PONTEIROS

DEF

Ponteiros são variáveis especiais que guardam **endereços de memória**. E acabou.

```
int x = 10;
int *p = &x;
```

O tipo `int *` especifica ao compilador que é esperado, neste momento, que o ponteiro guarde um endereço de memória de uma variável `int`.

Um ponteiro pode ser inicializado com o endereço de uma variável já criada, sem um endereço específico ou como um ponteiro nulo por meio da keyword `nullptr`, para indicar que um ponteiro não guarda nada ainda.

```
int *p = nullptr;
```

Se você não for atribuir um valor ao ponteiro naquele instante, é preferível inicializá-lo como `nullptr` por trazer clareza de que o ponteiro ainda não foi utilizado e impedir acesso a regiões indevidas do seu computador. Afinal, `int *p` pode explodir o computador, se brincar!

Vimos que cada tipo de dado ocupa uma quantidade diferente de bytes. Porém, os ponteiros são especiais! Por guardarem sempre um endereço de memória, eles sempre ocuparão 4 (arquitetura de 32 bits) ou 8 bytes (arquitetura de 64 bits), independente do que aquele endereço de memória guarda (um inteiro, um caractere, ...)

"Então por que eu preciso especificar o tipo do ponteiro, como em `int *`?"

Veremos isso mais tarde! Seguindo...

Detalhe: um ponteiro guarda um endereço de memória, mas ele próprio tem um endereço de memória. Logo, é possível criar um **ponteiro para ponteiro**, uma prática muito utilizada para criação de matrizes. Exemplo simples de uso:

```
int x = 10;
int *p1 = &x;
int **p2 = &p1; // p2 -> p1 -> x;
std::cout << "valor de p2: " << *p2;
// valor de p2: 0x7fff92679a44020
```

Para criar esse ponteiro, usamos **.

DERREFERENCIAÇÃO

"Certo, consigo guardar um endereço de uma variável em um ponteiro. E pra acessar o valor em si, como faço?"
Muito simples: basta derreferenciá-lo!

```
cout << "valor no ponteiro: " << *p;
// "valor no pointeiro: 10"
```

E só.

É importante não confundir o uso do operador `*`. Ele pode ser aplicado no contexto da **declaração**, como em `int *pointer;`, bem como na **derreferenciação**, como em `*pointer = 54;`; neste exemplo, estamos indo até o endereço que `pointer` guarda (um aleatório na memória, não foi inicializado) e armazenamos nele o valor 54.
Busque ler o operador `*` como "conteúdo de" e `&` como "endereço de".

EXERCÍCIO 1

Imprimir o endereço e o valor de um ponteiro qualquer.

PASSAGEM POR CÓPIA VS POR REFERÊNCIA

Na `stack`, quando uma função é chamada, ela copia os valores passados nos parâmetros, manipula-os e, quando é encerrada, apaga as cópias. Isso é conhecido como **passagem por cópia**. Mas e quanto a um endereço de memória?

Quando passamos um ponteiro para uma função, todas as mudanças feitas nele ocorrem tanto dentro quanto fora da função! Isso acontece porque a função recebe uma cópia do **endereço de memória**, mas as alterações são feitas diretamente no local da memória apontado por esse endereço. Esse conceito é conhecido como **passagem por referência**.

EXEMPLO

```c
void func1(int x, int y)
{
  x = 0;
  y = x + y;
}

void func2(int *x, int *y)
{
  *x = 0;
  *y = *x + *y;
}

int main()
{
  int x = 10;
  int y = 20;
  func1(x, y);   // x = 10, y = 20
  func2(&x, &y); // x = 0, y = 20
}
```

`func1` manipula somente as cópias dos valores. `func2` recebe os dois ponteiros e altera o valor dentro desses endereços!

VETORES E ARITMÉTICA DE PONTEIROS

Quando um vetor é criado e salvo na `stack`, a variável que criamos armazena somente o endereço-base (lembra dele? pois é...). Logo, todo vetor é um ponteiro por debaixo dos panos! Tanto é, que se você fizer isso:

```
int vetor[3] = {10, 20, 30};
int *p = vetor;
```

O compilador não indicará nenhum problema! Não foi necessário usar `&vetor` pois o `vetor` em si já é um ponteiro, só não aparenta ser.

"Mas tá, se eu sei o endereço-base do vetor como eu consigo acessar os outros elementos?"
Por aritmética de ponteiros! Sabendo o endereço-base, a quantidade de elementos do vetor E o tipo do dado (`int`, `char`), nós podemos avançar pelo vetor fazendo:

```
int vetor[3] = {10, 20, 30};
// Isso aqui
cout << "posicao 1: " << *(vetor+1) << "\n";
// Que é a mesma coisa que:
cout << "posicao 1: " << vetor[1] << "\n";
// Ou seja:
// Primeiro pulamos para a posição correta e
// Depois derreferenciamos.
```

Feio né? Fazer o quê, é assim que funciona...

O que aconteceu no código foi que, adicionando 1 ao ponteiro, nós estamos somando 1 "unidade de endereço" ao endereço atual. Não podemos dizer que é 1 byte necessariamente, pois, sabendo que cada tipo de dado ocupa uma quantidade de bytes diferente, nós somamos quantidades diferentes de bytes.
No exemplo atrás, ao fazer `vetor + 1`, estamos somando na verdade 4 bytes ao ponteiro para poder chegar no próximo número. Se `vetor` fosse um ponteiro para `char`, somaríamos 1 byte, e etc.

Lembra que eu disse que precisamos especificar o tipo do ponteiro mesmo que ele sempre ocupe 4 ou 8 bytes? É por causa da aritmética de ponteiros!

```
// Não se preocupe com a função malloc!
void *p = malloc(sizeof(int) * 4);
for (int i = 0; i < 4; i++)
{
  // Será que ele pularia 1 byte para ir ao próximo endereço
  // ou 4 para ir ao próximo número?
  std::cout << "Endereço de p: " << &p[i];
  // Não compila! O C++ é sabido!
}
```

Se não especificarmos o tipo, o compilador não vai saber quantos bytes pular para acessar o próximo número!

Então, querendo ou não, vamos sempre precisar especificar o tipo do ponteiro para podermos executar nossos programas.

```
int *p = (int *) malloc(sizeof(int) * 4);
for (int i = 0; i < 4; i++)
{
  // Agora, o compilador sabe que pra chegar no próximo
  // número, tem que pular 4 bytes!
  std::cout << "Endereço de p: " << &p[i];
}
```

A aritmética de ponteiros sempre esteve presente conosco, nós só não sabíamos!

EXERCÍCIO 2

Imprimir o endereço de memória e valor de cada elemento de um vetor qualquer.

EXERCÍCIO 3

Criar uma função que recebe um ponteiro para um vetor vazio, um número n e aloque os primeiros n naturais de forma crescente. Depois, faça a impressão de cada um deles usando aritmética de ponteiros.

EXERCÍCIO FINAL

Crie um algoritmo capaz de contar caracteres de uma frase. A frase deverá ser impressa pelo usuário e ter, no máximo 200 caracteres. O programa deve imprimir quantos caracteres há na frase, ignorando os espaços em branco.

Agora que vimos como ponteiros se comportam, podemos estudar sua principal utilidade: na alocação dinâmica de dados!

Speaker notes