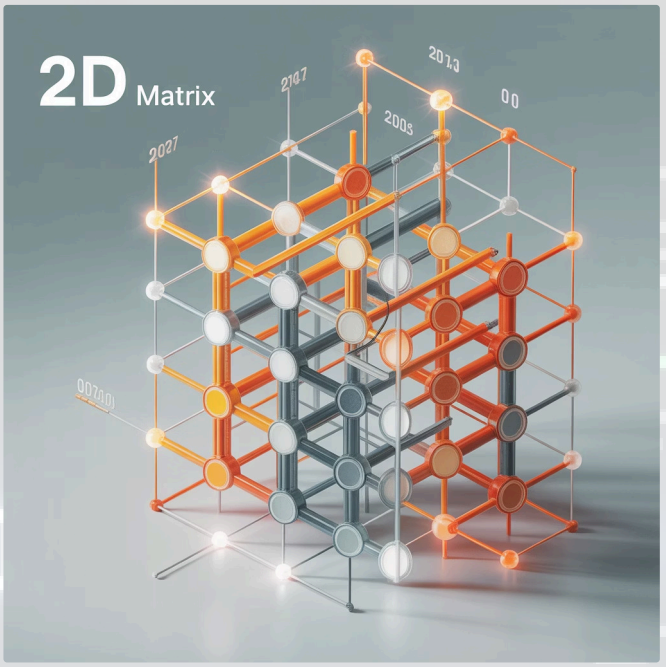
Capítulo 2 — Matrizes em C#

2.1 O que é uma matriz?

Uma **matriz** (ou *array bidimensional*) é uma coleção de elementos organizados em linhas e colunas, como uma tabela ou planilha. Se o vetor é uma linha de dados, a matriz é um quadro com várias linhas e colunas. Podemos pensar em uma matriz como um "vetor de vetores" ou uma tabela com células onde cada célula pode ser acessada através de dois índices: um para a linha e outro para a coluna.

As matrizes são fundamentais na programação porque muitos problemas do mundo real são naturalmente representados em duas dimensões. Por exemplo, um tabuleiro de xadrez, uma planilha de dados, uma imagem digital ou até mesmo a disposição de assentos em um teatro.

Em C#, as matrizes são implementadas como arrays multidimensionais e seguem a mesma lógica dos vetores, mas com a adição de uma dimensão extra. Assim como vetores, as matrizes em C# têm tamanho fixo após sua criação e todos os elementos devem ser do mesmo tipo.



Do ponto de vista da memória, uma matriz bidimensional é armazenada como uma sequência linear de elementos, mas o C# gerencia a tradução entre os índices bidimensionais (linha, coluna) e a posição real na memória, tornando o processo transparente para o programador.

A principal vantagem das matrizes é permitir a organização e acesso a dados que possuem relação lógica em duas dimensões, facilitando operações como busca, ordenação e processamento de informações estruturadas em formato tabular.

Bidimensionalidade

Organização em linhas e colunas, permitindo acesso a elementos através de dois índices.

Homogeneidade

Todos os elementos da matriz devem ser do mesmo tipo de dados.

Tamanho fixo

Assim como vetores, as matrizes em C# têm dimensões fixas após sua criação.

Armazenamento contíguo

Os elementos são armazenados sequencialmente na memória, com acesso rápido através de cálculos de deslocamento.

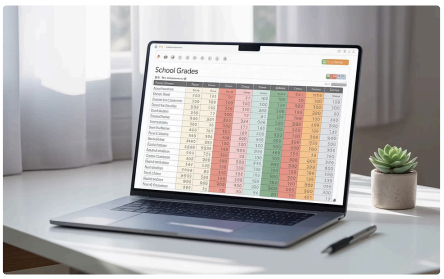
Visualizando uma matriz:

[0,0]	[0,1]	[0,2]	[0,3]
[1,0]	[1,1]	[1,2]	[1,3]
[2,0]	[2,1]	[2,2]	[2,3]

Cada célula é identificada por um par de índices [linha, coluna], começando do [0,0] no canto superior esquerdo. Esta notação permite acessar qualquer elemento da matriz de forma direta e eficiente.

Utilidade das matrizes

As matrizes são ferramentas extremamente versáteis na programação, oferecendo soluções elegantes para uma variedade de problemas do mundo real. Vamos explorar as principais aplicações e vantagens do uso de matrizes em seus programas.

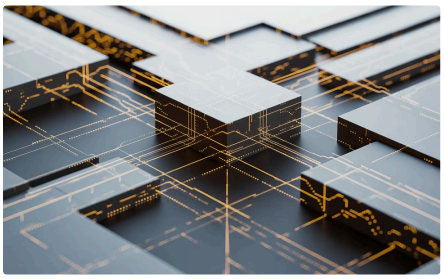


Tabelas de Dados

Ideal para representar informações tabulares como notas escolares (alunos nas linhas, disciplinas nas colunas), planilhas financeiras ou catálogos de produtos.

Jogos e Tabuleiros

Perfeitas para implementar jogos como xadrez, batalha naval, jogo da velha ou qualquer jogo baseado em grade, onde cada posição representa um estado.



Processamento de Imagens

Imagens digitais são essencialmente matrizes de pixels, onde cada elemento representa a cor ou intensidade de um ponto na imagem.

Vantagens e aplicações adicionais:

Cálculos matemáticos e estatísticos

Matrizes são fundamentais para álgebra linear, estatística e cálculos científicos. Operações como multiplicação de matrizes, determinantes e sistemas de equações lineares são facilmente implementáveis usando matrizes.

Implementação de algoritmos

Muitos algoritmos avançados, como os de busca em grafos (A*, Dijkstra), programação dinâmica e reconhecimento de padrões, utilizam matrizes como estrutura de dados fundamental.

Mapas e representações espaciais

Jogos, simulações e sistemas GIS (Geographic Information System) frequentemente usam matrizes para representar terrenos, mapas de calor, distribuições populacionais ou outros dados geoespaciais.

Análise de dados

Em ciência de dados, matrizes são usadas para armazenar conjuntos de dados multidimensionais, tabelas de contingência e matrizes de correlação entre variáveis.

Otimização de performance

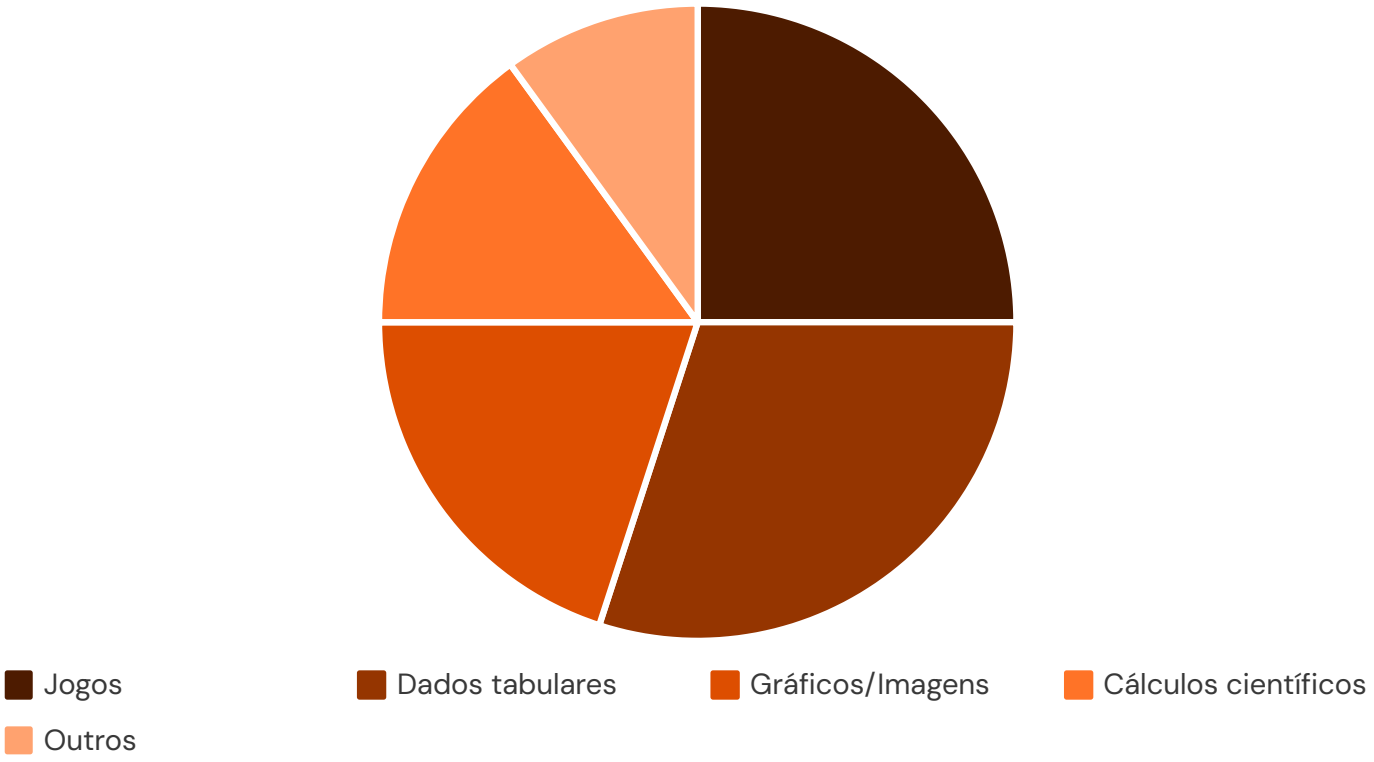
Para dados que possuem relação bidimensional natural, matrizes oferecem acesso mais rápido e lógico do que outras estruturas alternativas, como listas aninhadas.

Simplicidade e clareza

Em muitos casos, o uso de matrizes torna o código mais legível e intuitivo quando se trabalha com dados naturalmente organizados em linhas e colunas, melhorando a manutenibilidade do código.

> Em aplicações de inteligência artificial e aprendizado de máquina, matrizes são extensivamente utilizadas para representar dados de treinamento, pesos de redes neurais e transformações matemáticas, formando a base para bibliotecas como TensorFlow e PyTorch.

Dominar o uso de matrizes é essencial para programadores que trabalham com visualização de dados, sistemas de informação geográfica, desenvolvimento de jogos, modelagem matemática e muitas outras áreas que exigem manipulação eficiente de dados bidimensionais. Mesmo em aplicações mais simples, a capacidade de pensar em termos de matrizes pode levar a soluções mais elegantes e eficientes para problemas complexos.



Como declarar uma matriz em C#

A declaração de matrizes em C# segue uma sintaxe específica que indica tanto o tipo de dados quanto as dimensões da estrutura. Vamos explorar as diferentes formas de declarar matrizes e suas particularidades.

Sintaxe básica para declaração de matrizes:

```
tipo[,] nomeDaMatriz = new tipo[linhas, colunas];
```

1	2
tipo	nomeDaMatriz
O tipo de dado que a matriz irá armazenar (int, double, string, etc.). Todos os elementos da matriz serão deste tipo.	O identificador da variável que representa a matriz, seguindo as regras de nomenclatura do C#.
Exemplos: `int`, `string`, `double`, `bool`, ou tipos personalizados como classes próprias.	Boas práticas sugerem usar nomes que indiquem o propósito ou conteúdo da matriz, como `tabuleiro`, `notas` ou `mapa`.

3	4
linhas O número de linhas da matriz, determinando a primeira dimensão. Pode ser um valor literal ou uma variável/expressão que resulte em um número inteiro positivo.	colunas O número de colunas da matriz, determinando a segunda dimensão. Também pode ser um valor literal ou uma variável/expressão que resulte em um número inteiro positivo.

Exemplos de declarações de matrizes:

```
// Matriz de 3 linhas por 4 colunas para armazenar números inteiros
int[,] tabela = new int[3, 4];

// Matriz para um tabuleiro de xadrez (8x8)
string[,] tabuleiro = new string[8, 8];

// Matriz para armazenar temperaturas diárias por semana (7 dias x 3
// turnos)
double[,] temperaturas = new double[7, 3];

// Matriz cujas dimensões são determinadas por variáveis
int linhas = 10;
int colunas = 15;
bool[,] mapa = new bool[linhas, colunas];
```

Assim como vetores, quando você declara uma matriz em C#, todos os seus elementos são inicializados automaticamente com o valor padrão do tipo especificado (0 para tipos numéricos, false para booleanos, null para tipos de referência).

Matrizes com mais de duas dimensões

Embora matrizes bidimensionais sejam as mais comuns, o C# também suporta matrizes com três ou mais dimensões:

```
// Matriz tridimensional (cubo de dados)
int[,,] cubo = new int[4, 5, 3];

// Matriz tetradimensional
double[,,,] hiperCubo = new double[2, 3, 2, 4];
```

⊗ Matrizes com mais de duas dimensões rapidamente se tornam difíceis de visualizar e manipular. Use-as com moderação e apenas quando a natureza do problema realmente exigir esse tipo de estrutura.

Arrays irregulares (jagged arrays)

O C# também suporta arrays irregulares, que são arrays de arrays, onde cada linha pode ter um número diferente de colunas:

```
// Declaração de um array irregular
int[][] irregular = new int[3][];

// Inicializando cada linha com tamanhos diferentes
irregular[0] = new int[5];
irregular[1] = new int[3];
irregular[2] = new int[7];
```

Observe que arrays irregulares usam uma sintaxe diferente ([][] em vez de [,]) e são conceitualmente diferentes de matrizes regulares, embora possam ser usados para resolver problemas semelhantes em alguns casos.

Inicializando e acessando elementos

Inicializando elementos individualmente

Após declarar uma matriz, você pode atribuir valores a cada posição individualmente usando a notação [linha, coluna]:

```
int[,] tabela = new int[3, 4];

// Atribuindo valores a elementos específicos
tabela[0, 0] = 10;  // Primeira linha, primeira coluna
tabela[0, 1] = 20;  // Primeira linha, segunda coluna
tabela[2, 3] = 15;  // Terceira linha, quarta coluna

// Podemos também ler valores de forma similar
int valor = tabela[1, 2];  // Obtém o valor da segunda linha, terceira coluna
```

Inicializando com valores predefinidos

Você também pode inicializar uma matriz com valores já definidos no momento da declaração:

```
// Inicialização direta de matriz 3x3
int[,] matriz = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 }
};

// Inicialização de matriz 2x4
string[,] frutas = {
    { "Maçã", "Banana", "Laranja", "Uva" },
    { "Morango", "Abacaxi", "Pêra", "Melancia" }
};
```

Acessando elementos da matriz

Para acessar um elemento específico da matriz, você usa a notação [linha, coluna], onde ambos os índices começam em 0:

```
// Acessando elementos da matriz
Console.WriteLine(matriz[0, 0]);  // Imprime 1 (primeira linha, primeira coluna)
Console.WriteLine(matriz[1, 1]);  // Imprime 5 (segunda linha, segunda coluna)
Console.WriteLine(matriz[2, 2]);  // Imprime 9 (terceira linha, terceira coluna)

// Também podemos modificar valores
matriz[0, 2] = 30;  // Altera o valor da primeira linha, terceira coluna
```

Cuidados ao acessar elementos

Assim como com vetores, é importante evitar o acesso a índices fora dos limites da matriz para não causar uma exceção `IndexOutOfRangeException`. Sempre verifique se os índices estão dentro dos limites válidos, especialmente ao trabalhar com valores calculados dinamicamente.

> ⚠ Ao contrário de algumas outras linguagens, C# não usa a notação matriz[linha][coluna] para acessar elementos. A notação correta é sempre matriz[linha, coluna] com uma vírgula entre os índices.

Exemplo prático de inicialização e acesso

Percorrendo matrizes

Para processar todos os elementos de uma matriz, geralmente precisamos percorrê-la elemento por elemento. Em C#, isso é feito tipicamente usando dois laços for aninhados: um para percorrer as linhas e outro para percorrer as colunas em cada linha.

Método básico para percorrer uma matriz

```
for (int i = 0; i < matriz.GetLength(0); i++) // linhas
{
    for (int j = 0; j < matriz.GetLength(1); j++) // colunas
    {
        Console.Write(matriz[i, j] + " ");
    }
    Console.WriteLine(); // Nova linha após cada linha da matriz
}
```

> O método `GetLength(0)` retorna o número de linhas da matriz, enquanto `GetLength(1)` retorna o número de colunas. Usar esses métodos em vez de valores fixos torna seu código mais flexível e menos propenso a erros.

Exemplos de percorrimento

Percorrendo para exibir valores

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

// Exibindo a matriz em formato tabular
for (int i = 0; i < matriz.GetLength(0); i++)
{
    for (int j = 0; j < matriz.GetLength(1); j++)
    {
        Console.Write($"{matriz[i, j],3}"); // O 3 formata com largura fixa
    }
    Console.WriteLine();
}
```

Percorrendo para calcular somas

```
double[,] temperaturas = {
    { 22.5, 25.8, 27.3 },
    { 23.1, 26.5, 28.0 },
    { 21.8, 24.7, 26.2 },
    { 22.0, 25.5, 27.5 }
};

// Calculando a média de cada dia (linha)
for (int i = 0; i < temperaturas.GetLength(0); i++)
{
    double soma = 0;
    for (int j = 0; j < temperaturas.GetLength(1); j++)
    {
        soma += temperaturas[i, j];
    }
    double media = soma / temperaturas.GetLength(1);
    Console.WriteLine($"Média do dia {i+1}: {media:F1}°C");
}
```

Percorrendo para preenchimento

```
// Criando uma matriz 5x5 e preenchendo com um padrão
int[,] padrao = new int[5, 5];

for (int i = 0; i < padrao.GetLength(0); i++)
{
    for (int j = 0; j < padrao.GetLength(1); j++)
    {
        if (i == j)
            padrao[i, j] = 1; // Diagonal principal
        else if (i < j)
            padrao[i, j] = 2; // Acima da diagonal
        else
            padrao[i, j] = 3; // Abaixo da diagonal
    }
}
```

Percorrendo de maneira diferente

Além do percorrimento linha por linha, às vezes precisamos percorrer a matriz de maneiras alternativas:

```
// Percorrendo por colunas (coluna por coluna)
for (int j = 0; j < matriz.GetLength(1); j++)
{
    for (int i = 0; i < matriz.GetLength(0); i++)
    {
        Console.Write(matriz[i, j] + " ");
    }
    Console.WriteLine();
}

// Percorrendo apenas a diagonal principal
for (int i = 0; i < Math.Min(matriz.GetLength(0), matriz.GetLength(1)); i++)
{
    Console.Write(matriz[i, i] + " ");
}
```

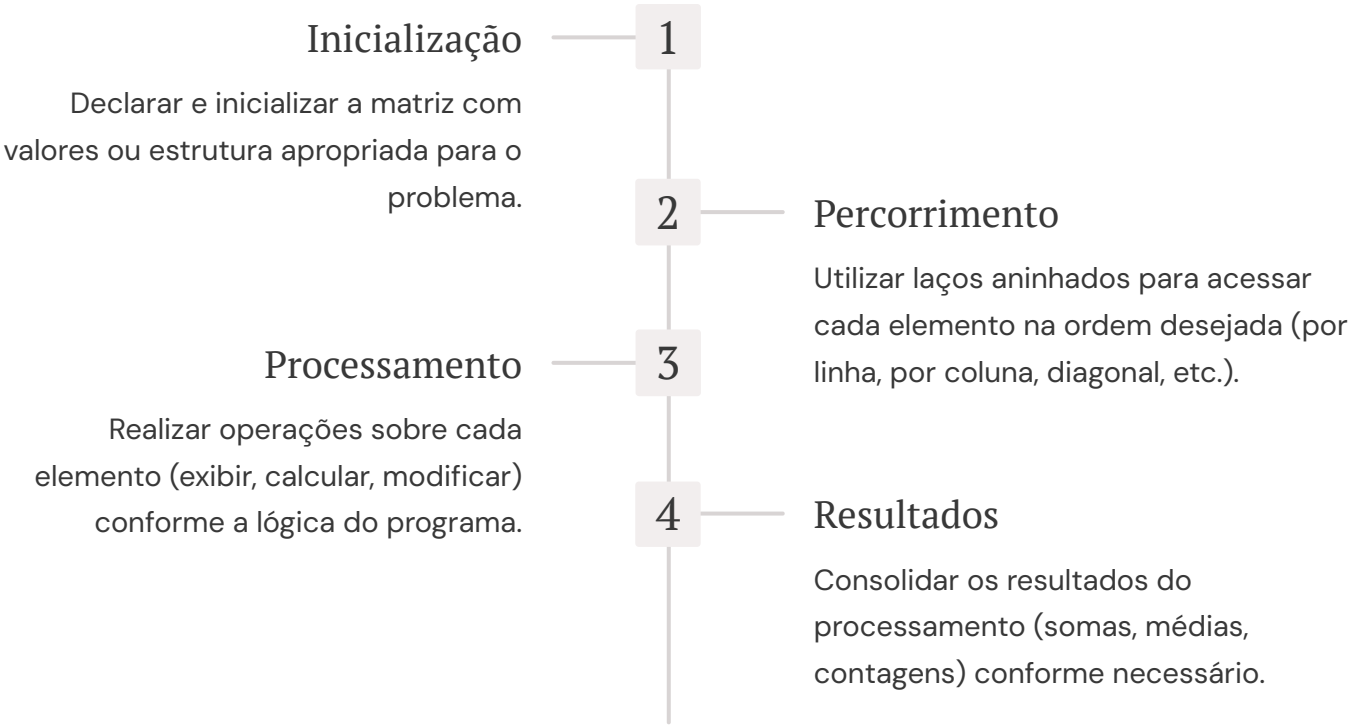
Percorrendo com foreach

Embora menos comum por não fornecer acesso aos índices, também é possível percorrer todos os elementos de uma matriz usando o laço foreach:

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

foreach (int elemento in matriz)
{
    Console.Write(elemento + " ");
}
// Saída: 1 2 3 4 5 6
```

Observe que o foreach percorre a matriz em ordem de linha principal (todas as colunas da primeira linha, depois todas as colunas da segunda linha, e assim por diante), mas não fornece informações sobre a posição (linha e coluna) de cada elemento.



Exemplo prático completo

Vamos criar um programa que lê notas de 3 alunos em 2 provas, calcula a média de cada aluno e determina sua situação (aprovado ou reprovado). Este exemplo ilustra a aplicação prática de matrizes em um cenário educacional comum.

```
using System;

class Program
{
   static void Main()
   {
      // Declaração da matriz para armazenar as notas (3 alunos x 2
      // provas)
      double[,] notas = new double[3, 2];
      string[] nomes = new string[3]; // Vetor para armazenar os nomes
                                      // dos alunos

      Console.WriteLine("=== Sistema de Notas Escolares ===\n");

      // Leitura dos nomes e notas
      for (int i = 0; i < 3; i++)
      {
         Console.Write($"Digite o nome do aluno {i + 1}: ");
         nomes[i] = Console.ReadLine();

         Console.WriteLine($"Notas do(a) {nomes[i]}:");
```

```
    for (int j = 0; j < 2; j++)
    {
        bool notaValida = false;

        // Validação para garantir notas entre 0 e 10
        while (!notaValida)
        {
            Console.Write($"  Nota da Prova {j + 1}: ");
            if (double.TryParse(Console.ReadLine(), out notas[i,
                j]))
            {
                if (notas[i, j] >= 0 && notas[i, j] <= 10)
                {
                    notaValida = true;
                }
                else
                {
                    Console.WriteLine("Nota inválida! Digite um
                        valor entre 0 e 10.");
                }
            }
            else
            {
                Console.WriteLine("  Entrada inválida! Digite um
                    número válido.");
            }
        }
    }

    Console.WriteLine(); // Linha em branco entre alunos
}

// Calculando e mostrando médias e situação
Console.WriteLine("\n=== Resultados ===");
Console.WriteLine("Aluno\t\tProva 1\tProva 2\tMédia\tSituação");
Console.WriteLine("-----------------------------------------------
    --");

int aprovados = 0;
double mediaTurma = 0;

for (int i = 0; i < 3; i++)
{
    // Calculando média do aluno
    double soma = 0;
```

```
            for (int j = 0; j < 2; j++)
            {
                soma += notas[i, j];
            }
            double media = soma / 2;
            mediaTurma += media;

            // Determinando situação (aprovado se média >= 7)
            string situacao = media >= 7 ? "Aprovado" : "Reprovado";
            if (media >= 7) aprovados++;

            // Exibindo resultados
            Console.WriteLine($"{nomes[i],-12}\t{notas[i, 0]:F1}\t{notas[i,
                1]:F1}\t{media:F1}\t{situacao}");
        }

        // Estatísticas gerais
        mediaTurma /= 3;
        Console.WriteLine("\n=== Estatísticas da Turma ===");
        Console.WriteLine($"Média geral da turma: {mediaTurma:F1}");
        Console.WriteLine($"Total de aprovados: {aprovados} de 3
            ({aprovados / 3.0 * 100:F1}%)");
    }
}
```

Explicação do código:

Declaração e inicialização

Criamos uma matriz 3x2 para armazenar as notas (3 alunos, 2 provas) e um vetor para os nomes dos alunos.

Entrada de dados

Utilizamos laços aninhados para ler o nome de cada aluno e suas notas nas duas provas, com validação para garantir valores entre 0 e 10.

Processamento

Calculamos a média de cada aluno somando suas notas e dividindo pelo número de provas, determinando também sua situação (aprovado ou reprovado).

Exibição dos resultados

Mostramos uma tabela com todos os dados e calculamos estatísticas gerais da turma, como média geral e porcentagem de aprovação.

Este exemplo demonstra como matrizes podem ser usadas para organizar e processar dados estruturados de forma eficiente. A matriz permite armazenar logicamente as relações entre alunos e suas respectivas notas, facilitando o acesso e processamento dessas informações.

Além disso, o programa mostra várias técnicas importantes:

- Acesso a elementos da matriz usando índices de linha e coluna
- Percorrimento de matrizes com laços aninhados
- Cálculos baseados em linhas (médias dos alunos)
- Combinação de matrizes com outros tipos de dados (como vetores de strings)
- Formatação de saída para apresentação clara dos resultados

Este tipo de programa pode ser expandido para incluir mais funcionalidades, como ordenação dos alunos por média, identificação do melhor e pior desempenho, ou mesmo para lidar com um número variável de alunos e provas definido pelo usuário.

Boas práticas com matrizes

Trabalhar com matrizes de forma eficiente e segura requer atenção a algumas práticas importantes. Seguir estas recomendações ajudará você a evitar erros comuns e a escrever código mais robusto e legível.

Use GetLength() para dimensões

Sempre utilize os métodos GetLength(0) e GetLength(1) para descobrir o tamanho real das dimensões da matriz, em vez de usar valores fixos no código.

```
// Correto
for (int i = 0; i < matriz.GetLength(0); i++)
{
    for (int j = 0; j < matriz.GetLength(1); j++)
    {
        // ...
    }
}

// Evite
for (int i = 0; i < 5; i++) // E se a matriz mudar de tamanho?
{
    for (int j = 0; j < 4; j++)
    {
        // ...
    }
}
```

Lembre-se: índices começam em 0

Assim como vetores, os índices das matrizes em C# começam em 0. O primeiro elemento está na posição [0,0], não [1,1].

Para uma matriz de 3x4, os índices válidos são de [0,0] até [2,3], não de [1,1] até [3,4].

Cuidado com acesso fora dos limites

Sempre verifique se os índices estão dentro dos limites válidos, especialmente quando são calculados dinamicamente ou vêm de entrada do usuário.

```
// Exemplo de verificação de limites
if (linha >= 0 && linha < matriz.GetLength(0) &&
    coluna >= 0 && coluna < matriz.GetLength(1))
{
    // Acesso seguro
    valor = matriz[linha, coluna];
}
else
{
    Console.WriteLine("Índices fora dos limites!");
}
```

Práticas adicionais recomendadas:

Inicialização adequada

Inicialize a matriz com valores padrão significativos ou preencha-a completamente antes de acessar os valores. Elementos não inicializados terão o valor padrão do tipo (0, false, null), o que pode causar comportamentos inesperados se não for considerado.

Encapsulamento em métodos

Crie métodos específicos para operações comuns em matrizes, como preenchimento, exibição, busca ou cálculos específicos. Isso melhora a legibilidade e a reutilização do código.

```
// Exemplo de método para exibir matriz
static void ExibirMatriz(int[,] matriz)
{
    for (int i = 0; i < matriz.GetLength(0); i++)
    {
        for (int j = 0; j < matriz.GetLength(1); j++)
        {
            Console.Write($"{matriz[i, j],4}");
        }
        Console.WriteLine();
    }
}
```

Documentação clara

Documente o significado das dimensões da matriz (o que representam as linhas e colunas) e quaisquer convenções especiais de indexação ou interpretação dos valores.

Considere arrays irregulares quando apropriado

Se cada linha da matriz tiver um número diferente de elementos, considere usar arrays irregulares (jagged arrays) em vez de matrizes regulares com espaço desperdiçado.

```
// Array irregular (cada linha pode ter comprimento diferente)
int[][] irregular = new int[3][];
irregular[0] = new int[5];
irregular[1] = new int[2];
irregular[2] = new int[7];
```

Tratamento de exceções

Implemente tratamento de exceções para operações críticas que possam gerar IndexOutOfRangeException, especialmente em código que lida com entrada do usuário ou cálculos complexos de índices.

Evite matrizes muito grandes na pilha

Matrizes muito grandes devem ser alocadas dinamicamente ou passadas como parâmetros para evitar estouro de pilha, especialmente em métodos recursivos.

> ⓘ Para operações avançadas como ordenação, busca ou transformações complexas em matrizes, considere utilizar bibliotecas especializadas ou implementar algoritmos específicos para o domínio do problema.

Seguindo estas práticas, você evitará erros comuns e criará código mais robusto, legível e eficiente ao trabalhar com matrizes em C#. Estas recomendações são especialmente importantes em aplicações mais complexas ou em situações onde o desempenho e a segurança são críticos.

Exercícios para fixação

Para consolidar seu conhecimento sobre matrizes, vamos explorar três exercícios práticos de diferentes níveis de dificuldade. Cada exercício inclui descrições detalhadas, dicas e soluções comentadas para auxiliar seu aprendizado.

1

Temperatura semanal

Crie uma matriz para armazenar temperaturas de uma semana (7 dias x 3 turnos) e calcule a temperatura média diária.

Dicas: Use uma matriz de 7x3 para armazenar as temperaturas. Percorra cada linha para calcular a média de cada dia.

2

Tabela de multiplicação

Monte um programa para gerar uma tabela de multiplicação (10x10) e mostrar na tela.

Dicas: Use dois laços aninhados, onde cada elemento [i,j] recebe o valor i * j.

3

Soma da diagonal principal

Some os valores da diagonal principal de uma matriz quadrada (ex: 3x3).

Dicas: A diagonal principal possui os elementos onde i = j. Use um único laço para percorrer esses elementos.

Soluções comentadas:

Exercício 1: Temperatura semanal

```
using System;

class Program
{
   static void Main()
   {
      // Declaração da matriz de temperaturas (7 dias x 3 turnos)
      double[,] temperaturas = new double[7, 3];
      string[] diasSemana = { "Domingo", "Segunda", "Terça", "Quarta",
                              "Quinta", "Sexta", "Sábado" };
      string[] turnos = { "Manhã", "Tarde", "Noite" };
```

```
// Entrada das temperaturas (nesta versão, usamos valores
// aleatórios)
Random rnd = new Random();
Console.WriteLine("Preenchendo a matriz com temperaturas
  aleatórias...");

for (int i = 0; i < 7; i++)
{
   for (int j = 0; j < 3; j++)
   {
      // Gera temperaturas entre 15°C e 35°C
      temperaturas[i, j] = 15 + rnd.NextDouble() * 20;
   }
}

// Exibindo as temperaturas registradas
Console.WriteLine("\nTemperaturas registradas:");
Console.WriteLine("Dia\t\tManhã\tTarde\tNoite\tMédia");
Console.WriteLine("-----------------------------------------------
  ");

// Cálculo das médias diárias
double[] mediasDiarias = new double[7];
double mediaGeral = 0;

for (int i = 0; i < 7; i++)
{
   double somaDia = 0;

   // Exibindo temperaturas do dia
   Console.Write($"{diasSemana[i],-10}\t");

   for (int j = 0; j < 3; j++)
   {
      Console.Write($"{temperaturas[i, j]:F1}°C\t");
      somaDia += temperaturas[i, j];
   }

   // Calculando e exibindo a média do dia
   mediasDiarias[i] = somaDia / 3;
   mediaGeral += mediasDiarias[i];
   Console.WriteLine($"{mediasDiarias[i]:F1}°C");
}
```

```
        // Calculando e exibindo a média geral da semana
        mediaGeral /= 7;
        Console.WriteLine("\nMédia semanal: " + mediaGeral.ToString("F1") +
          "°C");

        // Identificando o dia mais quente e o mais frio
        int diaMaisQuente = 0, diaMaisFrio = 0;
        for (int i = 1; i < 7; i++)
        {
            if (mediasDiarias[i] > mediasDiarias[diaMaisQuente])
                diaMaisQuente = i;
            if (mediasDiarias[i] < mediasDiarias[diaMaisFrio])
                diaMaisFrio = i;
        }

        Console.WriteLine($"Dia mais quente: {diasSemana[diaMaisQuente]} "
          + $"({mediasDiarias[diaMaisQuente]:F1}°C)");
        Console.WriteLine($"Dia mais frio: {diasSemana[diaMaisFrio]} " +
                    $"({mediasDiarias[diaMaisFrio]:F1}°C)");
    }
}
```

Exercício 2: Tabela de multiplicação

```
using System;

class Program
{
    static void Main()
    {
        // Declaração da matriz para a tabela de multiplicação
        int[,] tabelaMultiplicacao = new int[10, 10];

        // Preenchimento da tabela
        for (int i = 0; i < 10; i++)
        {
            for (int j = 0; j < 10; j++)
            {
                // Os índices começam em 0, mas queremos multiplicar 1x1,
                // 1x2, etc.
                tabelaMultiplicacao[i, j] = (i + 1) * (j + 1);
            }
        }
```

```
        // Exibição da tabela
        Console.WriteLine("Tabela de Multiplicação (10x10):");
        Console.WriteLine();

        // Exibindo cabeçalho das colunas
        Console.Write("    | ");
        for (int j = 1; j <= 10; j++)
        {
            Console.Write($"{j,3} ");
        }
        Console.WriteLine();
        Console.WriteLine("----+----------------------------------");

        // Exibindo linhas da tabela
        for (int i = 0; i < 10; i++)
        {
            Console.Write($"{i + 1,3} | ");

            for (int j = 0; j < 10; j++)
            {
                Console.Write($"{tabelaMultiplicacao[i, j],3} ");
            }

            Console.WriteLine();
        }
    }
}
```

Exercício 3: Soma da diagonal principal

```
using System;

class Program
{
    static void Main()
    {
        // Tamanho da matriz quadrada
        int tamanho = 3;

        // Declaração da matriz
        int[,] matriz = new int[tamanho, tamanho];

        // Preenchimento da matriz (neste exemplo, com valores aleatórios)
        Random rnd = new Random();
        Console.WriteLine($"Matriz {tamanho}x{tamanho}:");
```

```
        for (int i = 0; i < tamanho; i++)
        {
            for (int j = 0; j < tamanho; j++)
            {
                matriz[i, j] = rnd.Next(1, 10); // Valores entre 1 e 9
                Console.Write($"{matriz[i, j],3}");
            }
            Console.WriteLine();
        }

        // Calculando a soma da diagonal principal
        int somaDiagonal = 0;

        for (int i = 0; i < tamanho; i++)
        {
            // Na diagonal principal, os índices de linha e coluna são
            // iguais
            somaDiagonal += matriz[i, i];

            // Também poderíamos fazer:
            // somaDiagonal += matriz[i, i];
        }

        // Exibindo o resultado
        Console.WriteLine($"\nA soma da diagonal principal é:
          {somaDiagonal}");

        // Opcional: destacando a diagonal principal
        Console.WriteLine("\nDiagonal principal destacada:");

        for (int i = 0; i < tamanho; i++)
        {
            for (int j = 0; j < tamanho; j++)
            {
                if (i == j)
                    Console.Write($"[{matriz[i, j]}] ");
                else
                    Console.Write($"{matriz[i, j],3} ");
            }
            Console.WriteLine();
        }
    }
}
```

Estes exercícios ajudam a reforçar conceitos importantes sobre matrizes: declaração, acesso a elementos, percorrimento, processamento e visualização dos dados em formato tabular. Cada um explora diferentes aspectos do trabalho com matrizes e demonstra técnicas úteis para manipulação de dados bidimensionais.

> Desafio extra: Modifique o Exercício 3 para calcular também a soma da diagonal secundária (do canto superior direito ao canto inferior esquerdo) e compare as duas somas. Na diagonal secundária, a soma dos índices é sempre igual a n-1, onde n é o tamanho da matriz.

Ao praticar estes exercícios, você estará construindo uma base sólida para trabalhar com matrizes em aplicações mais complexas, como jogos, análise de dados, sistemas de informação e muito mais.

Capítulo 3 — Vetor x Matriz: Quando usar cada um?

A escolha entre vetores e matrizes deve ser baseada na estrutura natural dos dados que você está manipulando. Cada estrutura tem seu propósito e situações onde é mais adequada. Vamos analisar quando usar cada uma delas.

Use Vetores (Arrays)

Quando seus dados estiverem em sequência linear (apenas uma dimensão). Vetores são ideais para representar:

- Listas simples de elementos (nomes, números, objetos)
- Séries temporais (valores ao longo do tempo)
- Coleções homogêneas onde cada item tem o mesmo significado
- Sequências ordenadas onde a posição tem significado específico

Exemplos práticos: lista de alunos, notas de um único aluno, sequência de comandos, histórico de preços, conjunto de opções.

Use Matrizes (Arrays 2D)

Quando precisar relacionar informações em duas dimensões (linha x coluna). Matrizes são perfeitas para representar:

- Tabelas de dados (linhas e colunas)
- Grades, mapas e tabuleiros
- Relações entre dois conjuntos de entidades
- Dados que naturalmente formam uma estrutura retangular

Exemplos práticos: notas de vários alunos em várias disciplinas, tabuleiro de xadrez, mapa de um jogo, planilha de dados.

Comparativo detalhado:

Aspecto	Vetores	Matrizes
Dimensões	Uma dimensão (linear)	Duas ou mais dimensões (retangular/cúbica)
Acesso a elementos	Um índice: `vetor[i]`	Dois ou mais índices: `matriz[i,j]`
Complexidade	Mais simples de percorrer e manipular	Requer laços aninhados para percorrer
Uso de memória	Mais eficiente para dados lineares	Pode ter espaço não utilizado em dados esparsos
Aplicações típicas	Listas, sequências, históricos	Tabelas, grids, mapas bidimensionais

Considerações para a escolha:

Estrutura natural dos dados

O fator mais importante é como os dados se organizam naturalmente. Se eles formam uma lista linear, use vetor. Se formam uma tabela ou grid, use matriz.

Relações entre elementos

Se cada elemento se relaciona apenas com sua posição na sequência, vetores são suficientes. Se existem relações em duas dimensões (como em um gráfico de coordenadas x,y), matrizes são mais adequadas.

Necessidade de acesso

Se você precisa acessar elementos por uma única coordenada, vetores são mais simples. Se precisa acessar por pares de coordenadas (linha/coluna), matrizes são necessárias.

Eficiência de processamento

Vetores geralmente são mais eficientes para percorrer e processar, pois exigem apenas um laço. Matrizes requerem laços aninhados, o que pode ser mais custoso em grandes volumes de dados.

Clareza do código

Escolha a estrutura que torna seu código mais legível e intuitivo. Às vezes, é possível representar dados bidimensionais com vetores (usando cálculos de índice), mas isso pode tornar o código mais difícil de entender.

Necessidades futuras

Considere como os dados podem evoluir. Se uma estrutura unidimensional pode se tornar bidimensional no futuro, pode ser melhor começar com uma matriz.

Casos especiais:

Vetores de objetos complexos

Às vezes, um vetor de objetos complexos pode substituir uma matriz. Por exemplo, em vez de uma matriz de notas[aluno, disciplina], você pode ter um vetor de objetos Aluno, cada um com um vetor de notas por disciplina.

Arrays irregulares (jagged arrays)

Quando cada linha tem um número diferente de elementos, arrays irregulares (vetor de vetores) podem ser mais eficientes que matrizes com espaços não utilizados.

Coleções alternativas

Para casos mais complexos, considere estruturas como List<T>, Dictionary<K,V>, ou HashSet<T> em vez de vetores ou matrizes primitivas.

> Lembre-se que em C#, tanto vetores quanto matrizes têm tamanho fixo após a criação. Se precisar de estruturas dinâmicas que possam crescer ou diminuir, considere usar List<T> (para substituir vetores) ou List<List<T>> (para substituir matrizes).

A escolha correta entre vetores e matrizes é fundamental para um código eficiente e legível. Ao compreender as características e aplicações de cada estrutura, você estará mais preparado para modelar seus dados de forma apropriada, facilitando o desenvolvimento e manutenção de seus programas.