

Estruturas de Dados: Homogêneas, Heterogêneas e Ponteiros

Guia Simples e Didático com Análise de Complexidade

CONCEITOS FUNDAMENTAIS

O que são Estruturas de Dados?

Definição Simples:

Estruturas de dados são formas organizadas de armazenar e acessar informações na memória do computador.

Analogia do Dia a Dia:

- **Gaveta de roupas** = Array (tudo do mesmo tipo)
- **Mochila escolar** = Struct (coisas diferentes juntas)
- **Endereço da casa** = Ponteiro (indica onde algo está)

ESTRUTURAS HOMOGÊNEAS (Arrays)

 **Conceito:** Todos os elementos são do mesmo tipo

Exemplo Visual:

Array de inteiros:	[10]	[20]	[30]	[40]	[50]
	↑	↑	↑	↑	↑
Posições:	0	1	2	3	4

Implementação Simples em C

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

// Declaração simples
int numeros[5]; // Complexidade: O(1) - alocação estática

// Função para preencher array
void preencher_array(int arr[], int tamanho) {
    printf("Preenchendo array...\n");

    for (int i = 0; i < tamanho; i++) {
        arr[i] = (i + 1) * 10; // O(1) - atribuição direta
        printf("arr[%d] = %d\n", i, arr[i]); // O(1) - impressão
    }
    // Complexidade total: O(n)
```



ESTRUTURAS HETEROGÊNEAS (Structs)



Conceito: Elementos de tipos diferentes agrupados

Analogia: Como uma ficha de cadastro que tem nome (texto), idade (número), altura (decimal)



Implementação Simples em C

```
// Definição da estrutura
typedef struct {
    char nome[50];    // Array de caracteres
    int idade;        // Inteiro
    float altura;     // Número decimal
    char sexo;        // Caractere único
} Pessoa;

// Função para criar uma pessoa
Pessoa criar_pessoa(char* nome, int idade, float altura, char sexo) {
    Pessoa p;          // 0(1) - aloca espaço na stack

    strcpy(p.nome, nome); // 0(n) - copia string, onde n = tamanho do nome
    p.idade = idade;      // 0(1) - atribuição direta
    p.altura = altura;    // 0(1) - atribuição direta
    p.sexo = sexo;        // 0(1) - atribuição direta

    return p;           // 0(1) - retorna cópia da estrutura
    // Complexidade total: O(n) devido ao strcpy
}

// Função para imprimir dados
void imprimir_pessoa(Pessoa p) {
    printf("=== DADOS DA PESSOA ===\n"); // 0(1)
    printf("Nome: %s\n", p.nome);        // 0(1) - acesso direto ao campo
    printf("Idade: %d\n", p.idade);      // 0(1)
    printf("Altura: %.2f\n", p.altura);  // 0(1)
    printf("Sexo: %c\n", p.sexo);        // 0(1)
```

PONTEIROS

 **Conceito:** Variáveis que guardam endereços de memória

Analogia Simples:

Ponteiro é como o endereço da sua casa. Não é a casa, mas indica onde ela está.

Implementação Simples em C

```
// Exemplo básico de ponteiros
void exemplo_ponteiros_basico() {
    int numero = 42;                // 0(1) - cria variável
    int* ptr = &numero;             // 0(1) - ptr aponta para numero

    printf("Valor de numero: %d\n", numero);        // 0(1) - acesso direto
    printf("Endereço de numero: %p\n", &numero);    // 0(1) - pega endereço
    printf("Valor de ptr: %p\n", ptr);              // 0(1) - mostra endereço armazenado
    printf("Valor apontado por ptr: %d\n", *ptr);    // 0(1) - desreferenciamento

    // Modificar através do ponteiro
    *ptr = 100;                          // 0(1) - altera valor via ponteiro
    printf("Novo valor de numero: %d\n", numero);  // 0(1) - numero agora é 100
}

// Ponteiros com arrays
void ponteiros_com_arrays() {
    int numeros[] = {10, 20, 30, 40, 50};          // 0(1) - array estático
    int* ptr = numeros; // ptr aponta para o primeiro elemento // 0(1)

    printf("Array via ponteiro:\n");
    for (int i = 0; i < 5; i++) {                  // 0(n) - loop 5 vezes
        printf("numeros[%d] = %d\n", i, *(ptr + i)); // 0(1) - aritmética de ponteiro
        // *(ptr + i) é equivalente a numeros[i]
    }
}
```




COMPARAÇÃO PRÁTICA DAS ESTRUTURAS

Resumo de Complexidades

Estrutura	Acesso	Busca	Inserção	Uso de Memória	Exemplo
Array	$O(1)$	$O(n)$	$O(n)$	Contígua, eficiente	Lista de notas
Struct	$O(1)$	-	$O(1)$	Agrupada, organizada	Dados de pessoa
Ponteiro	$O(1)$	-	-	Referência, flexível	Navegação em listas

Quando Usar Cada Uma?

Array (Homogênea):

-  Coleção de elementos do mesmo tipo
-  Acesso frequente por índice
-  Tamanho conhecido ou fixo



DICAS PRÁTICAS



Boas Práticas

```
// ✅ BOM: Verificar ponteiros antes de usar
if (ptr != NULL) {
    printf("Valor: %d\n", *ptr);
}

// ❌ RUIM: Usar ponteiro sem verificar
printf("Valor: %d\n", *ptr); // Pode dar segmentation fault

// ✅ BOM: Liberar memória e anular ponteiro
free(ptr);
ptr = NULL;

// ❌ RUIM: Não liberar memória
// Causa memory leak!

// ✅ BOM: Inicializar arrays
int arr[5] = {0}; // Todos elementos = 0

// ✅ BOM: Usar const para ponteiros que não devem alterar dados
void imprimir_array(const int* arr, int tamanho) {
```

EXERCÍCIOS SIMPLES PARA PRATICAR

Exercício 1: Array Básico

```
// Complete a função para encontrar o maior elemento
int encontrar_maior(int arr[], int tamanho) {
    // Sua implementação aqui
    // Complexidade esperada: O(n)
}
```

Exercício 2: Struct Simples

```
// Crie uma struct para representar um produto
// e uma função para calcular desconto
typedef struct {
    // Defina os campos necessários
} Produto;

float calcular_preco_com_desconto(Produto p, float desconto) {
    // Sua implementação aqui
}
```


RESUMO EXECUTIVO


Pontos Principais

1. **Arrays:** Elementos iguais, acesso $O(1)$, busca $O(n)$
2. **Structs:** Elementos diferentes agrupados, acesso $O(1)$ aos campos
3. **Ponteiros:** Endereços de memória, flexibilidade máxima

Complexidades Essenciais

- **Acesso direto:** Sempre $O(1)$
- **Busca linear:** Sempre $O(n)$
- **Operações com strings:** Geralmente $O(n)$
- **Alocação/liberação:** $O(1)$

Regra de Ouro

 Este guia simples fornece uma base sólida para entender estruturas de dados com foco na praticidade e análise de complexidade.

Última atualização: 27 de agosto de 2025