Ponteiros na Linguagem C

1. Introdução

A linguagem C é amplamente utilizada em desenvolvimento de sistemas, drivers e aplicações de baixo nível devido ao seu alto desempenho e controle direto da memória. Um dos conceitos fundamentais em C são os **ponteiros**, que permitem manipular endereços de memória diretamente, fornecendo grande flexibilidade e eficiência.

Neste texto, exploraremos em detalhes o conceito de ponteiros, sua sintaxe, operações, aplicações e boas práticas.

---

2. O Que São Ponteiros?

Um **ponteiro** é uma variável que armazena um endereço de memória. Diferente de variáveis comuns que guardam valores, os ponteiros armazenam a localização desses valores na memória RAM.

Declaração e Inicialização de Ponteiros

A sintaxe básica para declarar um ponteiro é:

```
tipo *nome_do_ponteiro;
```

Exemplo:

```
int *ptr;  // Declara um ponteiro para inteiro
```

Um ponteiro **deve ser inicializado** antes do uso. Isso pode ser feito de duas maneiras:

1. **Apontando para uma variável existente**

   ```
   int x = 10;
   int *ptr = &x; // ptr armazena o endereço de x
   ```

2. **Alocando memória dinamicamente (explicado posteriormente)**

Operadores Relacionados a Ponteiros

Operador	Descrição
&	Operador de endereço: retorna o endereço de uma variável

Operador	Descrição
`*`	Operador de desreferência: acesse o valor armazenado no endereço

Exemplo ilustrativo:

```c
#include <stdio.h>

int main() {
    int x = 42;
    int *ptr = &x;

    printf("Valor de x: %d\n", x);
    printf("Endereço de x: %p\n", &x);
    printf("Valor armazenado em ptr: %p\n", ptr);
    printf("Valor apontado por ptr: %d\n", *ptr);

    return 0;
}
```

Saída esperada (o endereço de memória pode variar):

```
Valor de x: 42
Endereço de x: 0x7ffeefbff5dc
Valor armazenado em ptr: 0x7ffeefbff5dc
Valor apontado por ptr: 42
```

---

3. Tipos de Ponteiros

3.1 Ponteiros Nulos

Um ponteiro pode ser inicializado com `NULL`, o que significa que ele não aponta para nenhum endereço válido:

```c
int *ptr = NULL;
```

A verificação de ponteiros nulos antes de usá-los é uma boa prática para evitar falhas de segmentação (segmentation faults):

```c
if (ptr != NULL) {
    printf("%d", *ptr);
}
```

3.2 Ponteiros e Arrays

Em C, o nome de um array já é um ponteiro para o primeiro elemento:

```c
int arr[] = {10, 20, 30};
int *ptr = arr;  // ptr aponta para arr[0]

printf("%d\n", *(ptr + 1));  // Acessa arr[1]
```

O acesso a elementos de um array por meio de um ponteiro é equivalente à indexação tradicional:

```c
printf("%d\n", arr[1]);  // Equivalente a *(arr + 1)
```

3.3 Ponteiros e Alocação Dinâmica

O uso de `malloc()`, `calloc()` e `free()` permite alocar e liberar memória dinamicamente.

Exemplo:

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

int main() {
    int *ptr = (int *)malloc(sizeof(int));  // Aloca memória para um
inteiro

    if (ptr == NULL) {
        printf("Erro na alocação de memória!\n");
        return 1;
    }

    *ptr = 50;  // Armazena um valor na memória alocada
    printf("Valor armazenado: %d\n", *ptr);

    free(ptr);  // Libera a memória alocada
    return 0;
}
```

3.4 Ponteiros para Ponteiros

É possível criar um ponteiro que aponta para outro ponteiro, útil em manipulação de strings e alocação dinâmica.

```c
int x = 100;
int *ptr = &x;
int **ptr2 = &ptr;

printf("%d\n", **ptr2);  // Saída: 100
```

3.5 Ponteiros para Funções

Ponteiros podem armazenar o endereço de funções, permitindo a criação de callbacks.

```c
#include <stdio.h>

void mensagem() {
    printf("Olá, mundo!\n");
}

int main() {
    void (*func_ptr)() = mensagem;
    func_ptr();  // Chama a função através do ponteiro
    return 0;
}
```

---

4. Ponteiros e Estruturas (Structs)

Ponteiros são frequentemente usados com `structs` para criar estruturas dinâmicas.

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

typedef struct {
    int id;
    char nome[20];
} Aluno;

int main() {
    Aluno *a = (Aluno *)malloc(sizeof(Aluno));

    a->id = 1;
    printf("ID do aluno: %d\n", a->id);

    free(a);
    return 0;
}
```

---

5. Erros Comuns e Boas Práticas

5.1 Falha ao Inicializar Ponteiros

```
int *ptr; // Ponteiro não inicializado
*ptr = 10; // Erro! Acessando memória inválida
```

Correção: Sempre inicialize ponteiros.

5.2 Acesso a Memória Liberada

```
int *ptr = (int *)malloc(sizeof(int));
free(ptr);
printf("%d", *ptr);  // Erro: acessando memória já liberada
```

Correção: Após `free()`, defina o ponteiro como `NULL`.

```
free(ptr);
ptr = NULL;
```

5.3 Memory Leak (Fuga de Memória)

Ocorre quando a memória alocada dinamicamente não é liberada.

```
int *ptr = (int *)malloc(10 * sizeof(int));
// Esquecer de usar free(ptr) antes do fim do programa
```

Correção: Sempre use `free()` após o uso.

Exemplos com Ponteiros

Strings em C são arrays de caracteres terminados pelo caractere nulo (`\0`). Como os arrays são essencialmente ponteiros, podemos manipulá-los diretamente com ponteiros.

Exemplo: Imprimir caracteres de uma string usando ponteiros

```
#include <stdio.h>

int main() {
```

```c
    char str[] = "Ponteiros";
    char *ptr = str;  // Ponteiro para a primeira posição da string

    while (*ptr != '\0') {  // Percorre a string até encontrar o
caractere nulo
        printf("%c ", *ptr);
        ptr++;  // Avança para o próximo caractere
    }

    return 0;
}
```

Saída esperada:

```
P o n t e i r o s
```

> O ponteiro `ptr` é inicializado com o endereço do primeiro caractere da string e avança uma posição a cada iteração.

---

2. Passagem de Arrays para Funções Usando Ponteiros

Em C, quando passamos um array para uma função, na verdade estamos passando um ponteiro para o primeiro elemento do array.

Exemplo: Função que calcula a soma dos elementos de um array

```c
#include <stdio.h>

int soma(int *arr, int tamanho) {
    int total = 0;
    for (int i = 0; i < tamanho; i++) {
        total += *(arr + i);  // Acessando os elementos usando
aritmética de ponteiros
    }
    return total;
}

int main() {
    int numeros[] = {1, 2, 3, 4, 5};
    int resultado = soma(numeros, 5);

    printf("Soma dos elementos: %d\n", resultado);
    return 0;
}
```

Saída esperada:

```
Soma dos elementos: 15
```

> O array `numeros` é passado para a função `soma()`, que o trata como um ponteiro.

---

3. Ponteiros e Alocação Dinâmica com `malloc()`

A alocação dinâmica permite criar variáveis cujo tamanho é determinado em tempo de execução.

Exemplo: Criando um array dinamicamente

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

int main() {
    int *ptr;
    int n;

    printf("Digite o número de elementos: ");
    scanf("%d", &n);

    ptr = (int *)malloc(n * sizeof(int)); // Aloca memória para 'n' inteiros

    if (ptr == NULL) {
        printf("Erro na alocação de memória.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        ptr[i] = i + 1;  // Atribuindo valores ao array dinâmico
    }

    printf("Elementos alocados: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }

    free(ptr); // Liberando a memória alocada

    return 0;
}
```

Saída esperada (para entrada `5`):

```
Digite o número de elementos: 5
Elementos alocados: 1 2 3 4 5
```

> A mémoria é alocada dinamicamente com `malloc()` e liberada no final com `free()`.

---

4. Ponteiros para Ponteiros

Podemos armazenar o endereço de um ponteiro dentro de outro ponteiro. Isso é útil para modificar variáveis dentro de funções.

Exemplo: Modificando uma variável com um ponteiro para ponteiro

```c
#include <stdio.h>

void modificar(int **ptr) {
    **ptr = 50;  // Modifica o valor do inteiro original
}

int main() {
    int num = 10;
    int *ptr = &num;
    int **ptr2 = &ptr; // Ponteiro para ponteiro

    printf("Antes: %d\n", num);
    modificar(ptr2);
    printf("Depois: %d\n", num);

    return 0;
}
```

Saída esperada:

```
Antes: 10
Depois: 50
```

> `ptr2` armazena o endereço de `ptr`, permitindo que a função `modificar()` altere `num` indiretamente.

---

5. Estruturas e Ponteiros

Podemos usar ponteiros para acessar e manipular estruturas (`structs`).

Exemplo: Manipulando `struct` com ponteiros

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

```c
struct Pessoa {
    char nome[50];
    int idade;
};

void imprimir(struct Pessoa *p) {
    printf("Nome: %s\n", p->nome);
    printf("Idade: %d\n", p->idade);
}

int main() {
    struct Pessoa p1 = {"Carlos", 25};
    imprimir(&p1);

    return 0;
}
```

Saída esperada:

```
Nome: Carlos
Idade: 25
```

> Usamos `p->campo` para acessar os membros da estrutura via ponteiro.

---

6. Ponteiros para Funções

Ponteiros podem armazenar endereços de funções, permitindo chamadas dinâmicas.

Exemplo: Ponteiro para função

```c
#include <stdio.h>

void mensagem() {
    printf("Olá, mundo!\n");
}

int main() {
    void (*func_ptr)(); // Declara um ponteiro para função
    func_ptr = mensagem; // Armazena o endereço da função

    func_ptr(); // Chama a função através do ponteiro
    return 0;
}
```

Saída esperada:

```
Olá, mundo!
```

> Esse recurso é útil para callbacks e funções genéricas.

Alocação de Memória Dinâmica em C: `malloc`, `calloc` e `free`

Na linguagem C, a alocação de memória dinâmica permite que um programa solicite e libere memória durante a execução. Isso é útil para lidar com estruturas de dados de tamanho variável, como listas, árvores e matrizes, onde o tamanho pode não ser conhecido previamente.

Os principais métodos para alocação dinâmica são:

- `malloc` (Memory Allocation)
- `calloc` (Contiguous Allocation)
- `free` (Liberar memória alocada)

🔷 `malloc()` – Alocação de Memória Simples

A função `malloc` aloca um bloco de memória de tamanho especificado e retorna um ponteiro para o primeiro byte desse bloco. A memória alocada **não é inicializada**, ou seja, pode conter lixo.

Sintaxe:

```
void *malloc(size_t tamanho);
```

- `size_t` é um tipo de dado sem sinal usado para representar tamanhos de objetos.
- Retorna um ponteiro `void *`, que deve ser convertido para o tipo adequado.
- Se a alocação falhar, retorna `NULL`.

Exemplo:

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

int main() {
    int *ptr;

    // Aloca memória para 5 inteiros
    ptr = (int *) malloc(5 * sizeof(int));

    if (ptr == NULL) {
        printf("Falha na alocação de memória.\n");
        return 1;
    }
```

```c
    // Preenche os valores
    for (int i = 0; i < 5; i++)
        ptr[i] = i + 1;

    // Exibe os valores armazenados
    for (int i = 0; i < 5; i++)
        printf("%d ", ptr[i]);  // Pode conter valores lixo

    free(ptr); // Libera a memória alocada

    return 0;
}
```

🚨 Cuidados:

1. **A memória não é inicializada:** pode conter valores aleatórios (lixo).
2. **Se `malloc` falhar, retorna `NULL`**, então sempre verifique a alocação antes de usar a memória.
3. **Deve ser liberada com `free`** para evitar vazamento de memória.

🔷 `calloc()` – Alocação Contínua com Inicialização

A função `calloc` também aloca memória dinamicamente, mas tem duas diferenças principais em relação ao `malloc`:

1. **Zera a memória alocada** (preenche com `0`).
2. **Aceita dois argumentos:** número de elementos e tamanho de cada elemento.

Sintaxe:

```c
void *calloc(size_t num, size_t tamanho);
```

- `num`: número de elementos a serem alocados.
- `tamanho`: tamanho de cada elemento em bytes.

Exemplo:

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

int main() {
    int *ptr;

    // Aloca memória para 5 inteiros e inicializa com zero
    ptr = (int *) calloc(5, sizeof(int));
```

```c
    if (ptr == NULL) {
        printf("Falha na alocação de memória.\n");
        return 1;
    }

    // Exibe os valores (todos inicializados como 0)
    for (int i = 0; i < 5; i++)
        printf("%d ", ptr[i]); // Sempre 0

    free(ptr); // Libera a memória alocada

    return 0;
}
```

🚨 Cuidados:

1. **Mais seguro que `malloc`** porque inicializa a memória com zero.
2. **Pode ser mais lento** do que `malloc`, pois faz a inicialização dos bytes.
3. **Também deve ser liberado com `free`**.

🔷 `free()` – Liberando Memória

A função `free` é usada para liberar a memória alocada por `malloc` ou `calloc`, evitando vazamento de memória.

Sintaxe:

```c
void free(void *ptr);
```

- `ptr` deve ser um ponteiro previamente alocado por `malloc` ou `calloc`.
- Se `ptr` for `NULL`, `free` não faz nada.

Exemplo:

```c
#include <stdlib.h>

int main() {
    int *ptr = (int *) malloc(10 * sizeof(int));

    if (ptr == NULL)
        return 1;

    free(ptr); // Libera a memória

    return 0;
}
```

🚨 Cuidados:

1. **Após `free(ptr)`, o ponteiro ainda pode conter o endereço antigo**, então é comum definir `ptr = NULL` para evitar acesso acidental:

```
free(ptr);
ptr = NULL;
```

2. **Evite "double free"**, ou seja, liberar a mesma memória duas vezes:

```
free(ptr);
free(ptr); // ERRO! Comportamento indefinido.
```

3. **Não use memória após `free`**:

```
free(ptr);
printf("%d", ptr[0]); // ERRO! Acessando memória já liberada.
```

🔷 Comparação Entre `malloc` e `calloc`

Função	Inicializa Memória?	Parâmetros	Melhor para
`malloc`	❌ Não	`malloc(tamanho)`	Alocar memória rapidamente sem necessidade de inicialização.
`calloc`	✅ Sim (com zeros)	`calloc(n, tamanho)`	Alocar memória inicializada com zero.

🔷 Resumo e Boas Práticas

1. **Use `malloc` se não precisar de inicialização.**
2. **Use `calloc` se precisar que os valores iniciem em zero.**
3. **Sempre verifique se a alocação foi bem-sucedida (`NULL`).**
4. **Sempre libere a memória com `free` quando não for mais necessária.**
5. **Evite acessar memória após `free`.**
6. **Evite vazamentos de memória, principalmente em loops ou funções que alocam dinamicamente.**

Dicas Finais:

- Sempre inicialize ponteiros antes de usá-los.
- Verifique se `malloc()` retornou `NULL` antes de acessar a memória.
- Após `free()`, defina o ponteiro como `NULL` para evitar acessos inválidos.
- Use `sizeof()` para evitar alocações incorretas.
- Compreenda bem a aritmética de ponteiros para evitar acessos fora dos limites do array.

6. Conclusão

Ponteiros são uma ferramenta poderosa na linguagem C, permitindo manipulação eficiente da memória, alocação dinâmica e otimização do desempenho. No entanto, seu uso incorreto pode levar a erros graves, como **acesso inválido à memória e vazamento de memória**. Dominar ponteiros é essencial para programadores que trabalham com sistemas embarcados, desenvolvimento de drivers e aplicações de alto desempenho.

Os exemplos apresentados mostraram diferentes aplicações de ponteiros, incluindo:

- Manipulação de strings e arrays
- Passagem de parâmetros por referência
- Alocação dinâmica de memória
- Ponteiros para ponteiros
- Estruturas com ponteiros
- Ponteiros para funções

A prática é essencial para dominar ponteiros e evitar erros comuns como **acessar memória inválida** ou **não liberar memória alocada dinamicamente**.

Strings na Linguagem C: Uma Explicação Aprofundada

Em C, **strings** não são tipos de dados nativos como em linguagens de alto nível (Python, Java, etc.), mas sim representadas como **arrays de caracteres** terminados pelo caractere nulo (`\0`). Essa abordagem traz flexibilidade, mas também exige um gerenciamento manual da memória e da manipulação dos dados.

---

1. Representação de Strings em C

Uma string em C é, essencialmente, um **array de caracteres** que termina com o caractere `\0`, que indica o final da string.

Declaração de Strings

Podemos declarar strings de três formas principais:

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

```c
int main() {
    // 1. String como array de caracteres (inicialização automática)
    char str1[] = "Olá, Mundo!";

    // 2. String como array de caracteres (inicialização manual)
    char str2[] = {'O', 'l', 'á', ',', ' ', 'C', '!', '\0'};

    // 3. Ponteiro para string (string constante armazenada na memória
de leitura)
    char *str3 = "C é incrível!";

    printf("%s\n", str1);
    printf("%s\n", str2);
    printf("%s\n", str3);

    return 0;
}
```

Saída esperada:

```
Olá, Mundo!
Olá, C!
C é incrível!
```

> No terceiro caso (`char *str3`), a string é armazenada em uma região de memória somente leitura, então modificá-la diretamente pode causar um erro.

---

2. Manipulação de Strings

A linguagem C não fornece um tipo de dado `string` embutido como outras linguagens, mas a biblioteca `<string.h>` contém funções úteis para manipulá-las.

2.1. Cálculo do Tamanho da String

A função `strlen()` retorna o número de caracteres antes do caractere `\0`.

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

int main() {
    char texto[] = "Programação em C";
    printf("Tamanho da string: %lu\n", strlen(texto));

    return 0;
}
```

Saída esperada:

```
Tamanho da string: 17
```

> O `strlen()` **não** conta o caractere `\0`.

2.2. Cópia de Strings (`strcpy` e `strncpy`)

A função `strcpy()` copia uma string para outra.

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

int main() {
    char origem[] = "C é poderoso!";
    char destino[50];

    strcpy(destino, origem);  // Copia a string origem para destino

    printf("String copiada: %s\n", destino);
    return 0;
}
```

Saída esperada:

```
String copiada: C é poderoso!
```

> **Cuidado com `strcpy()`**, pois se a string de origem for maior do que o tamanho do destino, ocorrerá um **estouro de buffer**. Para evitar isso, usamos `strncpy()`:

```c
strncpy(destino, origem, sizeof(destino) - 1);
destino[sizeof(destino) - 1] = '\0';  // Garante terminação correta
```

2.3. Concatenação de Strings (`strcat` e `strncat`)

A função `strcat()` concatena duas strings.

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

int main() {
```

```c
    char saudacao[50] = "Olá, ";
    char nome[] = "Luis";

    strcat(saudacao, nome);

    printf("String final: %s\n", saudacao);
    return 0;
}
```

Saída esperada:

```
String final: Olá, Luis
```

> Para evitar estouro de buffer, utilize `strncat()`.

2.4. Comparação de Strings (`strcmp`)

Para comparar strings, usamos `strcmp()`.

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

int main() {
    char palavra1[] = "C";
    char palavra2[] = "Java";

    if (strcmp(palavra1, palavra2) == 0)
        printf("As strings são iguais\n");
    else
        printf("As strings são diferentes\n");

    return 0;
}
```

Saída esperada:

```
As strings são diferentes
```

> Retorno de `strcmp()`:

- `0`: strings são iguais.
- `< 0`: a primeira string é menor que a segunda.
- `> 0`: a primeira string é maior que a segunda.

2.5. Busca de Caracteres (`strchr`) e Substrings (`strstr`)

A função `strchr()` localiza um caractere dentro da string.

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

int main() {
    char frase[] = "Aprendendo C";
    char *resultado = strchr(frase, 'C');

    if (resultado)
        printf("Caractere encontrado na posição: %ld\n", resultado -
frase);
    else
        printf("Caractere não encontrado.\n");

    return 0;
}
```

Saída esperada:

```
Caractere encontrado na posição: 10
```

> `strstr()` pode ser usada para localizar substrings.

---

3. Manipulação Manual de Strings

3.1. Copiar String sem `strcpy()`

```
#include <stdio.h>

void copiar_string(char *dest, const char *orig) {
    while (*orig) {
        *dest = *orig;
        dest++;
        orig++;
    }
    *dest = '\0';  // Finaliza com o caractere nulo
}

int main() {
    char origem[] = "C manual!";
    char destino[20];

    copiar_string(destino, origem);
```

```
    printf("Cópia manual: %s\n", destino);

    return 0;
}
```

---

4. Alocação Dinâmica de Strings

Strings podem ser armazenadas dinamicamente usando `malloc()`.

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

int main() {
    char *dinamico;
    dinamico = (char *)malloc(50 * sizeof(char));

    if (dinamico == NULL) {
        printf("Erro de alocação de memória\n");
        return 1;
    }

    strcpy(dinamico, "Alocação dinâmica de strings!");
    printf("%s\n", dinamico);

    free(dinamico);  // Libera a memória alocada

    return 0;
}
```

> **Cuidado!** Sempre libere a memória com `free()` para evitar vazamentos.

---

5. Strings e Ponteiros

Strings podem ser manipuladas eficientemente com ponteiros.

```
#include <stdio.h>

void imprime_string(char *str) {
    while (*str) {
        printf("%c", *str);
        str++;  // Avança o ponteiro
    }
    printf("\n");
}
```

```c
int main() {
    char frase[] = "Ponteiros!";
    imprime_string(frase);

    return 0;
}
```

> O ponteiro `str` percorre a string sem precisar de um índice.

Exemplo do uso de `<string.h>`

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

int main() {
    // 1. strlen - Calcula o tamanho da string
    char str1[] = "Programação em C";
    printf("Tamanho de '%s': %lu\n\n", str1, strlen(str1));

    // 2. strcpy - Copia uma string para outra
    char destino[50];
    strcpy(destino, str1);
    printf("String copiada: %s\n\n", destino);

    // 3. strncpy - Copia parte da string, evitando overflow
    char origem[] = "C é rápido!";
    char destino2[10];
    strncpy(destino2, origem, sizeof(destino2) - 1);
    destino2[sizeof(destino2) - 1] = '\0';  // Garante terminação
correta
    printf("String copiada parcialmente: %s\n\n", destino2);

    // 4. strcat - Concatena strings
    char saudacao[30] = "Olá, ";
    strcat(saudacao, "Mundo!");
    printf("String concatenada: %s\n\n", saudacao);

    // 5. strncat - Concatena parte de uma string
    char str2[30] = "Hello, ";
    strncat(str2, "C Programming", 5);
    printf("String concatenada parcialmente: %s\n\n", str2);

    // 6. strcmp - Compara strings (case-sensitive)
    char s1[] = "Teste";
    char s2[] = "teste";
    printf("Comparação '%s' vs '%s': %d\n\n", s1, s2, strcmp(s1, s2));

    // 7. strncmp - Compara parte da string
```

```c
    printf("Comparação parcial: %d\n\n", strncmp(s1, s2, 4));

    // 8. strchr - Busca um caractere na string
    char *resultado = strchr(saudacao, 'M');
    if (resultado) {
        printf("Caractere encontrado: %c na posição %ld\n\n",
*resultado, resultado - saudacao);
    }

    // 9. strrchr - Busca última ocorrência de um caractere
    char frase[] = "banana";
    char *ultima_ocorrencia = strrchr(frase, 'a');
    if (ultima_ocorrencia) {
        printf("Última ocorrência de 'a': posição %ld\n\n",
ultima_ocorrencia - frase);
    }

    // 10. strstr - Busca substring dentro de string
    char texto[] = "Aprendendo C é divertido!";
    char *sub = strstr(texto, "C é");
    if (sub) {
        printf("Substring encontrada: %s\n\n", sub);
    }

    // 11. strtok - Divide string em tokens
    char frase2[] = "C;Python;Java;JavaScript";
    char *token = strtok(frase2, ";");
    printf("Tokens:\n");
    while (token != NULL) {
        printf(" - %s\n", token);
        token = strtok(NULL, ";");
    }
    printf("\n");

    // 12. memset - Preenche um bloco de memória com um valor
    char buffer[20];
    memset(buffer, '-', sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0';
    printf("Memória preenchida: %s\n\n", buffer);

    // 13. memcpy - Copia um bloco de memória
    char origem_mem[] = "Memória copiada!";
    char destino_mem[20];
    memcpy(destino_mem, origem_mem, strlen(origem_mem) + 1);
    printf("Memória copiada: %s\n\n", destino_mem);

    // 14. memmove - Move memória sem sobreposição de dados
    char sobreposicao[] = "123456";
    memmove(sobreposicao + 2, sobreposicao, 4);
    printf("Memória movida: %s\n\n", sobreposicao);

    // 15. memcmp - Compara blocos de memória
    char bloco1[] = "ABC";
```

```
    char bloco2[] = "ABD";
    int resultado_cmp = memcmp(bloco1, bloco2, 3);
    printf("Comparação de memória: %d\n\n", resultado_cmp);

    return 0;
}
```

Explicação de Cada Função

Manipulação de Strings

Função	Descrição
`strlen(s)`	Retorna o tamanho da string (sem contar `\0`)
`strcpy(dest, src)`	Copia `src` para `dest` (pode causar overflow)
`strncpy(dest, src, n)`	Copia no máximo `n` caracteres de `src` para `dest`
`strcat(dest, src)`	Concatena `src` ao final de `dest` (precisa de espaço suficiente)
`strncat(dest, src, n)`	Concatena no máximo `n` caracteres de `src` a `dest`

Comparação e Busca

Função	Descrição
`strcmp(s1, s2)`	Compara `s1` e `s2` (distingue maiúsculas e minúsculas)
`strncmp(s1, s2, n)`	Compara apenas os `n` primeiros caracteres
`strchr(s, c)`	Retorna ponteiro para primeira ocorrência de `c` em `s`
`strrchr(s, c)`	Retorna ponteiro para a **última** ocorrência de `c` em `s`
`strstr(s, sub)`	Retorna ponteiro para a primeira ocorrência da substring `sub`

Tokenização e Manipulação de Memória

Função	Descrição
`strtok(s, delim)`	Divide `s` em tokens separados por `delim`
`memset(ptr, valor, tamanho)`	Preenche `tamanho` bytes de `ptr` com `valor`
`memcpy(dest, src, tamanho)`	Copia `tamanho` bytes de `src` para `dest` (rápido, mas pode dar problema com sobreposição)
`memmove(dest, src, tamanho)`	Copia `tamanho` bytes de `src` para `dest`, **garantindo segurança contra sobreposição**

Função	Descrição
`memcmp(ptr1, ptr2, tamanho)`	Compara os `tamanho` bytes de `ptr1` e `ptr2`

Saída Esperada

```
Tamanho de 'Programação em C': 17

String copiada: Programação em C

String copiada parcialmente: C é rápi

String concatenada: Olá, Mundo!

String concatenada parcialmente: Hello, C Pro

Comparação 'Teste' vs 'teste': -32

Comparação parcial: 0

Caractere encontrado: M na posição 5

Última ocorrência de 'a': posição 5

Substring encontrada: C é divertido!

Tokens:
 - C
 - Python
 - Java
 - JavaScript

Memória preenchida: --------------------

Memória copiada: Memória copiada!

Memória movida: 121234

Comparação de memória: -1
```

Esse código cobre **todas as principais funções da `<string.h>`** e mostra como trabalhar com strings em C de maneira eficiente.

Dicas finais:

1. Sempre aloque **espaço suficiente** ao usar `strcpy()` e `strcat()`.
2. Prefira `strncpy()` e `strncat()` para **evitar estouro de buffer**.

3. **Use `memmove()` em vez de `memcpy()`** quando houver risco de sobreposição de memória.

Conclusão

Strings em C exigem um bom entendimento sobre **arrays, ponteiros e gerenciamento de memória**. Para trabalhar com elas de forma segura, siga boas práticas, como:

- **Evitar estouro de buffer** (`strncpy` e `strncat`).
- **Usar `free()` após `malloc()`**.
- **Manipular strings com ponteiros para maior eficiência**.