Guia Avançado de Funções em C

Funções Avançadas

Função sem retorno (`void`)

```c
#include <stdio.h>

/* Função que imprime uma mensagem */
void greet() {
    printf("Olá, bem-vindo!\n");
}

int main() {
    greet();
    return 0;
}
```

Explicação: Funções `void` não retornam valor. Usadas para executar ações ou efeitos colaterais.

Função com retorno

```c
#include <stdio.h>

/* Função que retorna soma de dois inteiros */
int sum(int a, int b) {
    return a + b;
}

int main() {
    int result = sum(5, 3);
    printf("Soma: %d\n", result);
    return 0;
}
```

Explicação: Funções podem retornar valores de tipos definidos, permitindo reutilização de resultados.

Função com múltiplos parâmetros

```c
#include <stdio.h>

/* Função que calcula média de três números */
float average(float a, float b, float c) {
    return (a + b + c) / 3.0;
```

```c
}

int main() {
    float avg = average(4.0, 5.0, 6.0);
    printf("Média: %.2f\n", avg);
    return 0;
}
```

Explicação: Funções podem receber múltiplos parâmetros de diferentes tipos, aumentando flexibilidade.

Função recursiva

```c
#include <stdio.h>

/* Função recursiva para fatorial */
int factorial(int n) {
    if(n <= 1) return 1;
    return n * factorial(n - 1);
}

int main() {
    printf("Fatorial de 5: %d\n", factorial(5));
    return 0;
}
```

Explicação: Funções podem chamar a si mesmas (recursão) para resolver problemas repetitivos ou hierárquicos.

Função inline (exemplo moderno em C99+)

```c
#include <stdio.h>

/* Função inline simples */
inline int square(int x) {
    return x * x;
}

int main() {
    printf("Quadrado de 4: %d\n", square(4));
    return 0;
}
```

Explicação: Funções `inline` sugerem ao compilador expandir o código no local da chamada, reduzindo overhead de chamadas.

Função com vetor como parâmetro

```c
#include <stdio.h>

/* Função que calcula a soma dos elementos de um vetor */
int sum_array(int arr[], int size) {
    int sum = 0;
    for(int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}

int main() {
    int numbers[5] = {1, 2, 3, 4, 5};
    int total = sum_array(numbers, 5);
    printf("Soma do vetor: %d\n", total);
    return 0;
}
```

Explicação: Arrays passados para funções **são tratados como ponteiros**, permitindo acessar e modificar elementos diretamente.

Função com ponteiro e referência simulada

```c
#include <stdio.h>

/* Função que troca valores usando ponteiros */
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;
    printf("Antes da troca: x=%d, y=%d\n", x, y);
    swap(&x, &y);
    printf("Depois da troca: x=%d, y=%d\n", x, y);
    return 0;
}
```

Explicação: Passando o endereço das variáveis (&x) permite que a função **modifique o valor original**, simulando passagem por referência.

Função com struct como parâmetro

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

/* Definição de struct */
struct Person {
    char name[50];
    int age;
};

/* Função que imprime informações da struct */
void print_person(struct Person p) {
    printf("Nome: %s\n", p.name);
    printf("Idade: %d\n", p.age);
}

/* Função que altera a struct via ponteiro (simulando referência) */
void birthday(struct Person *p) {
    p->age += 1;
}

int main() {
    struct Person person1;
    strcpy(person1.name, "Alice");
    person1.age = 30;

    print_person(person1);
    birthday(&person1);
    print_person(person1);

    return 0;
}
```

Explicação: Structs podem ser passadas por **valor** ou **ponteiro**.

- Por valor: função recebe uma cópia, alterações não afetam original.
- Por ponteiro: função pode modificar os dados originais, simulando passagem por referência.

Função que retorna struct

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

struct Point {
    int x;
    int y;
};

struct Point cria_ponto(int a, int b) {
    struct Point p;
```

```
    p.x = a;
    p.y = b;
    return p;
}

int main() {
    struct Point p1 = cria_ponto(5, 10);
    printf("Ponto: (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

Explicação: Funções podem retornar structs completas, permitindo encapsular múltiplos valores.

Função com ponteiro para ponteiro

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

/* Aloca dinamicamente um inteiro e atribui valor */
void aloca_inteiro(int **ptr, int valor) {
    *ptr = (int *)malloc(sizeof(int));
    **ptr = valor;
}

int main() {
    int *num = NULL;
    aloca_inteiro(&num, 42);
    printf("Valor alocado: %d\n", *num);
    free(num);
    return 0;
}
```

Explicação: Ponteiros para ponteiros permitem que funções **modifiquem endereços de memória** externos, útil em alocação dinâmica.

Função com array multidimensional

```
#include <stdio.h>

/* Soma todos os elementos de uma matriz 2x3 */
int soma_matriz(int mat[2][3]) {
    int sum = 0;
    for(int i=0; i<2; i++)
        for(int j=0; j<3; j++)
            sum += mat[i][j];
    return sum;
}
```

```c
int main() {
    int m[2][3] = {{1,2,3},{4,5,6}};
    printf("Soma da matriz: %d\n", soma_matriz(m));
    return 0;
}
```

Explicação: Funções podem receber arrays multidimensionais como parâmetros, mas a dimensão interna deve ser conhecida.

Função que recebe função como parâmetro

```c
#include <stdio.h>

/* Função que aplica uma função f a dois inteiros */
int aplicar(int a, int b, int (*f)(int,int)) {
    return f(a,b);
}

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

int main() {
    printf("Soma: %d\n", aplicar(3,4,soma));
    printf("Multiplica: %d\n", aplicar(3,4,multiplica));
    return 0;
}
```

Explicação: Funções podem receber **ponteiros para funções**, permitindo **passar comportamento como argumento**.

Função que modifica string via ponteiro

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

/* Converte string para maiúsculas */
void to_upper(char *str) {
    for(int i=0; str[i]; i++) {
        if(str[i]>='a' && str[i]<='z') str[i]-=32;
    }
}

int main() {
    char texto[] = "hello world";
    to_upper(texto);
    printf("%s\n", texto);
```

```
    return 0;
}
```

Explicação: Passando strings como ponteiros, funções podem **alterar o conteúdo original**.

Função genérica com ponteiro `void`

```c
#include <stdio.h>

/* Imprime valor inteiro ou float baseado em flag */
void imprime_valor(void *ptr, char tipo) {
    if(tipo=='i') printf("%d\n", *(int*)ptr);
    else if(tipo=='f') printf("%.2f\n", *(float*)ptr);
}

int main() {
    int a = 10;
    float b = 3.14;
    imprime_valor(&a,'i');
    imprime_valor(&b,'f');
    return 0;
}
```

Explicação: Ponteiros `void` permitem criar funções **genéricas**, que recebem diferentes tipos de dados.

Funções de Somatórios e Sequências em C

Somatórios e Sequências Matemáticas

Somatório de números inteiros de 1 até n

```c
#include <stdio.h>

/* Σ i de 1 até n = n*(n+1)/2 */
int somatorio_inteiros(int n) {
    return n * (n + 1) / 2;
}

int main() {
    printf("Somatório de 1 a 10: %d\n", somatorio_inteiros(10));
    return 0;
}
```

Explicação: Usa fórmula direta do somatório, sem loops.

Somatório de números pares até n

```c
#include <stdio.h>

/* Σ i par de 1 até n = n/2*(n/2 + 1) */
int somatorio_pares(int n) {
    int k = n / 2;
    return k * (k + 1);
}

int main() {
    printf("Somatório de pares até 10: %d\n", somatorio_pares(10));
    return 0;
}
```

Explicação: Calcula apenas a soma dos pares usando fórmula da soma dos primeiros k inteiros.

Somatório de números ímpares até n

```c
#include <stdio.h>

/* Σ i ímpar de 1 até n = (n+1)/2)^2 */
int somatorio_impares(int n) {
    int k = (n + 1) / 2;
    return k * k;
}

int main() {
    printf("Somatório de ímpares até 10: %d\n", somatorio_impares(10));
    return 0;
}
```

Explicação: Soma dos primeiros k números ímpares: k^2.

Sequência de Fibonacci usando fórmula iterativa

```c
#include <stdio.h>

/* Sequência de Fibonacci: 0,1,1,2,3,5,... */
void fibonacci(int n, int seq[]) {
    if(n>0) seq[0]=0;
    if(n>1) seq[1]=1;
    for(int i=2;i<n;i++)
        seq[i]=seq[i-1]+seq[i-2];
}
```

```c
int main() {
    int n=10;
    int seq[10];
    fibonacci(n,seq);
    printf("Fibonacci: ");
    for(int i=0;i<n;i++) printf("%d ",seq[i]);
    printf("\n");
    return 0;
}
```

Explicação: Gera sequência de Fibonacci iterativamente sem funções externas.

Somatório de termos de PA (Progressão Aritmética)

```c
#include <stdio.h>

/* Soma de n termos: n*(2*a1 + (n-1)*r)/2 */
double soma_pa(int a1, int r, int n) {
    return n * (2*a1 + (n-1)*r) / 2.0;
}

int main() {
    printf("Soma 10 termos PA (a1=2, r=3): %.2f\n", soma_pa(2,3,10));
    return 0;
}
```

Explicação: Usa fórmula direta da soma de PA, sem laços.

Somatório de termos de PG (Progressão Geométrica)

```c
#include <stdio.h>

/* Soma de n termos: a1*(1-q^n)/(1-q), q!=1 */
double soma_pg(double a1, double q, int n) {
    double q_n = 1;
    for(int i=0;i<n;i++) q_n *= q; // Calcula q^n
    return a1 * (1 - q_n) / (1 - q);
}

int main() {
    printf("Soma 5 termos PG (a1=2, q=3): %.2f\n", soma_pg(2,3,5));
    return 0;
}
```

Explicação: Aplica fórmula da soma de PG, calculando manualmente q^n.

Somatório de quadrados de 1 até n

```c
#include <stdio.h>

/* Σ i^2 de 1 até n = n*(n+1)*(2n+1)/6 */
int somatorio_quadrados(int n) {
    return n*(n+1)*(2*n+1)/6;
}

int main() {
    printf("Somatório de quadrados até 5: %d\n", somatorio_quadrados(5));
    return 0;
}
```

Explicação: Usa fórmula matemática direta sem loops.

Somatório de cubos de 1 até n

```c
#include <stdio.h>

/* Σ i^3 de 1 até n = (n*(n+1)/2)^2 */
int somatorio_cubos(int n) {
    int soma = n*(n+1)/2;
    return soma * soma;
}

int main() {
    printf("Somatório de cubos até 5: %d\n", somatorio_cubos(5));
    return 0;
}
```

Explicação: Soma de cubos baseada na fórmula matemática: $(\Sigma i)^2$.

Funções Matemáticas em C

Funções com Fórmulas Matemáticas

Função para calcular área de círculo

```c
#include <stdio.h>

#define PI 3.141592653589793

/* Área do círculo: π * r² */
```

```c
double area_circulo(double raio) {
    return PI * raio * raio;
}

int main() {
    double r = 5.0;
    printf("Área do círculo: %.2f\n", area_circulo(r));
    return 0;
}
```

Explicação: Usa a fórmula direta da geometria: área = $\pi \cdot r^2$, sem funções da biblioteca matemática.

Função para calcular hipotenusa (Teorema de Pitágoras)

```c
#include <stdio.h>

/* Hipotenusa de um triângulo retângulo: √(a² + b²) */
double hipotenusa(double a, double b) {
    double soma_quadrados = a*a + b*b;
    double raiz = soma_quadrados;
    double x = soma_quadrados / 2.0;
    for(int i=0; i<10; i++)  // Método iterativo simples para raiz quadrada
        x = 0.5 * (x + raiz/x);
    return x;
}

int main() {
    printf("Hipotenusa: %.2f\n", hipotenusa(3,4));
    return 0;
}
```

Explicação: Calcula $\sqrt{a^2 + b^2}$ usando **método de Newton-Raphson** iterativo, sem `math.h`.

Função para calcular área de triângulo (fórmula de Heron)

```c
#include <stdio.h>

/* Área usando lados a, b, c: √(s*(s-a)*(s-b)*(s-c)) */
double area_triangulo(double a, double b, double c) {
    double s = (a + b + c) / 2.0;
    double produto = s*(s-a)*(s-b)*(s-c);
    double x = produto;
    double raiz = produto / 2.0;
    for(int i=0; i<10; i++)
        raiz = 0.5 * (raiz + x/raiz);
    return raiz;
}
```

```c
int main() {
    printf("Área do triângulo: %.2f\n", area_triangulo(3,4,5));
    return 0;
}
```

Explicação: Fórmula de Heron aplicada manualmente, sem bibliotecas externas.

Função para calcular potência (x^y)

```c
#include <stdio.h>

/* Potência usando multiplicações */
double potencia(double base, int expoente) {
    double resultado = 1;
    int positivo = expoente >= 0 ? 1 : 0;
    if(!positivo) expoente = -expoente;
    for(int i=0; i<expoente; i++)
        resultado *= base;
    if(!positivo) resultado = 1 / resultado;
    return resultado;
}

int main() {
    printf("2^10 = %.2f\n", potencia(2,10));
    printf("2^-3 = %.2f\n", potencia(2,-3));
    return 0;
}
```

Explicação: Calcula potências inteiras e negativas sem usar `pow()`.

Função para calcular distância entre dois pontos 2D

```c
#include <stdio.h>

/* Distância: √((x2-x1)² + (y2-y1)²) */
double distancia(double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    double soma = dx*dx + dy*dy;
    double raiz = soma / 2.0;
    for(int i=0; i<10; i++)
        raiz = 0.5 * (raiz + soma/raiz);
    return raiz;
}

int main() {
```

```
    printf("Distância: %.2f\n", distancia(0,0,3,4));
    return 0;
}
```

Explicação: Calcula distância usando a **fórmula geométrica**, com raiz iterativa.

Função para calcular média geométrica

```
#include <stdio.h>

/* Média geométrica: n√(x1*x2*...*xn) */
double media_geometrica(double v[], int n) {
    double produto = 1.0;
    for(int i=0; i<n; i++)
        produto *= v[i];
    double raiz = produto / 2.0;
    for(int i=0; i<10; i++)
        raiz = 0.5 * (raiz + produto/raiz);
    return raiz;
}

int main() {
    double nums[3] = {2, 8, 4};
    printf("Média geométrica: %.2f\n", media_geometrica(nums,3));
    return 0;
}
```

Explicação: Produto dos números seguido de raiz n-ésima aproximada (n=2 simplificado) sem funções prontas.

Função para calcular progressão aritmética

```
#include <stdio.h>

/* n-ésimo termo: a1 + (n-1)*r */
double termo_pa(double a1, double r, int n) {
    return a1 + (n-1)*r;
}

/* Soma dos n primeiros termos: n*(a1+an)/2 */
double soma_pa(double a1, double r, int n) {
    double an = termo_pa(a1,r,n);
    return n*(a1+an)/2.0;
}

int main() {
    printf("10º termo PA: %.2f\n", termo_pa(2,3,10));
```

```
    printf("Soma 10 primeiros termos: %.2f\n", soma_pa(2,3,10));
    return 0;
}
```

Explicação: Usa **fórmulas matemáticas diretas da PA** sem loops para soma.

Função para calcular média ponderada

```
#include <stdio.h>

/* Média ponderada: Σ(xi*wi)/Σ(wi) */
double media_ponderada(double valores[], double pesos[], int n) {
    double soma_valor = 0, soma_peso = 0;
    for(int i=0; i<n; i++) {
        soma_valor += valores[i]*pesos[i];
        soma_peso += pesos[i];
    }
    return soma_valor / soma_peso;
}

int main() {
    double notas[3] = {7.0, 8.5, 9.0};
    double pesos[3] = {2, 3, 5};
    printf("Média ponderada: %.2f\n", media_ponderada(notas,pesos,3));
    return 0;
}
```

Explicação: Calcula média ponderada usando **somatória de produtos** diretamente, sem bibliotecas externas.
