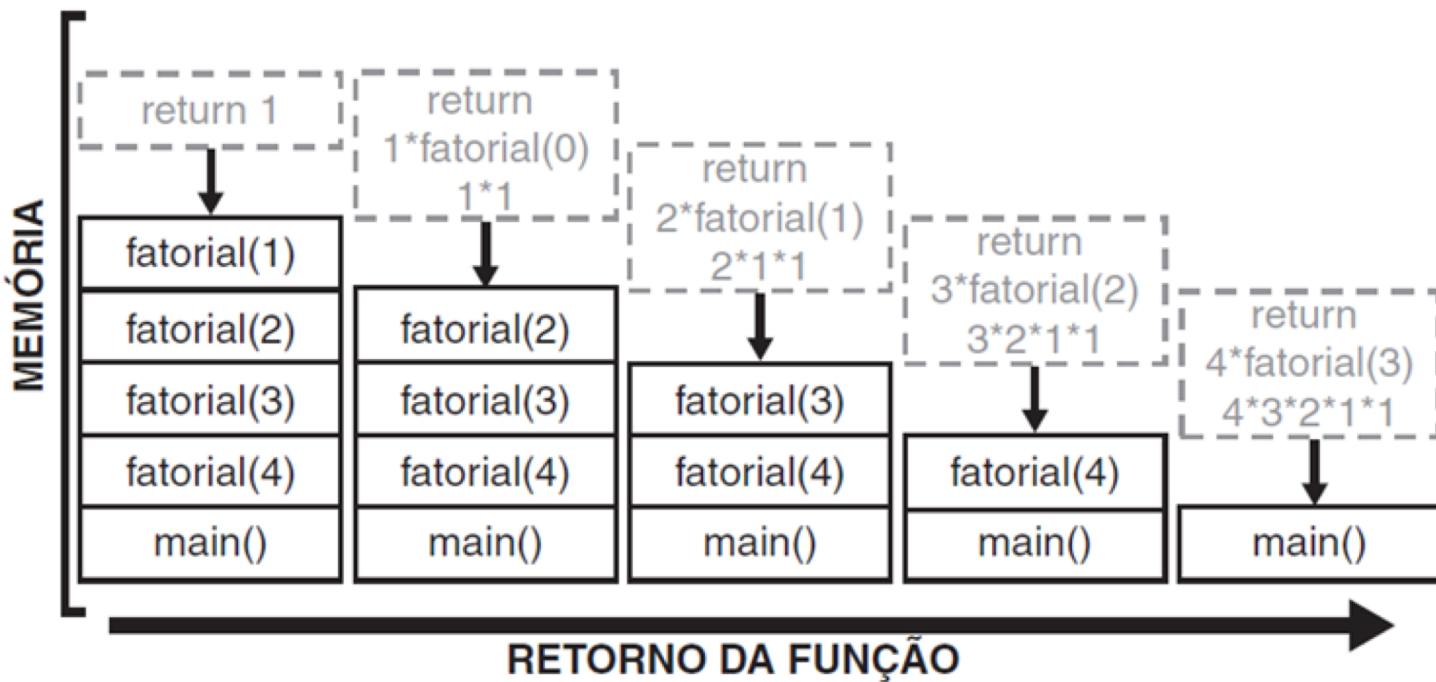
- O que acontece na chamada da função factorial com um valor como $n = 4$

```
int x = factorial(4);
```



Recursão

- Uma vez que chegamos ao caso-base, é hora de fazer o caminho de volta da recursão.



Problema de Fibonacci

- Essa seqüência é recursiva
 - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- A sequência de Fibonacci é definida como uma função recursiva utilizando a fórmula a seguir

$$F(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ F(n - 1) + F(n - 2), & \text{outros casos} \end{cases}$$

- Sua solução recursiva é mais interessante

Problema de Fibonacci

Com Recursão

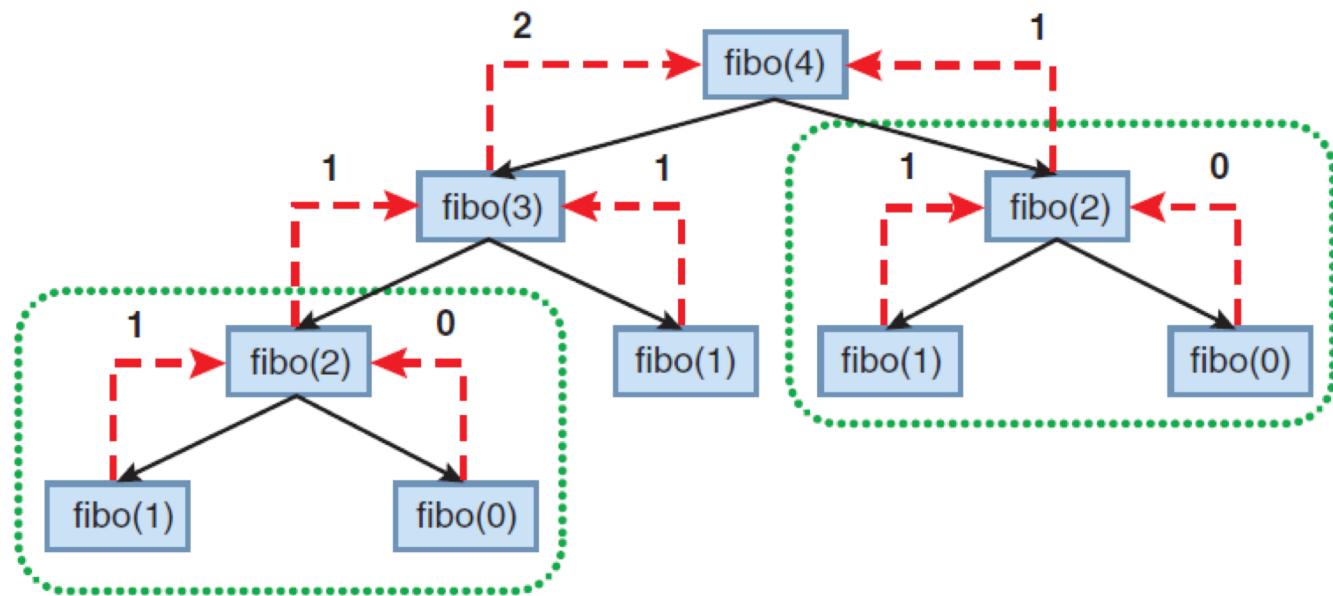
```
int fibo(int n){  
    int i, t, c, a = 0, b = 1;  
    for(i = 0; i < n; i++){  
        c = a + b;  
        a = b;  
        b = c;  
    }  
    return a;  
}
```

Sem Recursão

```
int fiboR(int n){  
    if (n == 0 || n == 1)  
        return n;  
    else  
        return fiboR(n-1) + fiboR(n-2);  
}
```

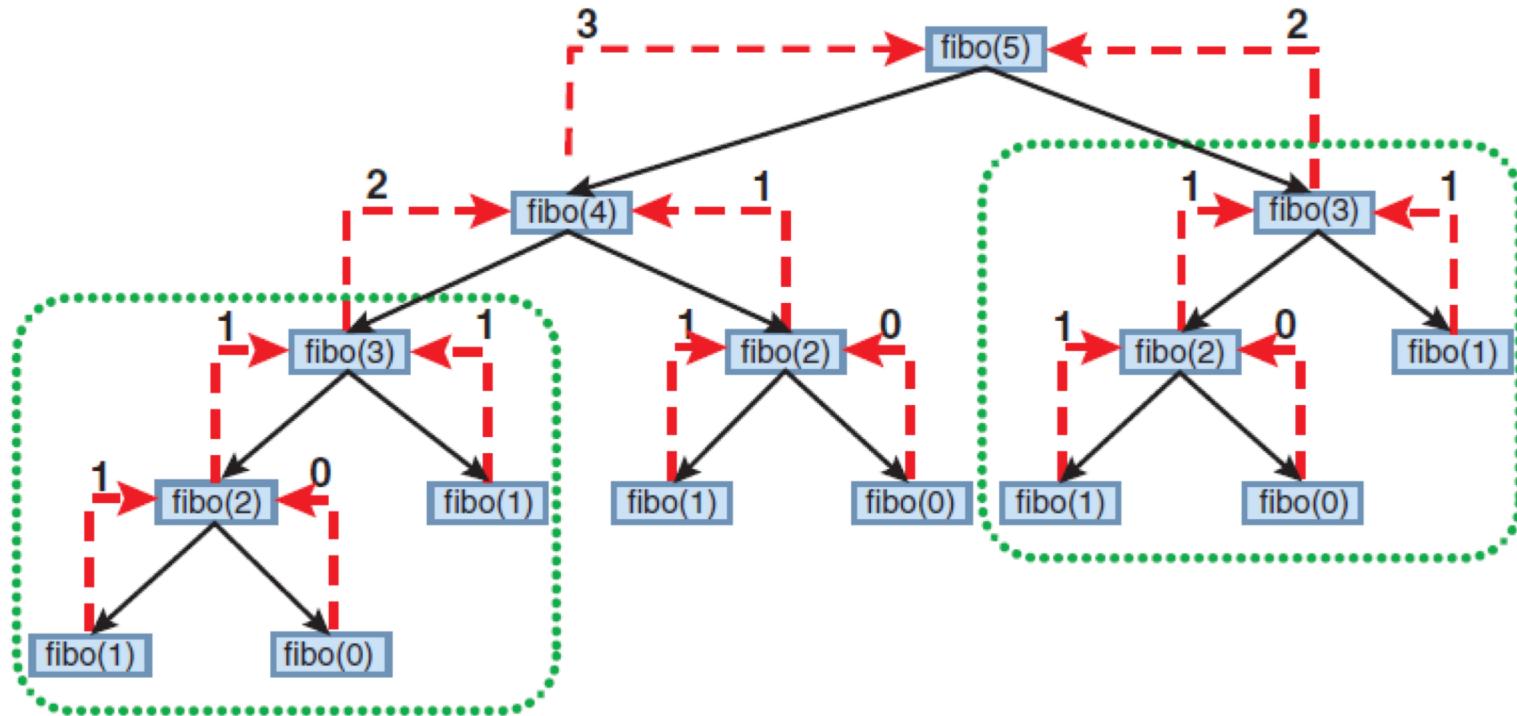
Problema de Fibonacci

Apesar de elegante, não é eficiente.



Problema de Fibonacci

Aumentado para 5



Dúvidas?



Referências

- André Luiz Villar Forbellone, Henri Frederico Eberspächer, Lógica de programação (terceira edição), Pearson, 2005, ISBN 9788576050247.
- Ulysses de Oliveira, Programando em C - Volume I - Fundamentos, editora Ciência Moderna, 2008, ISBN 9788573936599.
- **Slides baseados no material do site "Linguagem C Descomplicado"**
 - <https://programacaodescomplicada.wordpress.com/complementar/>