

Vetores e Matrizes em C#

Dominando o Armazenamento Estruturado de Dados

Este guia completo apresenta os conceitos fundamentais de vetores e matrizes na linguagem C#, desde declarações básicas até implementações práticas. Destinado a estudantes e desenvolvedores iniciantes, o material oferece explicações detalhadas, exemplos de código, exercícios práticos e recomendações de boas práticas para o armazenamento eficiente de dados estruturados.

Introdução

Ao programar, frequentemente precisamos armazenar e manipular grandes quantidades de dados, como listas de nomes, tabelas de notas ou mapas de jogos. Para isso, existem as **estruturas de dados**, ferramentas fundamentais para organizar informações de maneira eficiente e flexível. Neste material, você vai aprender a trabalhar com as estruturas mais usadas e essenciais no início de qualquer jornada em programação: **vetores** e **matrizes**.

Essas estruturas são como os blocos de construção de aplicações mais complexas, permitindo que você organize dados de forma lógica e acessível. Um vetor, por exemplo, é como uma fila de caixas numeradas onde você pode guardar informações do mesmo tipo, enquanto uma matriz é como uma grade ou tabela com linhas e colunas.

A importância dessas estruturas vai além da simples organização de dados. Elas permitem que você implemente algoritmos eficientes de busca, ordenação e processamento, melhorando significativamente o desempenho e a clareza do seu código.

O domínio desses conceitos abrirá caminho para a resolução de problemas mais complexos e será a base para seu avanço em Programação Orientada a Objetos. A partir destes fundamentos, você estará preparado para explorar estruturas de dados mais avançadas como listas encadeadas, pilhas, filas e árvores.

Ao longo deste material, vamos explorar desde os conceitos básicos até exemplos práticos e exercícios que ajudarão a consolidar seu conhecimento. Prepare-se para dar um importante passo na sua jornada como programador!

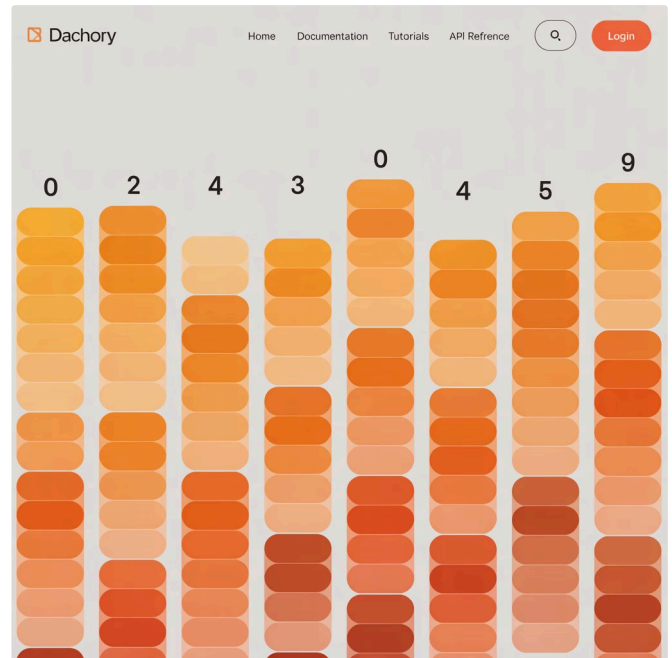
Capítulo 1 — Vetores em C#

1.1 O que é um vetor?

Um **vetor** (também chamado de *array* ou *arranjo*) é uma estrutura de dados que armazena uma sequência de elementos do mesmo tipo, organizados em uma linha, formando uma coleção unidimensional. Pense em uma fila de cadeiras, uma lista de compras ou uma sequência de notas. O vetor guarda vários valores, cada um acessível por sua posição (índice).

Vetores são blocos de memória contíguos, o que significa que todos os elementos são armazenados em sequência na memória do computador. Isso torna o acesso aos elementos extremamente rápido, pois o computador consegue calcular a posição exata de qualquer elemento com uma simples operação matemática.

Além disso, os vetores em C# são objetos de referência, o que significa que quando você declara um vetor, está criando uma referência a esse bloco de memória, não o bloco em si. Isso tem implicações importantes quando você passa vetores como parâmetros para métodos ou atribui um vetor a outro.



Em C#, os vetores têm tamanho fixo após sua criação. Isso significa que, uma vez criado um vetor com determinado tamanho, não é possível aumentá-lo ou diminuí-lo. Para contornar essa limitação, muitas vezes utilizamos coleções dinâmicas como `List<T>`, que veremos em outros estudos.

Um aspecto fundamental dos vetores é que todos os elementos devem ser do mesmo tipo. Isso permite que o compilador otimize o armazenamento e o acesso aos dados, mas também impõe uma restrição à flexibilidade da estrutura.

1

Organização sequencial

Os elementos são armazenados em sequência na memória, tornando o acesso rápido e eficiente.

2

Tamanho fixo

Uma vez criado, o tamanho do vetor não pode ser alterado durante a execução do programa.

3

Tipo homogêneo

Todos os elementos do vetor devem ser do mesmo tipo de dados (int, string, double, etc.).

4

Indexação baseada em zero

O primeiro elemento está na posição 0, o segundo na posição 1, e assim por diante.

Por que usar vetores?

Os vetores são ferramentas essenciais na programação moderna, oferecendo diversas vantagens que os tornam indispensáveis para desenvolvedores em todos os níveis. Vamos explorar os principais benefícios do uso de vetores em seus programas:

Armazenamento eficiente

Permitem guardar muitos valores de forma organizada, sem precisar criar uma variável para cada dado. Imagine ter que criar 1000 variáveis individuais para armazenar as temperaturas diárias de quase três anos – seria impraticável! Com um vetor, você pode fazer isso com uma única declaração.

Manipulação simplificada

Facilitam a aplicação de algoritmos, como busca, ordenação e cálculos em lote. Os vetores permitem que você processe grandes conjuntos de dados com loops simples, aplicando a mesma operação a cada elemento sem repetir código.


Código otimizado

Tornam o código mais limpo, reutilizável e eficiente. A capacidade de acessar elementos por índice e percorrer toda a estrutura com laços de repetição reduz a duplicação de código e melhora a legibilidade.

Além dessas vantagens principais, os vetores também oferecem:

- Acesso direto a qualquer elemento através do seu índice em tempo constante ($O(1)$)
- Suporte integrado a diversos métodos úteis como `Sort()`, `Reverse()`, `BinarySearch()`
- Facilidade para passar grandes conjuntos de dados como parâmetros de métodos
- Compatibilidade com diversos algoritmos e bibliotecas que esperam dados estruturados
- Base para estruturas de dados mais complexas como filas, pilhas e listas

Ao dominar o uso de vetores, você estará desenvolvendo uma habilidade fundamental que será utilizada em praticamente todos os projetos de software que você criar. Eles são especialmente úteis quando você precisa trabalhar com coleções de dados que têm tamanho conhecido e não precisam crescer ou diminuir durante a execução do programa.

 Para casos onde você precisa de uma coleção que possa mudar de tamanho, o C# oferece alternativas como `List<T>`, que é implementada internamente usando vetores, mas oferece métodos para adicionar e remover elementos dinamicamente.

Como declarar um vetor em C#

A declaração de vetores em C# segue uma sintaxe específica que combina o tipo de dados, colchetes para indicar que é um vetor, um nome para a variável e a alocação de memória para o número de elementos desejado.

```
tipo[] nomeDoVetor = new tipo[tamanho];
```

1

tipo

O tipo de dado que o vetor irá armazenar (int, double, string, etc.). Todos os elementos do vetor serão deste mesmo tipo.

Exemplos: int, string, double, bool, char, ou mesmo tipos complexos como DateTime ou classes personalizadas.

2

nomeDoVetor

O identificador da variável que seguirá as regras de nomenclatura do C#.

Boas práticas sugerem usar nomes no plural que indiquem o conteúdo do vetor, como notas, nomes, idades, etc.

3

tamanho

O número de elementos que o vetor irá armazenar. Este valor determina quanto espaço será alocado na memória.

Pode ser um número literal (ex: 5) ou uma variável/expressão que resulte em um número inteiro positivo.

Exemplos de declarações de vetores:

```
// Um vetor de 5 números inteiros  
int[] numeros = new int[5];
```

```
// Um vetor de 10 nomes (strings)  
string[] nomes = new string[10];
```

```
// Um vetor de 7 valores booleanos  
bool[] disponibilidade = new bool[7];
```

```
// Um vetor de 3 números decimais  
double[] precos = new double[3];
```

```
// Um vetor cujo tamanho é determinado por uma variável
int quantidade = 20;
float[] medicoes = new float[quantidade];
```

Quando você declara um vetor em C#, todos os seus elementos são inicializados automaticamente com o valor padrão do tipo especificado. Para tipos numéricos, o valor padrão é 0; para tipos booleanos, é false; para tipos de referência, como strings e objetos, é null.

❏ Os vetores em C# são objetos de primeira classe, o que significa que eles herdam da classe `System.Array` e têm acesso a diversos métodos e propriedades úteis, como `Length`, `GetValue`, `SetValue`, `Clone`, entre outros.

Inicializando e atribuindo valores

Existem várias maneiras de inicializar um vetor em C# e atribuir valores aos seus elementos. Vamos explorar as abordagens mais comuns e quando cada uma é mais adequada.

Atribuição individual de elementos

Após declarar um vetor, você pode atribuir valores a cada posição individualmente, utilizando o índice dentro de colchetes:

```
int[] numeros = new int[5];
numeros[0] = 10; // Primeiro elemento
numeros[1] = 20; // Segundo elemento
numeros[2] = 30; // Terceiro elemento
numeros[3] = 40; // Quarto elemento
numeros[4] = 50; // Quinto elemento
```

Inicialização na declaração

Uma maneira mais concisa é já declarar e inicializar o vetor com seus valores entre chaves. Neste caso, não é necessário especificar o tamanho do vetor, pois ele será determinado pela quantidade de elementos fornecidos:

```
int[] notas = { 7, 8, 6, 9, 10 };
string[] diasDaSemana = { "Domingo", "Segunda", "Terça", "Quarta", "Quinta", "Sexta", "Sábado" };
double[] alturas = { 1.75, 1.80, 1.65, 1.90 };
```

Inicialização com `new` e lista de valores

Você também pode combinar a palavra-chave `new` com a lista de valores entre chaves:

```
int[] numeros = new int[] { 10, 20, 30, 40, 50 };
```

Inicialização com valores padrão

Se você não atribuir valores específicos, o C# inicializa o vetor com os valores padrão para o tipo de dado:

0

Tipos numéricos

int, double, float,
decimal, etc.

false

Tipo booleano

bool

null

Tipos de
referência

string, arrays, objetos,
etc.

\0

Caracteres

char (caractere nulo)

Exemplo prático completo

```
using System;

class Program
{
    static void Main()
    {
        // Método 1: Declaração e depois atribuição
        double[] temperaturas = new double[4];
        temperaturas[0] = 23.5;
        temperaturas[1] = 27.8;
        temperaturas[2] = 25.1;
        temperaturas[3] = 22.7;

        // Método 2: Inicialização direta
        string[] frutas = { "Maçã", "Banana", "Laranja", "Uva", "Pêra" };

        // Método 3: Com new e lista de valores
        char[] vogais = new char[] { 'a', 'e', 'i', 'o', 'u' };

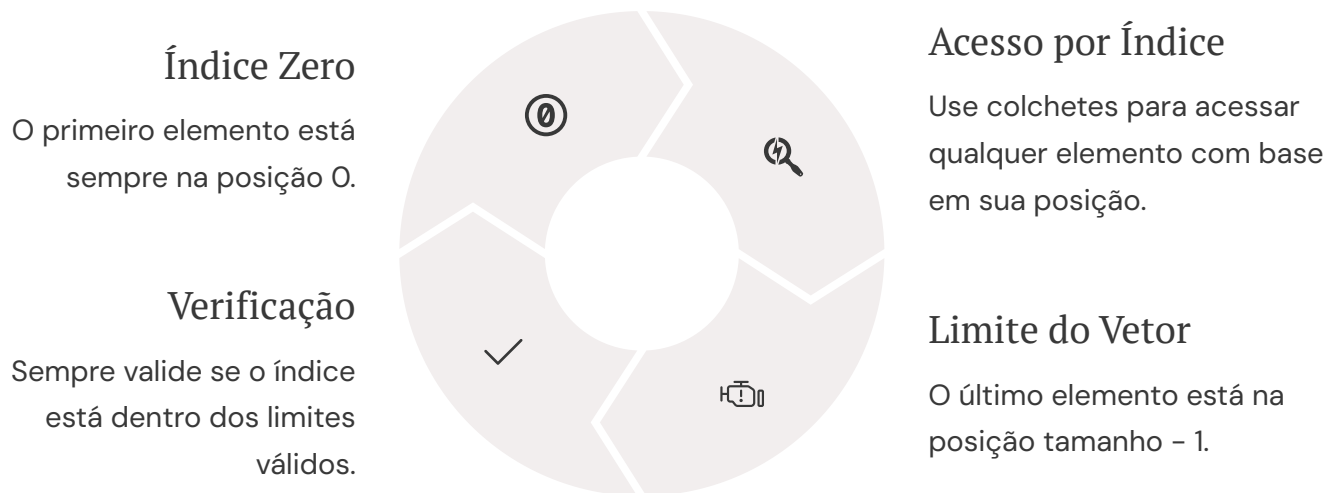
        // Imprimindo os valores
        Console.WriteLine("Temperaturas:");
```

```
for(int i = 0; i < temperaturas.Length; i++)
{
    Console.WriteLine($"Temperatura {i+1}: {temperaturas[i]}°C");
}
}
```

⚠ Cuidado com os índices ao atribuir valores! Tentar acessar um índice fora dos limites do vetor (negativo ou maior/igual ao tamanho) resultará em uma exceção `IndexOutOfRangeException`.

Acessando elementos

Acessar elementos em um vetor é uma operação fundamental que você utilizará constantemente em seus programas. Vamos entender como funciona o sistema de indexação e as práticas recomendadas para manipular elementos de um vetor.



Exemplo básico de acesso:

```
int[] numeros = { 10, 20, 30, 40, 50 };

// Acessando elementos específicos
Console.WriteLine(numeros[0]); // Mostra 10 (primeiro elemento)
Console.WriteLine(numeros[2]); // Mostra 30 (terceiro elemento)
Console.WriteLine(numeros[4]); // Mostra 50 (quinto elemento)

// O índice do último elemento é sempre o tamanho - 1
Console.WriteLine(numeros[numeros.Length - 1]); // Mostra 50 (último elemento)
```

Leitura e modificação de elementos:

Você pode tanto ler quanto modificar elementos utilizando a notação de colchetes:

```
string[] frutas = { "Maçã", "Banana", "Laranja", "Uva" };

// Lendo um valor
string frutaFavorita = frutas[2]; // "Laranja"

// Modificando um valor
frutas[1] = "Morango"; // Agora o array é { "Maçã", "Morango", "Laranja",
                        // "Uva" }

// Também podemos usar o valor atual para calcular um novo valor
int[] valores = { 1, 2, 3, 4 };
valores[0] = valores[0] * 10; // Multiplica o primeiro elemento por 10
                        // (agora é 10)
valores[3] += 5;           // Adiciona 5 ao último elemento (agora é 9)
```

⊗ Cuidado com o erro mais comum ao trabalhar com vetores: tentar acessar um elemento fora dos limites válidos. Se seu vetor tem tamanho 5, os índices válidos são 0, 1, 2, 3 e 4. Tentar acessar `numeros[5]` resultará em uma exceção `IndexOutOfRangeException`.

Boas práticas para acessar elementos:

- Sempre verifique se o índice está dentro dos limites antes de acessá-lo em situações dinâmicas
- Use a propriedade `Length` para determinar o tamanho do vetor em vez de valores fixos
- Considere usar estruturas condicionais para evitar acessos inválidos
- Em operações críticas, implemente tratamento de exceções com `try/catch`

Percorrendo um vetor

Percorrer um vetor é uma operação comum e essencial para processar todos os seus elementos. O C# oferece várias formas de iterar sobre um vetor, cada uma com suas vantagens em diferentes situações.

Utilizando o laço for tradicional

O método mais comum e versátil para percorrer um vetor é usando o laço for, que nos dá controle total sobre o processo de iteração:

```
int[] numeros = { 10, 20, 30, 40, 50 };

for (int i = 0; i < numeros.Length; i++)
{
    Console.WriteLine($"Posição {i}: {numeros[i]}");
}
```

Esta abordagem é especialmente útil quando você precisa do índice durante o processamento ou quando precisa percorrer apenas parte do vetor.

Utilizando o laço foreach

Quando você não precisa do índice e quer apenas processar cada elemento, o laço foreach oferece uma sintaxe mais limpa e menos propensa a erros:

```
string[] cores = { "Vermelho", "Verde", "Azul", "Amarelo" };

foreach (string cor in cores)
{
    Console.WriteLine($"Cor: {cor}");
}
```



O laço foreach é somente leitura. Você não pode modificar os elementos do vetor diretamente dentro dele. Se precisar modificar os elementos, use o laço for tradicional.

Utilizando métodos LINQ

Para operações mais avançadas, o C# oferece a biblioteca LINQ (Language Integrated Query), que fornece métodos poderosos para manipular coleções:

```
double[] valores = { 3.5, 2.7, 9.8, 1.5, 6.2 };

// Filtrando valores maiores que 5
var valoresAltos = valores.Where(v => v > 5);

// Ordenando o vetor
var valoresOrdenados = valores.OrderBy(v => v);
```

```
// Calculando média
double media = valores.Average();

Console.WriteLine($"Média dos valores: {media}");
```

Exemplo prático completo

Vamos ver um exemplo que combina diferentes formas de percorrer um vetor para resolver um problema prático:

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        int[] notas = { 7, 5, 8, 9, 6, 8, 10, 7 };

        // Calculando estatísticas usando for
        int soma = 0;
        int maior = int.MinValue;
        int menor = int.MaxValue;

        for (int i = 0; i < notas.Length; i++)
        {
            soma += notas[i];

            if (notas[i] > maior)
                maior = notas[i];

            if (notas[i] < menor)
                menor = notas[i];
        }

        double media = (double)soma / notas.Length;

        // Usando foreach para contar aprovações
        int aprovados = 0;
        foreach (int nota in notas)
        {
            if (nota >= 7)
                aprovados++;
        }
    }
}
```

```
// Usando LINQ para operações mais complexas
var notasOrdenadas = notas.OrderByDescending(n => n);
var segundaMaiorNota = notasOrdenadas.Skip(1).First();

// Exibindo resultados
Console.WriteLine($"Média da turma: {media:F1}");
Console.WriteLine($"Maior nota: {maior}");
Console.WriteLine($"Menor nota: {menor}");
Console.WriteLine($"Alunos aprovados: {aprovados} de
    {notas.Length}");
Console.WriteLine($"Segunda maior nota: {segundaMaiorNota}");
}
}
```

Laço for

Ideal quando você precisa do índice ou quer controlar o início e fim da iteração. Mais verboso, mas mais flexível.

Laço foreach

Perfeito para percorrer todos os elementos quando não precisa do índice. Sintaxe mais limpa e segura.

LINQ

Excelente para operações complexas como filtragem, ordenação, agrupamento e transformação em poucas linhas.

Exemplo prático completo

Vamos criar um programa que lê as notas de 5 alunos, calcula a média da turma e fornece algumas estatísticas adicionais. Este exemplo demonstra várias operações com vetores em um contexto real de aplicação.

```
using System;

class Program
{
    static void Main()
    {
        // Declaração do vetor para armazenar as notas
        int[] notas = new int[5];

        // Variáveis para cálculos
        int soma = 0;
        int notaMaxima = int.MinValue;
        int notaMinima = int.MaxValue;
        int alunosAprovados = 0;
```

```
// Leitura das notas e cálculos iniciais
Console.WriteLine("=== Sistema de Notas ===\n");

for (int i = 0; i < notas.Length; i++)
{
    bool notaValida = false;

    // Validação de entrada para garantir notas entre 0 e 10
    while (!notaValida)
    {
        Console.Write($"Digite a nota do aluno {i + 1} (0-10): ");
        if (int.TryParse(Console.ReadLine(), out notas[i]))
        {
            if (notas[i] >= 0 && notas[i] <= 10)
            {
                notaValida = true;
            }
            else
            {
                Console.WriteLine("Nota inválida! Digite um valor entre 0 e 10.");
            }
        }
        else
        {
            Console.WriteLine("Entrada inválida! Digite um número inteiro.");
        }
    }
}

// Atualiza a soma para cálculo posterior da média
soma += notas[i];

// Atualiza nota máxima e mínima
if (notas[i] > notaMaxima)
    notaMaxima = notas[i];

if (notas[i] < notaMinima)
    notaMinima = notas[i];

// Conta alunos aprovados (nota >= 7)
if (notas[i] >= 7)
    alunosAprovados++;
}
```

```

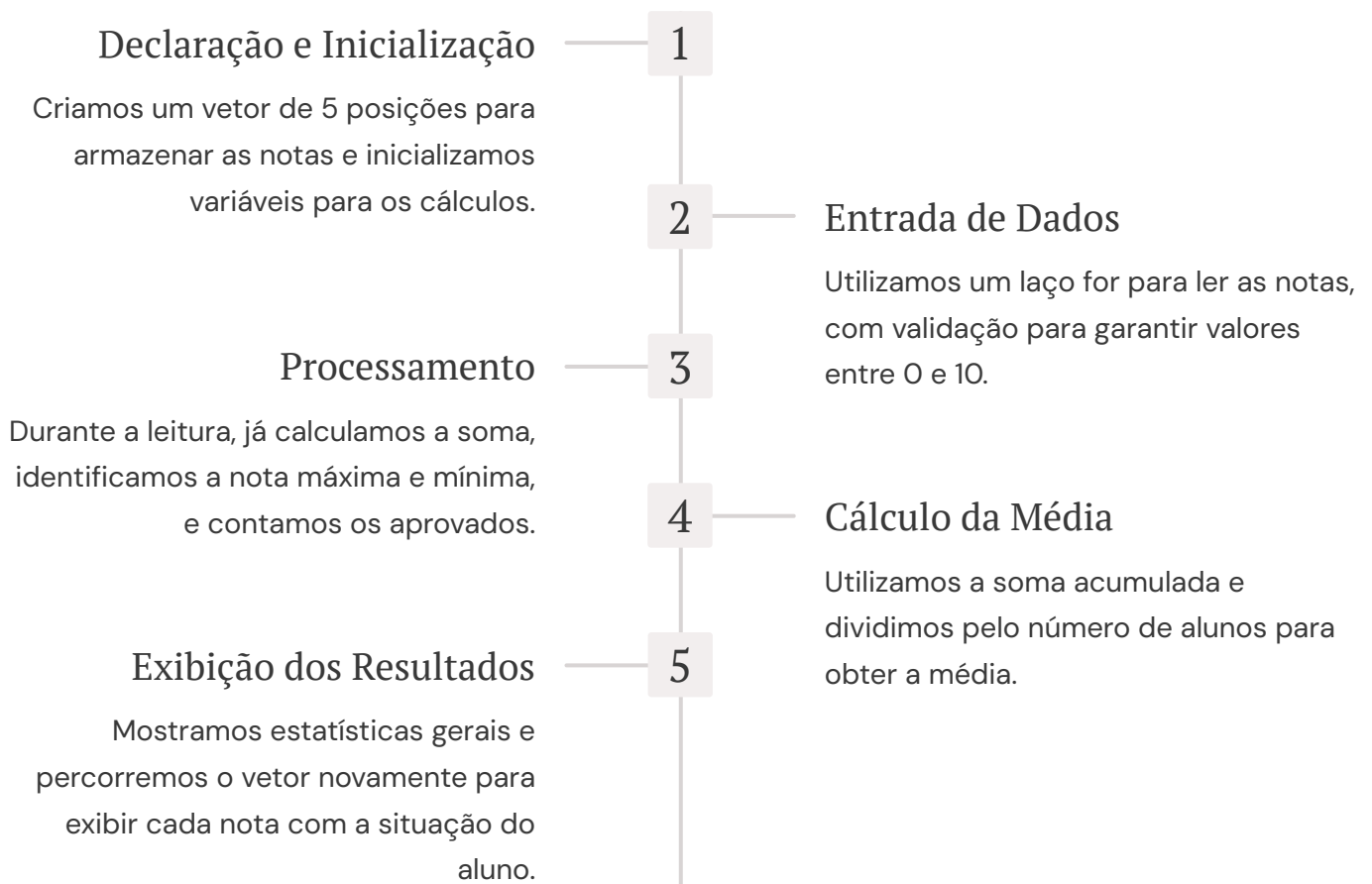
// Cálculo da média
double media = (double)soma / notas.Length;

// Exibição dos resultados
Console.WriteLine("\n=== Resultados ===");
Console.WriteLine($"Média da turma: {media:F2}");
Console.WriteLine($"Maior nota: {notaMaxima}");
Console.WriteLine($"Menor nota: {notaMinima}");
Console.WriteLine($"Alunos aprovados: {alunosAprovados} de
{notas.Length} ({(double)alunosAprovados/notas.Length*100:F1}%");

// Exibição de todas as notas em ordem
Console.WriteLine("\nNotas dos alunos:");
for (int i = 0; i < notas.Length; i++)
{
    string situacao = notas[i] >= 7 ? "Aprovado" : "Reprovado";
    Console.WriteLine($"Aluno {i + 1}: {notas[i]} - {situacao}");
}
}
}

```

Explicação do código:



Este exemplo demonstra várias técnicas importantes ao trabalhar com vetores:

- Declaração e inicialização do vetor
- Leitura de dados com validação
- Acesso aos elementos pelo índice
- Processamento dos dados para extrair informações úteis
- Percorrer o vetor múltiplas vezes para diferentes propósitos
- Formatação da saída para apresentar os resultados de forma clara

Este tipo de programa pode ser expandido para incluir mais funcionalidades, como ordenação das notas, cálculo de mediana, identificação de alunos em recuperação, entre outras possibilidades.

Dicas e boas práticas com vetores

O uso eficiente e seguro de vetores em C# requer atenção a alguns detalhes importantes. Seguir estas boas práticas ajudará você a evitar erros comuns e escrever código mais robusto e eficiente.

Inicialização adequada

Sempre inicie o vetor com o tamanho correto. Subestimar pode causar erros quando o programa precisar armazenar mais elementos do que o espaço alocado. Superestimar desperdiça memória.

Considere o uso de constantes ou variáveis para definir o tamanho do vetor, facilitando futuras alterações.

Validação de índices

Cuidado para não acessar índices fora do limite (`IndexOutOfRangeException`). Esta é uma das exceções mais comuns ao trabalhar com vetores.

Sempre verifique se o índice está dentro dos limites válidos antes de acessá-lo, especialmente quando o índice vem de entrada do usuário ou cálculos dinâmicos.

Uso da propriedade Length

Use o método `Length` para determinar o tamanho do vetor em vez de valores fixos. Isso torna seu código mais flexível e menos propenso a erros.

Exemplo: `for (int i = 0; i < array.Length; i++)` é melhor que `for (int i = 0; i < 10; i++)`.

Práticas adicionais recomendadas:

Nomes significativos

Dê nomes claros e descritivos aos seus vetores, geralmente no plural, que indiquem o tipo de dados que eles armazenam.

```
// Bom
string[] nomesDosAlunos;
double[] precosDosProdutos;

// Ruim
string[] array1;
double[] dados;
```

Tratamento de exceções

Use blocos try-catch para tratar exceções relacionadas a vetores, especialmente em código que lida com entradas de usuário ou dados externos.

```
try
{
    int indice = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine(numeros[indice]);
}
catch (IndexOutOfRangeException)
{
    Console.WriteLine("Índice inválido!");
}
catch (FormatException)
{
    Console.WriteLine("Digite um número válido!");
}
```

Métodos da classe Array

Aproveite os métodos e propriedades úteis disponíveis na classe Array:

- `Array.Sort(vetor)` – Ordena os elementos
- `Array.Reverse(vetor)` – Inverte a ordem
- `Array.BinarySearch(vetor, valor)` – Busca um valor
- `Array.Copy(origem, destino, tamanho)` – Copia elementos
- `Array.Clear(vetor, inicio, tamanho)` – Limpa elementos
- `Array.IndexOf(vetor, valor)` – Encontra a posição de um valor

Evite redimensionamento frequente

Vetores em C# têm tamanho fixo. Se você precisar de uma estrutura que mude de tamanho dinamicamente, considere usar `List<T>` em vez de recriar o vetor várias vezes.

i A classe `System.Array` em C# oferece métodos estáticos poderosos para manipulação de vetores. Familiarize-se com eles para escrever código mais eficiente e elegante. Consulte a documentação oficial da Microsoft para mais detalhes sobre esses métodos.

Seguindo estas práticas, você evitará erros comuns e criará código mais robusto, legível e eficiente ao trabalhar com vetores em C#. Lembre-se que boas práticas de programação não apenas melhoram a qualidade do seu código, mas também facilitam sua manutenção e expansão no futuro.

Exercícios para fixação

Para consolidar seu conhecimento sobre vetores, é essencial praticar com exercícios. Abaixo estão três exercícios de diferentes níveis de dificuldade, com dicas e soluções passo a passo para ajudar no seu aprendizado.

1

Armazenamento e exibição de cidades

Crie um vetor para armazenar os nomes de 4 cidades e imprima todos na tela.

Dica: Use um vetor de strings e um loop para mostrar os valores.

2

Encontrar o maior valor

Leia 10 números e mostre o maior valor digitado.

Dica: Use uma variável auxiliar para armazenar o maior valor encontrado até o momento.

3

Contagem de pessoas maiores de idade

Armazene as idades de um grupo de pessoas e calcule a quantidade de pessoas maiores de idade.

Dica: Use uma variável contadora e um loop para verificar cada idade.

Soluções comentadas:

Exercício 1: Armazenamento e exibição de cidades

```
using System;

class Program
{
    static void Main()
    {
        // Declaração e inicialização do vetor de cidades
        string[] cidades = new string[4];

        // Leitura dos nomes das cidades
        Console.WriteLine("Digite o nome de 4 cidades:");

        for (int i = 0; i < cidades.Length; i++)
        {
            Console.Write($"Cidade {i + 1}: ");
            cidades[i] = Console.ReadLine();
        }

        // Exibição dos nomes das cidades
        Console.WriteLine("\nCidades cadastradas:");

        for (int i = 0; i < cidades.Length; i++)
        {
            Console.WriteLine($"{i + 1}. {cidades[i]}");
        }

        // Alternativa usando foreach
        Console.WriteLine("\nUsando foreach:");
        int contador = 1;
        foreach (string cidade in cidades)
        {
            Console.WriteLine($"{contador++}. {cidade}");
        }
    }
}
```

Exercício 2: Encontrar o maior valor

```
using System;

class Program
{
    static void Main()
    {
        // Declaração do vetor para armazenar os números
        int[] numeros = new int[10];
        int maior = int.MinValue; // Inicializa com o menor valor possível

        Console.WriteLine("Digite 10 números inteiros:");

        // Leitura dos números e identificação do maior
        for (int i = 0; i < numeros.Length; i++)
        {
            Console.Write($"Número {i + 1}: ");
            if (int.TryParse(Console.ReadLine(), out numeros[i]))
            {
                // Verifica se o número atual é maior que o maior até agora
                if (numeros[i] > maior)
                {
                    maior = numeros[i];
                }
            }
            else
            {
                Console.WriteLine("Valor inválido! Digite novamente.");
                i--; // Repete a iteração atual
            }
        }

        // Exibição do resultado
        Console.WriteLine($"
O maior número digitado foi: {maior}");
    }
}
```

Exercício 3: Contagem de pessoas maiores de idade

```
using System;

class Program
{
    static void Main()
    {
        // Solicitação do tamanho do grupo
        Console.WriteLine("Quantas pessoas no grupo? ");
        int quantidadePessoas = int.Parse(Console.ReadLine());

        // Declaração do vetor para armazenar as idades
        int[] idades = new int[quantidadePessoas];
        int maioresDeldade = 0;

        // Leitura das idades
        Console.WriteLine("\nDigite a idade de cada pessoa:");

        for (int i = 0; i < idades.Length; i++)
        {
            Console.WriteLine($"Idade da pessoa {i + 1}: ");
            idades[i] = int.Parse(Console.ReadLine());

            // Verifica se a pessoa é maior de idade (>=18)
            if (idades[i] >= 18)
            {
                maioresDeldade++;
            }
        }

        // Exibição dos resultados
        Console.WriteLine($"No grupo de {quantidadePessoas} pessoas:");
        Console.WriteLine($"- {maioresDeldade} são maiores de idade");
        Console.WriteLine($"- {quantidadePessoas - maioresDeldade} são menores de idade");

        // Cálculo da porcentagem
        double percentualMaiores = (double)maioresDeldade /
            quantidadePessoas * 100;
        Console.WriteLine($"- {percentualMaiores:F1}% do grupo é maior de idade");
    }
}
```

Estes exercícios ajudam a reforçar os conceitos fundamentais de vetores: declaração, inicialização, acesso a elementos, percorrimento, e processamento de dados. À medida que você se torna mais confortável com esses conceitos básicos, pode avançar para exercícios mais complexos envolvendo algoritmos de ordenação, busca, e manipulações mais sofisticadas de vetores.

- ✔ Desafio extra: Modifique o Exercício 2 para mostrar também a posição (índice) do maior valor no vetor. Se houver valores repetidos, mostre todas as posições onde o maior valor aparece.

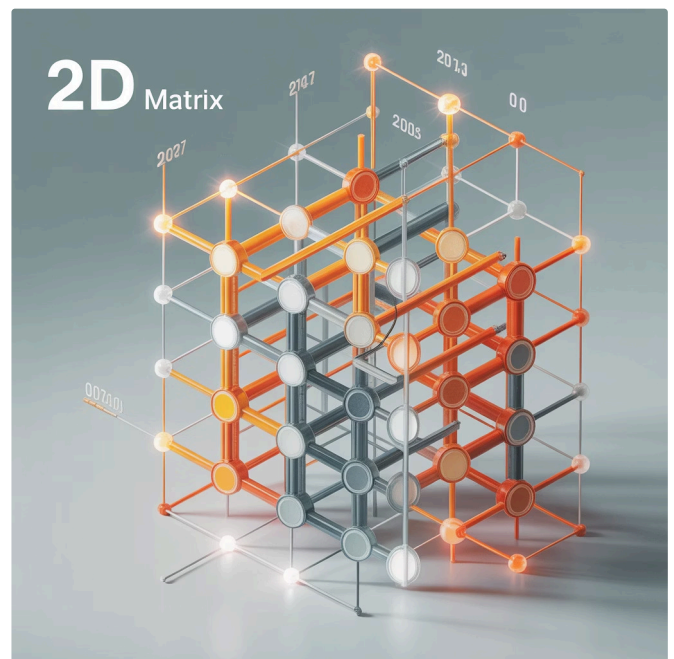
Capítulo 2 – Matrizes em C#

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.

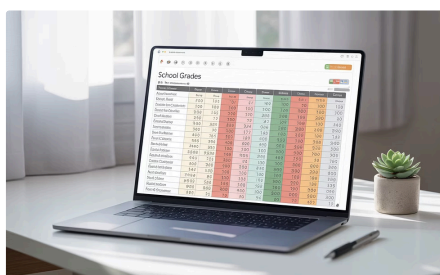
Visualizando uma matriz:

[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.



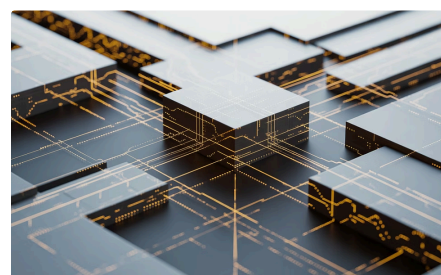
Tabelas de Dados

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

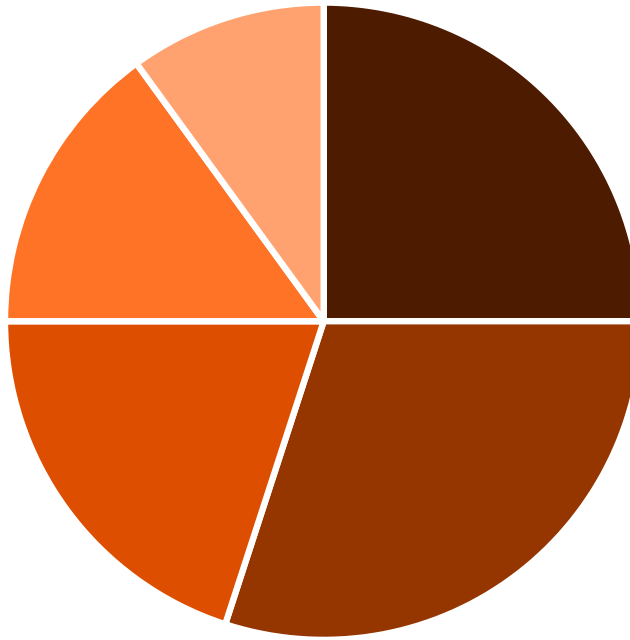
Para dados que possuem relação bidimensional natural, matrizes oferecem acesso mais rápido e lógico do que outras estruturas alternativas, como listas aninhadas.

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.

i 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.



■ Jogos

■ Dados tabulares

■ Gráficos/Imagens

■ Cálculos científicos

■ Outros

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

tipo

O tipo de dado que a matriz irá armazenar (int, double, string, etc.). Todos os elementos da matriz serão deste tipo.

Exemplos: int, string, double, bool, ou tipos personalizados como classes próprias.

2

nomeDaMatriz

O identificador da variável que representa a matriz, seguindo as regras de nomenclatura do C#.

Boas práticas sugerem usar nomes que indiquem o propósito ou conteúdo da matriz, como tabuleiro, notas ou mapa.

3

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.

4

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

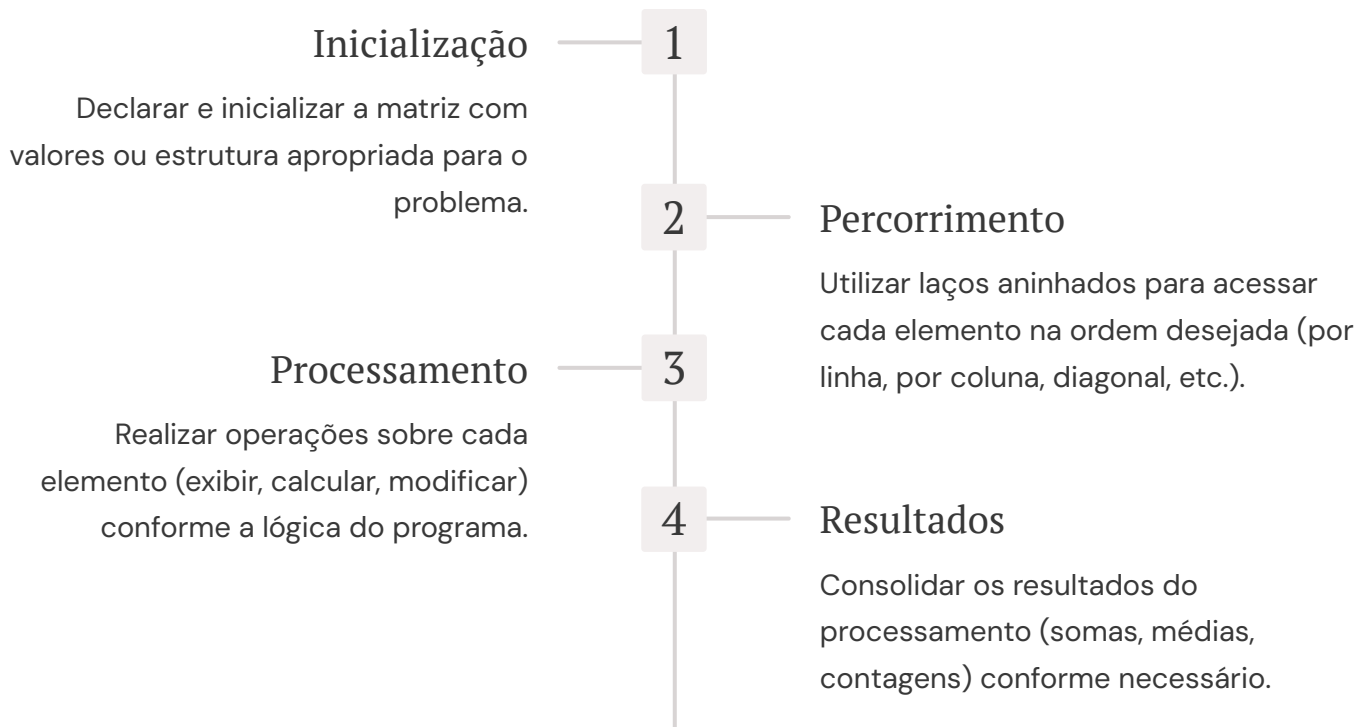
Embora menos comum por não fornecer acesso aos índices, também é possível percorrer todos os elementos de uma matriz usando o laço foreach:

```
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.WriteLine($" 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.

Processamento

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

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.

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($"
A soma da diagonal principal é:
{somaDiagonal}");
```

```
// Opcional: destacando a diagonal principal
Console.WriteLine("
Diagonal 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: <code>vetor[i]</code>	Dois ou mais índices: <code>matriz[i,j]</code>
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.

