


Guia de Estudo em C Puro

Este repositório contém exemplos organizados de código em **C** para auxiliar no estudo e conversão de conceitos. Cada tópico apresenta:

- ☒ Código completo e pronto para compilar.
 -  Explicação curta e direta.
-

1 Declaração de variáveis

```
#include <stdio.h>

int main() {
    int age = 25;           /* variável inteira */
    float height = 1.75;    /* variável de ponto flutuante */
    char grade = 'A';       /* variável caractere */

    printf("Idade: %d\n", age);
    printf("Altura: %.2f\n", height);
    printf("Nota: %c\n", grade);

    return 0;
}
```

Explicação: Variáveis armazenam valores na memória. É necessário declarar o tipo (`int`, `float`, `char`, etc.) antes do uso.

2 Funções em C

```
#include <stdio.h>

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

int main() {
    int x = 10, y = 20;
    int result = sum(x, y);

    printf("Resultado da soma: %d\n", result);
    return 0;
}
```

Explicação: Funções permitem organizar o código em blocos reutilizáveis. Recebem parâmetros, realizam operações e podem retornar valores.

2.1 Função sem retorno (**void**)

```
#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.

2.2 Função com retorno

```
#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.

2.3 Função com múltiplos parâmetros

```
#include <stdio.h>

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

```
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.

2.4 Função inline (exemplo moderno em C99+)

```
#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, útil para funções pequenas.

2.5 Função com vetor como parâmetro

```
#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.

2.6 Função com ponteiro e referência simulada

```
#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.

2.7 Função com struct como parâmetro

```
#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.

3 Parâmetros por valor

```

#include <stdio.h>

/* Função que tenta modificar os valores */
void increment(int a) {
    a = a + 1;
    printf("Dentro da função: a = %d\n", a);
}

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

    increment(num);

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

```

Explicação: Passagem por valor envia uma **cópia** da variável para a função. Modificações dentro da função **não alteram** o valor original.

4 Parâmetros simulando referência (usando ponteiros)

```

#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 = 5, y = 10;
    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 da variável (&x) e usando ponteiros (*a), a função consegue modificar o valor original, simulando passagem por referência.

5 Arrays

```

#include <stdio.h>

int main() {
    int numbers[5] = {10, 20, 30, 40, 50}; /* array de inteiros */

    for(int i = 0; i < 5; i++) {
        printf("Elemento %d: %d\n", i, numbers[i]);
    }

    return 0;
}

```

Explicação: Arrays armazenam múltiplos valores do mesmo tipo em posições contíguas de memória, acessíveis por índice, começando do zero.

6 Ponteiros e aritmética de ponteiros

```

#include <stdio.h>

int main() {
    int value = 42;
    int *ptr = &value; /* ponteiro para a variável value */

    printf("Endereço de value: %p\n", (void*)ptr);
    printf("Valor apontado por ptr: %d\n", *ptr);

    *ptr = 100; /* modifica o valor original */
    printf("Novo valor de value: %d\n", value);
}

```

```
ptr++; /* aritmética de ponteiros: move para o próximo endereço de int */
printf("Endereço após incrementar ponteiro: %p\n", (void*)ptr);

return 0;
}
```

Explicação: Ponteiros armazenam endereços de memória. Aritmética de ponteiros permite navegar pelos elementos de arrays ou posições contíguas, usando `*` para acessar o valor apontado.

7 Structs

```
#include <stdio.h>

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

int main() {
    struct Person person1 = {"Gabriel", 25, 1.75};

    printf("Nome: %s\n", person1.name);
    printf("Idade: %d\n", person1.age);
    printf("Altura: %.2f\n", person1.height);

    return 0;
}
```

Explicação: Structs agrupam diferentes tipos de dados em uma única unidade, facilitando o gerenciamento de informações relacionadas a um mesmo objeto.

8 Alocação dinâmica de memória

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

int main() {
    int n = 5;
    int *arr = (int*)malloc(n * sizeof(int)); /* aloca memória para 5 inteiros */

    if(arr == NULL) {
        printf("Erro ao alocar memória.\n");
        return 1;
    }
}
```

```

    for(int i = 0; i < n; i++) {
        arr[i] = i * 10;
        printf("arr[%d] = %d\n", i, arr[i]);
    }

    free(arr); /* libera a memória alocada */
    return 0;
}

```

Explicação: `malloc` aloca memória em tempo de execução. Sempre liberar com `free` para evitar vazamentos de memória.

9 Manipulação de strings em C

```

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

int main() {
    char str1[20] = "Olá";
    char str2[20];

    strcpy(str2, str1);           /* copia str1 para str2 */
    strcat(str2, " Mundo!");      /* concatena string */

    printf("str2: %s\n", str2);
    printf("Comprimento de str2: %lu\n", strlen(str2));

    return 0;
}

```

Explicação: Strings em C são arrays de caracteres terminados com `\0`. Funções de `string.h` permitem copiar, concatenar e medir comprimento.

10 Modularização (separação em headers e .c)

arquivo `math_utils.h`

```

#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int add(int a, int b);
int multiply(int a, int b);

#endif

```

arquivo `math_utils.c`


```
#include "math_utils.h"

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

int multiply(int a, int b) {
    return a * b;
}
```

arquivo main.c

```
#include <stdio.h>
#include "math_utils.h"

int main() {
    int x = 5, y = 3;
    printf("Soma: %d\n", add(x, y));
    printf("Multiplicação: %d\n", multiply(x, y));
    return 0;
}
```

Explicação: Modularização organiza o código em múltiplos arquivos (.c e .h), separando implementação e interface, facilitando manutenção e reutilização.

11 Operadores e expressões

```
#include <stdio.h>

int main() {
    int a = 10, b = 3;
    int sum = a + b;          /* adição */
    int diff = a - b;         /* subtração */
    int prod = a * b;         /* multiplicação */
    int div = a / b;          /* divisão inteira */
    int mod = a % b;          /* resto da divisão */

    printf("Soma: %d\n", sum);
    printf("Diferença: %d\n", diff);
    printf("Produto: %d\n", prod);
    printf("Divisão: %d\n", div);
    printf("Resto: %d\n", mod);

    return 0;
}
```

Explicação: Operadores realizam cálculos e comparações. Expressões combinam variáveis e operadores para produzir valores.

1 2 Estruturas de decisão (if, switch)

```
#include <stdio.h>

int main() {
    int num = 2;

    if(num > 0) {
        printf("Número positivo\n");
    } else if(num < 0) {
        printf("Número negativo\n");
    } else {
        printf("Zero\n");
    }

    switch(num) {
        case 1:
            printf("Valor é 1\n");
            break;
        case 2:
            printf("Valor é 2\n");
            break;
        default:
            printf("Outro valor\n");
    }

    return 0;
}
```

Explicação: `if` e `else` controlam fluxo baseado em condições. `switch` permite selecionar blocos de código com base em valores discretos.

1 3 Estruturas de repetição (for, while, do...while)

```
#include <stdio.h>

int main() {
    int i = 0;

    /* for */
    for(i = 0; i < 5; i++) {
        printf("for loop: %d\n", i);
    }

    /* while */
}
```

```

i = 0;
while(i < 5) {
    printf("while loop: %d\n", i);
    i++;
}

/* do...while */
i = 0;
do {
    printf("do...while loop: %d\n", i);
    i++;
} while(i < 5);

return 0;
}

```

Explicação: Estruturas de repetição executam blocos de código múltiplas vezes: **for** para contagem conhecida, **while** e **do...while** para condições.

14 Vetores e matrizes

```

#include <stdio.h>

int main() {
    int vector[5] = {1, 2, 3, 4, 5};           /* vetor unidimensional */
    int matrix[2][3] = {{1,2,3},{4,5,6}};      /* matriz 2x3 */

    printf("Vetor:\n");
    for(int i = 0; i < 5; i++) {
        printf("%d ", vector[i]);
    }
    printf("\nMatriz:\n");
    for(int i = 0; i < 2; i++) {
        for(int j = 0; j < 3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}

```

Explicação: Vetores armazenam elementos em uma dimensão; matrizes armazenam dados em múltiplas dimensões, acessíveis via índices.

14.1 Vetores e matrizes (avançado)

```

#include <stdio.h>

int main() {
    /* Vetor de floats */
    float temperatures[4] = {36.5, 37.2, 38.0, 36.8};

    /* Matriz 3x3 de doubles */
    double matrix[3][3] = {
        {1.1, 2.2, 3.3},
        {4.4, 5.5, 6.6},
        {7.7, 8.8, 9.9}
    };

    /* Matriz 3D de inteiros (2x2x3) */
    int cube[2][2][3] = {
        {{1,2,3}, {4,5,6}},
        {{7,8,9}, {10,11,12}}
    };

    /* Vetor de ponteiros para char (array de strings) */
    const char *names[3] = {"Alice", "Bob", "Charlie"};

    printf("Vetor de floats:\n");
    for(int i = 0; i < 4; i++) {
        printf("%.1f ", temperatures[i]);
    }

    printf("\nMatriz 3x3 de doubles:\n");
    for(int i = 0; i < 3; i++) {
        for(int j = 0; j < 3; j++) {
            printf("%.1f ", matrix[i][j]);
        }
        printf("\n");
    }

    printf("\nMatriz 3D de inteiros:\n");
    for(int i = 0; i < 2; i++) {
        for(int j = 0; j < 2; j++) {
            for(int k = 0; k < 3; k++) {
                printf("%d ", cube[i][j][k]);
            }
            printf("\n");
        }
        printf("---\n");
    }

    printf("\nVetor de strings:\n");
    for(int i = 0; i < 3; i++) {
        printf("%s\n", names[i]);
    }

    return 0;
}

```

Explicação: Vetores e matrizes podem ter múltiplas dimensões e tipos diferentes (`int`, `float`, `double`). É possível usar ponteiros para criar arrays de strings e manipular dados mais complexos. Índices permitem navegar em qualquer dimensão.

15 Funções recursivas

```
#include <stdio.h>

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

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

Explicação: Funções recursivas chamam a si mesmas. Sempre deve haver uma condição de parada para evitar loop infinito.

16 Arquivos (entrada/saída em disco)

```
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "w"); /* abre arquivo para escrita */
    if(file == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    fprintf(file, "Linha 1\n"); /* escreve no arquivo */
    fprintf(file, "Linha 2\n");

    fclose(file); /* fecha o arquivo */

    file = fopen("example.txt", "r"); /* abre arquivo para leitura */
    char line[100];
    while(fgets(line, sizeof(line), file)) {
        printf("%s", line); /* lê e imprime cada linha */
    }
}
```

```
    fclose(file);

    return 0;
}
```

Explicação: `FILE*` permite manipular arquivos. `fopen` abre, `fprintf` escreve, `fgets` lê e `fclose` fecha o arquivo.

17 Diretivas do pré-processador

```
#include <stdio.h>
#define PI 3.14159 /* define uma constante */
#define SQUARE(x) ((x) * (x)) /* define macro */

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

Explicação: Diretivas do pré-processador (`#define`, `#include`) são processadas antes da compilação. Servem para incluir arquivos, definir constantes e macros.

18 Enumerações (enum)

```
#include <stdio.h>

/* Definição de enumeração */
enum Color { RED, GREEN, BLUE };

int main() {
    enum Color favorite = GREEN;

    if(favorite == GREEN) {
        printf("Cor favorita é verde.\n");
    }

    return 0;
}
```

Explicação: `enum` cria um conjunto de constantes inteiras nomeadas, facilitando legibilidade e organização de valores discretos.

19 União (union)

```

#include <stdio.h>

/* Definição de union */
union Data {
    int i;
    float f;
    char str[20];
};

int main() {
    union Data data;
    data.i = 10;
    printf("Inteiro: %d\n", data.i);

    data.f = 3.14;
    printf("Float: %.2f\n", data.f);

    return 0;
}

```

Explicação: `union` permite armazenar diferentes tipos de dados no **mesmo espaço de memória**, mas somente um valor pode ser usado de cada vez.

20 Boas práticas de programação em C

```

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

/* Funções pequenas e bem definidas */
int add(int a, int b) {
    return a + b;
}

/* Função que verifica ponteiro nulo */
void print_message(const char *msg) {
    if(msg == NULL) {
        printf("Mensagem vazia!\n");
        return;
    }
    printf("%s\n", msg);
}

/* Uso de constantes e enum para legibilidade */
#define MAX_LENGTH 100
enum Status { SUCCESS, ERROR };

int main() {
    /* Declaração clara e inicialização de variáveis */

```

```

int numbers[MAX_LENGTH] = {0};
enum Status status = SUCCESS;

/* Modularização: uso de funções separadas para tarefas */
print_message("Iniciando programa...");

numbers[0] = add(10, 20);

/* Validação de entrada e checagem de erros */
if(numbers[0] < 0) {
    status = ERROR;
    print_message("Erro: número negativo.");
}

/* Liberação de memória alocada dinamicamente */
int *dynamic_array = (int*)malloc(5 * sizeof(int));
if(dynamic_array == NULL) {
    print_message("Falha ao alocar memória.");
    return 1;
}
free(dynamic_array);

/* Uso de comentários claros e significativos */
printf("Número calculado: %d\n", numbers[0]);

return 0;
}

```

Explicação: Boas práticas incluem:

- Funções pequenas e com responsabilidade única.
- Declaração e inicialização claras de variáveis.
- Uso de constantes, enums e macros para legibilidade.
- Modularização e reutilização de código.
- Validação de entradas e tratamento de erros.
- Liberação de memória alocada dinamicamente.
- Comentários claros e significativos.
- Evitar códigos complexos e duplicados, priorizando clareza e manutenção.

2.1 Controle de fluxo avançado: `goto`, `break`, `continue`

```

#include <stdio.h>

int main() {
    /* Exemplo de break e continue em loop */
    for(int i = 1; i <= 5; i++) {
        if(i == 3) continue; /* pula a iteração quando i = 3 */
        if(i == 5) break;     /* sai do loop quando i = 5 */
        printf("Loop: %d\n", i);
    }
}

```



```
}

/* Exemplo de goto */
int x = 0;
start:
    if(x >= 3) goto end;      /* pula para o label end */
    printf("Goto loop: %d\n", x);
    x++;
    goto start;
end:
    printf("Fim do programa.\n");

    return 0;
}
```

Explicação:

- **break** encerra o loop imediatamente.
- **continue** pula para a próxima iteração do loop.
- **goto** desvia o fluxo para um **label** definido, mas seu uso deve ser **evitado** em código moderno por reduzir legibilidade.