

5954001 - Introdução à Computação I (Optativa para Química)

Funções e Recursão

Prof. Dr. Denis M. L. Martins

DCM | FFCLRP | USP



Objetivos de Aprendizagem

- Justificar o uso de funções para **modularidade, reutilização e manutenção**.
- Explicar a passagem de argumentos **por valor** e a passagem de **endereços por valor**.
- Explicar o funcionamento da **Call Stack**.
- Compreender e implementar **recursão** com caso base correto.

Motivação

Produza um código em C que imprima um número definido de estrelas usando `int numOfStars`.

```
Para numOfStars = 1 -> *  
Para numOfStars = 5 -> *****
```

Imagine que essa mesma operação deve ser executada com valores diferentes e em **vários pontos do programa**.

```
#include <stdio.h>  
  
int main(void) {  
    int numOfStars = 5;  
    /* ... imprimir estrelas ... */  
    numOfStars = 2;  
    /* ... imprimir estrelas novamente ... */  
    return 0;  
}
```

Repetir código funciona, mas aumenta a chance de inconsistências e bugs.

Função

Uma **função** é um bloco de código nomeado que executa uma tarefa específica e bem definida.

```
void printStars(int numOfStars) {  
    for (int star = 1; star <= numOfStars; star++) {  
        printf("*");  
    }  
    printf("\n");  
}
```

- **printStars** encapsula a lógica de impressão.
- O parâmetro **numOfStars** torna a função reutilizável.
- Quem chama a função não precisa duplicar sua implementação.

printf() e **scanf()** são funções da biblioteca padrão. **sizeof** não é uma função: é um operador da linguagem C.

Por que usar Funções?

1. **Reutilização (DRY: Don't Repeat Yourself):** escreva a lógica uma vez e reutilize-a.
2. **Manutenibilidade:** alterações ficam concentradas em um ponto do programa.
3. **Legibilidade:** `main()` pode expressar a lógica do programa em alto nível.
4. **Testabilidade:** funções pequenas e bem definidas são mais fáceis de testar.
5. **Separação de responsabilidades:** cada função deve ter uma tarefa clara.

```
int main(void) {  
    lerDados();  
    processarDados();  
    imprimirRelatorio();  
    return 0;  
}
```

Uma boa função deve ter um nome que expresse **o que ela faz**.

Estrutura de uma Função em C

Toda definição de função possui:

1. **Tipo de retorno:** o tipo do valor devolvido (`int`, `float`, `double`, `void`, ...).
2. **Nome:** identificador usado para chamar a função.
3. **Parâmetros:** entradas declaradas pela função.
4. **Corpo:** instruções executadas pela função.

```
float calcularArea(float base, float altura) {  
    return base * altura;  
}
```

Parâmetros × Argumentos

Parâmetros aparecem na declaração/definição da função.

```
float calcularArea(float base, float altura) {  
    return base * altura;  
}
```

Argumentos são os valores ou expressões fornecidos na chamada.

```
float area = calcularArea(10.5f, 4.0f);
```

Neste exemplo:

- **base** e **altura** são **parâmetros**.
- **10.5f** e **4.0f** são **argumentos**.

Protótipo e Definição

O **protótipo** informa ao compilador a assinatura da função antes do seu uso.

```
// Protótipo  
float calcularArea(float base, float altura);
```

A **definição** contém a implementação real.

```
float calcularArea(float base, float altura) {  
    return base * altura;  
}
```

A assinatura permite ao compilador verificar:

- quantidade de argumentos;
- tipos dos argumentos;
- tipo de retorno.

Exemplo: Protótipo e Definição

```
#include <stdio.h>

int somar(int a, int b); // protótipo

int main(void) {
    printf("%d\n", somar(1, 2));
    return 0;
}

int somar(int a, int b) { // definição
    return a + b;
}
```

Sem uma declaração visível antes da chamada, o compilador moderno deve diagnosticar o uso incorreto.

Boa prática: mantenha os protótipos em arquivos de cabeçalho (**.h**) quando a função for compartilhada entre módulos.

Fluxo de Execução e **return**

Quando uma função é chamada, o fluxo de execução entra na função e retorna ao ponto seguinte à chamada quando ela termina.

```
float area = calcularArea(10.5f, 4.0f);  
printf("A área é: %.2f\n", area);
```

return possui duas funcionalidades importantes:

- encerra imediatamente a execução da função;
- em funções não **void**, fornece um valor ao chamador.

```
int maximo(int a, int b) {  
    if (a > b) {  
        return a;  
    }  
    return b;  
}
```

Exercícios

- a. Crie uma função que recebe um número inteiro como parâmetro e retorna o somatório de 1 até este número.
- b. Crie uma função que recebe 3 números inteiros referentes a dia, mês e ano. Caso os valores passados correspondam a uma data válida, a função deve retornar **1**. Caso contrário, ela deve retornar **0**.
- c. Crie uma função que recebe um número inteiro positivo **m** e retorna **1** se **m** é primo, **0** em caso contrário.

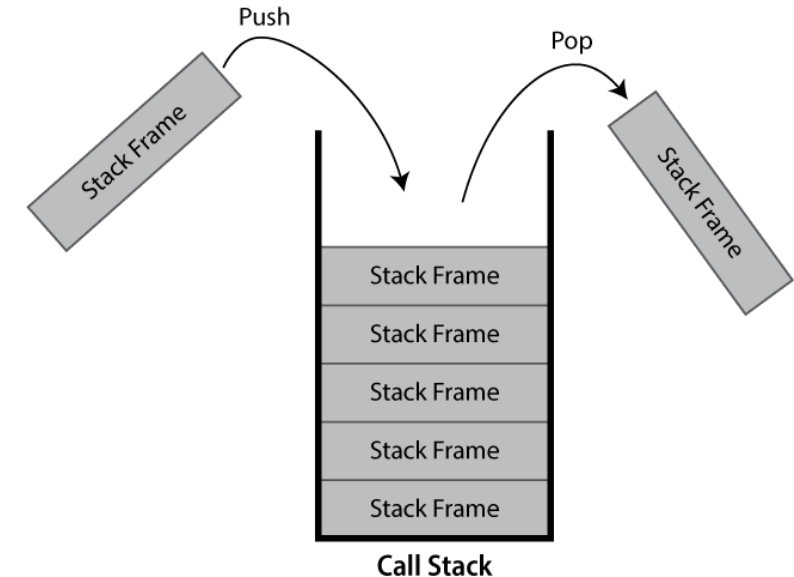
Call Stack

A **Call Stack** organiza as chamadas ativas de funções em uma estrutura LIFO (*Last-In, First-Out*).

Cada chamada normalmente possui um **stack frame** contendo informações como:

- parâmetros e/ou valores temporários;
- variáveis locais automáticas;
- estado necessário para restaurar a execução do chamador;
- endereço de retorno, conforme a ABI/plataforma.

A organização exata do frame depende do compilador, arquitetura e convenção de chamada.



Exemplo: Cadeia de Chamadas

```
#include <stdio.h>

void funcaoA(void) {
    printf("--- funcaoA ---\n");
}

int main(void) {
    printf("[MAIN] Início\n");

    funcaoA();

    printf("[MAIN] Fim\n");
    return 0;
}
```

Sequência:

main() começa → **funcaoA()** é chamada → **funcaoA()** executa e retorna → **main()** continua na instrução seguinte à chamada.

Exercício: Analise a Sequência de Chamadas

```
#include <stdio.h>

void funcaoC(void) { printf("C "); }

void funcaoB(void) {
    printf("B1 ");
    funcaoC();
    printf("B2 ");
}

void funcaoA(void) {
    printf("A1 ");
    funcaoB();
    printf("A2 ");
}

int main(void) {
    printf("M1 ");
    funcaoA();
    printf("M2 ");
    return 0;
}
```

Passagem por Valor

Em C, **todos os argumentos são passados por valor**.

```
void alterar(int x) {  
    x = 50;  
}  
  
int main(void) {  
    int n = 10;  
    alterar(n);  
    printf("%d\n", n); // 10  
}
```

A função recebe uma **cópia** do valor de **n**.

- **x** é uma variável local independente.
- alterar **x** não altera **n**.

Passando Endereços

Para permitir que uma função altere uma variável externa, passamos **seu endereço por valor**.

```
void alterar(int *p) {  
    *p = 50;  
}  
  
int main(void) {  
    int n = 10;  
    alterar(&n);  
    printf("%d\n", n); // 50  
}
```

- **&n** produz o endereço de **n**.
- **p** recebe uma **cópia desse endereço**.
- ***p** acessa o objeto localizado naquele endereço.

Mais detalhes na aula sobre [ponteiros](#).

Recursividade

Implemente a função abaixo para calcular o fatorial de um número inteiro positivo n .

```
int fatorial(int n);
```

O fatorial de um número é computado como $n! = n \times (n - 1) \times \dots \times 1$.

Note que o **fatorial(n)** utiliza o valor de **fatorial(n-1)**.

Recursividade

Recursão ocorre quando uma função chama a si mesma direta ou indiretamente.

Componentes essenciais:

1. **Caso base:** condição que encerra a cadeia de chamadas.
2. **Passo recursivo:** reduz o problema em direção ao caso base.

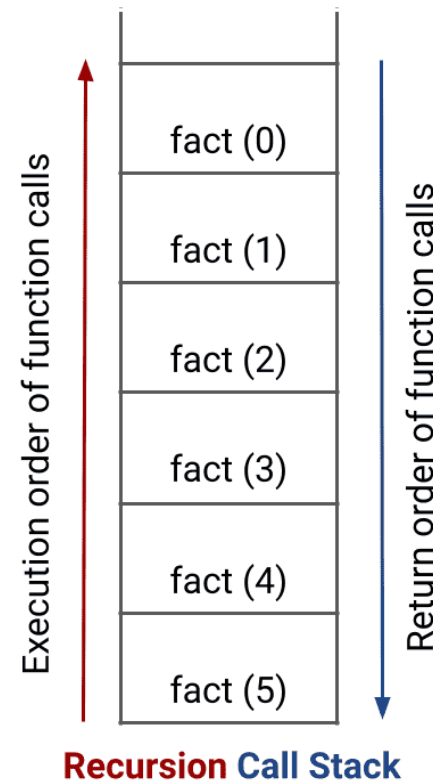
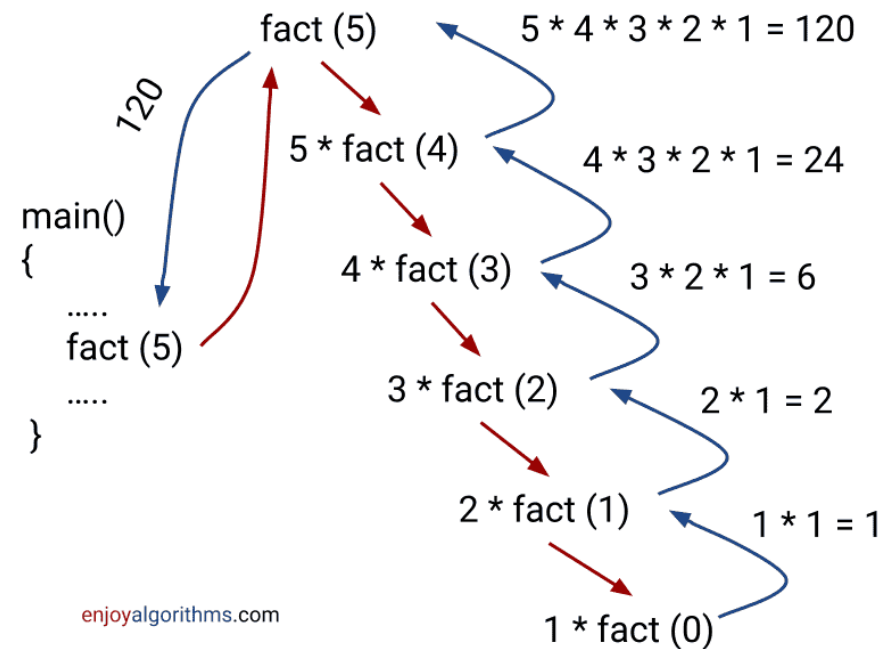
```
int fatorial(int n) {  
    if (n <= 1) {  
        return 1;  
    }  
  
    return n * fatorial(n - 1);  
}
```

Sem progresso em direção ao caso base, a recursão tende a consumir a pilha até falhar.

Passo a Passo: **fatorial(4)**

A Call Stack mantém o estado de cada chamada até que o caso base seja atingido.

Depois, as chamadas retornam em ordem inversa:



```
fatorial(1) = 1
fatorial(2) = 2 * 1
fatorial(3) = 3 * 2
fatorial(4) = 4 * 6
```

Exercício: Fibonacci

Implemente:

```
int fibonacci(int n);
```

A função deve retornar o **n-ésimo termo** da sequência, adotando:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n - 1) + F(n - 2) \end{aligned}$$

Depois:

1. use a função para imprimir os primeiros **N** termos;
2. conte quantas chamadas recursivas são realizadas;
3. discuta por que a versão recursiva simples se torna lenta para valores maiores.

Exercício: Soma Recursiva de um Vetor

Implemente sem **for** e sem **while** uma função que soma todos os elementos em um vetor de inteiros.

```
int somarElementosVetor(int indice, int tamanho, int vetor[]);
```

Exemplo:

```
int vetor[4] = {1, 6, 2, 1};  
somarElementosVetor(0, 4, vetor) // Resultado: 10
```

Defina explicitamente: caso base, passo recursivo e o argumento que diminui a cada chamada.

Extra: Reescreva o código para não precisar passar **indice** como argumento da função.

Exercício: Inverter String

Implemente a função abaixo para inverter uma string forma recursiva.

```
void inverterString(char palavra[], int inicio, int fim);
```

- Qual o caso base?
- Como inverter cada caractere?
- Qual o passo recursivo?

Limitações da Recursão

Cada chamada recursiva pode consumir espaço adicional na pilha.

```
void function(void) {  
    function();  
}  
  
int main(void) {  
    function();  
    return 0;  
}
```

Esse código não possui caso base. Possíveis consequências:

- crescimento indefinido da Call Stack;
- eventual falha de execução, frequentemente observada como *stack overflow* ou *segmentation fault*.

Separação em Módulos

Separar interface e implementação traz benefícios:

- **encapsulamento:** quem usa a função não precisa conhecer sua implementação;
- **reutilização:** o mesmo módulo pode atender vários programas;
- **compilação separada:** módulos podem ser recompilados independentemente;
- **organização:** contratos ficam concentrados no arquivo de cabeçalho.

Estrutura típica:

```
projeto/  
├── main.c  
├── mathutils.c  
└── mathutils.h
```

Arquivo de Cabeçalho (.h)

O cabeçalho expõe a **interface pública** do módulo.

```
// mathutils.h
#ifndef MATHUTILS_H
#define MATHUTILS_H

int somar(int a, int b);
void imprimirResultado(int x);

#endif
```

Cabeçalhos podem conter, conforme o projeto:

- protótipos de funções;
- **typedef**, **struct**, **enum**;
- macros e constantes;

Arquivo de Fonte (.c)

```
// mathutils.c
#include "mathutils.h"
#include <stdio.h>

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

void imprimirResultado(int x) {
    printf("Resultado: %d\n", x);
}
```

Por que o **.c** deve incluir seu próprio **.h**?

- o compilador verifica se a definição corresponde ao protótipo;
- inconsistências de parâmetros e retorno são detectadas cedo;
- a interface torna-se a fonte de verdade do módulo.

Arquivo Principal: Cliente

```
#include "mathutils.h"

int main(void) {
    int resultado = somar(2, 3);
    imprimirResultado(resultado);
    return 0;
}
```

O cliente conhece a **interface** declarada em **mathutils.h**.

Ele não precisa conhecer os detalhes de implementação de **mathutils.c**.

Essa separação prepara o caminho para bibliotecas maiores e APIs estáveis.

Compilação e Linking

Primeiro, cada arquivo fonte é compilado para código objeto:

1. **Compilação (.c -> .o)**: Transforma código fonte em objeto.

```
gcc -c mathutils.c      # Gera mathutils.o  
gcc -c main.c           # Gera main.o
```

Depois, os objetos são ligados:

2. **Linking (.o -> Executável)**: Junta os objetos e resolve endereços.

```
gcc main.o mathutils.o -o programa
```

O linker resolve referências entre símbolos definidos em módulos diferentes.

Conclusão

- Funções dividem problemas em unidades menores.
- Protótipos estabelecem contratos verificáveis pelo compilador.
- Argumentos são sempre passados por valor em C.
- A Call Stack organiza chamadas aninhadas e recursivas.
- Recursão exige caso base e progresso.
- Cabeçalhos e arquivos fonte separam interface e implementação.

Material Adicional: Explore recursividade visualmente em: c-lang-learning.com

Dúvidas e Discussão

Questões Teóricas

1. Por que protótipos de função são importantes para a verificação de tipos?
2. Qual a diferença entre **parâmetro** e **argumento**?
3. Explique por que **alterar(n)** não modifica **n**, mas **alterar(&n)** pode modificá-lo.
4. C possui passagem por referência nativa? Justifique.
5. O que normalmente acontece com a Call Stack quando **A()** chama **B()** e **B()** chama **C()**?
6. Qual é a diferença entre uma função **void** e uma função que retorna um valor?

Exercício 1

Implemente:

```
float calcularMedia(float n1, float n2, float n3);
```

Use a função em **main()** para calcular a média de um aluno.

Depois generalize para:

```
float calcularMediaVetor(const float notas[], int quantidade);
```

Requisitos:

- as notas são lidas em **main()**;
- a função não deve modificar o vetor;
- trate **quantidade <= 0** de forma definida pelo seu contrato.

Exercício 2

Implemente:

```
void trocar(int *a, int *b);
```

A função deve trocar os valores apontados por **a** e **b**.

Versão recomendada:

- use uma variável temporária;
- discuta por que ela é clara e segura.

Desafio opcional: pesquise estratégias sem variável temporária e explique problemas potenciais, como overflow ou comportamento inesperado quando os dois ponteiros são iguais.

Em código real, legibilidade e correção são mais importantes do que “truques”.

Exercício 3

Implemente:

```
int pot(int base, unsigned int expoente);
```

Para expoentes não negativos n e base b , temos:

$$b^0 = 1$$

$$b^n = b * b^{n-1}$$