5950257 - Programação de Computadores

Aula 11 - Ponteiros

Prof. Dr. Denis M. L. Martins
DCM | FFCLRP | USP

Objetivos de Aprendizagem

1. Compreender o modelo de memória da linguagem C e como variáveis são armazenadas.
2. Declarar, inicializar e manipular ponteiros corretamente.
3. Utilizar operadores de endereço (`&`) e de referência (`*`) com segurança.
4. Entender a relação intrínseca entre arrays e ponteiros em C.
5. Compreender como funciona alocação dinâmica de memória em C.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Motivação

Vamos escrever um programa que:

- inicializa lê dois valores inteiros e armazena o primeiro valor em **int x** e o segundo em **int y**
- troque os valores das variáveis.
- imprima o valor das variáveis após a troca.

Dica: use uma variável temporária **int temp** para guardar o valor original de uma das variáveis antes da troca.

> Ao longo dessa aula, vamos utilizar o C Tutor para visualizar como nossos programas alteram a memória.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Relembrando: Modelo de Memória

A memória RAM é modelada como um **array linear de bytes endereçáveis**.

- Cada variável ocupa um endereço base contíguo.
- O tamanho depende do tipo.

Exemplo (contém conceitos que veremos nos próximos slides):

```c
int x = 42;
printf("Endereço de x: %p\n", (void*)&x);
printf("Tamanho de int: %zu bytes\n", sizeof(int));
```

> Operador `&` (Address-of): Retorna o endereço de memória de uma variável.
>
> Note que endereços são impressos em hexadecimal.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

O Que é um Ponteiro?

Um ponteiro é uma variável que **armazena o endereço de memória** de outra variável. **Sintaxe:** **<tipo> *<identificador>;**

- O tipo do ponteiro deve indicar o tamanho dos dados que ele aponta (ex: **int ***).

```
int idade = 25;       // Variável 'idade' no endereço 0x7fff...
int *p_idade;         // Ponteiro para int (variável que guarda endereços)
p_idade = &idade;     // p_idade agora guarda o endereço de 'idade'
```

- O ***** vincula-se ao identificador, não ao tipo. Por exemplo, **int *p, q;** → **p** é ponteiro, **q** é inteiro.

> Note que um ponteiro **não é** uma variável normal. Ele contém um endereço.
>
> C é uma linguagem de alto nível que expõe explicitamente endereços de memória.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Operador de Endereço (`&`)

- Retorna o endereço de memória onde uma variável reside.
- Pode ser aplicado apenas a variáveis (não a expressões).

```c
int x = 10;
printf("%p", &x); // Exemplo: 0x7ffd4a2b...
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Operador de referência (`*`)

- Acessa o valor armazenado no endereço apontado.
- Sintaxe: **`*variavelPonteiro`**.

```c
int y = 5;
int *ptr = &y;
printf("%d", *ptr); // Imprime: 5 (o valor em 'y')
```

> O `*` na expressão (`*p`) significa "valor apontado por p".

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício: Rastreamento de Endereço

Considere o seguinte bloco de código em C (não precisa compilar, apenas analisar):

```c
int A = 5;
int B = 10;
int *pA = &A; // pA aponta para A
int *pB = &B; // pB aponta para B
```

Variável/Ponteiro	Tipo	Valor Armazenado	Endereço na Memória	O que *pA retorna?	O que &B é?
A	int	5	(Endereço X)	?	?
B	int	10	(Endereço Y)	N/A	?
pA	int*	?	(Endereço Z)	?	N/A
pB	int*	?	(Endereço W)	N/A	N/A

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício

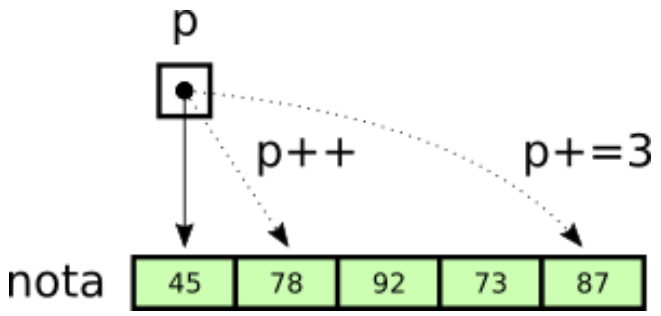
Escreva um programa que:

- Declare uma variável **`float temperatura = 36.5f;`**
- Crie um ponteiro para ela
- Imprima o endereço, o valor via ponteiro e altere o valor para **`37.2f`** usando apenas o ponteiro
- Imprima o valor final da variável original

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Aritmética de Ponteiros

- Ponteiros permitem navegar por estruturas de dados lineares (arrays, listas).
- Ponteiros suportam **+**, **-**, **++**, **--**.
- O incremento é **escalado** pelo tamanho do tipo apontado: **p + 1** avança **sizeof(*p)** bytes.



> Fonte da imagem: Prof. Maziero.

```c
int nota[5] = { 45, 78, 92, 73, 87 };
int *p = nota; // p aponta para nota[0]

printf("%d\n", *p);   // 45
p++;                  // Avança 4 bytes (em arquitetura típica)
printf("%d\n", *p);   // 78
p += 3;               // Pula para nota[4]
printf("%d\n", *p);   // 87
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Arrays e Ponteiros - Cálculo de Endereços

- **&nota[0]** é igual a **nota**.
- **nota[i]** é equivalente a ***(nota + i)**.
- A aritmética ignora o endereço base, calcula o deslocamento baseado no tamanho do ponteiro.

Exemplo:

```c
int nums[] = {1, 2, 3};
int *ptr = nums; // Equivalente a &nums[0]

printf("%d\n", ptr[1]); // Imprime 2 (sintaxe de array em ponteiro)
printf("%zu vs %zu\n", sizeof(nums), sizeof(ptr)); // 12 vs 8
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Arrays e Ponteiros (cont.)

```c
int arr[5] = {1, 2, 3, 4, 5};
int *p = arr; // p recebe o endereço de arr[0]
```

```c
arr[i]   == *(arr + i)
*(p + i) == p[i]
```

Decaimento de Array:

- **sizeof(arr)** retorna o tamanho total do array (ex: 20 bytes).
- **sizeof(p)** retorna o tamanho do ponteiro (ex: 8 bytes em 64-bits).

```c
printf("%zu", sizeof(arr)); // Tamanho do bloco inteiro
printf("%zu", sizeof(p));   // Tamanho da variável ponteiro
```

> Ao passar um array para uma função, ele perde o tamanho. O ponteiro não sabe onde termina.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício

Escreva um programa que inverta os elementos de um array de inteiros de 5 elementos usando apenas ponteiros (sem índices `[i]`).

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Casting e Ponteiro Genérico `void *`

- Um ponteiro que pode apontar para qualquer tipo de dado.
- Usado frequentemente em alocação dinâmica (**malloc**) e bibliotecas genéricas.

```c
void *p = malloc(100); // Aloca 100 bytes genéricos
int *pi = (int *)p;    // Cast necessário para usar como int*
char *pc = (char *)p;  // Pode ser usado como char* também
```

Por que casting é necessário?

- O compilador não sabe qual tipo de dado está no endereço genérico.
- Sem casting, o código gera erro ou comportamento indefinido.

> Sempre faça o casting explícito ao usar em funções específicas.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Cuidados: Ponteiro Nulo NULL

- Um ponteiro que não aponta para nada válido.
- Usado para indicar "não inicializado" ou "fim de lista".
- **Nunca** dereference um ponteiro nulo!

```c
int *p = NULL; // Segura
// int *p = 0; // Também seguro (0 é equivalente a NULL)
*p = 10;       // SEGMENTATION FAULT (Crash)
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Cuidados: Ponteiro Vazio (Dangling Pointer)

- Um ponteiro que aponta para uma memória já liberada ou inválida.
- Exemplo clássico: Ponteiros em variáveis locais após o fim do escopo.

```
void funcao() {
    int x = 10;
    int *p = &x; // p é válido dentro da função
}
// Após a execução de funcao(), 'x' não existe mais.
// Usar p fora daqui é um erro grave (Dangling Pointer).
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Cuidados: Inicialização

- Em C, ponteiros globais são inicializados para **NULL**.
- Ponteiros locais **não** são inicializados automaticamente.

```c
int *p; // p contém lixo na memória (endereço aleatório)
*p = 10; // Comportamento indefinido! Pode crashar.
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Alocação dinâmica de memória

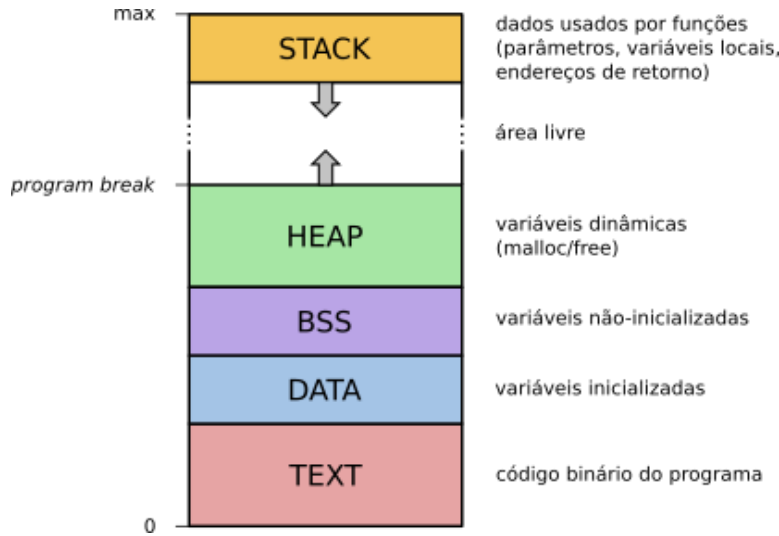
Pilha (Stack): aloca variáveis locais (ex: `int x;`). Tamanho fixo no momento da declaração.

- Liberado automaticamente ao sair do escopo.
- **Limitação:** Não podemos saber o tamanho exato de um array antes de rodar.

Heap: Área de memória dinâmica gerenciada pelo programador.

- Tamanho definido em tempo de execução.
- O programador deve liberar manualmente.

```c
int arr[10]; // Stack
int *arr = malloc(10 * sizeof(int)); // Heap
```



> Fonte da imagem: Prof. Maziero.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Alocação dinâmica de memória

Objetivo: Criar estruturas de dados com tamanho desconhecido em tempo de execução ou criar múltiplas cópias de objetos grandes sem estourar a pilha.

```c
#include <stdio.h>
#include <stdlib.h> // Biblioteca padrão para alocação

int main(void) {
    // Aloca espaço para 100 inteiros no Heap
    int *vetor = malloc(100 * sizeof(int));

    if (vetor == NULL) {
        printf("Erro: Memória insuficiente!\n");
    } else {
        vetor[0] = 42; // Agora é seguro escrever aqui
    }
    return 0;
}
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Função `malloc`: Alocação genérica

- Aloca um bloco de memória de tamanho especificado em bytes.
- Retorna um ponteiro genérico (`void *`) para o início do bloco.
- Se falhar (memória insuficiente), retorna `NULL`.
- **Conteúdo não é inicializado**. A memória contém "lixo" (valores aleatórios da memória anterior).

```c
int *nums = malloc(10 * sizeof(int)); // Aloca espaço para 10 inteiros
if (nums == NULL) {
    // Erro grave: não aloquei, não posso usar!
    return -1;
}
*nums = 5; // Perigoso! O valor é lixo.
```

> Sempre verifique se `malloc` retornou `NULL`. Em sistemas com RAM limitada, isso pode acontecer.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Função `free` para liberar memória

- Libera o bloco de memória alocado por **malloc** ou **calloc**.
- O ponteiro retornado torna-se inválido (não deve mais ser usado).

```c
int *p = malloc(sizeof(int));
*p = 10;
// ... uso da variável ...
free(p);
p = NULL; // Boa prática: evita dangling pointer se o código continuar usando p
```

> **Nunca libere memória da Pilha:** Tentar **free()** uma variável local gera erro.
>
> **Double Free**: Chamar **free(ptr)** duas vezes causa comportamento indefinido (crash).

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Função `free` para liberar memória (con.t)

Gerenciamento de Vazamentos (Memory Leaks):

- Um vazamento ocorre quando alocamos memória e esquecemos de liberar antes que o programa termine.
- Em programas pequenos, o SO libera ao fechar.
- Em servidores ou loops longos, vazamentos esgotam RAM e travam o sistema.

Padrão de Limpeza:

- Sempre pareie a alocação com uma liberação correspondente.
- Use `free()` na ordem inversa da alocação (regra geral).

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Conclusão

- Ponteiros armazenam endereços de memória, não valores diretos.
- `&` pega o endereço; `*` acessa o valor no endereço.
- Aritmética de ponteiro depende do tamanho do tipo apontado.
- Arrays "decadem" para ponteiros em expressões.
- `void *` é genérico, mas requer casting explícito.
- **Stack vs Heap:** Alocação dinâmica usa Heap e exige gerenciamento manual.
- `malloc`: Aloca bytes, conteúdo é lixo. Verifique retorno (`NULL`).
- `free`: Libera memória. Nunca use o ponteiro depois de liberar.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Material Adicional

- Aprenda ponteiros de uma vez por todas - Judson Santiago
- Alocação de Memória - Prof. Maziero
- Snefru: Learning Programming with C

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Dúvidas e Discussão

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício e Questões

Questões teóricas

1. Qual das opções abaixo descreve corretamente o comportamento do código abaixo?

```c
int a = 5;
int *p = &a;
*p = 10;
printf("%d", a);
```

(A) Imprime **5** (B) Imprime **10** (C) Gera erro (D) Imprime o endereço de **a**

2. Explique a diferença entre as seguintes declarações: **int *p;** **int p[5];** **int (*p)[5];**

3. Analise a afirmativa: "Um ponteiro inicializado com **NULL** é seguro para ser referenciado."

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Questões teóricas (cont.)

4. Explique o que acontece no código abaixo e por que é perigoso:

```c
int *p = malloc(10);
free(p);
printf("%d", *p); // O que acontece?
```

5. Analise a afirmativa: "Se eu alocar memória com **malloc**, posso usar **free** quantas vezes quiser."

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício 1

Crie um programa que receba um array de inteiros e imprima apenas os elementos ímpares usando ponteiros (não use índices **arr[i]**).

Exemplo esperado:

```
int arr[] = {1, 4, 7, 9, 12};
int *p;
for (p = arr; p < arr + 5; p++) {
    if (*p % 2 != 0) printf("%d ", *p);
}
```


Prof. Dr. Denis M. L. Martins | martins.denis@usp.br

Exercício 2

Escreva um programa que aloque memória para um inteiro usando **malloc**, inicialize-o, imprima o valor e libere a memória corretamente. Explique por que o **free** é obrigatório.

Prof. Dr. Denis M. L. Martins | martins.denis@usp.br