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 C#, os vetores são amplamente utilizados em diversas aplicações, como **algoritmos de ordenação, busca, armazenamento de grandes quantidades de dados e representação de matrizes.**

Nem todas as linguagens de programação possui o tipo **vetor** definido nessa aula, um exemplo clássico é a linguagem Python que não tem um tipo de dado nativo chamado **"vetor"** porque **sua estrutura principal para armazenar coleções de elementos é a lista (list)**, que é flexível e pode conter diferentes tipos de dados. No entanto, listas não são otimizadas para operações matemáticas vetorizadas, como soma e multiplicação elemento a elemento. Se você precisa de vetores no sentido matemático (como em álgebra linear), a solução mais eficiente é usar a biblioteca NumPy, que oferece a classe numpy.array.

Dica: É interessante ao estudar uma nova linguagem estudar como é definido a estrutura de dados para poder fazer o melhor uso.

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.

---

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.

Conceitos básico

- **Índice do vetor**: os elementos são identificados por seus índices. O índice da matriz começa a partir de 0.
- **Elemento de vetor**: Os elementos são itens armazenados e podem ser acessados por seu índice.
- **Comprimento do vetor**: O comprimento é determinado pelo número de elementos que ela pode conter.

Exemplo conceitual

Índice:	**0**	**1**	**2**	**3**	**4**
Valor:	10	20	30	40	50
Endereço Memória:	xB0451fa0	xB0451fa4	xB0451fa8	xB0415fac	xB0415fb0

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[0]); // Saída: 10
printf("%d\n", vetor[1]); // Saída: 20
printf("%d\n", vetor[3]); // Saída: 30
```

---

4. Operações com Vetores

Os vetores permitem diversas operações fundamentais:

4.1 Percorrer um Vetor

Percorrer um vetor é uma das operações mais comuns e consiste em passar pelos elementos das matriz. Podemos **usar qualquer estrutura de repetição**. Abaixo 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. Sintaxe para passar um vetor para uma função:

Em C, os **vetores (arrays)** são passados para funções de uma maneira diferente de variáveis comuns. Isso ocorre porque **o nome de um array em C é um ponteiro para seu primeiro elemento**. Assim, quando passamos um vetor como argumento para uma função, o que realmente estamos passando é um **ponteiro** para o primeiro elemento do vetor, e não uma cópia dos dados. Isso significa que qualquer modificação feita dentro da função **afeta o vetor original**.

5.1 Como Passar um Vetor para uma Função?

Para passar um vetor para uma função em C, utilizamos a seguinte sintaxe:

```c
void minhaFuncao(int array[], int tamanho);
```

Aqui, `array[]` indica que a função espera um vetor como argumento, enquanto `tamanho` representa a quantidade de elementos no vetor. Passar o tamanho do vetor como parâmetro é necessário porque C **não armazena automaticamente o tamanho de arrays passados para funções**.

5.2. Exemplo Simples: Exibir os Elementos de um Vetor

Vamos criar uma função que recebe um vetor e seu tamanho, e imprime seus elementos.

```c
#include <stdio.h>

// Função que recebe um vetor e imprime seus elementos
void imprimirVetor(int array[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int numeros[] = {1, 2, 3, 4, 5}; // Definição do vetor
    int tamanho = sizeof(numeros) / sizeof(numeros[0]); // Calcula o
tamanho do vetor

    imprimirVetor(numeros, tamanho); // Passa o vetor para a função

    return 0;
}
```

Saída:

```
1 2 3 4 5
```

> **Explicação:**
>
> - `numeros[]` é passado para `imprimirVetor()`, mas internamente ele é tratado como um ponteiro.
> - A função percorre o vetor usando um laço `for` e imprime seus elementos.

5.3. Modificando um Vetor dentro da Função

Como os vetores são passados **por referência**, qualquer alteração feita dentro da função **afeta o vetor original**. Veja um exemplo:

```c
#include <stdio.h>

// Função que multiplica cada elemento do vetor por 2
void dobrarElementos(int array[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        array[i] *= 2; // Modifica o valor diretamente no vetor original
    }
}

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

```c
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);

    dobrarElementos(numeros, tamanho); // Modifica os valores do vetor

    // Exibir vetor modificado
    for (int i = 0; i < tamanho; i++) {
        printf("%d ", numeros[i]);
    }
    printf("\n");

    return 0;
}
```

Saída:

```
2 4 6 8 10
```

> **Explicação:**
>
> - A função `dobrarElementos()` altera diretamente os valores do vetor original.
> - Isso acontece porque a função recebe **um ponteiro para o primeiro elemento do vetor**, e não uma cópia dos dados.

5.4. Passando Vetores como Ponteiros

Outra forma de definir uma função que recebe um vetor é utilizando **notação de ponteiros**:

```c
void minhaFuncao(int *array, int tamanho);
```

Essa abordagem é **equivalente** a `int array[]`, pois ambos representam um **ponteiro para o primeiro elemento do vetor**. Veja um exemplo:

```c
#include <stdio.h>

// Função que imprime os elementos do vetor usando notação de ponteiros
void imprimirComPonteiros(int *array, int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        printf("%d ", *(array + i)); // Acessa os elementos via
ponteiros
    }
    printf("\n");
}

int main() {
```

```
    int numeros[] = {10, 20, 30, 40, 50};
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);

    imprimirComPonteiros(numeros, tamanho); // Passa o vetor para a
função

    return 0;
}
```

Saída:

```
10 20 30 40 50
```

> **Explicação:**
>
> - `*(array + i)` acessa cada elemento do vetor diretamente na memória.
> - Isso mostra que `array[i]` e `*(array + i)` são equivalentes.

- **Vetores em C são passados por referência** porque o nome do vetor é um ponteiro para seu primeiro elemento.
- Para evitar problemas, **sempre passe o tamanho do vetor como parâmetro**.
- Podemos acessar os elementos usando **notação de índice (`array[i]`)** ou **notação de ponteiro (`*(array + i)`)**.
- **Matrizes precisam ter o número de colunas definido** na função.
- Se precisar de uma matriz de tamanho dinâmico, use **alocação dinâmica com ponteiros duplos (`int **matriz`)**.

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

---

6. Busca e Ordenação em Vetores

6.1 Busca Linear

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

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

6.2 Busca Binária

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

```
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;
}
```

6.3 Ordenação com Bubble Sort

```
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;
            }
        }
    }
}
```

---

7. 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**.

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.

Vetores Multidimensionais (Matrizes)

1. Matrizes na Matemática: Conceitos, Operações e Aplicações

1.1 Introdução às Matrizes

As **matrizes** são estruturas matemáticas fundamentais utilizadas para organizar e manipular dados numéricos em diversas áreas da matemática e ciências aplicadas. Elas são representadas como **tabelas retangulares de números**, organizadas em **linhas e colunas**.

Uma matriz com m linhas e n colunas é chamada de **matriz $m \times n$ (m por n)** e pode ser representada da seguinte forma:

$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \ a_{21} & a_{22} & \dots & a_{2n} \ \vdots & \vdots & \ddots & \vdots \ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$

Cada elemento da matriz, denotado por a_{ij}, representa o número na **linha i** e **coluna j**.

1.2. Tipos de Matrizes

As matrizes podem ser classificadas de acordo com suas propriedades estruturais. Algumas das principais são:

- **Matriz Linha**: Possui apenas uma linha, como $A = [a_{1} \quad a_{2} \quad \dots \quad a_{n}]$, sendo uma matriz $1 \times n$.
- **Matriz Coluna**: Possui apenas uma coluna, como
  $B = \begin{bmatrix} b_{1} \ b_{2} \ \vdots \ b_{m} \end{bmatrix}$
  sendo uma matriz $m \times 1$.
- **Matriz Quadrada**: O número de linhas é igual ao número de colunas ($m = n$). Por exemplo, uma matriz 2×2
- $I_3 = \begin{bmatrix} 1 & 4 \ 8 & 5 \end{bmatrix}$
-
- **Matriz Identidade** (I_n): Matriz quadrada onde os elementos da diagonal principal são 1 e os demais são 0.
  $I_3 = \begin{bmatrix} 1 & 0 & 0 \ 0 & 1 & 0 \ 0 & 0 & 1 \end{bmatrix}$

- **Matriz Diagonal**: Matriz quadrada onde todos os elementos fora da diagonal principal são nulos.
- **Matriz Nula** (0): Todos os elementos são iguais a zero.
- **Matriz Simétrica**: Uma matriz quadrada A é simétrica se $A^T = A$, ou seja, $a_{ij} = a_{ji}$.
- **Matriz Antissimétrica**: Uma matriz quadrada é antissimétrica se $A^T = -A$, ou seja, $a_{ij} = -a_{ji}$.

1.3. Operações com Matrizes

As operações entre matrizes seguem regras específicas e são amplamente utilizadas em álgebra linear.

1.4. Adição e Subtração de Matrizes

Dadas duas matrizes A e B de mesma dimensão $m \times n$, a soma $C = A + B$ e a subtração $D = A - B$ são obtidas somando ou subtraindo os elementos correspondentes:

$c_{ij} = a_{ij} + b_{ij}$

$d_{ij} = a_{ij} - b_{ij}$

Exemplo:
$A = \begin{bmatrix} 1 & 2 \ 3 & 4 \end{bmatrix}, \quad
B = \begin{bmatrix} 5 & 6 \ 7 & 8 \end{bmatrix}$

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

3.2. Multiplicação por Escalar

Multiplicar uma matriz por um escalar k significa multiplicar cada elemento da matriz por esse número:

$(kA)*{ij} = k \|cdot a*{ij}$

Exemplo:
$2 \times \begin{bmatrix} 1 & 2 \ 3 & 4 \end{bmatrix} =
\begin{bmatrix} 2 & 4 \ 6 & 8 \end{bmatrix}$

1.5. Multiplicação de Matrizes

A multiplicação de duas matrizes $A_{m \times n}$ e $B_{n \times p}$ resulta em uma matriz $C_{m \times p}$, onde cada elemento c_{ij} é obtido pelo produto escalar da linha i de A com a coluna j de B:

$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

Exemplo:
$A = \begin{bmatrix} 1 & 2 \ 3 & 4 \end{bmatrix}, \quad
B = \begin{bmatrix} 5 & 6 \ 7 & 8 \end{bmatrix}$

$A \times B =
\begin{bmatrix} (1 \times 5 + 2 \times 7) & (1 \times 6 + 2 \times 8) \ (3 \times 5 + 4 \times 7) & (3 \times 6 + 4 \times 8) \end{bmatrix}$

$C = \begin{bmatrix} 19 & 22 \ 43 & 50 \end{bmatrix}$

1.6. Matriz Transposta

A transposta de uma matriz A, denotada por A^T, é obtida trocando as linhas por colunas:

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

1.7. Determinante de uma Matriz

O **determinante** de uma matriz quadrada é um número associado à matriz e é essencial na álgebra linear. Para uma matriz 2×2:

$\det(A) = \begin{vmatrix} a & b \ c & d \end{vmatrix} = (a \cdot d - b \cdot c)$

Para matrizes $n \times n$, o determinante pode ser calculado por **expansão de Laplace** ou pelo **método de Gauss**.

---

2. Aplicações das Matrizes

As matrizes são amplamente aplicadas em diversas áreas, tais como:

- **Sistemas Lineares**: Resolver sistemas de equações lineares usando a matriz dos coeficientes.
- **Computação Gráfica**: Representação de transformações geométricas como rotação, escala e translação.
- **Inteligência Artificial**: Redes neurais usam operações matriciais para aprendizado profundo.
- **Engenharia e Física**: Simulações e modelagens em mecânica, eletromagnetismo e dinâmica de fluidos.
- **Economia e Estatística**: Análise de dados e previsão de tendências através de matrizes estocásticas.

As matrizes são ferramentas matemáticas fundamentais, com grande importância teórica e prática. Suas operações são essenciais para resolver problemas complexos em diversas áreas do conhecimento. Compreender as propriedades das matrizes permite explorar aplicações avançadas na ciência e na tecnologia.

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.

---

2. Aplicando matrizes em C

1. Declaração e Acesso a Vetores Bidimensionais

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

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

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

```c
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:

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

2. Percorrendo uma Matriz

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

```c
#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
```

3. Matrizes Dinâmicas

Em C, matrizes são tradicionalmente declaradas com tamanhos fixos, como `int matriz[3][3]`. No entanto, isso limita a flexibilidade do programa. Para criar **matrizes dinâmicas**, onde o tamanho é definido em tempo de execução, usamos **alocação dinâmica de memória** com `malloc` ou `calloc`.

3.1 Alocando uma matriz 2D com ponteiro para ponteiro (`int**`)

A maneira mais comum de criar uma matriz dinâmica em C é utilizando **ponteiros para ponteiros**. Isso permite alocar uma matriz cujo tamanho pode ser definido em tempo de execução.

Exemplo: Criando e liberando uma matriz dinâmica

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

int main() {
    int linhas = 3, colunas = 4;

    // Passo 1: Criar um ponteiro para um array de ponteiros
    int **matriz = (int **)malloc(linhas * sizeof(int *));

    // Passo 2: Para cada linha, alocar um array de inteiros
    for (int i = 0; i < linhas; i++) {
        matriz[i] = (int *)malloc(colunas * sizeof(int));
    }

    // Passo 3: Preenchendo a matriz
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            matriz[i][j] = i + j;  // Apenas um exemplo de preenchimento
        }
    }

    // Passo 4: Exibindo a matriz
    printf("Matriz:\n");
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }
```

```c
    // Passo 5: Liberando memória alocada
    for (int i = 0; i < linhas; i++) {
        free(matriz[i]);  // Libera cada linha
    }
    free(matriz);  // Libera o array de ponteiros

    return 0;
}
```

3.2 Alocando uma matriz em um único bloco de memória

Outra maneira eficiente de criar uma matriz dinâmica é alocando **um único bloco de memória** para todos os elementos. Isso reduz a fragmentação e melhora a localidade de cache.

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

int main() {
    int linhas = 3, colunas = 4;

    // Passo 1: Criando um ponteiro para um único bloco de memória
    int *matriz = (int *)malloc(linhas * colunas * sizeof(int));

    // Passo 2: Preenchendo a matriz
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            matriz[i * colunas + j] = i + j;
        }
    }

    // Passo 3: Exibindo a matriz
    printf("Matriz:\n");
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            printf("%d ", matriz[i * colunas + j]);
        }
        printf("\n");
    }

    // Passo 4: Liberando memória
    free(matriz);

    return 0;
}
```

Vantagens dessa abordagem:

- **Menos chamadas a `malloc`**: Um único bloco de memória é alocado.
- **Acesso mais rápido**: Melhor aproveitamento do cache, pois os elementos estão em sequência na memória.
- **Menos overhead de ponteiros**: Não há necessidade de armazenar vários ponteiros para as linhas.

A escolha entre **ponteiros para ponteiros** (`int **matriz`) e **um bloco único de memória** (`int *matriz`) depende do caso de uso:

- **`int **matriz`**: Mais intuitivo para manipular como matriz (`matriz[i][j]`), mas pode gerar fragmentação de memória.
- **`int *matriz`**: Melhor desempenho e uso eficiente de memória, mas requer cálculos manuais para acessar os elementos (`matriz[i * colunas + j]`).

Se estiver lidando com grandes matrizes ou performance é crítica, a segunda abordagem é geralmente mais eficiente.

4. Alocação Estática vs. Dinâmica

- **Alocação Estática**: O tamanho da matriz é fixo e determinado em tempo de compilação.

  ```
  int matriz[3][4]; // Sempre ocupa um espaço fixo na memória
  ```

  $ **Desvantagem**: Pode desperdiçar memória se for muito grande ou limitar o programa se for pequena.

- **Alocação Dinâmica**: A memória é alocada em tempo de execução, permitindo criar matrizes flexíveis.

  ```
  int **matriz = (int **)malloc(linhas * sizeof(int *));
  ```

  $ **Vantagem**: Eficiência no uso da memória e possibilidade de ajustar o tamanho dinamicamente.

4.1. Ponteiros e Matrizes

Em C, matrizes e ponteiros estão intimamente relacionados. Uma matriz bidimensional `int matriz[3][4]` pode ser vista como um **array de arrays**, onde cada linha é um array separado. Quando alocamos dinamicamente, criamos um **array de ponteiros para arrays**.

- **Ponteiro para ponteiro (`int **matriz`)**:
  - Cada linha é um array separado.
  - Cada posição da primeira dimensão aponta para um array de inteiros.
- **Bloco único de memória (`int *matriz`)**:

- Todos os elementos ficam alocados em um único espaço contíguo.

4.2. Funções para Gerenciamento de Memória

Na alocação dinâmica, usamos as seguintes funções da biblioteca `<stdlib.h>`:

Função	Descrição
`malloc(size)`	Aloca um bloco de memória sem inicialização.
`calloc(n, size)`	Aloca e inicializa a memória com zeros.
`free(ptr)`	Libera a memória alocada dinamicamente.

Exemplo:

```c
int *ptr = (int *)malloc(10 * sizeof(int)); // Aloca espaço para 10
inteiros
free(ptr); // Libera a memória alocada
```

```c
int *ptr = (int *)calloc(10 * sizeof(int)); // Aloca espaço para 10
inteiros preenchido com 0
free(ptr); // Libera a memória alocada
```

Matrizes dinâmicas são essenciais em algoritmos avançados e sistemas que precisam manipular grandes quantidades de dados. O uso adequado da **alocação dinâmica de memória** pode melhorar significativamente a eficiência de um programa.

Escolha entre ponteiros para ponteiros (`int **matriz`) ou bloco único (`int *matriz`) conforme a necessidade.
Gerencie a memória corretamente com `malloc` e `free` para evitar vazamentos de memória.

A escolha entre **ponteiros para ponteiros** (`int **matriz`) e **um bloco único de memória** (`int *matriz`) depende do caso de uso:

- **`int **matriz`**: Mais intuitivo para manipular como matriz (`matriz[i][j]`), mas pode gerar fragmentação de memória.
- **`int *matriz`**: Melhor desempenho e uso eficiente de memória, mas requer cálculos manuais para acessar os elementos (`matriz[i * colunas + j]`).

Se estiver lidando com grandes matrizes ou performance é crítica, a segunda abordagem é geralmente mais eficiente.

---

5. Operações com matrizes

5.1. 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;
}
```

5.2. 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];
            }
        }
    }
```

```c
}

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
```

5.3 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
```

5.4 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
```

5.5 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.

6. 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.

6.1 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>

#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},
```

```c
        {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
```

6.3 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
```

6.4 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

```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 * 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;

    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);
```

```c
    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.

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.

Exemplo Manipulação de Imagens

Para manipular imagens JPEG em C e aplicar filtros, precisamos usar uma biblioteca para ler e escrever arquivos no formato **JPEG**. A biblioteca **libjpeg** (do projeto **Independent JPEG Group - IJG**) é uma das mais populares para isso.

** O que este exemplo fará?**

1. **Carregar uma imagem JPEG** e armazená-la como uma **matriz de pixels**.
2. **Aplicar um filtro de escala de cinza** convertendo cada pixel.
3. **Salvar a imagem processada como um novo arquivo JPEG**.

Isso ajudará a entender como **imagens podem ser representadas como matrizes** e como **percorremos e manipulamos cada pixel**.

** Instalando a libjpeg (Linux/macOS)**

Antes de compilar o código, instale a biblioteca:

```
sudo apt-get install libjpeg-dev  # Debian/Ubuntu
brew install jpeg                 # macOS (Homebrew)
```

Se estiver no **Windows**, pode usar MinGW e baixar a **libjpeg-turbo**.

📜 Código em C: Carregando e Processando uma Imagem JPEG

Este código:

- **Lê uma imagem JPEG** e a armazena em uma **matriz bidimensional de pixels**.
- **Converte a imagem para tons de cinza** aplicando um filtro.
- **Salva a nova imagem como um arquivo JPEG**.

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

// Função para converter a imagem para tons de cinza
void aplicar_filtro_cinza(JSAMPLE *imagem, int largura, int altura) {
    for (int i = 0; i < largura * altura; i++) {
        // Pegando os valores RGB do pixel
        unsigned char r = imagem[i * 3];     // Vermelho
        unsigned char g = imagem[i * 3 + 1]; // Verde
        unsigned char b = imagem[i * 3 + 2]; // Azul

        // Calculando a média (fórmula para tons de cinza)
        unsigned char cinza = (r + g + b) / 3;

        // Aplicando o tom de cinza ao pixel
        imagem[i * 3] = imagem[i * 3 + 1] = imagem[i * 3 + 2] = cinza;
    }
}

// Função para ler uma imagem JPEG
JSAMPLE *ler_imagem_jpeg(const char *arquivo, int *largura, int *altura)
{
    struct jpeg_decompress_struct cinfo;
    struct jpeg_error_mgr jerr;

    FILE *arquivo_jpeg = fopen(arquivo, "rb");
    if (!arquivo_jpeg) {
        printf("Erro ao abrir arquivo %s\n", arquivo);
        return NULL;
    }

    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_decompress(&cinfo);
    jpeg_stdio_src(&cinfo, arquivo_jpeg);
    jpeg_read_header(&cinfo, TRUE);
    jpeg_start_decompress(&cinfo);

    *largura = cinfo.output_width;
    *altura = cinfo.output_height;
    int canais = cinfo.output_components; // Normalmente 3 para RGB

    // Criando matriz para armazenar pixels
    JSAMPLE *imagem = (JSAMPLE *)malloc((*largura) * (*altura) *
canais);
    JSAMPROW linha[1];

    // Lendo linha por linha da imagem
    while (cinfo.output_scanline < cinfo.output_height) {
        linha[0] = &imagem[cinfo.output_scanline * (*largura) * canais];
        jpeg_read_scanlines(&cinfo, linha, 1);
    }
```

```
    jpeg_finish_decompress(&cinfo);
    jpeg_destroy_decompress(&cinfo);
    fclose(arquivo_jpeg);

    return imagem;
}

// Função para salvar uma imagem JPEG
void salvar_imagem_jpeg(const char *arquivo, JSAMPLE *imagem, int
largura, int altura) {
    struct jpeg_compress_struct cinfo;
    struct jpeg_error_mgr jerr;

    FILE *arquivo_saida = fopen(arquivo, "wb");
    if (!arquivo_saida) {
        printf("Erro ao salvar arquivo %s\n", arquivo);
        return;
    }

    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_compress(&cinfo);
    jpeg_stdio_dest(&cinfo, arquivo_saida);

    cinfo.image_width = largura;
    cinfo.image_height = altura;
    cinfo.input_components = 3;  // RGB tem 3 componentes
    cinfo.in_color_space = JCS_RGB;

    jpeg_set_defaults(&cinfo);
    jpeg_set_quality(&cinfo, 100, TRUE);
    jpeg_start_compress(&cinfo, TRUE);

    JSAMPROW linha[1];
    while (cinfo.next_scanline < cinfo.image_height) {
        linha[0] = &imagem[cinfo.next_scanline * largura * 3];
        jpeg_write_scanlines(&cinfo, linha, 1);
    }

    jpeg_finish_compress(&cinfo);
    jpeg_destroy_compress(&cinfo);
    fclose(arquivo_saida);
}

int main() {
    const char *arquivo_entrada = "imagem.jpg";
    const char *arquivo_saida = "imagem_cinza.jpg";

    int largura, altura;

    // Lendo a imagem
    JSAMPLE *imagem = ler_imagem_jpeg(arquivo_entrada, &largura,
&altura);
```

```
    if (!imagem) return 1;

    // Aplicando filtro de cinza
    aplicar_filtro_cinza(imagem, largura, altura);

    // Salvando a imagem modificada
    salvar_imagem_jpeg(arquivo_saida, imagem, largura, altura);

    // Liberando memória
    free(imagem);

    printf("Imagem convertida para tons de cinza e salva como %s\n",
arquivo_saida);
    return 0;
}
```

Explicação Passo a Passo

Representação da Imagem como Matriz

A imagem é carregada em uma **matriz de pixels** no formato **RGB**, onde cada pixel é representado por **três valores (R, G, B)**.

Podemos imaginar a imagem como uma **tabela bidimensional**:

```
Índice:   | 0,0  | 0,1  | 0,2  | ... | 0, L-1 |
          | 1,0  | 1,1  | 1,2  | ... | 1, L-1 |
          | ...  | ...  | ...  | ... | ...    |
          | A-1,0| A-1,1| ...  | ... | A-1,L-1 |
```

Onde L é a **largura** e A a **altura** da imagem.

Cada posição da matriz contém **três números** que representam os valores **RGB** do pixel.

Percorrendo a Matriz e Aplicando o Filtro

Para converter a imagem para **tons de cinza**, percorremos cada pixel e aplicamos a média dos três valores RGB:

```
unsigned char cinza = (r + g + b) / 3;
```

Isso substitui os valores RGB pelo mesmo número, resultando em um tom de cinza.

Salvando a Nova Imagem

Após modificar os valores na matriz, salvamos a nova imagem usando **libjpeg**.

Como Compilar e Rodar

Salve o código como `filtro_imagem.c` e compile com `gcc` (incluindo a libjpeg):

```
gcc filtro_imagem.c -o filtro_imagem -ljpeg
```

Execute o programa com:

```
./filtro_imagem
```

Isso criará um novo arquivo chamado **`imagem_cinza.jpg`**, que será a versão convertida da imagem original.

- **Imagens podem ser representadas como matrizes bidimensionais**.
- **Cada pixel pode ser acessado e modificado percorrendo essa matriz**.
- **Usamos um filtro de cinza como exemplo de manipulação de imagem**.
- **Bibliotecas como libjpeg facilitam a leitura e escrita de arquivos de imagem**.

Para rodar o código de manipulação de imagem em **Windows**, você precisa instalar a biblioteca **libjpeg** e configurar o compilador corretamente. Aqui está um guia passo a passo para fazer isso.

Passo 1: Instalar o MinGW e a libjpeg

No Windows, o **GCC** pode ser usado através do **MinGW-w64**. Você também precisa da **libjpeg-turbo**, que é uma versão otimizada da **libjpeg**.

1️⃣ Instalar MinGW-w64

Se ainda não tem o **GCC** instalado no Windows, siga estes passos:

1. Baixe o instalador do **MinGW-w64**:
   👉 https://winlibs.com/
2. Escolha a versão **"UCRT"** (Universal C Runtime).
3. Instale e configure as variáveis de ambiente do Windows para incluir o caminho `C:\mingw-w64\bin`.

2️⃣ Instalar libjpeg-turbo

Agora, baixe e instale a biblioteca **libjpeg-turbo**:

1. Vá para 👉 https://libjpeg-turbo.org/
2. Baixe a versão **Windows 64-bit GNU** (`libjpeg-turbo-gcc.exe`).

3. Instale e anote o diretório onde a biblioteca foi instalada.

** Passo 2: Configurar o GCC para usar a libjpeg**

Agora você precisa configurar o compilador para encontrar a **libjpeg**.

1. Adicione o caminho dos arquivos **de inclusão** (`.h`) e **de biblioteca** (`.a`) ao seu compilador.
   Exemplo: Se a libjpeg-turbo foi instalada em `C:\libjpeg-turbo`, use:

   ```
   gcc filtro_imagem.c -o filtro_imagem.exe -I"C:\libjpeg-turbo\include" -L"C:\libjpeg-turbo\lib" -ljpeg
   ```

2. Se ocorrer erro ao executar (`.dll` não encontrada), copie os arquivos `.dll` da pasta `C:\libjpeg-turbo\bin` para o mesmo diretório onde está o `filtro_imagem.exe`.

💻 Compilando e Rodando

Agora que tudo está configurado, compile o código no terminal (CMD ou PowerShell):

```
gcc filtro_imagem.c -o filtro_imagem.exe -I"C:\libjpeg-turbo\include" -L"C:\libjpeg-turbo\lib" -ljpeg
```

E execute:

```
filtro_imagem.exe
```

Exemplo de **borramento (blur), realce de bordas (sharpen) e detecção de contornos (edge detection)** usando a biblioteca **libjpeg-turbo**.

O código lê uma imagem JPEG, aplica os filtros e salva novas imagens modificadas.

Requisitos

Antes de compilar, certifique-se de que você tem a **libjpeg-turbo** instalada. No Windows, o caminho padrão é `C:\libjpeg-turbo`, mas ajuste se necessário.

Código em C: Aplicação de Filtros em uma Imagem JPEG

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

// Definição de alguns filtros (Matrizes de convolução)
const int blurKernel[3][3] = {
    {1, 2, 1},
    {2, 4, 2},
    {1, 2, 1}
};

const int sharpenKernel[3][3] = {
    { 0, -1,  0},
    {-1,  5, -1},
    { 0, -1,  0}
};

const int edgeKernel[3][3] = {
    {-1, -1, -1},
    {-1,  8, -1},
    {-1, -1, -1}
};

// Função para carregar a imagem JPEG
unsigned char* load_jpeg(const char* filename, int* width, int* height,
int* channels) {
    FILE* file = fopen(filename, "rb");
    if (!file) {
        printf("Erro ao abrir a imagem!\n");
        return NULL;
    }

    struct jpeg_decompress_struct cinfo;
    struct jpeg_error_mgr jerr;
    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_decompress(&cinfo);
    jpeg_stdio_src(&cinfo, file);
    jpeg_read_header(&cinfo, TRUE);
    jpeg_start_decompress(&cinfo);

    *width = cinfo.output_width;
    *height = cinfo.output_height;
    *channels = cinfo.output_components;
    int row_stride = (*width) * (*channels);

    unsigned char* data = (unsigned char*)malloc((*width) * (*height) *
(*channels));
    unsigned char* buffer_array[1];

    while (cinfo.output_scanline < cinfo.output_height) {
```

```
        buffer_array[0] = data + (cinfo.output_scanline) * row_stride;
        jpeg_read_scanlines(&cinfo, buffer_array, 1);
    }

    jpeg_finish_decompress(&cinfo);
    jpeg_destroy_decompress(&cinfo);
    fclose(file);

    return data;
}

// Função para salvar uma imagem JPEG
void save_jpeg(const char* filename, unsigned char* data, int width, int
height, int channels, int quality) {
    FILE* file = fopen(filename, "wb");
    if (!file) {
        printf("Erro ao salvar a imagem!\n");
        return;
    }

    struct jpeg_compress_struct cinfo;
    struct jpeg_error_mgr jerr;
    cinfo.err = jpeg_std_error(&jerr);
    jpeg_create_compress(&cinfo);
    jpeg_stdio_dest(&cinfo, file);

    cinfo.image_width = width;
    cinfo.image_height = height;
    cinfo.input_components = channels;
    cinfo.in_color_space = JCS_RGB;
    jpeg_set_defaults(&cinfo);
    jpeg_set_quality(&cinfo, quality, TRUE);
    jpeg_start_compress(&cinfo, TRUE);

    int row_stride = width * channels;
    unsigned char* buffer_array[1];

    while (cinfo.next_scanline < cinfo.image_height) {
        buffer_array[0] = data + cinfo.next_scanline * row_stride;
        jpeg_write_scanlines(&cinfo, buffer_array, 1);
    }

    jpeg_finish_compress(&cinfo);
    jpeg_destroy_compress(&cinfo);
    fclose(file);
}

// Função para aplicar um filtro de convolução
void apply_filter(unsigned char* image, unsigned char* output, int
width, int height, int channels, const int kernel[3][3], int kernel_div)
{
    int offset = 1; // Para considerar os vizinhos na matriz 3x3
```

```
    for (int y = offset; y < height - offset; y++) {
        for (int x = offset; x < width - offset; x++) {
            int r = 0, g = 0, b = 0;

            for (int ky = -1; ky <= 1; ky++) {
                for (int kx = -1; kx <= 1; kx++) {
                    int pixelX = (x + kx);
                    int pixelY = (y + ky);
                    int index = (pixelY * width + pixelX) * channels;

                    r += image[index] * kernel[ky + 1][kx + 1];
                    g += image[index + 1] * kernel[ky + 1][kx + 1];
                    b += image[index + 2] * kernel[ky + 1][kx + 1];
                }
            }

            // Normaliza os valores
            int index = (y * width + x) * channels;
            output[index] = (unsigned char) fmin(fmax(r / kernel_div,
0), 255);
            output[index + 1] = (unsigned char) fmin(fmax(g /
kernel_div, 0), 255);
            output[index + 2] = (unsigned char) fmin(fmax(b /
kernel_div, 0), 255);
        }
    }
}

int main() {
    const char* input_file = "entrada.jpg";
    const char* blur_file = "blur.jpg";
    const char* sharpen_file = "sharpen.jpg";
    const char* edge_file = "edges.jpg";

    int width, height, channels;
    unsigned char* image = load_jpeg(input_file, &width, &height,
&channels);
    if (!image) return 1;

    unsigned char* blur_image = (unsigned char*)malloc(width * height *
channels);
    unsigned char* sharpen_image = (unsigned char*)malloc(width * height
* channels);
    unsigned char* edge_image = (unsigned char*)malloc(width * height *
channels);

    // Aplicando os filtros
    apply_filter(image, blur_image, width, height, channels, blurKernel,
16);
    apply_filter(image, sharpen_image, width, height, channels,
sharpenKernel, 1);
    apply_filter(image, edge_image, width, height, channels, edgeKernel,
1);
```

```
    // Salvando as imagens filtradas
    save_jpeg(blur_file, blur_image, width, height, channels, 90);
    save_jpeg(sharpen_file, sharpen_image, width, height, channels, 90);
    save_jpeg(edge_file, edge_image, width, height, channels, 90);

    // Liberar memória
    free(image);
    free(blur_image);
    free(sharpen_image);
    free(edge_image);

    printf("Imagens geradas com sucesso!\n");
    return 0;
}
```

Compilação no Windows (MinGW/GCC)

Se a **libjpeg-turbo** estiver instalada em `C:\libjpeg-turbo`, use este comando para compilar:

```
gcc manipulacao.c -o manipulacao.exe -I"C:\libjpeg-turbo\include" -
L"C:\libjpeg-turbo\lib" -ljpeg
```

Para rodar:

```
.\manipulacao.exe
```

Explicação

1. **Carrega** uma imagem JPEG (`entrada.jpg`).
2. **Aplica filtros de convolução** usando diferentes kernels:
    - **Borramento (blur)** → Suaviza a imagem.
    - **Realce de bordas (sharpen)** → Aumenta os detalhes.
    - **Detecção de contornos (edge detection)** → Destaca as bordas.
3. **Salva os resultados** em três novas imagens: `blur.jpg`, `sharpen.jpg` e `edges.jpg`.

** Dicas e Solução de Problemas**

Se tiver problemas:

1. Verifique se **libjpeg-turbo** está instalada corretamente.
2. Confirme que o **GCC** está no `PATH` (`gcc --version` deve funcionar no terminal).
3. Se faltar `.dll`, copie **`jpeg62.dll`** para a pasta onde está `filtro_imagem.exe`.

O erro **"undefined reference to `jpeg_xxx`"** significa que o **linker** não está encontrando a biblioteca `libjpeg`. Isso pode acontecer por dois motivos principais:

1. A biblioteca **não está instalada corretamente** ou não está no caminho esperado.
2. O **comando de compilação não está incluindo a biblioteca corretamente** na fase de linkedição.

Passo 1: Verificar se a libjpeg está instalada

Você instalou a **libjpeg-turbo**, certo? Agora, precisamos garantir que os arquivos da biblioteca estão no local correto.

1. Vá até a pasta da instalação, normalmente em:

   ```
   C:\libjpeg-turbo
   ```

2. Dentro dela, você deve encontrar:
   - `include/jpeglib.h` (cabeçalhos)
   - `lib/libjpeg.a` (biblioteca estática)
   - `bin/jpeg62.dll` (biblioteca dinâmica)

Se esses arquivos não estiverem presentes, pode ser necessário reinstalar a **libjpeg-turbo**.

Passo 2: Corrigir a Ordem do Linker no Comando de Compilação

No **GCC**, as bibliotecas **devem ser referenciadas por último** no comando de compilação. O erro ocorre porque o `-ljpeg` está no lugar errado.

Tente compilar assim:

```
gcc manipulacao.c -o manipulacao.exe -I"C:\libjpeg-turbo\include" -
L"C:\libjpeg-turbo\lib" -ljpeg
```

Se não funcionar, tente mover `-ljpeg` para o final:

```
gcc manipulacao.c -I"C:\libjpeg-turbo\include" -L"C:\libjpeg-turbo\lib"
-o manipulacao.exe -ljpeg
```

Se ainda der erro, tente a versão estática:

```
gcc manipulacao.c -I"C:\libjpeg-turbo\include" -L"C:\libjpeg-turbo\lib"
-o manipulacao.exe -static -ljpeg
```

Passo 3: Copiar a DLL para a Pasta do Executável

Caso você esteja tentando rodar o executável e receba um erro de **DLL ausente**, copie **jpeg62.dll** para o diretório onde está o `manipulacao.exe`:

```
copy C:\libjpeg-turbo\bin\jpeg62.dll .
```

Agora tente rodar novamente.

Passo 4: Adicionar a Biblioteca ao Path

Se mesmo após compilar corretamente ainda houver problemas ao executar o programa, tente adicionar o caminho da **libjpeg** ao `Path`:

1. Pressione **Win + R** e digite **sysdm.cpl**.
2. Vá até a aba **Avançado** > clique em **Variáveis de Ambiente**.
3. Encontre a variável **Path**, edite e adicione:

   ```
   C:\libjpeg-turbo\bin
   ```

Reinicie o terminal e tente compilar novamente.

Conclusão

Se você seguir esses passos, a compilação deve funcionar corretamente. Os pontos principais foram:

Verificar se a libjpeg está instalada corretamente
Corrigir a ordem dos argumentos do GCC
Certificar-se de que o linker está achando a libjpeg
Copiar a DLL para a pasta do executável
Adicionar a libjpeg ao Path do Windows

Exemplo em python

Para facilitar o código pode ser executado no Google Colab, mas com algumas adaptações. Como o Google Colab não possui interface gráfica para exibir imagens com `cv2.imshow()`, usaremos `cv2_imshow()` do módulo `google.colab.patches` para exibir os resultados.

Código Adaptado para Google Colab

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

# Fazer upload de uma imagem manualmente
from google.colab import files
uploaded = files.upload()

# Obter o nome do arquivo enviado
image_path = list(uploaded.keys())[0]

# Carregar a imagem
image = cv2.imread(image_path)

if image is None:
    print("Erro ao carregar a imagem!")
else:
    print("Imagem carregada com sucesso!")

# 1 **Filtro de Borramento (Gaussian Blur)**
blur_image = cv2.GaussianBlur(image, (5, 5), 0)

# 2 **Filtro de Suavização Média**
average_image = cv2.blur(image, (5, 5))

# 3 **Filtro de Realce de Bordas**
sharpen_kernel = np.array([[ 0, -1,  0],
                           [-1,  5, -1],
                           [ 0, -1,  0]], dtype=np.float32)
sharpen_image = cv2.filter2D(image, -1, sharpen_kernel)

# 4 **Filtro de Detecção de Bordas (Sobel)**
sobelx = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=5)  # Bordas
horizontais
sobely = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=5)  # Bordas verticais
sobel_image = cv2.magnitude(sobelx, sobely)

# 5 **Filtro de Detecção de Contornos (Canny)**
edges_image = cv2.Canny(image, 100, 200)

# 6 **Filtro de Emboss (Relevo)**
emboss_kernel = np.array([[-2, -1, 0],
                          [-1,  1, 1],
                          [ 0,  1, 2]], dtype=np.float32)
```

```
emboss_image = cv2.filter2D(image, -1, emboss_kernel)

# 7 **Filtro de Detecção de Borda Laplaciano**
laplacian_image = cv2.Laplacian(image, cv2.CV_64F)

# 📸 Exibir as imagens no Colab
cv2_imshow(image)           # Imagem original
cv2_imshow(blur_image)      # Filtro Gaussiano
cv2_imshow(average_image)   # Filtro de Média
cv2_imshow(sharpen_image)   # Filtro de Realce de Bordas
cv2_imshow(sobel_image)     # Filtro Sobel
cv2_imshow(edges_image)     # Filtro Canny
cv2_imshow(emboss_image)    # Filtro Emboss (Relevo)
cv2_imshow(laplacian_image) # Filtro Laplaciano
```

Como Executar no Google Colab

1. **Abra o Google Colab** (colab.research.google.com).
2. **Copie e cole o código acima** em uma célula.
3. **Execute a célula**: Ao rodar o código, será solicitado o **upload de uma imagem**.
4. **Selecione uma imagem** `.jpg` **ou** `.png` do seu computador.
5. **Os filtros serão aplicados** e os resultados serão exibidos diretamente no Colab.

Explicação do código acima

**** O que foi adaptado?****

1. **Adicionado** `files.upload()` para que o usuário possa carregar uma imagem manualmente.
2. **Substituído** `cv2.imshow()` **por** `cv2_imshow()` para exibição correta das imagens no Google Colab.

Agora, você pode testar filtros de imagem facilmente no Colab!

Explicação do Código de Processamento de Imagens em Python no Google Colab 🖼️ 📊

Este código demonstra como **carregar e processar imagens aplicando diferentes filtros** utilizando a biblioteca OpenCV (`cv2`) no Google Colab. Vamos detalhar cada parte do código para entender seu funcionamento.

---

1 Carregando a Imagem

```
from google.colab import files
uploaded = files.upload()
image_path = list(uploaded.keys())[0]
image = cv2.imread(image_path)
```

- `files.upload()` permite ao usuário carregar uma imagem manualmente no Colab.
- `cv2.imread(image_path)` lê a imagem carregada.
- Se a imagem não for encontrada, uma mensagem de erro será exibida.

---

2 Aplicação de Filtros

Agora, aplicamos **diferentes filtros de processamento de imagens** para transformar a imagem original.

**** Filtro 1: Borramento Gaussiano (Gaussian Blur)****

```
blur_image = cv2.GaussianBlur(image, (5, 5), 0)
```

- **Reduz o ruído** e **suaviza** a imagem.
- O `(5,5)` define o tamanho da matriz usada para calcular o desfoque.

**** Filtro 2: Suavização Média (Média Móvel)****

```
average_image = cv2.blur(image, (5, 5))
```

- Cada pixel é substituído pela **média** dos pixels vizinhos.
- Mais simples que o Gaussian Blur, mas pode causar **perda de detalhes**.

**** Filtro 3: Realce de Bordas (Sharpening)****

```
sharpen_kernel = np.array([[ 0, -1,  0],
                           [-1,  5, -1],
                           [ 0, -1,  0]], dtype=np.float32)
sharpen_image = cv2.filter2D(image, -1, sharpen_kernel)
```

- Usa **uma matriz (kernel)** que enfatiza diferenças de cor para **destacar os detalhes da imagem**.
- Muito utilizado para **realçar imagens desfocadas**.

**** Filtro 4: Detecção de Bordas (Sobel)****

```
sobelx = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=5)  # Bordas
horizontais
sobely = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=5)  # Bordas verticais
sobel_image = cv2.magnitude(sobelx, sobely)
```

- O operador **Sobel** detecta bordas na direção horizontal (`sobelx`) e vertical (`sobely`).
- A magnitude das bordas é calculada para destacar as diferenças.

**** Filtro 5: Detecção de Contornos (Canny)****

```
edges_image = cv2.Canny(image, 100, 200)
```

- **Identifica bordas nítidas** com base em gradientes de intensidade.
- `100` e `200` são os **limiares para detectar bordas** (ajustáveis para mais precisão).

**** Filtro 6: Relevo (Emboss)****

```
emboss_kernel = np.array([[-2, -1, 0],
                          [-1,  1, 1],
                          [ 0,  1, 2]], dtype=np.float32)
emboss_image = cv2.filter2D(image, -1, emboss_kernel)
```

- Realça **texturas e formas**, criando um **efeito de relevo**.
- Amplamente usado em **edição de fotos e arte digital**.

**** Filtro 7: Detecção de Bordas Laplaciano****

```
laplacian_image = cv2.Laplacian(image, cv2.CV_64F)
```

- O operador **Laplaciano** destaca regiões onde há mudanças bruscas de cor.
- Bom para **detecção de objetos** dentro da imagem.

---

3 Exibição das Imagens

```
cv2_imshow(image)           # Imagem original
cv2_imshow(blur_image)      # Filtro Gaussiano
cv2_imshow(average_image)   # Filtro de Média
cv2_imshow(sharpen_image)   # Filtro de Realce de Bordas
cv2_imshow(sobel_image)     # Filtro Sobel
cv2_imshow(edges_image)     # Filtro Canny
cv2_imshow(emboss_image)    # Filtro Emboss (Relevo)
cv2_imshow(laplacian_image) # Filtro Laplaciano
```

- O **Google Colab não suporta** `cv2.imshow()`, então usamos `cv2_imshow()` para exibir imagens.

🎯 Resumo

Filtro	Função
Borramento Gaussiano	Reduz ruído e suaviza
Suavização Média	Média dos pixels vizinhos
Realce de Bordas	Aumenta a nitidez da imagem
Sobel	Detecta bordas horizontais e verticais
Canny	Identifica bordas precisas
Emboss	Cria efeito de relevo
Laplaciano	Destaca transições bruscas

Com esse código, conseguimos **manipular imagens usando matrizes** e entender como **filtros podem modificar características visuais**.