



Professor
Maromo

LINGUAGEM DE PROGRAMAÇÃO

Material 008



GitHub
maromo71

C



Agenda



Ponteiros em C

- Entendendo e dominando a manipulação de memória"

Material: LP_008

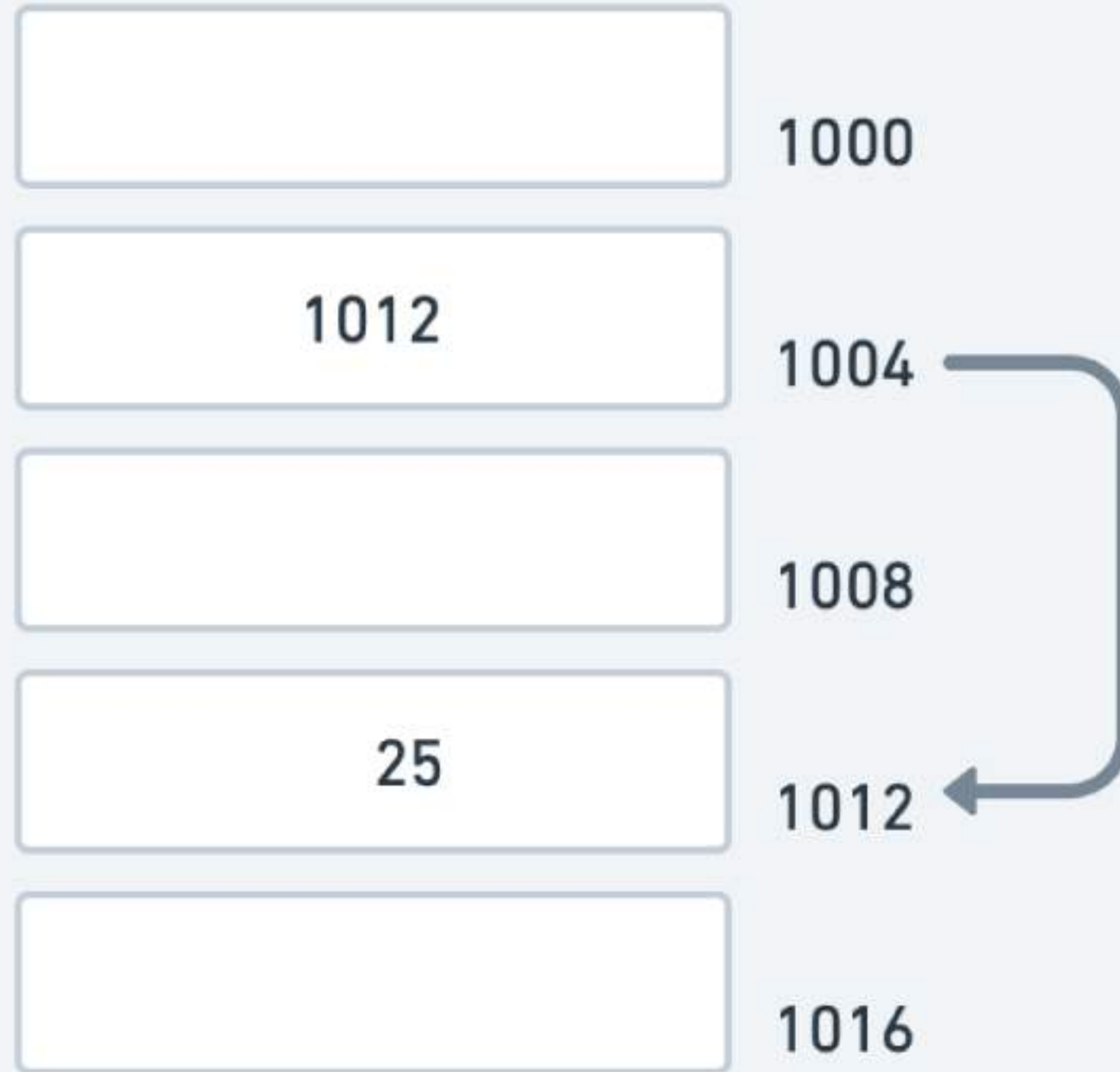
Ponteiros



De acordo com Herbert (1997):

"Um ponteiro é uma variável que armazena o endereço de memória de outra variável. Quando uma variável detém o endereço de outra, diz-se que a primeira 'aponta' para a segunda."

Exemplo



Variáveis – ponteiros

Para declarar uma variável ponteiro, deve-se colocar no tipo base um '*' (asterisco) e o nome da variável.

Sintaxe:

tipo *nome

Onde:

- o tipo pode ser qualquer tipo válido em C;
- e nome é o nome da variável ponteiro.

Operadores Ponteiros

Operadores especiais & e *;

- O & é o operador que devolve o endereço na memória do seu operando.
- O operador * devolve o valor da variável localizada no endereço que o segue. Ou seja:

& = endereço

e

* = conteúdo

Operadores Ponteiros

Faz-se através do operador & (Endereço de).

Exemplo:

```
int main(){  
    int x = 5;  
    float pi = 3.14;  
    int * ptr_x = &x;  
    float * pointer_to_pi = &pi;  
    return 0;  
}
```

NOTA:

Uma boa prática de programação é sempre inicializar ponteiros para evitar problemas futuros

Usando Ponteiros

Um ponteiro serve para acessar outros objetos através dos seus endereços.

```
int a = 5, b = 7;  
int * ptr = NULL;
```

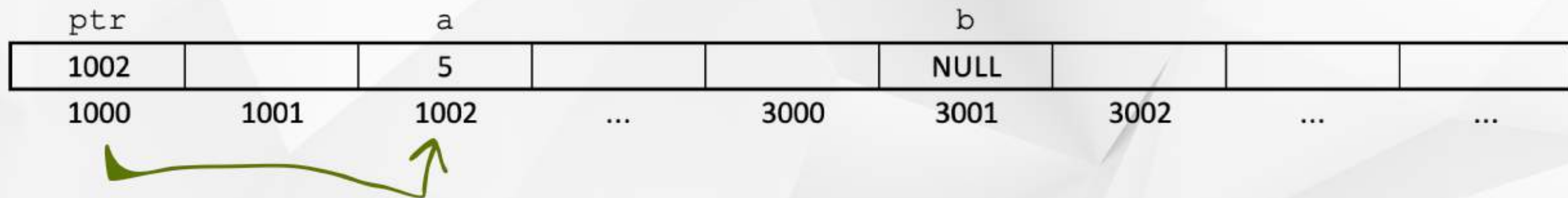
ptr		a		b					
		5		NULL					
1000	1001	1002	...	3000	3001	3002

Para se colocar ***ptr*** apontando para ***a*** faz-se:

ptr = &a

Usando Ponteiros

```
int a = 5, b = 7;  
int * ptr = NULL;
```



Nota:

Se **ptr** é um ponteiro, então ***ptr** nos permite obter o valor que é apontado por **ptr**, isto é, o valor da variável cujo endereço está armazenado em **ptr**.

***ptr** – Deve-se ler "**O APONTADO POR ptr**".

Observe o programa

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

```
int main(){
```

```
    int a = 12;
```

```
    int b = 0;
```

```
    int *pa = NULL;
```

```
    pa = &a;
```

```
    printf("Valor de pa: %d\n", *pa);
```

```
    a = 15;
```

```
    b = *pa;
```

```
    printf("Valor de b: %d\n", b);
```

```
    printf("Valor de a, b, pa e * pa --> %d %d %d %d\n", a,b,pa,*pa);
```

```
    return 0;
```

```
}
```

1000	1001	1002
a	b	pa
12	0	NULL
12	0	1000
15	15	1000

Ponteiros e Tipos de Dados

Ponteiros devem sempre possuir um tipo (e não ser genérico) devido ao fato de as variáveis ocuparem diferentes tamanhos em memória que os ponteiros para essas variáveis terão que saber quantos bytes terão que considerar.

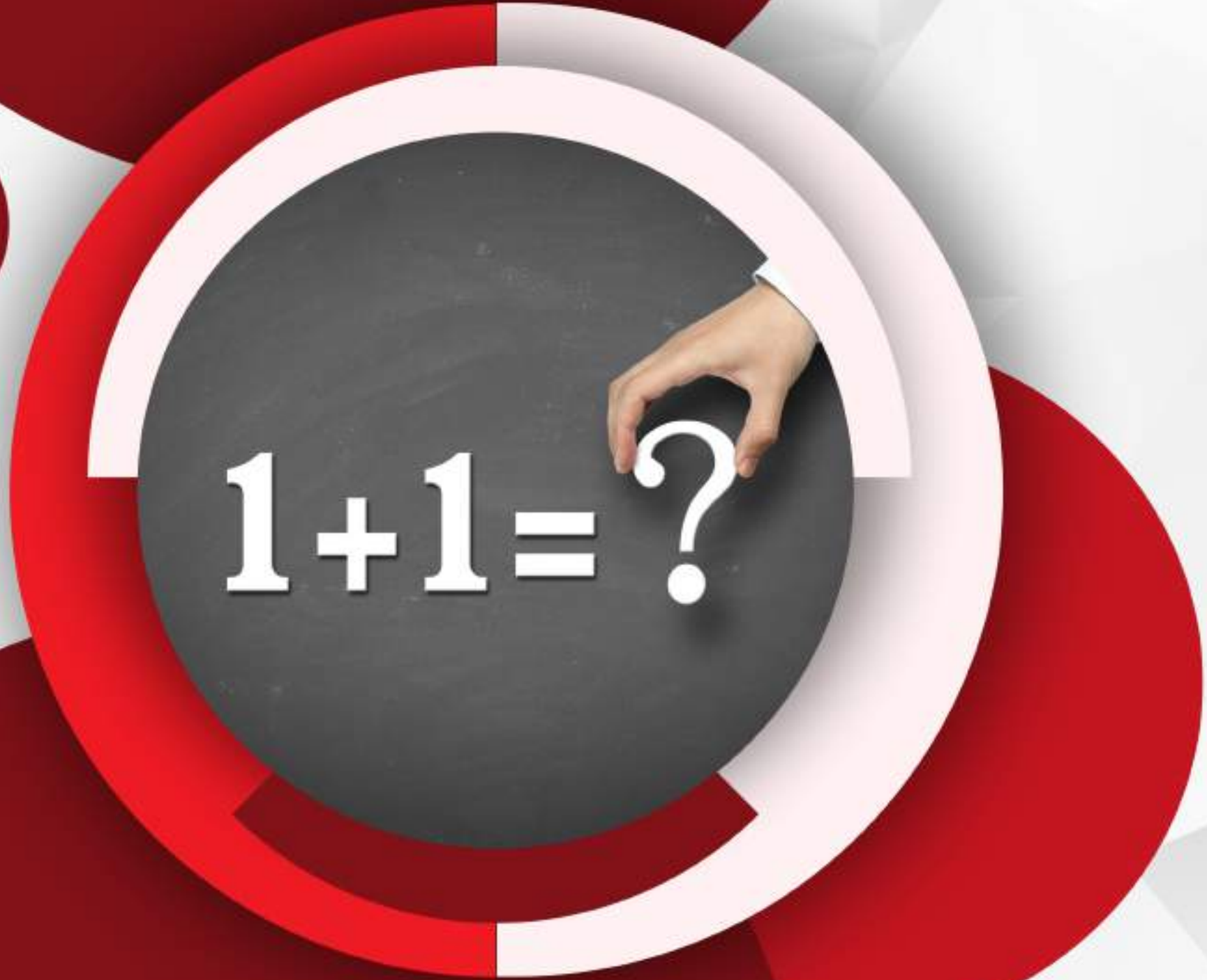
```
char *ptr_a = &a;  
int *ptr_n = &n;  
float *ptr_pi = &pi;
```

A declaração anterior nos diz que o número de *bytes* que irá ser considerado por cada um dos ponteiros é 1, 2 e 4, respectivamente, pois cada ponteiro terá que considerar o número de *bytes* do tipo que aponta.

Operações aritméticas válidas com ponteiros

Operação	Exemplo	Observações
Atribuição	<code>ptr = &x;</code>	Podemos atribuir um valor (endereço) a um ponteiro. Se quisermos que aponte para nada, podemos atribuir-lhe o valor da constante NULL.
Incremento	<code>ptr++;</code>	Incrementa o valor do ponteiro, fazendo-o apontar para o próximo endereço de memória.
Decremento	<code>ptr--;</code>	Decrementa o valor do ponteiro, fazendo-o apontar para o endereço de memória anterior.
Apontado por	<code>*ptr;</code>	Retorna o valor da variável para a qual o ponteiro está apontando.
Endereço de	<code>&x;</code>	Retorna o endereço de memória da variável.
Diferença	<code>ptr1 - ptr2;</code>	Retorna a diferença entre dois ponteiros, que representa a distância entre os dois endereços de memória.
Comparação	<code>ptr1 == ptr2;</code>	Compara se dois ponteiros apontam para o mesmo endereço de memória.

Incremento



1+1=?



Incremento

Um ponteiro pode ser incrementado como uma variável comum. No entanto, ao incrementar um ponteiro, ele avança não apenas um byte, mas sim pelo tamanho do tipo de dado para o qual aponta. Portanto, ao incrementar um ponteiro, seu valor aumenta pelo equivalente ao **sizeof(tipo)** do dado apontado.

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

```
int main(){
```

```
    short x = 5, *px = &x;
```

```
    long y = 5.0, *py = &y;
```

```
    printf("%d %ld\n", x, (long)px);
```

```
    printf("%d %ld\n", x + 1, (long)(px + 1));
```

```
    printf("%ld %ld\n", y, (long)py);
```

```
    printf("%ld %ld\n", y + 1, (long)(py + 1));
```

```
    return 0;
```

```
}
```

Exemplo de Saída

```
5 6162626074
```

```
6 6162626076
```

```
5 6162626056
```

```
6 6162626064
```


O decremento de um ponteiro opera de forma similar ao incremento. Quando um ponteiro do tipo **xxx** é decrementado, ele recua pelo tamanho, ou `sizeof(xxx)`, do tipo ao qual aponta. No exemplo a seguir, exibiremos uma string tanto na ordem original quanto na ordem inversa.

Exemplo do Decremento

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

```
int main(){
```

```
    char s[100];
```

```
    char *ptr = s; /* aponta para o 1o. caractere de s */
```

```
    printf("Introduza uma string: ");
```

```
    fgets(s, sizeof(s), stdin);
```

```
    if (*ptr == '\0') return 0;
```

```
    /*Imprimir string normalmente */
```

```
    while(*ptr!='\0')
```

```
        putchar(*ptr++);
```

```
    printf("\n\n");
```

```
    /*Imprimir ao contrário */
```

```
    ptr--; //por causa do '\0'
```

```
    while(ptr>=s) /*Enquanto ptr for >=que &s[0] */
```

```
        putchar(*ptr--);
```

```
    printf("\n\n");
```

```
    return 0;
```

```
}
```

Resumo:

O programa solicita ao usuário uma string e armazena-a no array **s**. Utilizando o ponteiro **ptr**, ele primeiro imprime a string na **ordem original, caracter por caracter**, e, em seguida, **inverte o ponteiro para imprimir a string na ordem inversa**. O ponteiro **ptr** é essencial para navegar pela string, **tanto na direção normal quanto na reversa**, enquanto a função `putchar()` imprime cada caractere individualmente.

Diferença entre ponteiros

A diferença entre dois ponteiros que apontam para elementos do mesmo tipo de dado **indica a quantidade de elementos que existem entre os endereços desses dois ponteiros na memória**. Isso é possível porque, em C, a **aritmética de ponteiros é baseada no tipo de dado para o qual o ponteiro aponta**. Portanto, ao subtrair um ponteiro de outro, o resultado reflete a diferença em termos de número de elementos e não apenas bytes. Vale ressaltar que só faz sentido e é válido subtrair ponteiros do mesmo tipo, pois a interpretação da diferença seria ambígua ou sem sentido para ponteiros de tipos diferentes.

Diferença entre ponteiros

Exemplo

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

```
int main() {
```

```
    int arr[] = {10, 20, 30, 40, 50};
```

```
    int *ptr1 = &arr[1]; // aponta para o elemento 20
```

```
    int *ptr2 = &arr[4]; // aponta para o elemento 50
```

```
    int diff = ptr2 - ptr1;
```

```
    printf("Diferença entre os ponteiros: %d\n", diff);
```

```
    // A saída será 3, pois existem 3 elementos entre 20 e 50, inclusive.
```

```
    return 0;
```

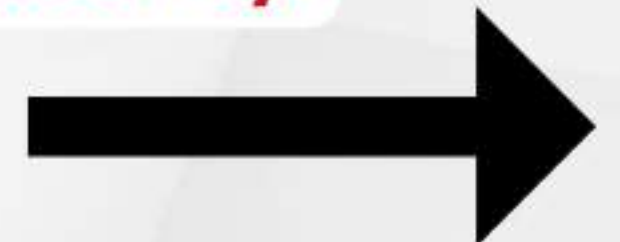
```
}
```

Neste exemplo, a diferença entre **ptr2** e **ptr1** é 3, indicando que há três elementos no array entre os elementos para os quais ptr2 e ptr1 estão apontando (30, 40 e 50).

Comparação

Em C, é possível comparar dois ponteiros do mesmo tipo usando operadores relacionais para determinar a relação entre os endereços de memória que eles armazenam. Esta comparação é especialmente útil quando se trabalha com arrays, onde você pode querer verificar se um ponteiro está antes ou depois de outro no mesmo array, ou se ambos apontam para o mesmo local.

Considere um array de inteiros e dois ponteiros apontando para diferentes posições desse array. Você pode querer verificar a relação entre esses ponteiros em relação ao array:



Comparação

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

```
int main() {
```

```
    int arr[] = {10, 20, 30, 40, 50};
```

```
    int *ptr1 = &arr[1]; // aponta para o elemento 20
```

```
    int *ptr2 = &arr[3]; // aponta para o elemento 40
```

```
    if (ptr1 < ptr2) {
```

```
        printf("ptr1 aponta para um elemento anterior ao elemento apontado por ptr2 no array.\n");
```

```
    }
```

```
    if (ptr1 == &arr[1]) {
```

```
        printf("ptr1 aponta para o segundo elemento do array.\n");
```

```
    }
```

```
    return 0;
```

```
}
```

Neste exemplo, a primeira condição avalia como verdadeira, indicando que ptr1 aponta para um elemento que vem antes do elemento para o qual ptr2 aponta no array. A segunda condição verifica se ptr1 está apontando para o segundo elemento do array, o que também é verdadeiro neste caso.

Desafio 1



Em linguagem C, um string é tipicamente representado como um array de caracteres, terminado pelo caractere nulo ('\\0'). O tamanho de um string é, portanto, o número de caracteres nele antes deste caractere nulo.

Neste exercício, você deve implementar uma função chamada **str_lens** que determina o tamanho de um string. A função deve usar aritmética de ponteiros em vez de acessar o string por índices de array. A função deve ter a seguinte assinatura:

```
int str_lens(char *s);
```



Instruções – Desafio 1



1. A função **str_lens** deve aceitar um ponteiro para um caractere, que aponta para o primeiro caractere de um string.
2. Usando aritmética de ponteiros, navegue pelo string até encontrar o caractere nulo ('**\0**').
3. A função deve retornar a diferença entre o ponteiro que aponta para o caractere nulo e o ponteiro inicial, que aponta para o primeiro caractere do string. Essa diferença é, de fato, o comprimento do string.
4. Teste sua função com diferentes strings para garantir que ela funcione corretamente.

Solução do problema



```
#include <stdio.h>

int str_lens(char *s) {
    char *start = s;
    while (*s != '\0') {
        s++;
    }
    return s - start;
}

int main() {
    char str[] = "Ola aluno"; // String de exemplo.
    printf("O tamanho do string '%s' é: %d\n", str, str_lens(str));
    return 0;
}
```

Notas importantes resumidas



1

O incremento ou decremento de um ponteiro faz com que ele se desloque na memória pelo tamanho do tipo de dado ao qual aponta.

Um ponteiro para int em sistemas de 32 bits, por exemplo, avança ou retrocede 4 bytes ao ser incrementado ou decrementado.

2

3

Ao trabalhar com arrays e memória dinâmica, é crucial entender a aritmética de ponteiros e como os ponteiros se movem na memória.

Ponteiros podem ser usados para verificar a relação espacial de dois elementos em um array ou se dois ponteiros apontam para o mesmo local na memória.

4

5

A diferença entre dois ponteiros indica a quantidade de elementos entre eles, sendo útil para operações em segmentos de arrays.

O valor para o qual um ponteiro aponta pode ser acessado e modificado usando o operador *, afetando diretamente a memória referenciada.

6

Desafios





Exercícios - Aula 08

Ponteiros

Desafio 1

Trocar valores usando ponteiros:

Escreva uma função **swap(int *a, int *b)** que troque os valores das duas variáveis inteiras para as quais os ponteiros a e b apontam.

Teste a função no main para garantir que os valores sejam trocados corretamente.



Exercícios - Aula 08

Ponteiros

Desafio 2

Calcular a média de um array:

Escreva uma função double **average(int *arr, int size)** que retorne a média dos valores de um array de inteiros. O ponteiro arr aponta para o primeiro elemento do array, e size indica o número de elementos no array.

Teste a função no main usando um array de inteiros.



Exercícios - Aula 08

Ponteiros

Desafio 3

Inverter uma string usando ponteiros:

Escreva uma função void **reverseString(char *str)** que inverta uma string in-place (ou seja, sem usar uma string auxiliar). A função deve usar aritmética de ponteiros para acessar os caracteres da string.

Teste a função no main com várias strings para garantir que elas sejam invertidas corretamente.

Exercícios - Aula 08

Resolução Exercício 01

```
#include <stdio.h>

void troca(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int num1 = 5, num2 = 10;
    troca(&num1, &num2);
    printf("%d %d\n", num1, num2);
    return 0;
}
```



Exercícios - Aula 08

Resolução Exercício 02

```
#include <stdio.h>

double media(int *arr, int tamanho) {
    int soma = 0;
    for (int i = 0; i < tamanho; i++) {
        soma += arr[i];
    }
    return (double) soma / tamanho;
}

int main() {
    int valores[] = {1, 2, 3, 4, 5};
    printf("%.2f\n", media(valores, 5));
    return 0;
}
```



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

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

```
void invertString(char *str) {  
    char *inicio = str;  
    char *fim = str + strlen(str) - 1;  
    while (inicio < fim) {  
        char temp = *inicio;  
        *inicio = *fim;  
        *fim = temp;  
        inicio++;  
        fim--;  
    }  
}  
  
int main() {  
    char texto[] = "ola alunos";  
    invertString(texto);  
    printf("%s\n", texto);  
    return 0;  
}
```

Exercícios - Aula 08

Resolução Exercício 03



Referências

DAMAS, L. M. D. Linguagem C. LTC, 2007.

HERBERT, S. C completo e total. 3a. ed. Pearson, 1997.

