



# Algoritmos e Complexidade

## Aula 02: Fundamentos Matemáticos de Estruturas de Dados









Prof. Vagner Cordeiro  
Sistemas de Informação  
2025.2

## Objetivos de Aprendizagem

Ao final desta aula, você será capaz de:

- **Compreender matematicamente** como estruturas de dados ocupam memória
- **Calcular complexidade** de acesso e operações em diferentes estruturas
- **Diferenciar teoricamente** estruturas homogêneas e heterogêneas
- **Analisar ponteiros** como referências matemáticas de endereços
- **Aplicar fórmulas** para cálculo de posições em arrays multidimensionais
- **Comparar trade-offs** entre diferentes representações de dados

## Agenda da Aula

1.  Fundamentos Matemáticos de Memória
2.  Arrays: Análise Matemática Detalhada
3.  Matrizes: Fórmulas de Indexação
4.  Ponteiros: Teoria e Cálculos de Endereços
5.  Estruturas Heterogêneas: Análise de Layout
6.  Complexidade: Provas Matemáticas
7.  Exemplos Práticos Simples
8.  Comparações e Trade-offs

## Objetivos de Aprendizagem

Ao final desta aula, o estudante será capaz de:

### Fundamentos Teóricos:

- **Definir** matematicamente estruturas de dados homogêneas e heterogêneas
- **Classificar** estruturas de dados segundo critérios de organização e acesso
- **Analisar** complexidade computacional de operações fundamentais

### Implementação Prática:

- **Implementar** arrays, matrizes e estruturas multidimensionais eficientemente
- **Dominar** conceitos avançados de ponteiros e aritmética de endereços
- **Criar** e manipular structs complexas e unions para otimização de memória

### Aplicações Avançadas:

- **Desenvolver** algoritmos otimizados para operações matriciais
- **Aplicar** estruturas de dados adequadas para problemas computacionais específicos
- **Comparar** performance entre diferentes implementações e linguagens

## 12 Fundamentos Matemáticos de Memória

### Como a Memória Funciona Matematicamente

A memória do computador é um **espaço linear indexado**:

$$\text{Memória} = \{M[0], M[1], M[2], \dots, M[n-1]\}$$

Onde cada posição  $M[i]$  armazena **exatamente 1 byte**.

### Tamanhos Fundamentais (em bytes):

Tipo	C	Python	Fórmula
char	1	-	$2^0$
int	4	28+	$2^2$
float	4	24+	$2^2$
double	8	-	$2^3$
pointer	8	8	$2^3$

### Alinhamento de Memória

O processador acessa dados mais eficientemente quando alinhados:

$$\text{endereço o\_alinhado} \equiv 0 \pmod{\text{sizeof}(T)}$$

**Exemplo:** `int` (4 bytes) deve estar em endereços múltiplos de 4: 0, 4, 8, 12...

## Arrays: Base Matemática Fundamental

### Definição Formal

Um array  $A$  de  $n$  elementos do tipo  $T$  é uma **função matemática**:

$$A : \{0, 1, 2, \dots, n - 1\} \rightarrow T$$

### Cálculo de Endereços

Se  $A$  inicia no endereço base  $\text{base}_A$ :

$$\text{endereço}(A[i]) = \text{base}_A + i \times \text{sizeof}(T)$$

### Exemplo Detalhado:

```
int numeros[5] = {10, 20, 30, 40, 50};  
// Se base = 1000, sizeof(int) = 4:  
// numeros[0]: 1000 + 0x4 = 1000 → valor 10  
// numeros[1]: 1000 + 1x4 = 1004 → valor 20  
// numeros[2]: 1000 + 2x4 = 1008 → valor 30  
// numeros[3]: 1000 + 3x4 = 1012 → valor 40  
// numeros[4]: 1000 + 4x4 = 1016 → valor 50
```

### Representação Visual:

```
Memória: [1000][1004][1008][1012][1016]  
Valores: [ 10 ][ 20 ][ 30 ][ 40 ][ 50 ]  
Índices:  [0]  [1]  [2]  [3]  [4]
```

# Propriedades Matemáticas Fundamentais

## Acesso e Indexação

Para uma estrutura indexada, o tempo de acesso é definido por:

$$T_{acesso}(i) = f(\text{posição}, \text{método\_acesso})$$

### Tipos de Acesso:

- **Direto:**  $T(i) = O(1)$  - Arrays convencionais
- **Sequencial:**  $T(i) = O(i)$  - Listas encadeadas
- **Associativo:**  $T(k) = O(\log n)$  - Árvores balanceadas

## Operações Fundamentais e Complexidade

Operação	Array	Lista	Árvore
Acesso	$O(1)$	$O(n)$	$O(\log n)$
Inserção	$O(n)$	$O(1)$	$O(\log n)$
Remoção	$O(n)$	$O(1)$	$O(\log n)$
Busca	$O(n)$	$O(n)$	$O(\log n)$

## 2. Arrays (Vetores) - Estruturas Homogêneas Fundamentais

### Definição Matemática Rigorosa

Um array é uma função bijetiva que estabelece correspondência entre índices e valores:

$$A : \{0, 1, 2, \dots, n - 1\} \rightarrow T$$

Onde:

- $n$  = Dimensão do array (cardinalidade do domínio)
- $T$  = Tipo uniforme dos elementos (contradomínio)
- Propriedade de injetividade: cada índice mapeia para exatamente um elemento



## ⚡ Complexidade de Acesso: Prova Matemática

### Teorema: Acesso a Array é $O(1)$

#### Prova:

Para acessar  $A[i]$ , o computador executa:

1. **Cálculo do endereço:** endereço = base +  $i \times \text{sizeof}(T)$ 
  - Operações: 1 multiplicação + 1 soma = **2 operações**
2. **Acesso à memória:** 1 operação de leitura = **1 operação**

**Total:** 3 operações, independente de  $i$  ou  $n$

$$\therefore T(n) = 3 = O(1) \text{ (tempo constante)}$$

### Exemplo Numérico:

- Array com 10 elementos: 3 operações
- Array com 1.000.000 elementos: 3 operações
- **Mesma performance!** ✅

### Comparação com Busca Linear:

```
// Acesso direto: O(1)
int valor = array[5]; // Sempre 3 operações

// Busca linear: O(n)
for(int i = 0; i < n; i++) {
    if(array[i] == target) return i; // Até n operações
}
```

## Matrizes: Matemática Multidimensional

### Representação Linear

Uma matriz  $M_{m \times n}$  é armazenada **linearmente** na memória:

**Row-major (C):**  $M[i][j] \rightarrow$  posição  $i \times n + j$

**Column-major (Fortran):**  $M[i][j] \rightarrow$  posição  $j \times m + i$

### Fórmula de Endereçamento Row-Major:

$$\text{endereço}(M[i][j]) = \text{base}_M + (i \times n + j) \times \text{sizeof}(T)$$

## Ponteiros: Teoria Matemática Avançada

### Definição Formal

Um ponteiro  $p$  é uma **variável que armazena um endereço**:

$$p : \text{Variável} \rightarrow \text{Endereço de Memória}$$

### Operações Matemáticas:

1. **Declaração:** `int *p;`  $\rightarrow p$  pode apontar para endereços de `int`
2. **Atribuição:** `p = &x;`  $\rightarrow p = \text{endereço de } x$
3. **Desreferenciamento:** `*p`  $\rightarrow$  valor armazenado em endereço  $p$

### Aritmética de Ponteiros:

Se  $p$  aponta para posição  $i$  de um array:

$$p + k \text{ aponta para posição } i + k$$

$$\text{endereço}(p + k) = \text{endereço}(p) + k \times \text{sizeof}(T)$$

### Exemplo Matemático Detalhado:

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;           // p aponta para arr[0]

// Endereços (supondo base = 3000):
// p      → 3000 (arr[0])
// p + 1  → 3004 (arr[1])
// p + 2  → 3008 (arr[2])
// p + 3  → 3012 (arr[3])
// p + 4  → 3016 (arr[4])
```

## Estruturas Heterogêneas: Análise de Layout

### Definição Matemática

Uma estrutura heterogênea  $S$  é uma tupla de tipos diferentes:

$$S = (T_1, T_2, \dots, T_k)$$

Onde  $T_i$  pode ser de qualquer tipo primitivo ou composto.

### Cálculo de Tamanho com Padding

O tamanho real considera **alinhamento de memória**:

$$\text{sizeof}(S) = \sum_{i=1}^k (\text{sizeof}(T_i) + \text{padding}_i)$$

### Exemplo Prático:

```
struct Pessoa {  
    char nome[20];    // 20 bytes  
    int idade;        // 4 bytes  
    double salario;   // 8 bytes  
};
```

**Sem padding:**  $20 + 4 + 8 = 32$  bytes

**Com padding:** Depende do alinhamento!

### Layout na Memória:

Offset: 0	20	24	32
-----------	----	----	----

## Complexidade: Análise Matemática Comparativa

### Operações em Arrays

Operação	Fórmula	Complexidade	Justificativa
Acesso	$T = c$	$O(1)$	Cálculo direto de endereço
Busca	$T = n/2$	$O(n)$	Média de comparações
Inserção	$T = n - i$	$O(n)$	Deslocamento de elementos
Remoção	$T = n - i - 1$	$O(n)$	Compactação necessária

### Demonstração: Inserção em Array

Para inserir na posição  $i$  de um array de tamanho  $n$ :

1. **Deslocar elementos:** de  $i$  até  $n - 1 \rightarrow (n - i)$  operações
2. **Inserir novo elemento:**  $\rightarrow 1$  operação

$$T_{\text{inserção}}(i) = (n - i) + 1 = O(n)$$

### Casos especiais:

- **Início** ( $i = 0$ ):  $n + 1$  operações (pior caso)
- **Final** ( $i = n$ ): 1 operação (melhor caso)
- **Meio** ( $i = n/2$ ):  $n/2 + 1$  operações (caso médio)

## 💡 Exemplos Práticos Simples

### Exemplo 1: Calculadora de Notas

```
// Estrutura homogênea
float notas[4] = {8.5, 7.0, 9.2, 6.8};

// Cálculo da média: O(n)
float soma = 0;
for(int i = 0; i < 4; i++) {
    soma += notas[i]; // Acesso O(1)
}
float media = soma / 4;

printf("Média: %.2f\n", media); // 7.88
```

### Exemplo 2: Cadastro de Estudante

```
// Estrutura heterogênea
struct Estudante {
    char nome[50];
    int matricula;
    float notas[4];
    char status; // 'A'=Aprovado, 'R'=Reprovado
};

struct Estudante aluno = {
    "João Silva",
    12345,
    {8.5, 7.0, 9.2, 6.8},
    'A'
};

// Acesso aos dados: O(1)
printf("Nome: %s\n", aluno.nome);
printf("Matrícula: %d\n", aluno.matricula);
```

# Comparações e Trade-offs

## Homogêneas vs Heterogêneas

Aspecto	Homogêneas (Arrays)	Heterogêneas (Structs)
Simplicidade	★★★★★	★★★
Flexibilidade	★★	★★★★★
Eficiência	★★★★★	★★★★★
Uso de Memória	★★★★★	★★★★
Facilidade Debug	★★★★★	★★★

## Quando Usar Cada Uma?

### Arrays (Homogêneas):

- ✓ Cálculos matemáticos (vetores, matrizes)
- ✓ Processamento de sinais/imagens
- ✓ Algoritmos numéricos
- ✓ Performance crítica

### Structs (Heterogêneas):

- ✓ Modelagem de entidades reais
- ✓ Bancos de dados
- ✓ Interfaces de usuário
- ✓ Sistemas complexos

## Resumo: Fundamentos Matemáticos

### Fórmulas Essenciais

1. **Endereçamento:**  $\text{addr}(A[i]) = \text{base} + i \times \text{sizeof}(T)$
2. **Matriz 2D:**  $\text{addr}(M[i][j]) = \text{base} + (i \times n + j) \times \text{sizeof}(T)$
3. **Complexidade de Acesso:**  $T_{\text{array}} = O(1), T_{\text{busca}} = O(n)$
4. **Tamanho de Struct:**  $\text{sizeof}(S) = \sum \text{sizeof}(T_i) + \text{padding}$

### Princípios Fundamentais

1. **Arrays oferecem acesso  $O(1)$**  por cálculo matemático direto
2. **Estruturas heterogêneas** modelam realidade com flexibilidade
3. **Ponteiros** permitem indireção e estruturas dinâmicas
4. **Alinhamento de memória** afeta performance e tamanho
5. **Trade-offs** sempre existem entre simplicidade e flexibilidade

### Próxima Aula

 **Algoritmos de Ordenação:** Bubble Sort, Selection Sort, Quick Sort

 **Análise de Complexidade:** Melhor, médio e pior caso

 **Matemática:** Análise assintótica detalhada

### Vantagens Fundamentais:

- **Acesso Aleatório:**  $T_{\text{acesso}}(i) = O(1)$  constante
- **Localidade Espacial:** Elementos contíguos na memória
- **Eficiência de Cache:** Alta taxa de cache hits



## Endereçamento e Layout de Memória

### Cálculo Matemático de Endereços

Para um array unidimensional, o endereço do elemento  $A[i]$  é:

$$\text{endereço}(A[i]) = \text{base} + i \times \text{sizeof}(T)$$

Onde:

- $\text{base}$  = Endereço inicial do array
- $i$  = Índice do elemento desejado
- $\text{sizeof}(T)$  = Tamanho em bytes do tipo  $T$

### Exemplo Prático de Endereçamento

```
int numeros[5] = {10, 20, 30, 40, 50};  
// Se base = 0x1000 e sizeof(int) = 4:  
// numeros[0] → 0x1000 (base + 0×4)  
// numeros[1] → 0x1004 (base + 1×4)  
// numeros[2] → 0x1008 (base + 2×4)  
// numeros[3] → 0x100C (base + 3×4)  
// numeros[4] → 0x1010 (base + 4×4)
```

## Implementações Avançadas em C

### Declaração e Inicialização Otimizada

```
// Diferentes métodos de inicialização
int array_basico[5];           // Não inicializado
int array_inicializado[5] = {1,2,3,4,5}; // Inicialização completa
int array_parcial[5] = {1,2};  // Parcial: {1,2,0,0,0}
int array_zero[5] = {0};       // Todos zeros
int array_automatico[] = {1,2,3,4,5}; // Tamanho inferido
```

### Validação e Tratamento de Erros

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

typedef enum {
    ARRAY_OK = 0,
    ARRAY_INDICE_INVALIDO = -1,
    ARRAY_PONTEIRO_NULO = -2,
    ARRAY_TAMANHO_INVALIDO = -3
} CodigoErroArray;

CodigoErroArray acessar_elemento_seguro(int array[], int tamanho,
                                       int indice, int *resultado) {
    // Validação de pré-condições
    if (array == NULL) return ARRAY_PONTEIRO_NULO;
    if (resultado == NULL) return ARRAY_PONTEIRO_NULO;
    if (tamanho <= 0) return ARRAY_TAMANHO_INVALIDO;
    if (indice < 0 || indice >= tamanho) return ARRAY_INDICE_INVALIDO;

    // Operação segura
    *resultado = array[indice];
    return ARRAY_OK;
}

int main() {
    int numeros[5] = {10, 20, 30, 40, 50};
```

# Algoritmos Fundamentais com Arrays

## 1. Busca Linear com Análise Matemática

```
int busca_linear_otimizada(int array[], int tamanho, int elemento) {  
    // Análise: T(n) = O(n), S(n) = O(1)  
    for (int i = 0; i < tamanho; i++) {  
        if (array[i] == elemento) {  
            return i; // Índice do elemento encontrado  
        }  
    }  
    return -1; // Elemento não encontrado  
}
```

### Análise de Complexidade:

- **Melhor caso:**  $T(n) = O(1)$  - elemento na primeira posição
- **Caso médio:**  $T(n) = O(n/2) = O(n)$  - elemento no meio
- **Pior caso:**  $T(n) = O(n)$  - elemento inexistente ou última posição

## 2. Busca Binária para Arrays Ordenados

```
int busca_binaria_recursiva(int array[], int inicio, int fim, int elemento) {  
    if (inicio > fim) {  
        return -1; // Elemento não encontrado  
    }  
  
    int meio = inicio + (fim - inicio) / 2; // Evita overflow  
  
    if (array[meio] == elemento) {  
        return meio; // Elemento encontrado  
    }  
  
    if (array[meio] > elemento) {  
        return busca_binaria_recursiva(array, inicio, meio - 1, elemento);  
    } else {  

```

## Operações Estatísticas Avançadas

### Cálculo de Estatísticas Descritivas

```
#include <math.h>

typedef struct {
    double media;
    double mediana;
    double desvio_padrao;
    int minimo;
    int maximo;
} EstatisticasArray;

EstatisticasArray calcular_estatisticas(int array[], int tamanho) {
    EstatisticasArray stats = {0};

    if (tamanho == 0) return stats;

    // Cálculo da média:  $\mu = (1/n) * \sum x_i$ 
    double soma = 0;
    stats.minimo = stats.maximo = array[0];

    for (int i = 0; i < tamanho; i++) {
        soma += array[i];
        if (array[i] < stats.minimo) stats.minimo = array[i];
        if (array[i] > stats.maximo) stats.maximo = array[i];
    }

    stats.media = soma / tamanho;

    // Cálculo do desvio padrão:  $\sigma = \sqrt{(1/n) * \sum (x_i - \mu)^2}$ 
    double soma_quadrados = 0;
    for (int i = 0; i < tamanho; i++) {
        double diferenca = array[i] - stats.media;
        soma_quadrados += diferenca * diferenca;
    }

    stats.desvio_padrao = sqrt(soma_quadrados / tamanho);

    return stats;
}
```

### 3. Matrizes e Arrays Bidimensionais

#### Definição Matemática Formal

Uma matriz é uma função que mapeia pares ordenados para valores:

$$M : \{0, 1, \dots, m - 1\} \times \{0, 1, \dots, n - 1\} \rightarrow T$$

Onde:

- $m$  = Número de linhas (dimensão vertical)
- $n$  = Número de colunas (dimensão horizontal)
- $T$  = Tipo dos elementos da matriz

#### Endereçamento em Matrizes

Row-major order (C/C++):

$$\text{endereco}(M[i][j]) = \text{base} + (i \times n + j) \times \text{sizeof}(T)$$

Column-major order (Fortran):

$$\text{endereco}(M[i][j]) = \text{base} + (j \times m + i) \times \text{sizeof}(T)$$

## Implementação de Matrizes em C

### Declaração e Inicialização

```
// Diferentes formas de declarar matrizes
int matriz_fixa[3][4]; // 3 linhas, 4 colunas
int matriz_inicializada[2][3] = {{1,2,3}, {4,5,6}};
int matriz_parcial[2][3] = {{1,2}, {4}}; // Resto preenchido com 0

// Matriz usando array unidimensional
int matriz_linear[12]; // Simula matriz 3x4
#define ACESSO(i,j,colunas) ((i)*(colunas)+(j))
// Acesso: matriz_linear[ACESSO(i,j,4)] para matriz[i][j]
```

### Operações Matriciais Fundamentais

```
// Adição de matrizes: C = A + B
void adicionar_matrizes(int A[][4], int B[][4], int C[][4],
                       int linhas, int colunas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    // Complexidade: O(m × n)
}

// Multiplicação de matrizes: C = A × B
void multiplicar_matrizes(int A[][4], int B[][4], int C[][4],
                          int m, int n, int p) {
    // A: m×n, B: n×p, C: m×p
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

## Algoritmos Avançados para Matrizes

### Transposição de Matriz

```
void transpor_matriz(int original[][4], int transposta[][3],
                    int linhas, int colunas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            transposta[j][i] = original[i][j];
        }
    }
}
```

Definição Matemática:

$$A^T[j][i] = A[i][j]$$

```
tipo nome[tamanho];
```

```
// Exemplos
```

```
int numeros[10];           // Array de 10 inteiros
```

```
float notas[5];           // Array de 5 floats
```

```
char texto[100];          // String (array de chars)
```

## Inicialização

```
int arr1[5] = {1, 2, 3, 4, 5};    // Inicialização completa
```

```
int arr2[5] = {1, 2};            // Parcial: {1, 2, 0, 0, 0}
```

```
int arr3[] = {1, 2, 3};          // Tamanho automático: 3
```



## Operações Fundamentais em Arrays

### 1. Acesso e Modificação

```
int arr[5] = {10, 20, 30, 40, 50};  
  
// Acesso: O(1)  
int valor = arr[2];    // valor = 30  
  
// Modificação: O(1)  
arr[2] = 35;           // arr[2] agora é 35
```

### Cálculo de Endereço

$$\text{endereço}(\text{arr}[i]) = \text{base} + i \times \text{sizeof}(\text{tipo})$$

## 2. Algoritmos Básicos em Arrays

### Busca Linear

```
int busca_linear(int arr[], int n, int x) {  
    for (int i = 0; i < n; i++) {  
        if (arr[i] == x) {  
            return i; // Retorna índice  
        }  
    }  
    return -1; // Não encontrado  
}
```

**Complexidade:**  $T(n) = O(n)$

**Melhor caso:**  $O(1)$  (primeiro elemento)

**Pior caso:**  $O(n)$  (último ou não existe)

## Busca Binária (Array Ordenado)

```
int busca_binaria(int arr[], int l, int r, int x) {  
    while (l <= r) {  
        int m = l + (r - l) / 2;  
  
        if (arr[m] == x) return m;  
  
        if (arr[m] < x)  
            l = m + 1;  
        else  
            r = m - 1;  
    }  
    return -1;  
}
```

**Complexidade:**  $T(n) = O(\log n)$

**Recorrência:**  $T(n) = T(n/2) + O(1)$

### 3. Inserção e Remoção em Arrays

#### Inserção no Final

```
int inserir_final(int arr[], int *tamanho, int elemento, int capacidade) {  
    if (*tamanho >= capacidade) return 0; // Array cheio  
  
    arr[*tamanho] = elemento;  
    (*tamanho)++;  
    return 1; // Sucesso  
}
```

**Complexidade:**  $O(1)$

## Inserção em Posição Específica

```
int inserir_posicao(int arr[], int *tamanho, int pos, int elemento, int cap) {  
    if (*tamanho >= cap || pos > *tamanho) return 0;  
  
    // Desloca elementos para direita  
    for (int i = *tamanho; i > pos; i--) {  
        arr[i] = arr[i-1];  
    }  
  
    arr[pos] = elemento;  
    (*tamanho)++;  
    return 1;  
}
```

**Complexidade:**  $O(n)$  - devido ao deslocamento

## Remoção de Elementos

### Remoção por Índice

```
int remover_indice(int arr[], int *tamanho, int indice) {  
    if (indice >= *tamanho || indice < 0) return 0;  
  
    // Desloca elementos para esquerda  
    for (int i = indice; i < *tamanho - 1; i++) {  
        arr[i] = arr[i + 1];  
    }  
  
    (*tamanho)--;  
    return 1;  
}
```

**Complexidade:**  $O(n)$

## 4. Matrizes (Arrays Bidimensionais)

### Definição Matemática

Uma matriz é uma função:

$$M : \{0, 1, \dots, m - 1\} \times \{0, 1, \dots, n - 1\} \rightarrow T$$

### Representação na Memória

**Row-major (C):**  $M[i][j] \rightarrow base + (i \times cols + j) \times sizeof(T)$

**Column-major (Fortran):**  $M[i][j] \rightarrow base + (j \times rows + i) \times sizeof(T)$

## Declaração de Matrizes em C

```
// Matriz estática
int matriz[3][4]; // 3 linhas, 4 colunas

// Inicialização
int mat[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Alocação dinâmica
int **matriz_dinamica = malloc(linhas * sizeof(int*));
for (int i = 0; i < linhas; i++) {
    matriz_dinamica[i] = malloc(colunas * sizeof(int));
}
```



## Operações com Matrizes

### Multiplicação de Matrizes

$$C[i][j] = \sum_{k=0}^{p-1} A[i][k] \times B[k][j]$$

```
void multiplicar_matrizes(int A[][3], int B[][3], int C[][3], int n) {  
    for (int i = 0; i < n; i++) {  
        for (int j = 0; j < n; j++) {  
            C[i][j] = 0;  
            for (int k = 0; k < n; k++) {  
                C[i][j] += A[i][k] * B[k][j];  
            }  
        }  
    }  
}
```

**Complexidade:**  $O(n^3)$

## Transposição de Matriz

### Conceito Matemático

$$A^T[j][i] = A[i][j]$$

```
void transpor_matriz(int A[][MAX], int T[][MAX], int linhas, int colunas) {  
    for (int i = 0; i < linhas; i++) {  
        for (int j = 0; j < colunas; j++) {  
            T[j][i] = A[i][j];  
        }  
    }  
}
```

**Complexidade:**  $O(m \times n)$

## 4. Ponteiros: Fundamentos Matemáticos e Aplicações Avançadas

### Definição Formal de Ponteiro

Um ponteiro é uma abstração matemática para endereçamento indireto:

$$ptr : \text{Endereço} \rightarrow \text{Valor}$$

### Propriedades Fundamentais:

- **Indireção:** Acesso ao valor através do endereço
- **Aritmética:** Operações matemáticas sobre endereços
- **Tipagem:** Conhecimento do tipo do dado apontado
- **Nulidade:** Possibilidade de não apontar para local válido

### Anatomia de um Ponteiro em C

```
int valor = 42;           // Variável comum
int *ponteiro = &valor;   // Ponteiro para inteiro

// Análise dos operadores:
// &valor    → Endereço de 'valor' (operador address-of)
// *ponteiro → Valor apontado por 'ponteiro' (operador dereference)
// ponteiro  → Endereço armazenado no ponteiro
```

### Representação Visual:



## Aritmética de Ponteiros: Formalização Matemática

### Operações Fundamentais

#### Incremento/Decremento:

$$p \pm n = p \pm n \times \text{sizeof}(\text{tipo\_apontado})$$

#### Diferença entre Ponteiros:

$$p_1 - p_2 = \frac{\text{endereco}(p_1) - \text{endereco}(p_2)}{\text{sizeof}(\text{tipo})}$$

### Implementação e Exemplos

```
int array[5] = {10, 20, 30, 40, 50};
int *p = array; // p aponta para array[0]

// Navegação por aritmética
printf("p[0] = %d\n", *p);      // 10
printf("p[1] = %d\n", *(p+1)); // 20
printf("p[2] = %d\n", *(p+2)); // 30

// Equivalências matemáticas
assert(array[i] == *(array + i)); // Sempre verdadeiro
assert(&array[i] == array + i);   // Sempre verdadeiro

// Cálculo de distância
int *inicio = &array[0];
int *fim = &array[4];
ptrdiff_t distancia = fim - inicio; // Resultado: 4 elementos
```

## Ponteiros para Ponteiros (Indireção Múltipla)

### Conceito e Aplicações

```
int valor = 100;
int *ptr1 = &valor;      // Ponteiro simples
int **ptr2 = &ptr1;      // Ponteiro para ponteiro
int ***ptr3 = &ptr2;     // Ponteiro triplo

// Acesso através de diferentes níveis de indireção
printf("Valor: %d\n", valor);    // 100
printf("Valor: %d\n", *ptr1);    // 100
printf("Valor: %d\n", **ptr2);   // 100
printf("Valor: %d\n", ***ptr3);  // 100
```

### Aplicação Prática: Matriz Dinâmica

```
int **criar_matriz_dinamica(int linhas, int colunas) {
    // Aloca array de ponteiros para linhas
    int **matriz = malloc(linhas * sizeof(int*));

    // Aloca cada linha individualmente
    for (int i = 0; i < linhas; i++) {
        matriz[i] = malloc(colunas * sizeof(int));
    }

    return matriz;
}

void liberar_matriz_dinamica(int **matriz, int linhas) {
    for (int i = 0; i < linhas; i++) {
        free(matriz[i]); // Libera cada linha
    }
    free(matriz); // Libera array de ponteiros
}
```

## Ponteiros para Funções: Programação Funcional em C

### Declaração e Uso

```
// Declaração de ponteiro para função
int (*operacao)(int, int);

// Funções de diferentes operações
int somar(int a, int b) { return a + b; }
int multiplicar(int a, int b) { return a * b; }
int elevar(int base, int expoente) {
    int resultado = 1;
    for (int i = 0; i < expoente; i++) resultado *= base;
    return resultado;
}

// Array de ponteiros para funções
int (*operacoes[])(int, int) = {somar, multiplicar, elevar};

// Uso dinâmico
int main() {
    int resultado = operacoes[0](5, 3); // Chama somar(5,3) = 8
    return 0;
}
```

### Implementação de Calculadora Flexível

```
typedef int (*OperacaoMatematica)(int, int);

typedef struct {
    char simbolo;
    OperacaoMatematica funcao;
    const char *nome;
} CalculadoraOperacao;

CalculadoraOperacao calculadora[] = {
    {'+', somar, "Adição"},
    {'*', multiplicar, "Multiplicação"},
    {'^', elevar, "Exponenciação"}
```

# Ponteiros e Gerenciamento de Memória

## Alocação Dinâmica Segura

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

typedef enum {
    MEMORIA_OK = 0,
    MEMORIA_INSUFICIENTE = -1,
    PONTEIRO_INVALIDO = -2
} StatusMemoria;

StatusMemoria alocar_array_seguro(int **array, int tamanho) {
    if (array == NULL) return PONTEIRO_INVALIDO;

    *array = malloc(tamanho * sizeof(int));
    if (*array == NULL) return MEMORIA_INSUFICIENTE;

    // Inicializa com zeros para segurança
    memset(*array, 0, tamanho * sizeof(int));

    return MEMORIA_OK;
}

void liberar_array_seguro(int **array) {
    if (array != NULL && *array != NULL) {
        free(*array);
        *array = NULL; // Evita double-free
    }
}

// Uso seguro
int main() {
    int *meu_array = NULL;

    if (alocar_array_seguro(&meu_array, 100) == MEMORIA_OK) {
        // Usar o array...
        meu_array[0] = 42;

        // Limpeza segura
        liberar_array_seguro(&meu_array);
    }

    return 0;
}
```

## Validação e Debugging de Ponteiros

### Técnicas de Validação

```
#include <assert.h>

void validar_ponteiro(void *ptr, const char *nome_variavel) {
    if (ptr == NULL) {
        fprintf(stderr, "ERRO: Ponteiro %s é NULL\n", nome_variavel);
        abort();
    }
}

void copiar_array_seguro(int *origem, int *destino, int tamanho) {
    // Validações de pré-condição
    validar_ponteiro(origem, "origem");
    validar_ponteiro(destino, "destino");
    assert(tamanho > 0);

    // Cópia segura
    for (int i = 0; i < tamanho; i++) {
        destino[i] = origem[i];
    }
}

// Macro para debugging
#define DEBUG_PONTEIRO(ptr) \
    printf("DEBUG: %s = %p, valor = %d\n", #ptr, (void*)ptr, \
        (ptr != NULL) ? *ptr : 0)
```



## 5. Estruturas Heterogêneas (Structs): Organização de Dados Complexos

### Definição Matemática

Uma struct é um produto cartesiano de tipos:

$$\text{struct} = T_1 \times T_2 \times \dots \times T_n$$

Onde cada  $T_i$  representa um tipo de dado diferente.

### Motivação: Modelagem de Entidades do Mundo Real

```
// Problema: Representar um estudante
// Solução inadequada: arrays separados
char nomes[100][50];
int idades[100];
float notas[100];
// Problema: Relacionamento implícito, propenso a erros

// Solução elegante: struct
typedef struct {
    char nome[50];
    int idade;
    float nota;
    char curso[30];
} Estudante;
```

## Definição e Uso de Structs

### Sintaxe e Declaração

```
// Definição da estrutura
struct Ponto {
    double x;
    double y;
    double z;
};

// Diferentes formas de declaração
struct Ponto p1; // Declaração simples
struct Ponto p2 = {1.0, 2.0, 3.0}; // Inicialização
struct Ponto p3 = {.x = 5.0, .z = 10.0}; // Inicialização designada

// Usando typedef para simplificação
typedef struct {
    double x, y, z;
} Ponto3D;

Ponto3D origem = {0.0, 0.0, 0.0};
```

### Acesso aos Membros

```
// Acesso direto (operador ponto)
p1.x = 10.5;
p1.y = 20.3;
printf("Coordenada X: %.2f\n", p1.x);

// Acesso via ponteiro (operador seta)
Ponto3D *ptr = &origem;
ptr->x = 15.0; // Equivale a (*ptr).x = 15.0
ptr->y = 25.0;
```

## Operações Avançadas com Structs

### Cópia e Comparação

```
typedef struct {
    int dia, mes, ano;
} Data;

// Cópia de estruturas (assignment)
Data data1 = {25, 12, 2024};
Data data2 = data1; // Cópia byte-a-byte automática

// Comparação personalizada
int comparar_datas(Data d1, Data d2) {
    if (d1.ano != d2.ano) return d1.ano - d2.ano;
    if (d1.mes != d2.mes) return d1.mes - d2.mes;
    return d1.dia - d2.dia;
}

int datas_iguais(Data d1, Data d2) {
    return (d1.dia == d2.dia) &&
           (d1.mes == d2.mes) &&
           (d1.ano == d2.ano);
}
```

### Função de Inicialização (Constructor Pattern)

```
typedef struct {
    char nome[100];
    double salario;
    int id;
    Data data_contratacao;
} Funcionario;

Funcionario criar_funcionario(const char *nome, double salario, Data contratacao) {
    Funcionario funcionario;

    // Cópia segura do nome
    strncpy(funcionario.nome, nome, sizeof(funcionario.nome) - 1);
```

## Arrays de Structs: Bases de Dados Simples

### Implementação de Sistema de Gerenciamento

```
#define MAX_FUNCIONARIOS 1000

typedef struct {
    Funcionario funcionarios[MAX_FUNCIONARIOS];
    int total_funcionarios;
} SistemaRH;

// Inicialização do sistema
SistemaRH inicializar_sistema() {
    SistemaRH sistema = {{0}, 0}; // Inicializa com zeros
    return sistema;
}

// Adição de funcionário
int adicionar_funcionario(SistemaRH *sistema, Funcionario funcionario) {
    if (sistema->total_funcionarios >= MAX_FUNCIONARIOS) {
        return -1; // Sistema cheio
    }

    sistema->funcionarios[sistema->total_funcionarios] = funcionario;
    sistema->total_funcionarios++;

    return sistema->total_funcionarios - 1; // Retorna índice
}

// Busca por ID
Funcionario* buscar_funcionario_por_id(SistemaRH *sistema, int id) {
    for (int i = 0; i < sistema->total_funcionarios; i++) {
        if (sistema->funcionarios[i].id == id) {
            return &sistema->funcionarios[i];
        }
    }
    return NULL; // Não encontrado
}
```

# Alinhamento de Memória e Padding

## Conceito e Importância

```
// Struct sem consideração de alinhamento
struct SemPadding {
    char a;    // 1 byte
    int b;     // 4 bytes
    char c;    // 1 byte
    double d;  // 8 bytes
};
// Tamanho esperado: 14 bytes
// Tamanho real: 24 bytes (com padding)

// Struct otimizada para alinhamento
struct ComPadding {
    double d;  // 8 bytes (alinhamento de 8)
    int b;     // 4 bytes (alinhamento de 4)
    char a;    // 1 byte
    char c;    // 1 byte
    char padding[2]; // Padding explícito
};
// Tamanho: 16 bytes (mais eficiente)
```

## Fórmula de Cálculo de Tamanho

$$\text{sizeof}(\text{struct}) \geq \sum \text{sizeof}(\text{membros})$$

### Regra de Alinhamento:

- Cada membro deve estar alinhado em múltiplo de seu tamanho
- Struct inteira deve ter tamanho múltiplo do maior alinhamento

## Unions: Economia de Memória

### Conceito e Aplicação

```
// Union: todos os membros compartilham a mesma memória
union Valor {
    int inteiro;
    float decimal;
    char caractere[4];
};

// Uso prático: interpretação de dados
union Conversor {
    float numero;
    unsigned char bytes[4];
};

void analisar_float(float f) {
    union Conversor conv;
    conv.numero = f;

    printf("Float: %f\n", f);
    printf("Representação em bytes: ");
    for (int i = 0; i < 4; i++) {
        printf("%02X ", conv.bytes[i]);
    }
    printf("\n");
}
```

### Union Discriminada (Tagged Union)

```
typedef enum {
    TIPO_INTEIRO,
    TIPO_DECIMAL,
    TIPO_STRING
} TipoValor;

typedef struct {
    TipoValor tipo;
```

## 6. Aplicações Práticas: Sistema de Coordenadas

### Implementação Completa

```
#include <math.h>

typedef struct {
    double x, y;
} Ponto2D;

typedef struct {
    double x, y, z;
} Ponto3D;

// Operações vetoriais
double distancia_2d(Ponto2D p1, Ponto2D p2) {
    double dx = p2.x - p1.x;
    double dy = p2.y - p1.y;
    return sqrt(dx*dx + dy*dy);
}

Ponto2D somar_pontos_2d(Ponto2D p1, Ponto2D p2) {
    Ponto2D resultado = {p1.x + p2.x, p1.y + p2.y};
    return resultado;
}

double produto_escalar_2d(Ponto2D v1, Ponto2D v2) {
    return v1.x * v2.x + v1.y * v2.y;
}

// Sistema de partículas
typedef struct {
    Ponto2D posicao;
    Ponto2D velocidade;
    double massa;
    double energia;
} Particula;

void atualizar_particula(Particula *p, double dt) {
    // Integração de Euler simples
    p->posicao.x += p->velocidade.x * dt;
    p->posicao.y += p->velocidade.y * dt;

    // Cálculo de energia cinética
    double velocidade_magnitude = sqrt(p->velocidade.x * p->velocidade.x +
                                         p->velocidade.y * p->velocidade.y);
    p->energia = 0.5 * p->massa * velocidade_magnitude * velocidade_magnitude;
}
```

## Exercícios Práticos Avançados

### 1. Sistema de Biblioteca Digital

```
typedef struct {
    char titulo[200];
    char autor[100];
    int ano_publicacao;
    char isbn[20];
    int disponivel; // 1 = disponível, 0 = emprestado
} Livro;

typedef struct {
    char nome[100];
    int numero_cartao;
    char email[100];
    Data data_cadastro;
} Usuario;

typedef struct {
    int id_livro;
    int id_usuario;
    Data data_emprestimo;
    Data data_devolucao_prevista;
    int devolvido; // 0 = não devolvido, 1 = devolvido
} Emprestimo;

// Implementar funções:
// - cadastrar_livro()
// - buscar_livros_por_autor()
// - realizar_emprestimo()
// - calcular_multa_atraso()
```

### 2. Sistema de Geometria Computacional

```
typedef struct {
    Ponto2D vertices[3];
} Triangulo;
```



## 7. Comparação C vs Python: Estruturas de Dados

### Arrays/Listas: Análise Comparativa

#### C - Arrays Estáticos:

```
int numeros[1000]; // Alocação em stack, acesso O(1)
numeros[500] = 42; // Acesso direto, sem verificação de bounds
```

#### Python - Listas Dinâmicas:

```
numeros = [0] * 1000 # Alocação em heap, mais flexível
numeros[500] = 42    # Acesso O(1) com verificação automática
numeros.append(43)   # Redimensionamento automático O(1) amortizado
```

Aspecto	C Arrays	Python Lists
Performance	Máxima	Boa
Segurança	Manual	Automática
Flexibilidade	Limitada	Alta
Gerência Memória	Manual	Automática

## Structs vs Classes: Paradigmas de Organização

### C - Structs com Funções Associadas

```
typedef struct {
    double x, y;
    char nome[50];
} Ponto;

Ponto criar_ponto(double x, double y, const char *nome) {
    Ponto p = {x, y, ""};
    strncpy(p.nome, nome, sizeof(p.nome) - 1);
    return p;
}

double distancia_origem(Ponto p) {
    return sqrt(p.x * p.x + p.y * p.y);
}
```

### Python - Classes com Métodos Integrados

```
class Ponto:
    def __init__(self, x, y, nome):
        self.x = x
        self.y = y
        self.nome = nome

    def distancia_origem(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5

    def __str__(self):
        return f"Ponto({self.x}, {self.y}) - {self.nome}"

# Uso mais expressivo
p = Ponto(3.0, 4.0, "A")
print(p.distancia_origem()) # 5.0
```

## Análise de Performance: Benchmarks Reais

### Operações com Arrays (1 milhão de elementos)

#### Teste: Soma de elementos

```
// C - Versão otimizada
double somar_array_c(double *array, int tamanho) {
    double soma = 0.0;
    for (int i = 0; i < tamanho; i++) {
        soma += array[i];
    }
    return soma;
}
// Tempo: ~2.3ms
```

```
# Python - Versão nativa
def somar_array_python(array):
    return sum(array)
# Tempo: ~45ms

# Python com NumPy - Versão otimizada
import numpy as np
def somar_array_numpy(array):
    return np.sum(array)
# Tempo: ~3.1ms
```

**Resultado:** C é ~20x mais rápido que Python puro, comparável ao NumPy

## Gerenciamento de Memória: Trade-offs

### C - Controle Total, Responsabilidade Total

```
typedef struct {
    int *dados;
    int tamanho;
    int capacidade;
} ArrayDinamico;

ArrayDinamico* criar_array(int capacidade_inicial) {
    ArrayDinamico *arr = malloc(sizeof(ArrayDinamico));
    if (!arr) return NULL;

    arr->dados = malloc(capacidade_inicial * sizeof(int));
    if (!arr->dados) {
        free(arr);
        return NULL;
    }

    arr->tamanho = 0;
    arr->capacidade = capacidade_inicial;
    return arr;
}

void destruir_array(ArrayDinamico *arr) {
    if (arr) {
        free(arr->dados);
        free(arr);
    }
}
```

### Python - Automação com Overhead

```
class ArrayDinamico:
    def __init__(self, capacidade_inicial=10):
        self._dados = [None] * capacidade_inicial
        self._tamanho = 0
        self._capacidade = capacidade_inicial
```

## 8. Aplicações Avançadas: Processamento de Imagens

### Representação de Imagem como Matriz

```
typedef struct {
    unsigned char r, g, b; // Red, Green, Blue (0-255)
} Pixel;

typedef struct {
    Pixel **pixels; // Matriz bidimensional
    int largura;
    int altura;
} Imagem;

Imagem* criar_imagem(int largura, int altura) {
    Imagem *img = malloc(sizeof(Imagem));
    if (!img) return NULL;

    img->largura = largura;
    img->altura = altura;

    // Aloca matriz de ponteiros para linhas
    img->pixels = malloc(altura * sizeof(Pixel*));
    if (!img->pixels) {
        free(img);
        return NULL;
    }

    // Aloca cada linha
    for (int i = 0; i < altura; i++) {
        img->pixels[i] = malloc(largura * sizeof(Pixel));
        if (!img->pixels[i]) {
            // Limpa alocações parciais
            for (int j = 0; j < i; j++) {
                free(img->pixels[j]);
            }
            free(img->pixels);
            free(img);
            return NULL;
        }
    }

    return img;
}
```

## 9. Estruturas de Dados Avançadas: Preview

### Listas Ligadas: Fundamentos

```
typedef struct No {
    int dados;
    struct No *proximo;
} No;

typedef struct {
    No *cabeca;
    No *cauda;
    int tamanho;
} ListaLigada;

// Inserção no início: O(1)
void inserir_inicio(ListaLigada *lista, int valor) {
    No *novo = malloc(sizeof(No));
    if (!novo) return;

    novo->dados = valor;
    novo->proximo = lista->cabeca;
    lista->cabeca = novo;

    if (lista->cauda == NULL) {
        lista->cauda = novo;
    }

    lista->tamanho++;
}

// Busca: O(n)
No* buscar(ListaLigada *lista, int valor) {
    No *atual = lista->cabeca;
    while (atual != NULL) {
        if (atual->dados == valor) {
            return atual;
        }
        atual = atual->proximo;
    }
    return NULL;
}
```

## 10. Conclusões e Próximos Passos

### Conhecimentos Fundamentais Adquiridos

#### Estruturas Homogêneas:

- Arrays e matrizes com análise matemática rigorosa
- Algoritmos de busca e operações otimizadas
- Endereçamento e layout de memória

#### Ponteiros e Gerenciamento:

- Aritmética de ponteiros e indireção múltipla
- Alocação dinâmica segura
- Ponteiros para funções e programação funcional

#### Estruturas Heterogêneas:

- Structs para modelagem de entidades complexas
- Unions para economia de memória
- Sistemas práticos de gerenciamento de dados

### Preparação para Aulas Futuras

#### Próxima Aula: Algoritmos de Ordenação e Busca

- QuickSort, MergeSort, HeapSort
- Análise de complexidade comparativa
- Estruturas de dados especializadas (heaps, árvores)

#### Conceitos Avançados em Desenvolvimento:

## Bibliografia e Recursos Complementares

### Referências Técnicas Essenciais

#### Livros Clássicos:

- Kernighan, B. W.; Ritchie, D. M. *The C Programming Language*
- Cormen, T. H. et al. *Introduction to Algorithms*
- Sedgewick, R. *Algorithms in C*

#### Recursos Práticos:

- Projeto GNU: Documentação oficial da glibc
- IEEE Standards: Padrões C11/C18
- Exercícios online: HackerRank, LeetCode, Codeforces

### Ferramentas de Desenvolvimento

#### Compiladores e IDEs:

- GCC com flags de otimização (-O2, -O3)
- Clang Static Analyzer para detecção de bugs
- Valgrind para análise de memória
- VS Code com extensões C/C++



## Exercícios Práticos Finais

### Projeto Integrador: Sistema de Partículas

#### Especificação:

1. Implementar sistema de N partículas em 2D
2. Cada partícula tem posição, velocidade e massa
3. Simular colisões elásticas entre partículas
4. Visualização em modo texto das trajetórias
5. Análise de energia total do sistema

#### Critérios de Avaliação:

- Uso correto de structs e arrays
- Gerenciamento eficiente de memória
- Implementação de algoritmos físicos
- Tratamento de casos extremos
- Documentação técnica completa

### Desafio Avançado: Interpretador de Comandos

#### Funcionalidades Mínimas:

- Parser de comandos com structs
- Sistema de variáveis dinâmicas
- Operações matemáticas básicas
- Gerenciamento de memória robusto
- Interface de debugging

## Encerramento da Aula

### Algoritmos e Complexidade - Aula 02

*Estruturas de Dados - Homogêneas, Heterogêneas e Ponteiros*

**Próxima Sessão:** Algoritmos de Ordenação e Análise de Performance

**Material Complementar:** Exercícios práticos no repositório GitHub

### Contato e Suporte

**Prof. Vagner Cordeiro**

**GitHub:** [github.com/cordeirotelecom/algoritimos\\_e\\_complexidade](https://github.com/cordeirotelecom/algoritimos_e_complexidade)

**Horários de Atendimento:** [Conforme cronograma da disciplina]

```
void *realloc(void *ptr, size_t novo_tamanho);
```

// Liberação

```
void free(void *ptr);
```

```
---  
  
## Exemplo: Array Dinâmico  
  
```c  
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct {  
    int *dados;  
    int tamanho;  
    int capacidade;  
} ArrayDinamico;  
  
ArrayDinamico* criar_array(int capacidade_inicial) {  
    ArrayDinamico *arr = malloc(sizeof(ArrayDinamico));  
    arr->dados = malloc(capacidade_inicial * sizeof(int));  
    arr->tamanho = 0;  
    arr->capacidade = capacidade_inicial;  
    return arr;  
}  
  
void liberar_array(ArrayDinamico *arr) {  
    free(arr->dados);  
    free(arr);  
}
```

## Redimensionamento Automático

```
int adicionar_elemento(ArrayDinamico *arr, int elemento) {
    if (arr->tamanho >= arr->capacidade) {
        // Dobra a capacidade
        int nova_capacidade = arr->capacidade * 2;
        int *novos_dados = realloc(arr->dados,
                                   nova_capacidade * sizeof(int));

        if (novos_dados == NULL) return 0; // Erro

        arr->dados = novos_dados;
        arr->capacidade = nova_capacidade;
    }

    arr->dados[arr->tamanho++] = elemento;
    return 1;
}
```

**Complexidade Amortizada:**  $O(1)$

## 7. Estruturas Heterogêneas (Structs)

### Definição

Uma struct agrupa dados de tipos diferentes:

```
struct Pessoa {  
    char nome[50];  
    int idade;  
    float altura;  
    char sexo;  
};  
  
// Usando typedef  
typedef struct {  
    int x, y;  
} Ponto;
```

## Operações com Structs

### Declaração e Inicialização

```
// Declaração
struct Pessoa p1;
Ponto origem = {0, 0};

// Inicialização designada (C99)
struct Pessoa p2 = {
    .nome = "João",
    .idade = 25,
    .altura = 1.75,
    .sexo = 'M'
};
```

### Acesso aos Membros

```
p1.idade = 30;
strcpy(p1.nome, "Maria");

// Com ponteiros
struct Pessoa *ptr = &p1;
ptr->idade = 35; // Equivale a (*ptr).idade = 35;
```

## Struct com Arrays e Ponteiros

```
typedef struct {
    int *notas;
    int num_notas;
    char nome[50];
    float media;
} Estudante;

void calcular_media(Estudante *e) {
    int soma = 0;
    for (int i = 0; i < e->num_notas; i++) {
        soma += e->notas[i];
    }
    e->media = (float)soma / e->num_notas;
}
```

## 8. Unions - Compartilhamento de Memória

### Conceito

Uma union permite que diferentes tipos compartilhem a mesma área de memória:

```
union Valor {  
    int inteiro;  
    float real;  
    char caractere;  
};  
  
union Valor v;  
v.inteiro = 42;  
printf("%d\n", v.inteiro);    // 42  
  
v.real = 3.14;  
printf("%f\n", v.real);      // 3.14  
// v.inteiro agora tem valor indefinido
```



## Exemplo Prático: Sistema de Tipos

```
typedef enum {
    TIPO_INT,
    TIPO_FLOAT,
    TIPO_STRING
} TipoDado;

typedef struct {
    TipoDado tipo;
    union {
        int valor_int;
        float valor_float;
        char valor_string[100];
    } dados;
} Variavel;

void imprimir_variavel(Variavel *v) {
    switch (v->tipo) {
        case TIPO_INT:
            printf("%d\n", v->dados.valor_int);
            break;
        case TIPO_FLOAT:
            printf("%.2f\n", v->dados.valor_float);
            break;
        case TIPO_STRING:
            printf("%s\n", v->dados.valor_string);
            break;
    }
}
```

## 9. Comparação C vs Python

### Arrays em Python (Lists)

```
# Lista dinâmica
numeros = [1, 2, 3, 4, 5]
numeros.append(6)      # O(1) amortizado
numeros.insert(2, 10)   # O(n)
del numeros[1]          # O(n)

# List comprehension
quadrados = [x**2 for x in range(10)]

# Slicing
sub_lista = numeros[1:4] # [2, 10, 4]
```

### Numpy Arrays (Homogêneos)

```
import numpy as np

# Array homogêneo
arr = np.array([1, 2, 3, 4, 5])
matriz = np.array([[1, 2], [3, 4]])

# Operações vetorizadas
resultado = arr * 2      # [2, 4, 6, 8, 10]
produto = matriz @ matriz # Multiplicação matricial
```

# 10. Análise de Complexidade

## Comparação de Operações

Operação	Array C	Python List	Numpy Array
Acesso	$O(1)$	$O(1)$	$O(1)$
Busca	$O(n)$	$O(n)$	$O(n)$
Inserção final	$O(1)$	$O(1)$ amort.	$O(n)$
Inserção meio	$O(n)$	$O(n)$	$O(n)$
Remoção	$O(n)$	$O(n)$	$O(n)$

## Consumo de Memória

### C - Array Estático

```
int arr[1000]; // 4000 bytes (exato)
```

### C - Struct

```
struct Exemplo {  
    char c;    // 1 byte  
    int i;     // 4 bytes  
    double d;  // 8 bytes  
}; // Total: pode ser 16 bytes (com padding)
```

### Padding e Alinhamento

$$\text{sizeof(struct)} \geq \sum \text{sizeof(membros)}$$

## 11. Algoritmos de Ordenação

### Bubble Sort

```
void bubble_sort(int arr[], int n) {  
    for (int i = 0; i < n-1; i++) {  
        int houve_troca = 0;  
        for (int j = 0; j < n-i-1; j++) {  
            if (arr[j] > arr[j+1]) {  
                trocar(&arr[j], &arr[j+1]);  
                houve_troca = 1;  
            }  
        }  
        if (!houve_troca) break; // Otimização  
    }  
}
```

#### Complexidade:

- Melhor caso:  $O(n)$  (já ordenado)
- Caso médio/pior:  $O(n^2)$

## Selection Sort

```
void selection_sort(int arr[], int n) {  
    for (int i = 0; i < n-1; i++) {  
        int min_idx = i;  
  
        // Encontra o menor elemento  
        for (int j = i+1; j < n; j++) {  
            if (arr[j] < arr[min_idx]) {  
                min_idx = j;  
            }  
        }  
  
        // Troca se necessário  
        if (min_idx != i) {  
            trocar(&arr[i], &arr[min_idx]);  
        }  
    }  
}
```

**Complexidade:**  $O(n^2)$  sempre

## 12. Strings em C

### Representação

```
char str1[20] = "Hello";    // Array de caracteres
char *str2 = "World";       // Ponteiro para literal
char str3[] = {'H', 'i', '\0'}; // Inicialização char por char
```

### Funções da string.h

```
#include <string.h>

strlen(str);           // Comprimento
strcpy(dest, src);     // Cópia
strcat(dest, src);     // Concatenação
strcmp(str1, str2);    // Comparação
```

## Implementação de strlen

```
size_t meu_strlen(const char *str) {
    size_t len = 0;
    while (str[len] != '\0') {
        len++;
    }
    return len;
}

// Versão com ponteiro
size_t strlen_ptr(const char *str) {
    const char *inicio = str;
    while (*str) str++;
    return str - inicio;
}
```

**Complexidade:**  $O(n)$  onde  $n$  é o comprimento da string



## 13. Estruturas de Dados Avançadas

### Lista Ligada com Struct

```
typedef struct No {  
    int dado;  
    struct No *proximo;  
} No;  
  
typedef struct {  
    No *cabeca;  
    int tamanho;  
} Lista;  
  
void inserir_inicio(Lista *lista, int valor) {  
    No *novo = malloc(sizeof(No));  
    novo->dado = valor;  
    novo->proximo = lista->cabeca;  
    lista->cabeca = novo;  
    lista->tamanho++;  
}
```

## Pilha (Stack) com Array

```
#define MAX_SIZE 100

typedef struct {
    int dados[MAX_SIZE];
    int topo;
} Pilha;

void push(Pilha *p, int valor) {
    if (p->topo < MAX_SIZE - 1) {
        p->dados[++p->topo] = valor;
    }
}

int pop(Pilha *p) {
    if (p->topo >= 0) {
        return p->dados[p->topo--];
    }
    return -1; // Pilha vazia
}
```

**Complexidade:**  $O(1)$  para push e pop

## 14. Aplicações Práticas

### Sistema de Cadastro

```
typedef struct {
    int id;
    char nome[50];
    char email[100];
    float salario;
} Funcionario;

typedef struct {
    Funcionario *funcionarios;
    int quantidade;
    int capacidade;
} BaseDados;

int buscar_por_id(BaseDados *db, int id) {
    for (int i = 0; i < db->quantidade; i++) {
        if (db->funcionarios[i].id == id) {
            return i;
        }
    }
    return -1;
}
```

## Matriz Esparsa

```
typedef struct {
    int linha;
    int coluna;
    double valor;
} Elemento;

typedef struct {
    Elemento *elementos;
    int num_elementos;
    int linhas;
    int colunas;
} MatrizEsparsa;

void adicionar_elemento(MatrizEsparsa *m, int i, int j, double valor) {
    if (valor != 0.0) {
        m->elementos[m->num_elementos++] = (Elemento){i, j, valor};
    }
}
```

## 15. Otimizações e Considerações

### Cache Locality

```
// Bom para cache (row-major)
for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
        matriz[i][j] = i + j;
    }
}

// Ruim para cache (column-major em C)
for (int j = 0; j < COLS; j++) {
    for (int i = 0; i < ROWS; i++) {
        matriz[i][j] = i + j; // Acesso não sequencial
    }
}
```

## Memory Alignment

```
// Sem padding otimizado
struct Ineficiente {
    char a;    // 1 byte
    double b;  // 8 bytes (7 bytes de padding)
    char c;    // 1 byte (7 bytes de padding no final)
}; // Total: 24 bytes

// Com padding otimizado
struct Eficiente {
    double b;  // 8 bytes
    char a;    // 1 byte
    char c;    // 1 byte (6 bytes de padding)
}; // Total: 16 bytes
```

## 16. Depuração e Testes

### Validação de Ponteiros

```
void funcao_segura(int *ptr) {  
    if (ptr == NULL) {  
        printf("Erro: ponteiro nulo!\n");  
        return;  
    }  
  
    // Usar o ponteiro seguramente  
    *ptr = 42;  
}
```

### Detecção de Memory Leaks

```
void testar_memoria() {  
    int *arr = malloc(100 * sizeof(int));  
  
    // ... usar array ...  
  
    free(arr); // IMPORTANTE: sempre liberar  
    arr = NULL; // Boa prática  
}
```

## 17. Benchmarking

### Medição de Performance

```
#include <time.h>

void benchmark_algoritmo() {
    clock_t inicio = clock();

    // Algoritmo a ser testado
    for (int i = 0; i < 1000000; i++) {
        // Operação repetitiva
    }

    clock_t fim = clock();
    double tempo = ((double)(fim - inicio)) / CLOCKS_PER_SEC;
    printf("Tempo: %.6f segundos\n", tempo);
}
```



## 18. Padrões de Design

### Factory Pattern para Structs

```
Pessoa* criar_pessoa(const char *nome, int idade) {
    Pessoa *p = malloc(sizeof(Pessoa));
    if (p != NULL) {
        strncpy(p->nome, nome, sizeof(p->nome) - 1);
        p->nome[sizeof(p->nome) - 1] = '\0';
        p->idade = idade;
    }
    return p;
}

void destruir_pessoa(Pessoa *p) {
    free(p);
}
```

## 19. Comparação Final: C vs Python

### Vantagens do C:

- **Performance:** 10-100x mais rápido
- **Controle de memória:** Gestão precisa
- **Previsibilidade:** Comportamento determinístico
- **Eficiência espacial:** Menor overhead

### Vantagens do Python:

- **Produtividade:** Desenvolvimento mais rápido
- **Flexibilidade:** Tipos dinâmicos
- **Bibliotecas:** NumPy, SciPy, Pandas
- **Expressividade:** Código mais conciso

## 20. Conclusões e Próximos Passos

### O que Aprendemos:

- **Estruturas homogêneas** (arrays) e heterogêneas (structs)
- **Ponteiros** e gerenciamento de memória
- **Algoritmos** fundamentais em estruturas
- **Análise de complexidade** e otimizações
- **Comparações** entre linguagens

### Próxima Aula:

- **Análise de Algoritmos** e complexidade computacional
- **Notações assintóticas** avançadas
- **Técnicas de análise** matemática
- **Casos práticos** de otimização

## Exercícios Propostos

1. Implemente uma calculadora de matrizes completa
2. Crie um sistema de gerenciamento de estudantes usando structs
3. Desenvolva um array dinâmico genérico (void\*)
4. Compare performance: array estático vs dinâmico vs Python list
5. Implemente ordenação eficiente para structs

## Bibliografia

- **Cormen, T. H.** et al. *Introduction to Algorithms*, 4<sup>a</sup> ed.
- **Kernighan, B. W.; Ritchie, D. M.** *The C Programming Language*, 2<sup>a</sup> ed.
- **Sedgewick, R.** *Algorithms in C*, 3<sup>a</sup> ed.
- **Tanenbaum, A. S.** *Structured Computer Organization*, 6<sup>a</sup> ed.

## Contato e Dúvidas

**Prof. Vagner Cordeiro**

 **Email:** [email do professor]

 **Atendimento:** [horários de atendimento]

 **Material:** [github.com/cordeirotelecom/algoritimos\\_e\\_complexidade](https://github.com/cordeirotelecom/algoritimos_e_complexidade)

**Próxima aula:** Análise de Algoritmos e Prática de Análise

**Obrigado!**

**Perguntas?**

**Algoritmos e Complexidade - Aula 02**

*Estruturas de Dados - Homogêneas, Heterogêneas e Ponteiros*