Ponteiros em C

Conceitos, Aritmética, Alocação Dinâmica e Boas Práticas

02.08.2025

Gabriela Cota, Jéssica Pereira, Lucas Ramos, Mayara Barbosa & Rian Carlos

Conteúdos

1 Uso de &(address-of) e *(dereference).

O que são os operadores & e *?

- `&` (address-of): obtém o endereço de memória de uma variável.
- `*` (dereference): acessa ou modifica o valor armazenado no endereço apontado por um ponteiro.

Essenciais para acesso indireto à memória e para diversas APIs em C.

Passo a passo


```c
int numero = 42;                    // 1) variável comum
printf("valor: %d\n", numero);

printf("endereco: %p\n", (void*)&numero); // 2) &numero -> endereço

int *ponteiro_para_numero;          // 3) declara um ponteiro p/ int
ponteiro_para_numero = &numero;     // 4) armazena o endereço de numero

int lido = *ponteiro_para_numero;  // 5) lê indiretamente (dereference)
printf("lido via ponteiro: %d\n", lido);

*ponteiro_para_numero = 100;        // 6) modifica indiretamente
printf("novo valor de numero: %d\n", numero);
```

- `%p` imprime endereços; converta para `(void*)` por portabilidade.
- `*ponteiro` lê/escreve o valor na posição apontada.

Visualizando a relação


```
                [numero]      [ponteiro_para_numero]
                   42   <-- guarda &numero (um endereço)

*ponteiro_para_numero   ==  numero
&numero                 ==  ponteiro_para_numero (após a atribuição)
```

Saída esperada


```
1. Variável 'numero' declarada.
   - Valor de 'numero': 42
2. Usando o operador '&' (address-of).
   - Endereço de memória de 'numero': 0x...
3. Ponteiro 'ponteiro_para_numero' declarado.
4. O ponteiro agora armazena o endereço de 'numero'.
   - Valor do ponteiro: 0x...
5. Usando o operador '*' (dereference) para LER o valor.
   - Valor no endereço apontado: 42
6. Usando o operador '*' para MODIFICAR o valor.
   - Modificando o valor para 100 via ponteiro: *ponteiro_para_numero = 100;
7. Verificando o valor da variável 'numero' original.
   - Novo valor de 'numero': 100
```

2 Relação entre arrays e ponteiros.

A Conexão Fundamental

Em C, o nome de um array é, na maioria dos contextos, um **ponteiro constante** para o seu primeiro elemento.

Isto significa que `v` e `&v[0]` são equivalentes e apontam para o mesmo endereço de memória.

```c
int v[4] = {25, 50, 75, 100};

// O código abaixo imprimirá o mesmo endereço duas vezes:
printf("Endereco do array (v): %p\n", v);
printf("Endereco do primeiro elemento (&v[0]): %p\n", &v[0]);
```

Acesso: Índice vs. Ponteiro

A notação de colchetes (`v[i]`) é, na verdade, "açúcar sintático" para a aritmética de ponteiros.

A expressão `v[i]` é internamente convertida para `*(v + i)`.

```c
// As duas formas de acesso produzem o mesmo resultado.
for (int i = 0; i < 4; i++) {
    printf(
      "Acesso por indice v[%d] = %d | "
      "Acesso por ponteiro *(v+%d) = %d\n",
      i, v[i], i, *(v+i)
    );
}
```

Saída Esperada

```
v[0] = 25 | *(v+0) = 25
v[1] = 50 | *(v+1) = 50
...
```

Aritmética de Ponteiros

Quando incrementamos um ponteiro, ele não avança 1 byte, mas sim o tamanho do tipo para o qual ele aponta (`sizeof(tipo)`).

- Se `p` é um `int*` no endereço `0x100`, `p+1` aponta para `0x104` (assumindo `sizeof(int)` de 4 bytes).
- Se `p` é um `char*` no endereço `0x100`, `p+1` aponta para `0x101`.

```c
int *p = v; // p aponta para o endereço de v[0]

// (p+i) avança o ponteiro para o próximo elemento.
// *(p+i) acessa o valor nesse endereço.
printf("Endereco: %p | Valor: %d\n", (p+i), *(p+i));
```

Navegando e Modificando com Ponteiros

Podemos percorrer e até modificar os elementos de um array usando apenas um ponteiro.

```c
// Percorrendo o array
int *p;
for (p = v; p < v + 4; p++) {
    printf("Endereco: %p | Valor: %d\n", p, *p);
}

// Modificando valores
*v = 13;        // Altera v[0] para 13
*(v + 1) = 17;  // Altera v[1] para 17
```

Ponto de Atenção

C **não** impede que um ponteiro acesse memória fora dos limites de um array.

```c
int v[4];
int *p = v;

// Isso compila, mas é um erro grave!
// Estamos escrevendo em uma área de memória que não nos pertence.
*(p + 10) = 999;
```

Este é um dos erros mais comuns e perigosos em C, podendo causar falhas de segmentação (**segmentation faults**) ou corrupção de dados.

3 Diferença entre char s[] e const char *.

A Diferença Fundamental: Mutabilidade e Memória

A forma como você declara uma string em C muda onde ela é armazenada e se você pode alterá-la.

- `char s[] = "texto";`
  - Cria um **array** na **stack**.
  - O conteúdo de `"texto"` é **copiado** para este array.
  - O array é **mutável**: você pode alterar seus caracteres.
- `const char *s = "texto";`
  - Cria um **ponteiro** que aponta para a string literal `"texto"`.
  - A string literal é armazenada em uma área de **memória somente leitura**.
  - O conteúdo é **imutável**: tentar alterar a string causa um erro.

Caso 1: `char s[]` (Array Mutável)

Neste caso, `s` é um array na stack, contendo uma **cópia** da string. A modificação é segura e permitida.

```c
#include <stdio.h>

int main() {
    // 's' é um array na stack, uma cópia de "Gabriela".
    char s[] = "Gabriela";
    printf("Original: %s\n", s);

    // Modificar a cópia local é permitido.
    s[0] = 'g';
    printf("Modificado: %s\n", s);

    // sizeof(s) retorna o tamanho do array (7 chars + '\0').
    printf("Tamanho do array: %zu bytes\n", sizeof(s));
    return 0;
}
```

Saída Esperada

A string é modificada com sucesso e o `sizeof` reflete o tamanho total do array.

```
Original: Gabriela
Modificado: gabriela
Tamanho do array: 8 bytes
```

Caso 2: `const char *s` (Ponteiro para Constante)

Aqui, `s` é um ponteiro para uma string literal em memória somente leitura. Tentar modificar o conteúdo resulta em erro.

```c
#include <stdio.h>

int main() {
    // 's' aponta para a string literal em memória somente leitura.
    const char *s = "Gabriela";
    printf("Nome: %s\n", s);

    // TENTATIVA ILEGAL DE MODIFICAÇÃO
    // s[0] = 'g'; // Causa erro de compilação ou falha em execução!

    // sizeof(s) retorna o tamanho do ponteiro, não da string.
    printf("Tamanho do ponteiro: %zu bytes\n", sizeof(s));
    return 0;
}
```

Saída Esperada

A modificação é ilegal. O `sizeof` retorna o tamanho de um ponteiro no sistema (geralmente 4 ou 8 bytes).

```
Nome: Gabriela
Tamanho do ponteiro: 8 bytes
```

Resumo Visual

`char s[] = "abc";`

Stack

`s: | 'a' | 'b' | 'c' | '\0' |`

Array mutável na stack.

`const char *s = "abc";`

Stack

`s: | endereço |`

`|`

`v`

Memória Somente Leitura

`| 'a' | 'b' | 'c' | '\0' |`

Ponteiro para dados imutáveis.

4 Função swap com ponteiros.

Por que usar swap com ponteiros?

- Passagem por referência: a função recebe ENDEREÇOS, não cópias de valores.
- Permite modificar, dentro da função, as variáveis originais do chamador.
- Padrão essencial para manipular dados em C.

Protótipo e funcionamento


```c
// Recebe dois ponteiros para int e troca os valores apontados
void swap(int* ptr_a, int* ptr_b) {
    printf("  [Dentro da função swap]\n");
    printf("  - Endereço recebido em ptr_a: %p\n", (void*)ptr_a);
    printf("  - Endereço recebido em ptr_b: %p\n", (void*)ptr_b);

    int temp = *ptr_a;  // 1) guarda o valor apontado por ptr_a
    *ptr_a = *ptr_b;    // 2) copia o valor apontado por ptr_b para ptr_a
    *ptr_b = temp;      // 3) restaura o valor antigo de ptr_a em ptr_b

    printf("  - Troca realizada dentro da função.\n");
}
```

- `ptr_a` e `ptr_b` são `int*`: acessamos os valores com `*` (dereference).
- Ao modificar `*ptr_a` e `*ptr_b`, alteramos as variáveis originais.

Uso na prática


```c
int valor1 = 10;
int valor2 = 99;

printf("Valores ANTES da troca:\n");
printf(" - valor1 = %d (no endereço %p)\n", valor1, (void*)&valor1);
printf(" - valor2 = %d (no endereço %p)\n\n", valor2, (void*)&valor2);

printf("Chamando swap(&valor1, &valor2) ...\n");
swap(&valor1, &valor2);  // passamos os ENDEREÇOS com '&'

printf("Valores DEPOIS da troca:\n");
printf(" - valor1 = %d\n", valor1);
printf(" - valor2 = %d\n", valor2);
```

Saída esperada


```
Valores ANTES da troca:
 - valor1 = 10 (no endereço 0x...)
 - valor2 = 99 (no endereço 0x...)

Chamando a função swap() e passando os ENDEREÇOS de valor1 e valor2...
  [Dentro da função swap]
  - Endereço recebido em ptr_a: 0x...
  - Endereço recebido em ptr_b: 0x...
  - Troca realizada dentro da função.
...retornamos para a main.

Valores DEPOIS da troca:
 - valor1 = 99
 - valor2 = 10
```

Pontos de atenção

- Passe os ENDEREÇOS com `&` (ex.: `swap(&a, &b)`), não os valores.
- Não dereferencie ponteiros nulos; valide entradas se necessário.
- Ao imprimir endereços com `%p`, converta para `(void*)` por portabilidade.
- Para outros tipos, ajuste a assinatura (ex.: `void swap_double(double*, double*)`).

5 Função que aloca dinamicamente um array com T (ponteiro para ponteiro).**

O que é uma Matriz Dinâmica?

Em C, uma matriz (array 2D) pode ser representada como um "ponteiro para ponteiro", por exemplo `char**`.

- Diferente de uma matriz estática (`char matriz[5][10]`), suas dimensões (`linhas` e `colunas`) podem ser definidas em **tempo de execução**.
- A memória é alocada na **heap**, não na **stack**.
- Para isso, usamos `malloc` para solicitar a memória e `free` para liberá-la.

A Estrutura: Array de Ponteiros

A ideia central é que uma matriz é um **array de ponteiros**, onde cada ponteiro aponta para uma **linha**.

- `matriz` (`char**`): Ponteiro que aponta para o início de um array de ponteiros.
- `matriz[i]` (`char*`): Cada elemento desse array é um ponteiro que aponta para o início de uma linha (um array de `char`).
- `matriz[i][j]` (`char`): O caractere na linha `i` e coluna `j`.

Alocando a Matriz: Passo a Passo

A alocação ocorre em duas fases:

```c
char **inicia_matriz(int linhas, int colunas) {
    // 1. Aloca um array de ponteiros (um para cada linha).
    char **matriz = malloc(linhas * sizeof(*matriz)); // ou sizeof(char*)
    if (matriz == NULL) { /* Tratar erro */ }

    // 2. Para cada linha, aloca um array de chars (as colunas).
    for (int i = 0; i < linhas; i++) {
        matriz[i] = malloc(colunas * sizeof(*matriz[i])); // ou sizeof(char)
        if (matriz[i] == NULL) { /* Tratar erro */ }

        // Preenche a matriz...

    return matriz;
}
```

Liberando a Memória: O Processo Inverso

Para evitar vazamentos de memória, a memória alocada com `malloc` deve ser liberada com `free`.

O processo é o **inverso** da alocação:

- Liberar cada uma das linhas.
- Liberar o array de ponteiros.

```c
void libera_matriz(char **matriz, int linhas) {
    if (!matriz) return;

    // 1. Libera cada linha (array de colunas).
    for (int i = 0; i < linhas; i++) {
        free(matriz[i]);
    }
    // 2. Libera o array de ponteiros.
    free(matriz);
}
```

Saída Esperada

- **Saída (para 5 linhas e 10 colunas):**

```
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
{L}{L}{L}{L}{L}{L}{L}{L}{L}{L}
```

6 Ponteiro para função: exemplo com `qsort`

A função `qsort`

A função `qsort` da biblioteca padrão (`stdlib.h`) é uma função de ordenação genérica.

- **Genérica** porque pode ordenar arrays de **qualquer tipo de dado**: inteiros, floats, structs, e até mesmo strings (que são arrays de `char`).
- Para conseguir essa flexibilidade, `qsort` não sabe como comparar os elementos do array.
- Nós é que precisamos fornecer a lógica de comparação através de um **ponteiro para função**.

Sua assinatura é:

```c
void qsort(
  void *base,
  size_t num,
  size_t size,
  int (*compar)(const void *, const void *)
);
```

O problema: como ordenar strings?

Temos um array de strings que queremos ordenar em ordem alfabética.

```
char valores_string[5][10] = {
  "Jessica", "Lucas", "Rian", "Gabriela", "Mayara"
};
```

A função `strcmp` (de `string.h`) sabe como comparar duas strings. Mas sua assinatura é `int strcmp(const char *, const char *)`.

A `qsort` espera uma função com a assinatura `int (*compar)(const void *, const void *)`.

Não podemos passar `strcmp` diretamente para `qsort` devido à incompatibilidade de tipos dos ponteiros.

A solução: uma função “wrapper”

Criamos uma função “wrapper” (ou adaptadora) que compatibiliza a `strcmp` com o que a `qsort` espera.

```c
#include <string.h>

// Esta função segue a assinatura exigida por qsort.
int compara_string(void const *ponteiro_a, void const *ponteiro_b) {
    // 1. Converte os ponteiros genéricos (void *)
    //    para o tipo que realmente estamos trabalhando (char *).
    char const *primeira = (char const *)ponteiro_a;
    char const *segunda = (char const *)ponteiro_b;

    // 2. Usa strcmp para fazer a comparação real.
    return strcmp(primeira, segunda);
}
```

Juntando tudo: a chamada da `qsort`

Agora, no `main`, podemos chamar a `qsort` e passar nossa função de comparação.

```c
int main() {
    char valores_string[5][10] = {"Jessica", "Lucas", "Rian", "Gabriela", "Mayara"};

    // ... código para imprimir o array desordenado ...

    qsort(
      valores_string, // 1. O array a ser ordenado.
      5,              // 2. O número de elementos no array.
      10,             // 3. O tamanho de cada elemento (em bytes).
      compara_string  // 4. O ponteiro para nossa função de comparação.
    );

    // ... código para imprimir o array ordenado ...
    return 0;
}
```

Saída Esperada

- **Saída:**

```
Strings desordenadas: Jessica Lucas Rian Gabriela Mayara
Strings ordenadas: Gabriela Jessica Lucas Mayara Rian
```