

# Trabalhar com os dados em C#

A linguagem de programação C# depende extensivamente de tipos de dados. Os tipos de dados restringem os tipos de valores que podem ser armazenados em uma determinada variável, que pode ser útil ao tentar criar um código sem erros.

Um tipo de dados é um constructo de linguagem de programação que define a quantidade de memória a ser reservada para um valor.

## Tipos de valor e de referência.

A diferença fundamental entre tipos de valor e de referência diz respeito ao local em que esses valores estão temporariamente armazenados na memória conforme o aplicativo é executado. O local em que o valor é armazenado afeta a maneira como o runtime do .NET gerencia a vida útil do valor, incluindo sua declaração (nascimento), atribuição e recuperação (vida útil) e finalização (morte).

## Tipos de valor simples.

Os tipos de valor simples são um conjunto de tipos predefinidos fornecidos pelo C# como palavras-chave. Essas palavras-chave são meramente aliases para tipos predefinidos definidos na Biblioteca de Classes do .NET. Por exemplo, a palavra-chave **int** do C# é um alias de um tipo de valor definido na Biblioteca de Classes do .NET como **System.Int32**.

## Tipos Integrais.

Um **tipo integral** é um tipo de valor simples que representa números inteiros (não fracionários).

### Tipos integrais com sinal.

Um *tipo com sinal* usa seus bytes para representar uma quantidade igual de números positivos e negativos.

Editor do .NET

```
1 Console.WriteLine("Signed integral types:");
2
3 Console.WriteLine($"sbyte : {sbyte.MinValue} to {sbyte.MaxValue}");
4 Console.WriteLine($"short : {short.MinValue} to {short.MaxValue}");
5 Console.WriteLine($"int : {int.MinValue} to {int.MaxValue}");
6 Console.WriteLine($"long : {long.MinValue} to {long.MaxValue}");
```

Saída

```
Signed integral types:
sbyte : -128 to 127
short : -32768 to 32767
int : -2147483648 to 2147483647
long : -9223372036854775808 to 9223372036854775807
```

O código utiliza o `MinValue` e `MaxValue` para expor os intervalos de valores para os vários tipos de dados integrais com sinal.

## Tipos integrais sem sinal.

Um *tipo sem sinal* usa seus bytes para representar apenas números positivos.

Editor do .NET

```
1 Console.WriteLine("Unsigned integral types:");
2
3 Console.WriteLine($"byte      : {byte.MinValue} to {byte.MaxValue}");
4 Console.WriteLine($"ushort    : {ushort.MinValue} to {ushort.MaxValue}");
5 Console.WriteLine($"uint      : {uint.MinValue} to {uint.MaxValue}");
6 Console.WriteLine($"ulong     : {ulong.MinValue} to {ulong.MaxValue}");
```

Saída

```
Unsigned integral types:
byte      : 0 to 255
ushort    : 0 to 65535
uint      : 0 to 4294967295
ulong     : 0 to 18446744073709551615
```

O código utiliza o `MinValue` e `MaxValue` para expor os intervalos de valores para os vários tipos de dados integrais sem sinal.

## Tipos de ponto flutuante.

Um ponto flutuante é um tipo de valor simples que representa números fracionários.

Primeiro, você também deve considerar os dígitos de precisão que cada um permite. Precisão é o número de valores que podem ser armazenados após o ponto decimal.

Em segundo lugar, você deve considerar a maneira como os valores são armazenados e o impacto sobre a precisão do valor.

**Float** e **double** são úteis porque números grandes podem ser armazenados usando um pequeno volume de memória, mas só deverão ser usados quando um valor aproximado for útil.

Quando você precisar de uma resposta mais precisa, deverá usar **decimal**. Cada valor do tipo **decimal** tem um volume de memória relativamente grande; contudo, efetuar operações matemáticas dá a você um resultado mais preciso.

```
Console.WriteLine("Floating point types:");
Console.WriteLine($"float      : {float.MinValue} to {float.MaxValue} (with ~6-9 digits of precision)");
Console.WriteLine($"double    : {double.MinValue} to {double.MaxValue} (with ~15-17 digits of precision)");
Console.WriteLine($"decimal   : {decimal.MinValue} to {decimal.MaxValue} (with ~28-29 digits of precision)");
```

Saída

```
Floating point types:
float      : -3.402823E+38 to 3.402823E+38 (with ~6-9 digits of precision)
double     : -1.79769313486232E+308 to 1.79769313486232E+308 (with ~15-17 digits of precision)
decimal    : -79228162514264337593543950335 to 79228162514264337593543950335 (with ~28-29
digits of precision)
```

Como você pode ver, **float** e **double** usam uma notação diferente de **decimal** para representar seus maiores e menores valores possíveis. Mas o que essa notação significa?

Com os tipos de ponto flutuante podem armazenar grandes números com muita precisão, seus valores podem ser representados usando a “notação E”, que é uma forma de notação científica que significa “vezes dez elevado à potência de”. Assim, um valor como  $5E+2$  seria o valor 500, porque é o equivalente de  $5 * 10^2$  ou  $5 * 10 * 10$ .

1. Um tipo de ponto flutuante é um tipo de dados de valor simples que pode armazenar números fracionários.
2. Escolher o tipo de ponto flutuante certo para seu aplicativo exige que você considere mais do que apenas os valores máximo e mínimo que ele pode armazenar. Você também deve considerar quantos valores podem ser preservados após o decimal, como os números são armazenados e como o armazenamento interno afeta o resultado de operações matemáticas.
3. Os valores de ponto flutuante às vezes poderão ser representados usando a “notação E” quando os números ou expoentes crescerem muito.
4. Há uma diferença fundamental em como o compilador e o tempo de execução manipulam o **decimal** em oposição ao **float** ou ao **double**, principalmente ao determinar quanta precisão é necessária de operações matemáticas.

## Tipos de referência.

Os tipos de referência incluem matrizes, classes e cadeias de caracteres. Os tipos de referência são tratados de maneira diferente dos tipos de valor com relação a como os valores são armazenados quando o aplicativo está em execução.

Uma variável de tipo de valor armazenará seus valores diretamente em uma área de armazenamento chamada a *pilha*. A pilha é a memória alocada ao código que está em execução, no momento, na CPU. Quando a execução do registro de ativação for concluída, os valores na pilha serão removidos.

Uma variável de tipo de referência armazenará seus valores em uma região de memória separada chamada de *heap*. O *heap* é uma área de memória compartilhada entre muitos aplicativos em execução no sistema operacional ao mesmo tempo. O .NET comunica-se com o sistema operacional para determinar quais endereços de memória estão disponíveis. O runtime do .NET armazena o valor e, em seguida, retorna o endereço de memória à variável. Quando o código usa a variável, o .NET pesquisa sem interrupção o endereço armazenado na variável e recupera o valor que está armazenado lá.

C#

```
int[] data;  
data = new int[3];
```

A primeira linha de código reserva um endereço na memória no *heap*.

A palavra-chave **new** informa ao runtime do .NET para criar uma instância da matriz **int** e, em seguida, coordena-se com o sistema operacional para armazená-la na memória. O runtime do .NET cumpre e retorna um endereço de memória da nova matriz **int**. Finalmente, o endereço de memória é armazenado nos dados de variável.

O tipo de dados **string** também é um tipo de referência. Mas não usamos o operador **new** ao declarar uma cadeia de caracteres.

Como o tipo de dados **string** é usado com frequência, em segundo plano, no entanto, uma nova instância de `System.String` é criada e inicializada. Isso é meramente uma conveniência proporcionada pelos designers do C#.

#### ❗ Observação

Quando os designers de linguagem criam um atalho simplificado, às vezes ele é conhecido como "açúcar sintático". Você provavelmente verá essa frase usada em artigos, vídeos e em apresentações.

#### Editor do .NET

```
1  string msg = "Hello World!";  
2  Console.WriteLine(msg);  
3
```

#### Saída

Hello World!

## Como você escolhe o tipo de dados certo?

Ao avaliar suas opções, é necessário ponderar várias considerações importantes. Geralmente, não há uma única resposta correta, mas algumas delas estão mais corretas do que outras.

**Escolha o tipo de dados que tem o intervalo de valor desejado:** Por exemplo, se você sabe que uma variável específica deve armazenar apenas um número entre 1 e 10 mil, caso contrário ela estaria fora dos limites do que seria esperado, você provavelmente evitaria **byte**, **sbyte** pois seus intervalos são muito baixos. Além disso, provavelmente você não precisaria de **int**, **long**, **uint** e **ulong**, porque eles podem armazenar muito mais dados do que é necessário. Da mesma forma, você provavelmente ignoraria **float**, **double** e **decimal** se não precisasse de valores fracionários. Você poderia restringir isso para **short** e **ushort**, ambos podendo ser viáveis dadas as circunstâncias.

**Escolha o tipo de dados com base na interação com funções de biblioteca e os tipos de dados de suas entradas e saídas:** Suponha que você queira trabalhar com um intervalo de anos entre duas datas. Considerando que esse é um aplicativo de negócios, você pode determinar que só precisa de um intervalo entre cerca de 1960 a 2200. Isso pode fazer você trabalhar com o **byte**, pois ele pode representar números entre 0 e 255. No entanto, quando você examina métodos internos nas classes **System.TimeSpan** e **System.DateTime**, percebe que elas aceitam valores do tipo **double** e **int**. Se escolher **sbyte**, você constantemente estará transmitindo para frente e para trás entre **sbyte** e **double** ou **int**. Nesse caso, poderá fazer mais sentido escolher **int** se você não precisar de uma precisão de subsegundos e **double** se você precisar de uma precisão de subsegundos.

**Escolha o tipo de dados com base no impacto em outros sistemas, como armazenamento de longo prazo em um banco de dados:** Às vezes, é necessário considerar como as informações serão consumidas por outros aplicativos ou outros sistemas, como um banco de dados. Por exemplo, o sistema de tipo do SQL Server é diferente do sistema de tipo do C#. Como resultado, um mapeamento entre os dois deve acontecer antes que você possa salvar dados nesse banco de dados. Se a finalidade do seu aplicativo é criar uma interface com um banco de dados, você provavelmente precisará considerar como os dados serão armazenados, quantos dados serão armazenados e como a escolha de um tipo de dados maior poderia afetar a quantidade (e o custo) do armazenamento físico necessário para armazenar todos os dados que seu aplicativo gerará.

## Converter tipos de dados usando técnicas de conversão cast em C#.

Ao trabalhar com dados, com frequência você precisará alterar um valor de um tipo de dados para outro. Há muitas razões pelas quais é preciso fazer isso.

### Conversão cast de tipo de dados e conversão de tipo de dados.

```
Editor do .NET

1  int first = 2;
2  string second = "4";
3  int result = first + second;
4  Console.WriteLine(result);

Saída

(3,14): error CS0029: Cannot implicitly convert type 'string' to 'int'
```

A mensagem de erro nos informa que o problema é com o nosso uso do tipo de dados **string**. Mas por que o Compilador C# não pode apenas lidar com isso?

A variável **second** é do tipo **string**, portanto, é possível defini-la com um valor diferente, como "hello". Se o compilador C# tentou converter "hello" em um número que causaria uma exceção em tempo de execução. Para evitar essa possibilidade, o compilador C# não executará implicitamente a conversão de **string** em **int**.

Se a sua intenção é fazer uma adição usando uma cadeia de caracteres, o compilador C# exige que você assuma um controle mais explícito do processo de conversão de dados.

Para executar a conversão de dados, você pode usar uma de várias técnicas:

1. Usar um método auxiliar no tipo de dados.
2. Usar um método auxiliar na variável.
3. Usar os métodos da classe Convert.

### Conversão de expansão.

Significa que você está tentando converter um valor **de** um tipo de dados que poderia armazenar *menos* informações **em** um tipo de dados que pode armazenar *mais* informações.

Exemplo: converter um **int** em um **decimal**.

```
Editor do .NET
1  int myInt = 3;
2  Console.WriteLine($"int: {myInt}");
3
4  decimal myDecimal = myInt;
5  Console.WriteLine($"decimal: {myDecimal}");
```

```
Saída
int: 3
decimal: 3
```

Quando você sabe que executará uma conversão de expansão, pode depender da **conversão implícita**. A conversão implícita é tratada pelo compilador.

Como qualquer valor **int** pode se ajustar facilmente dentro de um **decimal**, o compilador executa a conversão.

### Conversão de restrição.

Significa que você está tentando converter um valor de um tipo de dados que pode armazenar mais informações em um tipo de dados que pode armazenar menos informações. Nesse caso, você pode perder informações.

```
Editor do .NET
1  decimal myDecimal = 3;
2  Console.WriteLine($"int: {myDecimal}");
3
4  int myInt = myDecimal;
5  Console.WriteLine($"decimal: {myInt}");
```

```
Saída
(4,13): error CS0266: Cannot implicitly convert type 'decimal' to 'int'. An explicit conversion exists (are you missing a cast?)
```

Quando você sabe que executará uma conversão de restrição, precisará executar uma **conversão cast**. A conversão cast é uma instrução ao compilador C# de que você sabe que talvez a precisão seja perdida, mas que se dispõe a aceitar isso.



## Conversão cast.

Para executar uma conversão **cast**, use o operador de conversão **cast** () para envolver um tipo de dados e colocá-lo ao lado da variável que você deseja converter.

```
Editor do .NET
1  decimal myDecimal = 3.14m;
2  Console.WriteLine($"decimal: {myDecimal}");
3
4  int myInt = (int)myDecimal;
5  Console.WriteLine($"int: {myInt}");
6

Saída
decimal: 3.14
int: 3
```

A variável **myDecimal** armazena um valor que tem dois locais de precisão. Ao adicionar a instrução de conversão cast (**int**) na linha 4, estamos informando ao compilador C# que entendemos que é possível que percamos essa precisão.

## Como eu sei se uma conversão é “de expansão” ou “de restrição”?

Se você não tiver certeza se perderá dados ou não, poderá consultar os artigos do docs.

Tabelas de Conversão de Tipo no .NET

<https://docs.microsoft.com/pt-br/dotnet/standard/base-types/conversion-tables>

Tabela de tipos internos

<https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/keywords/built-in-types-table>

Os desenvolvedores frequentemente escrevem pequenos testes para entender melhor as propensões de duas técnicas semelhantes.

Por exemplo, você pode fazer algo como isto:

#### Editor do .NET

```
1 decimal myDecimal = 1.23456789m;  
2 float myFloat = (float)myDecimal;  
3  
4 Console.WriteLine($"Decimal: {myDecimal}");  
5 Console.WriteLine($"Float : {myFloat}");
```

#### Saída

```
Decimal: 1.23456789  
Float : 1.234568
```

Com base nisso, posso ver que a conversão **cast** de um **decimal** em um **float** é uma conversão de restrição, pois estou perdendo precisão.

## Executar conversões de dados.

Usar **ToString()** para converter um número em uma cadeia de caracteres.

Cada variável de tipo de dados tem um método **ToString()**. O que o método **ToString()** faz depende de como ele é implementado em um determinado tipo.

#### Editor do .NET

```
1 int first = 5;  
2 int second = 7;  
3 string message = first.ToString() + second.ToString();  
4 Console.WriteLine(message);
```

#### Saída

```
57
```

Esse código mostra como usar o método **ToString()** para converter explicitamente valores **int** em strings.

## Converter explicitamente uma cadeia de caracteres em um número.

A maioria dos tipos de dados numéricos tem um método **Parse()**, que converte uma cadeia de caracteres no tipo de dados fornecido.

Editor do .NET

```
1 string first = "5";
2 string second = "7";
3 int sum = int.Parse(first) + int.Parse(second);
4 Console.WriteLine(sum);
```

Saída

12

O código utiliza o método **Parse()** para converter duas cadeias de caracteres em valores **int** e depois somá-las.

### Conversão de dados usando a classe **Convert**.

A classe **Convert** tem muitos métodos auxiliares para converter um valor de um tipo em outro.

Editor do .NET

```
1 string value1 = "5";
2 string value2 = "7";
3 int result = Convert.ToInt32(value1) * Convert.ToInt32(value2);
4 Console.WriteLine(result);
```

Saída

35

O método **ToInt32()** tem 19 versões sobrecarregadas que permitem que ele aceite praticamente todos os tipos de dados.

#### 📌 Observação

Por que o nome do método é **ToInt32()**? Por que não **ToInt()**? **System.Int32** é o nome do tipo de dados subjacente na Biblioteca de Classes do .NET que a linguagem de programação C# mapeia para a palavra-chave **int**. Como a classe **Convert** também faz parte da Biblioteca de Classes do .NET, ela é chamada por seu nome completo, não pelo seu nome em C#. Definindo tipos de dados como parte da Biblioteca de Classes do .NET, várias linguagens .NET como Visual Basic, F#, IronPython e outras podem compartilhar os mesmos tipos de dados e as mesmas classes na Biblioteca de Classes do .NET. No final deste módulo, publicaremos alguns links para que você possa aprender mais sobre o Common Type System do .NET.

Então, quando devemos usar a classe **Convert**? A classe **Convert** é melhor para converter números fracionários em números inteiros (**int**) porque ela arredonda da maneira esperada.

Editor do .NET

```
1  int value = (int)1.5m; //casting truncates
2  Console.WriteLine(value);
3
4  int value2 = Convert.ToInt32(1.5m); //converting rounds up
5  Console.WriteLine(value2);
```

Saída

```
1
2
```

Durante a conversão **cast**, o valor de **float** é truncado, o que significa que o valor depois do decimal é ignorado completamente.

Ao converter usando **Convert.ToInt32()**, o valor de **float** literal é corretamente arredondado para **2**. Se alterássemos o valor literal para **1.499m**, ele seria arredondado para baixo para **1**.

## Recapitulação:

1. Executar uma conversão de dados quando for possível que isso possa causar um erro em runtime.
2. Executar uma conversão cast explícita para informar ao compilador que você entende o risco de perder dados.
3. Depender do compilador para executar uma conversão cast implícita ao executar uma conversão de expansão.
4. Use o operador cast () e o tipo de dados para executar uma conversão (por exemplo, (int)myDecimal).
5. Use a classe Convert quando desejar executar uma conversão de restrição, mas desejar executar arredondamento, e não truncamento das informações.

## Método TryParse()

Como o tipo de dados de cadeia de caracteres pode armazenar um valor não numérico, é possível que executar a conversão de uma **string** em um tipo de dados numérico causará um erro de tempo de execução.

C# Copiar Executar

```
string name = "Bob";
Console.WriteLine(int.Parse(name));
```

Saída Copiar

```
System.FormatException: 'Input string was not in a correct format.'
```

Para se proteger contra isso, você deve usar o método **TryParse()** no tipo de dados de destino.

O método **TryParse()** faz várias coisas simultaneamente:

1. Ele tenta analisar uma cadeia de caracteres sobre o tipo de dados numérico fornecido.
2. Se for bem-sucedido, ele armazenará o valor convertido em um **parâmetro out**.
3. Ele retorna um **bool** para indicar se ele teve êxito ou não.

O método **int.TryParse()** retornará **true** se ele tiver convertido com êxito nossa **string** de variável **value** em um **int**; caso contrário, ele retornará **false**.

Editor do .NET

```
1 string value = "102";
2 int result = 0;
3 if (int.TryParse(value, out result))
4 {
5     Console.WriteLine($"Measurement: {result}");
6 }
7 else
8 {
9     Console.WriteLine("Unable to report the measurement.");
10 }
11
12 Console.WriteLine($"Measurement (w/ offset): {50 + result}");
```

Saída

```
Measurement: 102
Measurement (w/ offset): 152
```

Evolvendo a instrução em uma instrução **if** conseguimos fazer um controle. Quando o valor for convertido será armazenado no **int** da variável **result**.

A palavra-chave **out** instrui o compilador de que o método **TryParse()** não retornará apenas um valor mas será populado pelo parâmetro **out**, podendo ser usado posteriormente em seu código.

## Desafio.

Desafio para dividir os dados dependendo do seu tipo e a concatenar ou adicionar os dados de forma adequada.

Dados: "12.3", "45", "ABC", "11", "DEF".

Dica: Iterar por meio de cada valor em uma cadeia de caracteres de valores.

Regras de negócio:

- Regra 1: se o valor for alfanumérico, concatene-o para formar uma mensagem
- Regra 2: se o valor for numérico, adicione-o ao total
- Regra 3: verifique se o resultado corresponde à seguinte saída:

Message: ABCDEF

Total: 68.3


```
23 string[] values = {"12.3", "45", "ABC", "11", "DEF"};
24 decimal result = 0m;
25 decimal total = 0;
26 string message = "";
27
28 foreach(string valores in values )
29 {
30     if (decimal.TryParse(valores, out result))
31     {
32         total += result;
33     }
34     else
35     {
36         message += valores;
37     }
38 }
39
40
41 Console.WriteLine($"Message: {message}");
42 Console.WriteLine($"Total: {total}");
43
```

Saída

Message: ABCDEF  
Total: 68.3

## Desafio.

O desafio a seguir forçará você a entender as implicações da conversão cast de valores considerando o impacto de conversões de restrição e expansão, exibindo a saída a seguir:

Saída	 Copiar
<pre>Divide value1 by value2, display the result as an int: 1 Divide value2 by value3, display the result as a decimal: 1.4418604651162790697674418605 Divide value3 by value1, display the result as a float: 0.3583333</pre>	

Dos seguintes dados:

```
C#

int value1 = 12;
decimal value2 = 6.2m;
float value3 = 4.3f;
```

## Resultado.

Editor do .NET
<pre>1  int value1 = 12; 2  decimal value2 = 6.2m; 3  float value3 = 4.3f; 4 5  decimal divideValue = (decimal)value1 / value2; 6  int result1 = (int)divideValue; 7  Console.WriteLine(\$"Divide value1 by value2, display the result as an int: {result1}"); 8 9  decimal result2 = value2 / (decimal)value3; 10 Console.WriteLine(\$"Divide value2 by value3, display the result as a decimal: {result2}"); 11 12 float result3 = value3 / value1; 13 Console.WriteLine(\$"Divide value3 by value1, display the result as a float: {result3}");</pre>
Saída
<pre>Divide value1 by value2, display the result as an int: 1 Divide value2 by value3, display the result as a decimal: 1.4418604651162790697674418605 Divide value3 by value1, display the result as a float: 0.3583333</pre>

# Métodos auxiliares para matrizes.

As matrizes em C# permitem armazenar sequências de valores em uma única estrutura de dados. Depois que os dados estiverem em uma matriz, você poderá manipular a ordem e o conteúdo da matriz. Além disso, você pode executar operações avançadas de cadeia de caracteres usando métodos auxiliares da matriz.

## Classificar a matriz.

Editor do .NET

```
1  string[] pallets = {"B14", "A11", "B12", "A13"};
2
3  Console.WriteLine("Sorted...");
4  Array.Sort(pallets);
5  foreach (var pallet in pallets)
6  {
7      Console.WriteLine($"-- {pallet}");
8  }
```


Saída

```
Sorted...
-- A11
-- A13
-- B12
-- B14
```

Nesse código na linha **Array.Sort(pallets);** estamos usando o método **Sort()** da classe **Array** para classificar os itens na matriz de forma alfanumérica.

## Invertendo a ordem.

C#

 Copiar

```
string[] pallets = { "B14", "A11", "B12", "A13" };

Console.WriteLine("Sorted...");
Array.Sort(pallets);
foreach (var pallet in pallets)
{
    Console.WriteLine($"-- {pallet}");
}

Console.WriteLine("");
Console.WriteLine("Reversed...");
Array.Reverse(pallets);
foreach (var pallet in pallets)
{
    Console.WriteLine($"-- {pallet}");
}
```



#### Saída

Sorted...

```
-- A11
-- A13
-- B12
-- B14
```

Reversed...

```
-- B14
-- B12
-- A13
-- A11
```

Concentre-se na linha de código **Array.Reverse(pallets)**; Aqui, estamos usando o método **Reverse()** da classe **Array** para inverter a ordem dos itens.

## Limpar itens da matriz.

#### Editor do .NET

```
1 string[] pallets = {"B14", "A11", "B12", "A13"};
2 Console.WriteLine("");
3
4 Array.Clear(pallets, 0, 2);
5 Console.WriteLine($"Clearing 2 ... count: {pallets.Length}");
6 foreach (var pallet in pallets)
7 {
8     Console.WriteLine($"-- {pallet}");
9 }
```

#### Saída

```
Clearing 2 ... count: 4
--
--
-- B12
-- A13
```

Aqui, estamos usando o método **Array.Clear()** para limpar os valores armazenados nos elementos da matriz **pallets** começando no índice **0** e limpando 2 elementos.

Quando você usa **Array.Clear()**, os elementos que foram limpos não referenciam mais uma cadeia de caracteres na memória.

Os valores armazenados nos dois primeiros elementos da matriz foram limpos. Como podemos ver na propriedade **Length** e na instrução **foreach**, os elementos ainda existem, mas agora estão vazios.

Se tentar utilizar um método em um elemento limpo o C# retornará uma exceção, pois se torna nulo.

## Redimensionar a matriz para adicionar mais elementos.

```
C# Copiar

string[] pallets = { "B14", "A11", "B12", "A13" };
Console.WriteLine("");

Array.Clear(pallets, 0, 2);
Console.WriteLine($"Clearing 2 ... count: {pallets.Length}");
foreach (var pallet in pallets)
{
    Console.WriteLine($"-- {pallet}");
}

Console.WriteLine("");
Array.Resize(ref pallets, 6);
Console.WriteLine($"Resizing 6 ... count: {pallets.Length}");

pallets[4] = "C01";
pallets[5] = "C02";

foreach (var pallet in pallets)
{
    Console.WriteLine($"-- {pallet}");
}
```

### Saída

```
Clearing 2 ... count: 4
--
--
-- B12
-- A13

Resizing 6 ... count: 6
--
--
-- B12
-- A13
-- C01
-- C02
```

Concentre-se na linha **Array.Resize(ref pallets, 6);** Aqui, estamos chamando o método **Resize()** passando a matriz **pallets** por referência, usando a palavra-chave **ref**.

Nesse caso, estamos redimensionando a matriz **pallets** de quatro elementos para 6. Os novos elementos são adicionados ao final dos elementos atuais. Os dois novos elementos serão nulos até atribuirmos um valor a eles.

## Redimensionar a matriz para remover elementos.

```
C# Copiar Executar

string[] pallets = { "B14", "A11", "B12", "A13" };
Console.WriteLine("");

Array.Clear(pallets, 0, 2);
Console.WriteLine($"Clearing 2 ... count: {pallets.Length}");
foreach (var pallet in pallets)
{
    Console.WriteLine($"-- {pallet}");
}

Console.WriteLine("");
Array.Resize(ref pallets, 6);
Console.WriteLine($"Resizing 6 ... count: {pallets.Length}");

pallets[4] = "C01";
pallets[5] = "C02";

foreach (var pallet in pallets)
{
    Console.WriteLine($"-- {pallet}");
}

Console.WriteLine("");
Array.Resize(ref pallets, 3);
Console.WriteLine($"Resizing 3 ... count: {pallets.Length}");

foreach (var pallet in pallets)
{
    Console.WriteLine($"-- {pallet}");
}
```

Saída

```
Clearing 2 ... count: 4
--
--
-- B12
-- A13

Resizing 6 ... count: 6
--
--
-- B12
-- A13
-- C01
-- C02

Resizing 3 ... count: 3
--
--
-- B12
```

Observe que a última chamada de **Array.Resize()** não eliminou os dois primeiros elementos nulos. Em vez disso, ela removeu os três últimos elementos. Pois a matriz foi redefinida para ter 3 posições.

## Métodos Array do tipo de dados String

As variáveis do tipo **string** têm muitos métodos internos que convertem uma única cadeia de caracteres em uma matriz de cadeias de caracteres menores ou em uma matriz de caracteres individuais.

### Usar ToCharArray() para inverter uma cadeia de caracteres

Aqui estamos usando o método **ToCharArray()** para criar uma matriz de **char**. Cada elemento da matriz tem um caractere da cadeia de caracteres original.

Editor do .NET

```
1 string value = "abc123";
2 char[] valueArray = value.ToCharArray();
3
4 foreach (var values in valueArray)
5 {
6     Console.WriteLine($"-- {values}");
7 }
```

Saída

```
-- a
-- b
-- c
-- 1
-- 2
-- 3
```

Inverter e combinar a matriz char em uma nova cadeia de caracteres.

Editor do .NET

```
1 string value = "abc123";
2 char[] valueArray = value.ToCharArray();
3
4 Array.Reverse(valueArray);
5 string result = new string(valueArray);
6 Console.WriteLine(result);
```

Saída

```
321cba
```

Com **Array.Reverse()** invertemos os elementos da matriz e a **expressão new string(valueArray)** cria uma instância vazia da classe **System.String** (que é a mesma do tipo de dados string em C#) e passa a matriz char como um construtor.

## Cadeia de caracteres de valores separados por vírgula usando Join.

Talvez seja necessário separar cada elemento da matriz char usando uma vírgula. Usaremos o método **Join()** da classe String, passando o char que desejamos para delimitar cada segmento (a vírgula) e a própria matriz.

Editor do .NET

```
1 string value = "abc123";
2 char[] valueArray = value.ToCharArray();
3
4 Array.Reverse(valueArray);
5 string result = String.Join(",", valueArray);
6 Console.WriteLine(result);
```

Saída

3,2,1,c,b,a

## Dividir a nova cadeia de caracteres de valores separados por vírgula em uma matriz de cadeias de caracteres.

Por fim, vamos usar o método **Split()** disponível para variáveis do tipo cadeia de caracteres para criar uma matriz de cadeias de caracteres. Usaremos a vírgula como o delimitador para dividir uma cadeia de caracteres longa em cadeias de caracteres menores. Por fim, executaremos um loop em cada elemento da nova matriz de cadeia de caracteres.

Editor do .NET

```
1 string value = "abc123";
2 char[] valueArray = value.ToCharArray();
3
4 Array.Reverse(valueArray);
5
6 string result = String.Join(",", valueArray);
7 Console.WriteLine(result);
8
9 string[] items = result.Split(',');
10 foreach (string item in items)
11 {
12     Console.WriteLine(item);
13 }
```

Saída

3,2,1,c,b,a  
3  
2  
1  
c  
b  
a

## Desafio – selecionar itens de um array.

O desafio a seguir transforma uma cadeia de caracteres em uma matriz utilizando **Split()**. Depois percorre o array com foreach com a condição que será exibido na saída do console apenas os itens da matriz que se iniciem com a letra **B**. Para isso utilizamos **StartsWith()**.

```
Editor do .NET
1  string orderStream = "B123,C234,A345,C15,B177,G3003,C235,B179";
2
3  string[] orderStreamArray = orderStream.Split(',');
4  foreach(string items in orderStreamArray){
5      if(items.StartsWith("B"))
6      {
7          Console.WriteLine($"{items}");
8      }
9  }
```

Saída

B123  
B177  
B179

## Desafio – invertendo palavras.

Escreva o código necessário para inverter as letras de cada palavra no lugar e exibir o resultado. Em outras palavras, não basta inverter todas as letras na variável **pangram**. Em vez disso, você precisará inverter apenas as letras de cada palavra, mas imprimir a palavra invertida na posição original na mensagem.

Se você tiver êxito, deverá ver a saída a seguir.

```
Saída
ehT kciuq nworb xof spmuj revo eht yzal god
```

### ❗ Importante

Esse é um desafio particularmente difícil. Você precisará combinar muitos dos conceitos aprendidos neste exercício, incluindo o uso de `Split()`, `ToCharArray()`, `Array.Reverse()` e `String.Join()`. Você também precisará criar várias matrizes e, pelo menos, uma instrução de iteração.

## Código 1

```
Editor do .NET
1  string pangram = "The quick brown fox jumps over the lazy dog";
2
3  char[] pangramArray = pangram.ToCharArray();
4  Array.Reverse(pangramArray);
5  string resultPangramArray = new string(pangramArray);
6
7  string[] itemsPangram = resultPangramArray.Split(' ');
8
9  Array.Reverse(itemsPangram);
10 string result = String.Join(" ", itemsPangram);
11 Console.WriteLine(result);
12
```

Esse código transforma a string **pangram** em uma matriz de char com **ToCharArray()** e logo em seguida inverte todo array com **Array.Reverse()**, depois cria uma nova **string** com esse resultado utilizando a palavra **new** e passando a array em questão como parametro.

O código cria um novo array com essa nova string com **Split( )** utilizando como delimitador “espaço”. Esse array é novamente invertido com **Array.Reverse()** e depois utilizamos o **Join()** passando o novo array como parametro para criar uma única cadeia de caracteres.

## Código 2

```
Editor do .NET
1  string pangram = "The quick brown fox jumps over the lazy dog";
2
3  string[] message = pangram.Split(' ');
4
5  string[] newMessage = new string[message.Length];
6
7  for (int i = 0; i < message.Length; i++)
8  {
9      char[] letters = message[i].ToCharArray();
10     Array.Reverse(letters);
11     newMessage[i] = new string(letters);
12 }
13
14 string result = String.Join(" ", newMessage);
15 Console.WriteLine(result);
16
```

O código em questão utiliza o **Split( )** para transformar a cadeia de caracteres em um array utilizando “espaço” como delimitador.

Depois cria um novo array que define o tamanho do array com o **Length** para iterar no loop **for**.

O loop **for** executará seu bloco de código até percorrer todo array.

Dentro do **for** a cada iteração no item do array ele será transformado em um novo array, invertido com **Array.Reverse( )** e será guardado em um outro array.

Ao final das iterações o array que recebeu os valores invertidos será passado como parametro para o **Join( )** criar uma única cadeia de caracteres na saída do console.

# Formatação de dados para apresentação.

De uma perspectiva de alto nível, os desenvolvedores de software estão preocupados com:

- a **entrada de dados**, incluindo dados digitados por um usuário de um teclado, usando seu mouse, um dispositivo ou por outro sistema de software por meio de uma solicitação de rede
- o **processamento de dados**, incluindo a lógica de decisão, manipulação de dados, execução de cálculos e assim por diante
- a **saída de dados**, incluindo apresentação para um usuário final por meio de uma mensagem de linha de comando, uma janela, uma página da Web ou salvando os dados processados em um arquivo, enviando-os para um serviço de rede e assim por diante

## Formatação Composta.

A *formatação composta* usa espaços reservados numerados dentro de uma cadeia de caracteres. Em tempo de execução, tudo dentro das chaves será resolvido para um valor que também é passado com base em sua posição.

```
Editor do .NET
1  string first = "Hello";
2  string second = "World";
3  string result = string.Format("{0} {1}!", first, second);
4  Console.WriteLine(result);

Saída
Hello World!
```

Exemplo de formatação composta usando um método **Format()** interno na palavra-chave do tipo de dados **string**.

1. A cadeia de caracteres literal "{0} {1}!" forma um modelo, partes das quais serão substituídas em tempo de execução.
2. O token **{0}** é substituído pelo primeiro argumento depois do modelo da cadeia de caracteres; em outras palavras, o valor da variável **first**.
3. O token **{1}** é substituído pelo segundo argumento depois do modelo da cadeia de caracteres; em outras palavras, o valor da variável **second**.

### 🚨 Observação

Você pode achar estranho começar com o número 0. Na verdade, isso é muito comum no desenvolvimento de software. Sempre que houver uma sequência de itens que podem ser identificados usando um número, a numeração normalmente começará em 0.



## Interpolação de cadeia de caracteres.

A *interpolação de cadeia de caracteres* é uma técnica mais recente que simplifica a formatação composta.

Em vez de usar um **token** numerado e incluir o valor literal ou nome da variável em uma lista de argumentos em **String.Format()** ou **Console.WriteLine()**, você pode apenas usar o nome da variável dentro das chaves.

Para que uma cadeia de caracteres seja interpolada, você deve prefixá-la com a diretiva \$.

Editor do .NET

```
1 string first = "Hello";
2 string second = "World";
3 Console.WriteLine($"{first} {second}!");
```

Saída

Hello World!

## Moeda de formatação.

No exemplo a seguir, o especificador de formato de moeda **:C** é usado para apresentar as variáveis **price** e **discount** como moeda.

Editor do .NET

```
1 decimal price = 123.45m;
2 int discount = 50;
3 Console.WriteLine($"Price: {price:C} (Save {discount:C})");
```

Saída

Price: ₱123.45 (Save ₱50.00)

O símbolo ₱ usado em vez do símbolo do dinheiro do seu país. Esse é um símbolo genérico usado para denotar a “moeda” independentemente do *tipo* de moeda. Você vê esse símbolo no Editor do .NET porque ele ignora seu local atual.

No entanto, se você executasse esse código em um computador nos EUA com o Idioma de Exibição do Windows definido como inglês, veria a seguinte saída.

Saída

Price: \$123.45 (Save \$50.00)

E se você executar o código (acima) em um computador na França que tem o Idioma de Exibição do Windows definido como francês? Nesse caso, você veria a seguinte saída.

```
Saída

Price: 123,45 € (Save 50,00 €)
```

Formatar valores para exibição podem ser consideradas uma linguagem e cultura específicas. O uso desses recursos de formatação de cadeia de caracteres dependem da *cultura* computacional. Nesse contexto, o termo “cultura” refere-se ao país e ao idioma do usuário final. O *código de cultura* é uma cadeia de cinco caracteres que os computadores usam para notificar o local e o idioma do usuário final para garantir que determinadas informações, como datas e moeda, possam ser devidamente apresentadas.

Por exemplo:

- O código de cultura de um falante de inglês nos EUA é **en-US**.
- O código de cultura de um falante de francês na França é **fr-FR**.
- O código de cultura de um falante de francês no Canadá é **fr-CA**.

A cultura afeta o sistema de escrita, o calendário usado, a ordem de classificação das cadeias de caracteres e a formatação de datas e números (como a formatação de moeda).

Verificar se o código funciona corretamente em todos os computadores, independentemente do país ou do idioma do usuário final. Esse processo é conhecido como *localização* (ou *globalização*).

## Formatar números.

Ao trabalhar com os dados numéricos, convém formatar o número para facilitar a leitura incluindo vírgulas para delinear milhares, milhões, bilhões e assim por diante.

O especificador de formato numérico **N** fará isso.

```
Editor do .NET

1 decimal measurement = 123456.78912m;
2 Console.WriteLine($"Measurement: {measurement:N} units");
```

```
Saída

Measurement: 123,456.79 units
```

Por padrão, o especificador de formato numérico **N** exibe apenas dois dígitos após o ponto decimal.

Se desejar mostrar mais precisão, adicione um número após o especificador **N**.

Editor do .NET

```
1 decimal measurement = 123456.78912m;  
2 Console.WriteLine($"Measurement: {measurement:N4} units");
```

Saída

Measurement: 123,456.7891 units

## Formatar percentuais.

Use o especificador de formato **P** para formatar percentuais. Adicione um número posteriormente para controlar o número de valores exibidos após o ponto decimal.

Editor do .NET

```
1 decimal tax = .36785m;  
2 Console.WriteLine($"Tax rate: {tax:P}");
```

Saída

Tax rate: 36.79 %

Ou

Editor do .NET

```
1 decimal tax = .36785m;  
2 Console.WriteLine($"Tax rate: {tax:P3}");
```

Saída

Tax rate: 36.785 %

## Interpolação de cadeia de caracteres.

```
Editor do .NET
1  int invoiceNumber = 1201;
2  decimal productMeasurement = 25.4568m;
3  decimal subtotal = 2750.00m;
4  decimal taxPercentage = .15825m;
5  decimal total = 3185.19m;
6
7  Console.WriteLine($"Invoice Number: {invoiceNumber}");
8  Console.WriteLine($"    Measurement: {productMeasurement:N3} mg");
9  Console.WriteLine($"    Sub Total: {subtotal:C}");
10 Console.WriteLine($"    Tax: {taxPercentage:P2}");
11 Console.WriteLine($"    Total Due: {total:C}");

Saída

Invoice Number: 1201
    Measurement: 25.457 mg
    Sub Total: ₺2,750.00
    Tax: 15.83 %
    Total Due: ₺3,185.19
```

Aqui temos diversas formatações em interpolação de cadeia de caracteres, incluindo especificadores de formato de moeda, percentual e números.

## Métodos internos.

Anteriormente, usamos o método **string.Format()** para executar a formatação composta.

Há muitos métodos semelhantes no tipo de dados string, bem como qualquer cadeia de caracteres literal ou variável do tipo String.

1. Métodos que adicionam espaços em branco para fins de formatação **PadLeft()**, **PadRight()**
2. Métodos que comparam duas cadeias de caracteres ou facilitam a comparação **Trim()**, **TrimStart()**, **TrimEnd()**, **GetHashCode()**, a propriedade **Length**
3. Métodos que ajudam a determinar o que há dentro de uma cadeia de caracteres ou até mesmo recuperar apenas uma parte da cadeia de caracteres **Contains()**, **StartsWith()**, **EndsWith()**, **Substring()**
4. Métodos que alteram o conteúdo da cadeia de caracteres substituindo, inserindo ou removendo partes **Replace()**, **Insert()**, **Remove()**
5. Métodos que transformam uma cadeia de caracteres em uma matriz de cadeias de caracteres ou de caracteres **Split()**, **ToCharArray()**

## Espaços em branco.

O método **PadLeft()** adicionará espaços em branco ao lado esquerdo da cadeia de caracteres para que o número total de caracteres seja igual ao argumento enviado por você. Para adicionar espaço ou caracteres ao lado direito da cadeia de caracteres, use o método **PadRight()**.

Nesse caso, queremos que o comprimento total da cadeia de caracteres seja 12 caracteres.

Editor do .NET

```
1 string input = "Pad this";
2 Console.WriteLine(input.PadLeft(12));
3 Console.WriteLine(input.PadRight(12));
```

Saída

```
    Pad this
Pad this
```

## Método sobrecarregado.

No C#, um *método sobrecarregado* é outra versão de um método com argumentos diferentes ou adicionais que modificam a funcionalidade do método ligeiramente.

Editor do .NET

```
1 string input = "Pad this";
2 Console.WriteLine(input.PadLeft(12, '-'));
3 Console.WriteLine(input.PadRight(12, '-'));
```

Saída

```
----Pad this
Pad this----
```

Essa é uma versão sobrecarregada dos métodos **PadLeft()** e **PadRight()**, preenchendo o espaço extra com o caractere de traço.

## Trabalhar com cadeias de caracteres preenchidas.

Vamos supor que trabalhamos para uma empresa de processamento de pagamentos, Geralmente, esses sistemas exigem que os dados sejam inseridos em colunas específicas.

Por exemplo, a ID do pagamento deve ser armazenada nas colunas 1 a 6, o nome do favorecido nas colunas 7 a 30 e o valor do pagamento nas colunas 31 a 40. Além disso, é importante que o Valor do Pagamento esteja alinhado à direita.

### Editor do .NET

```
1 string paymentID = "769";
2 string payeeName = "Mr. Stephen Ortega";
3 string paymentAmount = "$5,000.00";
4
5 var formattedLine = paymentID.PadRight(6);
6 formattedLine += payeeName.PadRight(24);
7 formattedLine += paymentAmount.PadLeft(10);
8
9 Console.WriteLine(formattedLine);
```

### Saída

```
769   Mr. Stephen Ortega      $5,000.00
```

O código expressa as regras citadas.

1. O pagamento terá 6 colunas adicionando um **PadRight(6)** a variável.
2. O nome do beneficiário terá 24 colunas “**PadRight(24)**” ou seja da coluna 7 até a coluna 30.
3. O valor do pagamento da coluna 31 a 40 alinhado a direita com **PadLeft(10)**.
4. Todos os valores são concatenados a variável **formattedLine**.

## Desafio.

O objetivo desse código é mesclará informações personalizadas dos clientes.

Para promover os produtos de investimento mais recentes da empresa, é preciso criar uma mensagem contendo informações, como o portfólio existente, e comparará seus retornos atuais com retornos projetados se o cliente precisar investir no uso dos nossos novos produtos.

Criar a saída formatada observando as variáveis com informações do cliente e produto.

saída:

Dear Mr. Jones,

As a customer of our Magic Yield offering we are excited to tell you about a new financial product that would dramatically increase your return.

Currently, you own 2,975,000.00 shares at a return of 12.75 %.

Our new product, Glorious Future offers a return of 13.13 %. Given your current volume, your potential profit would be ₺63,000,000.00.

Here's a quick comparison:

Magic Yield	12.75 %	₺55,000,000.00
-------------	---------	----------------

Glorious Future	13.13 %	₺63,000,000.00
-----------------	---------	----------------

```
//Informações do cliente e produto.
string customerName = "Mr. Jones";

string currentProduct = "Magic Yield";
int currentShares = 2975000;
decimal currentReturn = 0.1275m;
decimal currentProfit = 55000000.0m;

string newProduct = "Glorious Future";
decimal newReturn = 0.13125m;
decimal newProfit = 63000000.0m;

//msg
Console.WriteLine($"Dear {customerName},");
Console.WriteLine($"As a customer of our {currentProduct} offering we are excited to tell you about a new financial product that would dramatically increase your return.\n");
Console.WriteLine($"Currently, you own {currentShares:C} shares at a return of {currentReturn:P}.\n");
Console.WriteLine($"Our new product, {newProduct} offers a return de {newReturn:P}. Given your current volume, your potential profit would be {newProfit:C}.\n");

Console.WriteLine("Here's a quick comparison:\n");

string comparisonMessage = "";

comparisonMessage = currentProduct.PadRight(20);
comparisonMessage += String.Format($"{currentReturn:P}").PadRight(10);
//comparisonMessage += String.Format("{0:P}", currentReturn).PadRight(10);
comparisonMessage += String.Format($"{currentProfit:C}").PadRight(20);
//comparisonMessage += String.Format("{0:C}", currentProfit).PadRight(20);

comparisonMessage += "\n";

comparisonMessage += newProduct.PadRight(20);
comparisonMessage += String.Format($"{newReturn:P}").PadRight(10);
//comparisonMessage += String.Format("{0:P}", newReturn).PadRight(10);
comparisonMessage += String.Format($"{newProfit:C}").PadRight(20);
//comparisonMessage += String.Format("{0:C}", newProfit).PadRight(20);

Console.WriteLine(comparisonMessage);
```

saída:

```
Dear Mr. Jones,
As a customer of our Magic Yield offering we are excited to tell you about a new financial product that would dramatically increase your return.

Currently, you own 2,975,000.00 shares at a return of 12.75 %.

Our new product, Glorious Future offers a return of 13.13 %. Given your current volume, your potential profit would be ₺63,000,000.00.

Here's a quick comparison:

Magic Yield      12.75 %   ₺55,000,000.00
Glorious Future  13.13 %   ₺63,000,000.00
```

## Modificar o conteúdo de cadeia de caracteres.

Geralmente, os dados com os quais você precisará trabalhar serão provenientes de outros sistemas de software, como uma cadeia de caracteres. Em geral, elas são iniciadas em um formato inutilizável, contendo informações estranhas que tornam difícil a extração das informações importantes. Quando isso acontecer, você precisará de ferramentas e técnicas para analisar dados de cadeia de caracteres, isolar as informações necessárias e remover as informações de que não precisa.

### Métodos auxiliares **IndexOf()** e **Substring()** da cadeia de caracteres.

Nesses exemplos iremos utilizar o método **IndexOf()** e suas variantes para localizar a posição de uma cadeia de caracteres dentro de uma cadeia de caracteres maior.

Depois de localizar a posição use o método **Substring()** para retornar o restante da cadeia de caracteres após a posição.

```
Editor do .NET
1  string message = "Find what is (inside the parentheses)";
2
3  int openingPosition = message.IndexOf('(');
4  int closingPosition = message.IndexOf(')');
5
6  Console.WriteLine(openingPosition);
7  Console.WriteLine(closingPosition);
8
9  int length = closingPosition - openingPosition;
10 Console.WriteLine(message.Substring(openingPosition, length));
```

```
Saída
13
36
(inside the parentheses
```

Nesse caso, o índice do caractere ( é 13. Lembre-se de que esses valores são baseados em zero, portanto, é o 14º caractere na cadeia de caracteres. O índice do caractere ) é 36.

Agora que temos os dois índices utilizamos o método **Substring()** os definindo como limite para recuperar o valor.

O método **Substring()** precisa da posição inicial e do número de caracteres, ou comprimento, para recuperar. Portanto, calculamos o comprimento em uma variável temporária chamada **length** e a passamos com o valor **openingPosition** para recuperar a cadeia de caracteres dentro dos parênteses.



No entanto, a saída inclui o parêntese de abertura. Nessa situação em particular, isso não é desejado. Para corrigir isso, precisaremos atualizar o código **openingPosition += 1;**

Editor do .NET

```
1  string message = "Find what is (inside the parentheses)";
2
3  int openingPosition = message.IndexOf('(');
4  int closingPosition = message.IndexOf(')');
5
6  openingPosition += 1;
7
8  int length = closingPosition - openingPosition;
9  Console.WriteLine(message.Substring(openingPosition, length));
```

Saída

inside the parentheses

Ao aumentar a **openingPosition** em (1), você ignora o caractere de parêntese de abertura.

Se estivéssemos tentando localizar um valor iniciado após uma cadeia de caracteres mais longa, por exemplo, **<div>** usaríamos o comprimento dessa cadeia (5) de caracteres em vez de (1).

Editor do .NET

```
1  string message = "What is the value <span>between the tags</span>?";
2
3  int openingPosition = message.IndexOf("<span>");
4  int closingPosition = message.IndexOf("</span>");
5
6  openingPosition += 6;
7  int length = closingPosition - openingPosition;
8  Console.WriteLine(message.Substring(openingPosition, length));
```

Saída

between the tags

## Evitar valores mágicos.

As cadeias de caracteres codificadas como "<span>" na lista de códigos anterior são conhecidas como "cadeias de caracteres mágicas" e valores numéricos codificados como 6 são conhecidos como "números mágicos". Os valores "mágicos" são indesejáveis por vários motivos e você deve tentar evitá-los, se possível.

Se você errar a digitação uma vez como "<sapn>". O compilador não detectará isso em tempo de compilação. Além disso, se você alterar a cadeia de caracteres "<span>" para "<div>", mas esquecer de alterar o número 6, o código produzirá resultados indesejáveis.

Em vez disso, você deve usar uma constante com a palavra-chave **const**. Uma constante permite definir e inicializar uma variável cujo valor nunca pode ser alterado.

Editor do .NET

```
1  string message = "What is the value <span>between the tags</span>?";
2
3  const string openSpan = "<span>";
4  const string closeSpan = "</span>";
5
6  int openingPosition = message.IndexOf(openSpan);
7  int closingPosition = message.IndexOf(closeSpan);
8
9  openingPosition += openSpan.Length;
10
11 int length = closingPosition - openingPosition;
12 Console.WriteLine(message.Substring(openingPosition, length));
13
```

Saída

between the tags

## LastIndexOf()

Vamos escrever um código para recuperar o conteúdo dentro do **último** conjunto de parênteses.

```
string message = "(What if) I am (only interested) in the last (set of parentheses)?";  
int openingPosition = message.LastIndexOf('(');  
openingPosition += 1;  
int closingPosition = message.LastIndexOf(')');  
int length = closingPosition - openingPosition;  
Console.WriteLine(message.Substring(openingPosition, length));
```

Saída

set of parentheses

O segredo deste exemplo é o uso de **LastIndexOf()**, que usamos para obter as posições dos últimos parênteses de abertura e fechamento.

## Recuperar qualquer valor entre um ou mais conjuntos de parênteses.

Desta vez o **message** terá três conjuntos de parênteses e escreveremos o código para extrair qualquer texto dentro deles. Precisaremos adicionar uma instrução **while** para iterar pela cadeia de caracteres até que todos os conjuntos de parênteses sejam descobertos, extraídos e exibidos.

```
8 string message = "(What if) there are (more than) one (set of parentheses)?";  
9  
10 while (true)  
11 {  
12     int openingPosition = message.IndexOf('(');  
13     if (openingPosition == - 1) break;  
14  
15     openingPosition += 1;  
16  
17     int closingPosition = message.IndexOf(')');  
18     int length = closingPosition - openingPosition;  
19     Console.WriteLine(message.Substring(openingPosition, length));  
20  
21     message = message.Substring(closingPosition + 1);  
22 }  
23
```

Saída

What if  
more than  
set of parentheses

## Trabalhar com tipos diferentes de conjuntos de símbolos.

Desta vez, atualizaremos a cadeia de caracteres **message** adicionando tipos diferentes de símbolos, como colchetes e chaves. Dependemos de **IndexOfAny()** para fornecer uma matriz de caracteres que representa os símbolos de abertura. **IndexOfAny()** retornará a primeira correspondência encontrada na cadeia de caracteres.

Quando um símbolo for encontrado, precisaremos localizar seu símbolo de fechamento correspondente. Depois de fazer isso, o restante deve ser semelhante. Usaremos uma tática diferente em vez de modificar o valor original de **message**. Desta vez, usaremos a posição de fechamento da iteração anterior como a posição de abertura da iteração atual.

```
string message = "(What if) I have [different symbols] but every {open symbol} needs a [matching closing symbol]?";
char[] openSymbols = {'[', '{', '('};
int closingPosition = 0;

while (true)
{
    int openingPosition = message.IndexOfAny(openSymbols, closingPosition);

    if (openingPosition == -1) break;

    string currentSymbols = message.Substring(openingPosition, 1);

    char matchingSymbol = ' ';

    switch (currentSymbols)
    {
        case "[":
            matchingSymbol = ']';
            break;
        case "{":
            matchingSymbol = '}';
            break;
        case "(":
            matchingSymbol = ')';
            break;
    }

    openingPosition += 1;
    closingPosition = message.IndexOf(matchingSymbol, openingPosition);

    int length = closingPosition - openingPosition;
    Console.WriteLine(message.Substring(openingPosition, length));
}
```

Saída



```
What if
different symbols
open symbol
matching closing symbol
```

## Recapitulação

Abordamos muitos aspectos nessa unidade. Veja os fatores mais importantes a serem lembrados:

- **IndexOf()** informa a primeira posição de um caractere ou uma cadeia de caracteres dentro de outra cadeia de caracteres.
- **IndexOf()** retornará **-1** se não for possível encontrar uma correspondência.
- **Substring()** retorna apenas a parte especificada de uma cadeia de caracteres usando uma posição inicial e um comprimento opcional.
- **LastIndexOf()** retorna a última posição de um caractere ou uma cadeia de caracteres dentro de outra cadeia de caracteres.
- **IndexOfAny()** retorna a primeira posição de uma matriz de **char** que ocorre dentro de outra cadeia de caracteres.
- Geralmente, há mais de uma maneira de resolver um problema. Usamos duas técnicas diferentes para localizar todas as instâncias de um determinado caractere ou cadeia de caracteres.
- Evite valores mágicos codificados. Em vez disso, defina uma variável **const**. O valor de uma variável constante não pode ser alterado após a inicialização.

## Usar os métodos Remove() e Replace()

### Remove()

Editor do .NET

```
1 string data = "12345John Smith          5000 3 ";
2 string updatedData = data.Remove(5, 20);
3 Console.WriteLine(updatedData);
```

Saída

```
123455000 3
```

O código utiliza o **Remove()** para remover caracteres em locais específicos de uma cadeia de caracteres. Nesse caso da posição 5 até a 20.

O método **Remove()** funciona de forma semelhante ao método **Substring()**. Defina uma posição inicial e o comprimento para remover esses caracteres da cadeia de caracteres.

### Replace()

Usaremos o método **Replace()** para substituir um ou mais caracteres por um caractere diferente (ou nenhum caractere). O método **Replace()** é diferente dos outros métodos que usamos até agora, pois ele substituirá todas as instâncias dos caracteres especificados, e não apenas a primeira ou a última instância.

Editor do .NET

```
1 string message = "This--is--ex-amp-le--da-ta";
2
3 message = message.Replace("--", " ");
4 message = message.Replace("-", "");
5
6 Console.WriteLine(message);
```

Saída

```
This is example data
```

Aqui, usamos o método **Replace()** duas vezes. Na primeira vez, substituímos a cadeia de caracteres -- por um espaço vazio. Na segunda vez, substituímos a cadeia de caracteres - por uma cadeia de caracteres vazia, que remove completamente o caractere da cadeia de caracteres.

## Desafio.

Neste desafio, você trabalhará com uma cadeia de caracteres que contém um fragmento de HTML. Você extrairá dados do fragmento HTML, substituirá parte de seu conteúdo e removerá outras partes de seu conteúdo para obter a saída desejada.

Dado o ponto de partida na lista de códigos a seguir, você adicionará código para extrair, substituir e remover partes do valor **input** para produzir a saída desejada.

```
C# Copiar Executar

const string input = "<div><h2>Widgets &trade;</h2><span>5000</span></div>";

string quantity = "";
string output = "";

// Your work here

Console.WriteLine(quantity);
Console.WriteLine(output);
```

A lista a seguir é a saída desejada depois que você modificou o código no ponto de partida.

```
Saída Copiar

Quantity: 5000
Output: <h2>Widgets &reg;</h2><span>5000</span>
```

Você só pode adicionar código à lista de códigos do ponto de partida. Não altere as declarações da variável. Todo o seu trabalho deve ficar abaixo do comentário //

Your work here.

Você executará três operações na entrada usando as ferramentas e as técnicas aprendidas neste módulo.

## Operações

Defina a variável **quantity** como o valor extraído entre as marcas **<span>** e **</span>**.

Defina a variável **output** como o valor de entrada e, em seguida, remova as marcas **<div>** e **</div>**.

Substitua o caractere HTML **&trade;** por **&reg;** na variável **output**.

## Solução.

```
1  string input = "<div class='product'><h2>Widgets &trade;</h2><span>5000</span></div>";
2
3  string quantity = "";
4  string output = "";
5
6  // Your work here
7
8  const string spanTag = "<span>";
9
10 // Extract the quantity
11 int quantityStart = input.IndexOf(spanTag);
12 int quantityEnd = input.IndexOf("</span>");
13 quantityStart += spanTag.Length;
14 int quantityLength = quantityEnd - quantityStart;
15 quantity = input.Substring(quantityStart, quantityLength);
16
17 // Set output to input, replacing the trademark symbol with the registered trademark symbol
18 output = input.Replace("&trade;", "&reg;");
19
20 // Remove the opening <div> tag
21 int divStart = input.IndexOf("<div>");
22 int divEnd = input.IndexOf(">");
23 int divLength = divEnd - divStart;
24 divLength += 1;
25 output = output.Remove(divStart, divLength);
26
27 // Remove the closing <div> tag
28 int divCloseStart = output.IndexOf("</div>");
29 int divCloseEnd = output.IndexOf(">", divCloseStart);
30 int divCloseLength = divCloseEnd - divCloseStart;
31 divCloseLength += 1;
32 output = output.Remove(divCloseStart, divCloseLength);
33
34 Console.WriteLine($"Quantity: {quantity}");
35 Console.WriteLine($"Output: {output}");
```

Saída



Quantity: 5000  
Output: <h2>Widgets &reg;</h2><span>5000</span>