Vetores em Estruturas de Dados: Conceitos, Implementação e Aplicações

1. Introdução aos Vetores

Vetores, também chamados de **arrays**, são uma das estruturas de dados mais fundamentais em computação. Eles consistem em uma sequência **contígua** de elementos do mesmo tipo, armazenados na memória de forma ordenada. Essa organização permite **acesso rápido a qualquer elemento** por meio de um índice, tornando os vetores uma escolha eficiente para armazenamento e manipulação de dados.

Em muitas linguagens de programação, incluindo C, C++, Java e Python, os vetores são amplamente utilizados em diversas aplicações, como **algoritmos de ordenação, busca, armazenamento de grandes quantidades de dados e representação de matrizes.**

---

2. Características dos Vetores

Os vetores possuem características fundamentais que os diferenciam de outras estruturas de dados, como listas encadeadas ou pilhas:

1. **Acesso direto e rápido**: Qualquer elemento pode ser acessado diretamente por meio de seu índice em tempo constante **O(1)**.
2. **Tamanho fixo**: Em muitas linguagens, o tamanho do vetor precisa ser definido no momento da alocação e não pode ser alterado dinamicamente sem realocação.
3. **Eficiência na leitura e escrita**: Operações de leitura e escrita são extremamente rápidas devido à alocação contígua na memória.
4. **Dificuldade na inserção e remoção de elementos**: Adicionar ou remover elementos no meio do vetor exige deslocamento de dados, resultando em complexidade **O(n)** no pior caso.
5. **Uso eficiente de memória**: Como os elementos são armazenados de forma contígua, o uso da memória é otimizado e não há sobrecarga de ponteiros, como acontece em listas encadeadas.

---

3. Declaração e Inicialização de Vetores

Em C, um vetor pode ser declarado de maneira simples especificando seu tipo e tamanho:

```c
int numeros[5]; // Vetor de 5 inteiros
char letras[10]; // Vetor de 10 caracteres
float valores[3] = {1.5, 2.3, 4.7}; // Vetor inicializado
```

O índice dos elementos começa em `0` e vai até `n-1`, onde `n` é o tamanho do vetor.

Exemplo de acesso a elementos:

```c
int vetor[3] = {10, 20, 30};
printf("%d\n", vetor[1]); // Saída: 20
```

---

4. Operações com Vetores

Os vetores permitem diversas operações fundamentais:

4.1 Percorrer um Vetor

Usamos um loop `for` para percorrer todos os elementos do vetor:

```c
#include <stdio.h>

int main() {
    int numeros[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; i++) {
        printf("%d ", numeros[i]);
    }

    return 0;
}
```

4.2 Inserção de Elementos

A inserção em um vetor estático só pode ser feita **substituindo valores existentes** ou **realocando memória** em um vetor dinâmico.

Para adicionar um elemento no final de um vetor dinâmico:

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

int main() {
    int capacidade = 2, tamanho = 0;
    int *vetor = (int *)malloc(capacidade * sizeof(int));

    for (int i = 0; i < 5; i++) {
        if (tamanho == capacidade) {
            capacidade *= 2; // Dobra a capacidade
            vetor = (int *)realloc(vetor, capacidade * sizeof(int));
        }
        vetor[tamanho++] = i * 10;
    }

    free(vetor); // Libera memória alocada
```

```
    return 0;
}
```

4.3 Remoção de Elementos

A remoção de um elemento requer o deslocamento dos elementos à direita:

```c
#include <stdio.h>

void removerElemento(int vetor[], int *tamanho, int indice) {
    for (int i = indice; i < *tamanho - 1; i++) {
        vetor[i] = vetor[i + 1];
    }
    (*tamanho)--;
}

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

    removerElemento(vetor, &tamanho, 2);

    for (int i = 0; i < tamanho; i++) {
        printf("%d ", vetor[i]);
    }

    return 0;
}
```

---

5. Busca e Ordenação em Vetores

5.1 Busca Linear

A busca linear percorre todo o vetor até encontrar o elemento desejado. Tem complexidade **O(n)**.

```c
int buscaLinear(int vetor[], int tamanho, int chave) {
    for (int i = 0; i < tamanho; i++) {
        if (vetor[i] == chave) return i;
    }
    return -1;
}
```

5.2 Busca Binária

Requer um vetor ordenado e tem complexidade **O(log n)**.

```c
int buscaBinaria(int vetor[], int esq, int dir, int chave) {
    while (esq <= dir) {
        int meio = esq + (dir - esq) / 2;
        if (vetor[meio] == chave) return meio;
        if (vetor[meio] < chave) esq = meio + 1;
        else dir = meio - 1;
    }
    return -1;
}
```

5.3 Ordenação com Bubble Sort

```c
void bubbleSort(int vetor[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (vetor[j] > vetor[j + 1]) {
                int temp = vetor[j];
                vetor[j] = vetor[j + 1];
                vetor[j + 1] = temp;
            }
        }
    }
}
```

---

6. Aplicações Práticas de Vetores

Os vetores são usados em diversos cenários:

1. **Armazenamento de dados estruturados** (exemplo: armazenar notas de alunos).
2. **Manipulação de imagens e áudio**, onde os dados são armazenados em arrays multidimensionais.
3. **Algoritmos de machine learning** utilizam arrays para armazenar vetores de entrada.
4. **Gerenciamento de filas e pilhas** em estruturas de dados mais complexas.
5. **Simulação e modelagem científica** onde grandes volumes de dados precisam ser manipulados.

Os **vetores** são uma estrutura de dados poderosa e eficiente para armazenar e acessar elementos sequenciais. Apesar de apresentarem dificuldades em operações de inserção e remoção, seu **acesso direto em tempo constante** os torna ideais para muitas aplicações. Além disso, a **alocação dinâmica de memória** permite superar a limitação de tamanho fixo, tornando-os ainda mais versáteis.

Compreender **busca, ordenação e manipulação dinâmica** de vetores é essencial para programadores que desejam desenvolver software eficiente e otimizado.

8. Vetores Multidimensionais

Até agora, discutimos vetores unidimensionais (ou seja, arrays simples). No entanto, muitas aplicações exigem a manipulação de **dados em múltiplas dimensões**, como matrizes (tabelas de valores), imagens e grafos.

8.1 Declaração e Acesso a Vetores Bidimensionais

Em C, podemos declarar um vetor bidimensional (matriz) da seguinte forma:

```
int matriz[3][3]; // Matriz 3x3 de inteiros
```

Cada elemento da matriz pode ser acessado por meio de dois índices:

```
matriz[0][1] = 5; // Define o elemento na primeira linha, segunda coluna
printf("%d\n", matriz[0][1]); // Saída: 5
```

Também é possível inicializar uma matriz diretamente:

```
int matriz[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

8.2 Percorrendo uma Matriz

Para percorrer todos os elementos de uma matriz, usamos um **loop aninhado**:

```
#include <stdio.h>

int main() {
    int matriz[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Saída:

```
1 2 3
4 5 6
7 8 9
```

8.3 Matrizes Dinâmicas

Como os vetores simples, as matrizes também podem ser alocadas dinamicamente. No entanto, para alocar uma matriz dinamicamente, usamos **ponteiros duplos**.

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

int main() {
    int linhas = 3, colunas = 3;
    int **matriz = (int **)malloc(linhas * sizeof(int *));

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

    // Preenchendo a matriz com valores
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            matriz[i][j] = i * colunas + j + 1;
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    // Liberando a memória alocada
    for (int i = 0; i < linhas; i++) {
        free(matriz[i]);
    }
    free(matriz);

    return 0;
}
```

Aqui, alocamos um vetor de ponteiros (`int **matriz`), onde cada elemento aponta para um vetor de inteiros representando uma linha da matriz.

---

9. Algoritmos Aplicados a Vetores e Matrizes

Agora que entendemos como vetores e matrizes funcionam, vamos explorar alguns algoritmos comuns que usam essas estruturas.

9.1 Soma de Todos os Elementos de um Vetor

```c
#include <stdio.h>

int somaVetor(int vetor[], int tamanho) {
    int soma = 0;
    for (int i = 0; i < tamanho; i++) {
        soma += vetor[i];
    }
    return soma;
}

int main() {
    int numeros[] = {10, 20, 30, 40, 50};
    int resultado = somaVetor(numeros, 5);
    printf("Soma dos elementos: %d\n", resultado);
    return 0;
}
```

9.2 Multiplicação de Matrizes

```c
#include <stdio.h>

#define N 2 // Tamanho da matriz

void multiplicarMatrizes(int A[N][N], int B[N][N], int C[N][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];
            }
        }
    }
}

void imprimirMatriz(int matriz[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }
}

int main() {
```

```c
    int A[N][N] = {{1, 2}, {3, 4}};
    int B[N][N] = {{5, 6}, {7, 8}};
    int C[N][N];

    multiplicarMatrizes(A, B, C);
    printf("Resultado da multiplicação de matrizes:\n");
    imprimirMatriz(C);

    return 0;
}
```

Saída:

```
Resultado da multiplicação de matrizes:
19 22
43 50
```

---

10. Vetores vs. Outras Estruturas de Dados

Embora os vetores sejam eficientes em termos de acesso direto aos elementos, eles apresentam algumas desvantagens quando comparados a outras estruturas de dados, como **listas encadeadas e árvores**.

Característica	Vetor	Lista Encadeada	Árvore Binária
Acesso Direto	O(1)	O(n)	O(log n)
Inserção/Remoção	O(n)	O(1) (em qualquer posição)	O(log n)
Uso de Memória	Contígua	Fragmentada	Estruturada
Busca Sequencial	O(n)	O(n)	O(n)
Busca Binária	O(log n) (se ordenado)	O(n)	O(log n)

Os **vetores são ideais para cenários onde acesso rápido a elementos individuais é necessário**, enquanto **listas encadeadas são melhores quando há inserção e remoção frequente**.

---

11. Conclusão

Os vetores são uma das estruturas de dados mais fundamentais da ciência da computação. São fáceis de usar e extremamente eficientes para armazenar e acessar dados sequenciais. No entanto, exigem cuidado especial para gerenciar seu tamanho e memória alocada dinamicamente.

Resumo dos pontos abordados:

- **Vetores unidimensionais e bidimensionais**
- **Acesso, inserção, remoção e manipulação de dados**
- **Busca e ordenação em vetores**
- **Alocação dinâmica e operações complexas**
- **Comparação entre vetores e outras estruturas de dados**

Dominar vetores é essencial para programadores e cientistas da computação, pois servem de base para algoritmos avançados e otimização de desempenho.

9. Operações Comuns em Matrizes

Além das operações básicas de soma e multiplicação, existem diversas outras operações úteis e fundamentais ao trabalhar com matrizes. Aqui estão algumas das operações mais comuns e suas explicações:

9.1 Transposição de uma Matriz

A **transposta** de uma matriz é uma nova matriz obtida trocando suas linhas por colunas. Se a matriz A é de ordem $m \times n$, a transposta de A será uma matriz A^T de ordem $n \times m$.

Exemplo:
Se temos uma matriz A:

$$
A =
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
5 & 6 \\
\end{bmatrix}
$$

A sua transposta A^T será:

$$
A^T =
\begin{bmatrix}
1 & 3 & 5 \\
2 & 4 & 6 \\
\end{bmatrix}
$$

Código para transposição:

```c
#include <stdio.h>

void transposta(int A[3][2], int T[2][3]) {
    for (int i = 0; i < 3; i++) {
```

```
            for (int j = 0; j < 2; j++) {
                T[j][i] = A[i][j];
            }
        }
    }

    int main() {
        int A[3][2] = {
            {1, 2},
            {3, 4},
            {5, 6}
        };
        int T[2][3];

        transposta(A, T);

        // Imprimindo a matriz transposta
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
                printf("%d ", T[i][j]);
            }
            printf("\n");
        }

        return 0;
    }
```

Saída:

```
1 3 5
2 4 6
```

9.2 Multiplicação de Matrizes

A multiplicação de matrizes é uma operação que combina duas matrizes para gerar uma nova. Para que duas matrizes A e B possam ser multiplicadas, o número de **colunas** de A deve ser igual ao número de **linhas** de B. O produto da multiplicação resulta em uma matriz C, onde cada elemento $c[i][j]$ é a soma do produto de elementos correspondentes das linhas de A e das colunas de B.

Exemplo de multiplicação:
Se A for uma matriz 2×3 e B for uma matriz 3×2:

$$
A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad
B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}
$$

A multiplicação $C = A \times B$ resultará em uma matriz 2×2:

$$
C = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}
$$

Código para multiplicação de matrizes:

```c
#include <stdio.h>

void multiplicarMatrizes(int A[2][3], int B[3][2], int C[2][2]) {
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            C[i][j] = 0;
            for (int k = 0; k < 3; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int A[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };
    int B[3][2] = {
        {7, 8},
        {9, 10},
        {11, 12}
    };
    int C[2][2];

    multiplicarMatrizes(A, B, C);

    // Imprimindo a matriz resultante
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Saída:

```
58 64
139 154
```

9.3 Determinante de uma Matriz

O **determinante** de uma matriz quadrada $n \times n$ é um número que pode ser calculado a partir de seus elementos, com várias aplicações em álgebra linear, como resolver sistemas de equações lineares e verificar a inversibilidade de uma matriz. O cálculo do determinante é mais simples para matrizes de 2×2 e 3×3, mas pode ser complexo para matrizes maiores, geralmente sendo calculado usando recursão ou o método de eliminação de Gauss.

Fórmula para o determinante de uma matriz 2×2:

$$
\text{det}(A) = a \times d - b \times c
$$

Onde:

$$
A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}
$$

Código para calcular o determinante de uma matriz 2×2:

```c
#include <stdio.h>

int determinante(int A[2][2]) {
    return A[0][0] * A[1][1] - A[0][1] * A[1][0];
}

int main() {
    int A[2][2] = {
        {1, 2},
        {3, 4}
    };

    int det = determinante(A);
    printf("Determinante: %d\n", det);

    return 0;
}
```

Saída:

```
Determinante: -2
```

9.4 Inversão de Matrizes

A **inversão de uma matriz** é o processo de encontrar uma matriz A^{-1} tal que $A \times A^{-1} = I$, onde I é a matriz identidade (uma matriz com 1s na diagonal principal e 0s em outros lugares). Somente matrizes quadradas possuem inversa, e a matriz deve ser **não singular** (determinante diferente de zero).

Cálculo para matrizes 2×2:
Se A for uma matriz 2×2:

$$
A = \begin{bmatrix} a & b \ c & d \end{bmatrix}
$$

A inversa de A é dada por:

$$
A^{-1} = \frac{1}{\text{det}(A)} \begin{bmatrix} d & -b \ -c & a \end{bmatrix}
$$

Código para encontrar a inversa de uma matriz 2×2:

```c
#include <stdio.h>

void inversa(int A[2][2], float A_inv[2][2]) {
    int det = A[0][0] * A[1][1] - A[0][1] * A[1][0];

    if (det != 0) {
        float inv_det = 1.0 / det;
        A_inv[0][0] = A[1][1] * inv_det;
        A_inv[0][1] = -A[0][1] * inv_det;
        A_inv[1][0] = -A[1][0] * inv_det;
        A_inv[1][1] = A[0][0] * inv_det;
    } else {
        printf("Matriz singular, não pode ser invertida.\n");
    }
}

int main() {
    int A[2][2] = {
        {1, 2},
        {3, 4}
    };
    float A_inv[2][2];

    inversa(A, A_inv);

    // Imprimindo a matriz inversa
    printf("Matriz Inversa:\n");
    printf("%.2f %.2f\n", A_inv[0][0], A_inv[0][1]);
    printf("%.2f %.2f\n", A_inv[1][0], A_inv[1][1]);
```

```
    return 0;
}
```

Saída:

```
Matriz Inversa:
-2.00 1.00
1.50 -0.50
```

Em C, é comum passar vetores e matrizes para funções para manipulação de dados. A passagem de vetores e matrizes funciona de maneira semelhante, mas há algumas nuances a serem compreendidas. Aqui, abordaremos a passagem de vetores e matrizes para funções, detalhando os conceitos e fornecendo exemplos.

10. Passagem de Vetores para Funções

Vetores em C são, na verdade, ponteiros para o primeiro elemento da lista de dados. Quando passamos um vetor para uma função, estamos passando o endereço do primeiro elemento do vetor, e qualquer modificação feita dentro da função afetará o vetor original.

Sintaxe para passar um vetor para uma função:

```c
void minha_funcao(int vetor[], int tamanho) {
    // Aqui você pode manipular o vetor
}
```

Observe que, ao passar um vetor para uma função, passamos o nome do vetor, que na verdade é um ponteiro para o primeiro elemento.

Exemplo 1: Passando um Vetor para uma Função

Aqui está um exemplo de como passar um vetor para uma função que altera seus valores:

```c
#include <stdio.h>

void incrementar(int vetor[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        vetor[i] += 1; // Incrementa cada elemento
    }
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int tamanho = sizeof(arr) / sizeof(arr[0]);
```

```
    printf("Antes de incrementar: ");
    for (int i = 0; i < tamanho; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Passando o vetor para a função
    incrementar(arr, tamanho);

    printf("Depois de incrementar: ");
    for (int i = 0; i < tamanho; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Saída:

```
Antes de incrementar: 1 2 3 4 5
Depois de incrementar: 2 3 4 5 6
```

10.1 Passagem de Matrizes para Funções

Matrizes também são passadas para funções como ponteiros, mas devido à sua estrutura bidimensional, a forma de passagem é ligeiramente diferente.

Sintaxe para passar uma matriz para uma função:

```
void minha_funcao(int matriz[][COLUNAS], int linhas) {
    // Aqui você pode manipular a matriz
}
```

Note que precisamos especificar o número de colunas na definição da matriz, mas o número de linhas pode ser flexível. Também é possível usar o ponteiro para uma matriz bidimensional, mas a forma mais comum é usar a notação de `matriz[][]`.

Exemplo 2: Passando uma Matriz para uma Função

Aqui está um exemplo de como passar uma matriz para uma função e realizar uma operação, como somar uma constante a todos os seus elementos:

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

```c
#define LINHAS 3
#define COLUNAS 3

void somar_constante(int matriz[LINHAS][COLUNAS], int constante) {
    for (int i = 0; i < LINHAS; i++) {
        for (int j = 0; j < COLUNAS; j++) {
            matriz[i][j] += constante; // Soma a constante a cada elemento
        }
    }
}

int main() {
    int matriz[LINHAS][COLUNAS] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    printf("Matriz antes da soma:\n");
    for (int i = 0; i < LINHAS; i++) {
        for (int j = 0; j < COLUNAS; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    // Passando a matriz para a função
    somar_constante(matriz, 5);

    printf("Matriz depois da soma de 5:\n");
    for (int i = 0; i < LINHAS; i++) {
        for (int j = 0; j < COLUNAS; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Saída:

```
Matriz antes da soma:
1 2 3
4 5 6
7 8 9
Matriz depois da soma de 5:
6 7 8
9 10 11
12 13 14
```

10.3 Passagem de Matrizes Dinâmicas para Funções

Se você estiver usando alocação dinâmica para criar uma matriz (com `malloc` ou `calloc`), a passagem para funções será um pouco diferente. Em vez de passar o nome da matriz, você passará o ponteiro para o primeiro elemento da matriz alocada dinamicamente.

Exemplo 3: Matrizes Dinâmicas

Aqui vamos alocar dinamicamente uma matriz 2x2 e passar para uma função para alterar seus elementos.

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

void modificar_matriz(int **matriz, int linhas, int colunas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            matriz[i][j] *= 2; // Multiplica cada elemento por 2
        }
    }
}

int main() {
    int linhas = 2, colunas = 2;

    // Alocando memória para a matriz dinamicamente
    int **matriz = (int **)malloc(linhas * sizeof(int *));
    for (int i = 0; i < linhas; i++) {
        matriz[i] = (int *)malloc(colunas * sizeof(int));
    }

    // Inicializando a matriz
    matriz[0][0] = 1; matriz[0][1] = 2;
    matriz[1][0] = 3; matriz[1][1] = 4;

    printf("Matriz antes da modificação:\n");
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }

    // Passando a matriz para a função
    modificar_matriz(matriz, linhas, colunas);

    printf("Matriz depois da modificação:\n");
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            printf("%d ", matriz[i][j]);
        }
```

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

    // Liberando memória alocada
    for (int i = 0; i < linhas; i++) {
        free(matriz[i]);
    }
    free(matriz);

    return 0;
}
```

Saída:

```
Matriz antes da modificação:
1 2
3 4
Matriz depois da modificação:
2 4
6 8
```

10.4 Passagem de Matrizes para Funções com Ponteiros

Em vez de passar uma matriz bidimensional diretamente para uma função, você pode passar um ponteiro para um bloco de memória contínuo, o que pode ser útil em certas situações de alocação dinâmica.

Exemplo 4: Usando Ponteiros para Passar Matrizes

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

void modificar_matriz(int *matriz, int linhas, int colunas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            matriz[i * colunas + j] += 10; // Adiciona 10 a cada elemento
        }
    }
}

int main() {
    int linhas = 2, colunas = 2;

    // Alocando memória para a matriz dinamicamente
    int *matriz = (int *)malloc(linhas * colunas * sizeof(int));

    // Inicializando a matriz
    matriz[0] = 1; matriz[1] = 2;
    matriz[2] = 3; matriz[3] = 4;
```

```c
    printf("Matriz antes da modificação:\n");
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            printf("%d ", matriz[i * colunas + j]);
        }
        printf("\n");
    }

    // Passando a matriz para a função
    modificar_matriz(matriz, linhas, colunas);

    printf("Matriz depois da modificação:\n");
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            printf("%d ", matriz[i * colunas + j]);
        }
        printf("\n");
    }

    // Liberando memória alocada
    free(matriz);

    return 0;
}
```

Saída:

```
Matriz antes da modificação:
1 2
3 4
Matriz depois da modificação:
11 12
13 14
```

Considerações Finais

- **Vetores** são passados por referência para funções, ou seja, qualquer modificação dentro da função afetará o vetor original.
- **Matrizes** funcionam de forma semelhante a vetores em termos de passagem de dados, mas, por serem bidimensionais, exigem um pouco mais de atenção na manipulação e nas funções.
- **Matrizes dinâmicas** podem ser alocadas dinamicamente usando `malloc` ou `calloc`, e sua passagem para funções pode ser feita com ponteiros.

Conclusão

Matrizes são uma parte fundamental das estruturas de dados e são amplamente utilizadas em computação científica, processamento de imagens, gráficos, álgebra linear e em muitas outras áreas. O conhecimento sobre como manipular matrizes é essencial para a construção de algoritmos eficientes e para a resolução de problemas complexos em muitas disciplinas da ciência da computação.

A compreensão de operações como soma, multiplicação, transposição, inversão e determinantes é uma habilidade valiosa, e ao dominá-las, o programador pode implementar soluções poderosas e eficientes para problemas que envolvem manipulação de dados em múltiplas dimensões.