# Глава 5. Оператори и изрази

## В тази тема...

В настоящата тема ще се запознаем с **операторите в C#** и действията, които те извършват върху различните типове данни. Ще разясним приоритетите на операторите и ще разгледаме видовете оператори според броя на аргументите, които приемат и какви действия извършват. Във втората част на темата ще разгледаме **преобразуването на типове**, ще обясним кога и защо се налага да се извършва и как да работим с типове. В края на темата ще обърнем внимание на изразите и как да работим с тях. Най-накрая сме приготвили упражнения, за да затвърдим знанията си по материала от тази глава.

## Оператори

Във всички езици за програмиране се използват оператори, чрез които се извършват някакви действия върху данните. Нека разгледаме операторите в C# и да видим за какво служат и как се използват.

### Какво е оператор?

След като научихме как да декларираме и да задаваме стойности на про­менливи в [предходната глава](#_Глава_2._Примитивни), сега ще разгледаме как да извърш­ваме различни операции върху тях. За целта ще се запознаем с опера­торите.

**Операторите** позволяват обработка на прими­тивни типове данни и обекти. Те приемат като вход един или няколко операнда и връщат като резултат някаква стойност. Операторите в C# представляват **специални символи** (като например "+", ".", "^" и други) и извършат специфични преобра­зувания над един, два или три операнда. Пример за оператори в C# са знаците за събиране, изваждане, умножение и делене в математиката (+, - , \*, /) и операциите, които те извършват върху целите и реалните числа.

### Операторите в C#

Операторите в C# могат да бъдат разделени в няколко различни категории:

* Аритметични – също както в математиката, служат за извършване на прости математически операции.
* Оператори за присвояване – позволяват присвояването на стойност на променливите.
* Оператори за сравнение – дават възможност за сравнение на два литерала и/или променливи.
* Логически оператори – оператори за работа с булеви типове данни и булеви изрази.
* Побитови оператори – използват се за извършване на операции върху двоичното представяне на числови данни.
* Оператори за преобразуване на типовете – позволяват преобразу­ването на данни от един тип в друг.

#### Категории оператори

Следва списък с операторите, разделени по категории:

|  |  |
| --- | --- |
| **Категория** | **Оператори** |
| аритметични | -, +, \*, /, %, ++, -- |
| логически | &&, ||, !, ^ |
| побитови | &, |, ^, ~, <<, >> |
| за сравнение | ==, !=, >, <, >=, <= |
| за присвояване | =, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>= |
| съединяване на символни низове | + |
| за работа с типове | (type), as, is, typeof, sizeof |
| други | ., new, (), [], ?:, ?? |

#### Оператори според броя аргументи

Операторите могат да се разделят на типове според броя на аргументите, които приемат:

|  |  |
| --- | --- |
| **Тип оператор** | **Брой на аргументите (операндите)** |
| едноаргументни (unary) | приемат един аргумент |
| двуаргументни (binary) | приемат два аргумента |
| триаргументни (ternary) | приемат три аргумента |

Всички двуаргументни оператори в C# са ляво-асоциативни, т.е. изра­зите, в които участват се изчисляват от ляво на дясно, освен операторите за присвояване на стойности. Всички оператори за присвояване на стойности и условните оператори ?: и ?? са дясно-асоциативни (изчисляват се от дясно на ляво). Едноаргументните оператори нямат асоциативност.

Някои оператори в C# извършват различни операции, когато се приложат върху различен тип данни. Пример за това е **операторът** **+**. Когато се използва върху числени типове данни (int, long, float и др.), операторът извършва операцията **математическо събиране**. Когато обаче използваме оператора върху **символни низове**, той **слепва съдържанието** на двете про­менливи / литерали и връща новополучения низ.

#### Оператори – пример

Ето един пример за използване на оператори:

|  |
| --- |
| int a = 7 + 9;  Console.WriteLine(a); // 16  string firstName = "Dilyan";  string lastName = "Dimitrov";  // Do not forget the interval between them  string fullName = firstName + " " + lastName;  Console.WriteLine(fullName); // Dilyan Dimitrov |

Примерът показва как при използването на оператора + върху числа той връща числова стойност, а при използването му върху низове връща низ.

### Приоритет на операторите в C#

Някои оператори имат **приоритет** над други. Например, както е в математиката, умножението има приоритет пред събирането. Операторите с по-висок приоритет се изчисляват преди тези с по-нисък. Операторът () служи за промяна на приоритета на операторите и се изчислява пръв, също както в математиката.

В таблицата са показани приоритетите на операторите в C#:

|  |  |
| --- | --- |
| **Приоритет** | **Оператори** |
| най-висок  ...  най-нисък | (, ) |
| ++, -- (като постфикс), new, (type), typeof, sizeof |
| ++, -- (като префикс), +, - (едноаргументни), !, ~ |
| \*, /, % |
| + (свързване на низове) |
| +, - |
| <<, >> |
| <, >, <=, >=, is, as |
| ==, != |
| &, ^, | |
| && |
| || |
| ?:, ?? |
| =, \*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |= |

Операторите, намиращи се по-нагоре в таблицата, имат по-висок прио­ритет от тези, намиращи се след тях, и съответно имат предимство при изчисляването на даден израз. За да променим приоритета на даден оператор, може да използваме скоби.

Когато пишем по-сложни изрази или такива съдържащи повече оператори се препоръчва използването на скоби, за да се избегнат трудности при четене и разбиране на кода. Ето един пример:

|  |
| --- |
| // Ambiguous  x + y / 100  // Unambiguous, recommended  x + (y / 100) |

Първата операция, която се изпълнява в примера, е делението, защото то има **по-висок приоритет** от оператора за събиране. Въпреки това използването на скоби е добра идея, защото кодът става по-лесен за четене и възможността да се допусне грешка намалява.

### Аритметични оператори

Аритметичните оператори в C# +, -, \* са същите като в математика. Те извършват съответно събиране, изваждане и умножение върху числови стойности и резултатът е отново числова стойност.

Операторът за деление / има **различно действие върху цели и реални числа**. Когато се извършва деление на целочислен с целочислен тип (например int, long, sbyte, …), върнатият резултат е отново целочислен (без закръгляне, с отрязване на дробната част). Такова деление се нарича целочислено. Например при целочислено деление 7 / 3 = 2. **Целочислено деление на 0 не е позволено** и при опит да бъде извършено, се получава грешка по време на изпълнение на програ­мата DivideByZeroException. Остатъкът от целочислено делене на цели числа може да се получи чрез оператора %. Например 7 % 3 = 1, а -10 % 2 = 0.

При деление на две реални числа или на две числа, от които едното е реално, се извършва реално делене (не целочислено) и резултатът е реално число с цяла и дробна част. Например 5.0 / 2 = 2.5. При делене на реални числа е позволено да се дели на 0.0 и резултатът е съответно +∞, -∞ или NaN.

Операторът за увеличаване с единица (increment) ++добавя единица към стойността на променливата, а съответно операторът -- (decrement) изважда единица от стойността. Когато използваме операторите ++ и -- като **префикс** (поставяме ги непосредствено преди променливата), първо се пресмята новата стой­ност, а после се връща резултата, докато при използването на операто­рите като **постфикс** (поста­вяме оператора непосредствено след променли­вата) първо се връща ори­гиналната стойност на операнда, а после се добавя или изважда единица към нея.

#### Аритметични оператори – примери

Ето няколко примера за аритметични оператори и тяхното действие:

|  |
| --- |
| int squarePerimeter = 17;  double squareSide = squarePerimeter / 4.0;  double squareArea = squareSide \* squareSide;  Console.WriteLine(squareSide); // 4.25  Console.WriteLine(squareArea); // 18.0625  int a = 5;  int b = 4;  Console.WriteLine(a + b); // 9  Console.WriteLine(a + b++); // 9  Console.WriteLine(a + b); // 10  Console.WriteLine(a + (++b)); // 11  Console.WriteLine(a + b); // 11  Console.WriteLine(14 / a); // 2  Console.WriteLine(14 % a); // 4  int one = 1;  int zero = 0;  Console.WriteLinе(one / zero); // DivideByZeroException  double dMinusOne = -1.0;  double dZero = 0.0;  Console.WriteLine(dMinusOne / dZero); // -Infinity  Console.WriteLine(one / dZero); // Infinity |

### Логически оператори

Логическите оператори приемат булеви стойности и връщат булев резултат (true или false). Основните булеви оператори са "И" (&&), "ИЛИ" (||), изключващо "ИЛИ" (^) и логическо отрицание (!).

Следва таблица с логическите оператори в C# и операциите, които те извършват:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** | **y** | **!x** | **x && y** | **x || y** | **x ^ y** |
| true | true | false | true | true | false |
| true | false | false | false | true | true |
| false | true | true | false | true | true |
| false | false | true | false | false | false |

От таблицата, както и от следващия пример, става ясно, че логическото "И" (&&) връща истина, само тогава, когато и двете променливи съдържат истина. **Логическото "ИЛИ" (||)** връща истина, когато поне един от операндите е истина. Операторът за **логическо отрицание (!)** сменя стойността на аргумента. Например, ако операндът е имал стойност true и приложим оператор за отрицан ие, новата стойност ще бъде false. Операторът за отрицание е едноаргументен и се слага пред аргумента. **Изключващото "ИЛИ" (^)** връща резултат true, когато само един от двата операнда има стойност true. Ако двата операнда имат различни стойности, изключващото "ИЛИ" ще върне резултат true, ако имат еднакви стойности, ще върне false.

#### Логически оператори – пример

Следва пример за използване на логически оператори, който илюстрира тяхното действие:

|  |
| --- |
| bool a = true;  bool b = false;  Console.WriteLine(a && b); // False  Console.WriteLine(a || b); // True  Console.WriteLine(!b); // True  Console.WriteLine(b || true); // True  Console.WriteLine((5 > 7) ^ (a == b)); // False |

#### Закони на Де Морган

Логическите операции се подчиняват на законите на Де Морган от математическата логика:

|  |
| --- |
| !(a && b) == (!a || !b)  !(a || b) == (!a && !b) |

Първият закон твърди, че отрицанието на конюнкцията (логическо И) на две съждения е равна на дизюнкцията (логическо ИЛИ) на техните отри­цания.

Вторият закон твърди, че отрицанието на дизюнкцията на две съждения е равно на конюнкцията на техните отрицания.

### Оператор за съединяване на низове

Операторът + се използва за съединяване на символни низове (string). Той слепва два или повече низа и връща резултата като нов низ. Ако поне един от аргументите в израза е от тип string, и има други операнди, които не са от тип string, то те автоматично ще бъдат преобразувани към тип string.

#### Оператор за съединяване на низове – пример

Ето един пример, в който съединяваме няколко символни низа, както и стрингове с числа:

|  |
| --- |
| string csharp = "C#";  string dotnet = ".NET";  string csharpDotNet = csharp + dotnet;  Console.WriteLine(csharpDotNet); // C#.NET  string csharpDotNet4 = csharpDotNet + " " + 4;  Console.WriteLine(csharpDotNet4); // C#.NET 4 |

В примера инициализираме две променливи от тип string и им задаваме стойности. На третия и четвъртия ред съединяваме двата стринга и подаваме резул­тата на метода Console.WriteLine(), за да го отпечата на конзолата. На следващия ред съединяваме полученият низ с интервал и числото 4. Върнатия резултат записваме в променливата csharpDotNet4, който автоматично ще бъде преобразуван към тип string. На последния ред подаваме резултата за отпечатване.

|  |  |
| --- | --- |
|  | Конкатенацията (слепването на два низа) на стрингове е бавна операция и трябва да се използва внимателно. Препоръчва се използването на класа **StringBuilder** при нужда от итеративни (повтарящи се) операции върху сим­волни низове. |

В главата "[Символни низове](#_Глава_13._Символни)" ще обясним в детайли защо при операции над символни низове, изпълнени в цикъл, задължително трябва да се използва гореспоменатия клас StringBuilder.

### Побитови оператори

**Побитов оператор** (**bitwise operator**) означава оператор, който действа над двоичното пред­ставяне на числовите типове. В компютрите всички данни и в част­ност числовите данни се представят като поредица от нули и единици. За целта се използва **двоичната бройна система**. Например числото 55 в двоична бройна система се представя като 00110111.

Двоичното представяне на данните е удобно, тъй като нулата и единицата в електро­никата могат да се реализират чрез **логически схеми**, в които нулата се представя като "няма ток" или например с напрежение -5V, а единицата се представя като "има ток" или например с напрежение +5V.

Ще разгледаме в дълбочина двоичната бройна система в главата "[Бройни системи](#_Глава_8._Бройни_1)", а за момента можем да считаме, че числата в компют­рите се **представят като нули и единици** и че побитовите опера­тори служат за анализиране и промяна на точно тези нули и единици.

**Побитовите оператори** много **приличат на логическите**. Всъщност можем да си представим, че логическите и побитовите оператори извършат едно и също нещо, но върху различни типове данни.Логическите опера­тори работят над стойностите true и false (булеви стойности), докато побито­вите работят над числови стойности и се прилагат побитово над тяхното двоично представяне, т.е. работят върху битовете на числото (съставя­щите го цифри 0 и 1). Също както при логическите оператори, в C# има оператори за побитово "И" (&), побитово "ИЛИ" (|), побитово отрицание (~) и изключващо "ИЛИ" (^).

#### Побитови оператори и тяхното действие

Действието на побитовите оператори над двоичните цифри 0 и 1 е показано в следната таблица:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** | **y** | **~x** | **x & y** | **x | y** | **x ^ y** |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 |

Както виждаме, побитовите и логическите оператори си **приличат много**. Разликата в изписването на "И" и "ИЛИ" е, че при логическите оператори се пише двоен амперсанд (&&) и двойна вертикална черта (||), а при битовите – единични (& и |). Побитовият и логическият оператор за изключващо или е един и същ "^". За логическо отрицание се използва "!", докато за побитово отрицание (инвертиране) се използва "~".

В програмирането има **още два побитови оператора**, които нямат аналог при логическите. Това са побитовото изместване в ляво (<<) и побитовото изместване вдясно (>>). Използвани над числови стойности те преместват всички битове на стойността, съответно на ляво или надясно, като цифрите, излезли извън обхвата на числото, се губят и се заместват с 0.

Операторите за преместване се използват по следния начин: от ляво на оператора слагаме промен­ливата (операнда), над която ще извършим операцията, вдясно на оператора поставяме число, указващо с колко знака искаме да отместим битовете. Например 3 << 2 означава, че искаме да преместим **два пъти наляво битовете на числото 3**. Числото 3 представено в битове изглежда така: "0000 0011". Когато го преместим два пъти в ляво, неговата двоична стойност ще изглежда така: "0000 1100", а на тази поредица от битове отговаря числото 12. Ако се вгледаме в примера можем да забележим, че реално сме **умножили числото по 4**. Самото побитово преместване може да се представи като умножение (побитово премест­ване вляво) или делене (преместване в дясно) някаква степен на числото 2. Това явление е следствие от природата на двоичната бройна система. Пример за преместване надясно е 6 >> 2, което означава да преместим двоичното число "0000 0110" с две позиции надясно. Това означава, че ще изгубим двете най-десни цифри и ще допълним с две нули отляво. Резултатът е "0000 0001", т.е. числото 1.

#### Побитови оператори – пример

Ето един пример за работа с побитови оператори. Двоичното представяне на числата и резултатите от различните оператори е дадено в коментари:

|  |
| --- |
| byte a = 3; // 0000 0011 = 3  byte b = 5; // 0000 0101 = 5  Console.WriteLine(a | b); // 0000 0111 = 7  Console.WriteLine(a & b); // 0000 0001 = 1  Console.WriteLine(a ^ b); // 0000 0110 = 6  Console.WriteLine(~a & b); // 0000 0100 = 4  Console.WriteLine(a << 1); // 0000 0110 = 6  Console.WriteLine(a << 2); // 0000 1100 = 12  Console.WriteLine(a >> 1); // 0000 0001 = 1 |

В примера първо създаваме и инициализираме стойностите на две променливи a и b. След това отпечатваме на конзолата, резултатите от няколко побитови операции над двете променливи. Първата операция, която прилагаме, е "ИЛИ". От примера се вижда, че за всички позиции, на които е имало 1 в двоичното представяне на променливите a и b, има 1 и в резултата. Втората операция е "И". Резултатът от операцията съдържа 1 само в най-десния бит, защото двете променливи имат едновременно 1 само в най-десния си бит. Изключващото "ИЛИ" връща единици само на позициите, където a и b имат различни стойности на двоичните си битове. След това в примера е илюстрирана работата на логическото отрицание и побитовото преместване вляво и вдясно.

### Оператори за сравнение

Операторите за сравнение в C# се използват за сравняване на два или повече операнда. C# поддържа следните оператори за сравнение:

* по-голямо (>)
* по-малко (<)
* по-голямо или равно (>=)
* по-малко или равно (<=)
* равенство (==)
* различие (!=)

Всички оператори за сравнение в C# са двуаргументни (приемат два операнда), а върнатият от тях резултат е булев (true или false). Операторите за сравнение имат **по-малък приоритет** **от аритметичните**, но са с по-голям приоритет от операторите за присвояване на стойност.

#### Оператори за сравнение – пример

Следва пример, който демонстрира употребата на операторите за сравнение в C#:

|  |
| --- |
| int x = 10, y = 5;  Console.WriteLine("x > y : " + (x > y)); // True  Console.WriteLine("x < y : " + (x < y)); // False  Console.WriteLine("x >= y : " + (x >= y)); // True  Console.WriteLine("x <= y : " + (x <= y)); // False  Console.WriteLine("x == y : " + (x == y)); // False  Console.WriteLine("x != y : " + (x != y)); // True |

В примерната програма първо създаваме две променливи x и y и им присвояваме стойностите 10 и 5. На следващия ред отпечатваме на конзо­лата посредством метода Console.WriteLine() резултата от сравня­ването на двете променливи x и y посредством оператора >. Върнатият резултат е true, защото x има по-голяма стойност от y. Аналогично в следващите редове се отпечатват резултатите от останалите 5 оператора за сравнение между променливите x и y.

### Оператори за присвояване

Операторът за присвояване на стойност на променливите е "=" (символът равно). Синтаксисът, който се използва за присвояване на стойности, е следният:

|  |
| --- |
| операнд1 = литерал, израз или операнд2; |

#### Оператори за присвояване – пример

Ето един пример, в който използваме оператора за присвояване на стойност:

|  |
| --- |
| int x = 6;  string helloString = "Здравей стринг.";  int y = x; |

В горния пример присвояваме стойност 6 на променливата x. На втория ред присвояваме текстов литерал на променливата helloString, а на третия ред копираме стойността от променливата x в променливата y.

#### Каскадно присвояване

Операторът за присвояване може да се използва и **каскадно** (да се използва повече от веднъж в един и същ израз). В този случай присвоя­ванията се извършват последователно отдясно наляво. Ето един пример:

|  |
| --- |
| int x, y, z;  x = y = z = 25; |

На първия ред от примера създаваме три променливи, а на втория ред ги инициализираме със стойност 25.

|  |  |
| --- | --- |
|  | Операторът за присвояване в C# е "**=**", докато операто­рът за сравнение е "**==**". Размяната на двата оператора е честа причина за грешки при писането на код. Внимавайте да не объркате оператора за сравнение с оператора за присво­яване, тъй като те много си приличат. |

#### Комбинирани оператори за присвояване

Освен оператора за присвояване в C# има и **комбинирани оператори** за присвояване. Те спомагат за съкращаване на обема на кода чрез изписване на две операции заедно с един оператор: операция и присвояване. Комбинира­ните оператори имат следния синтаксис:

|  |
| --- |
| операнд1 оператор = операнд2; |

Горният израз е идентичен със следния:

|  |
| --- |
| операнд1 = операнд1 оператор операнд2; |

Ето един пример за комбиниран оператор за присвояване:

|  |
| --- |
| int x = 2;  int y = 4;  x \*= y; // Same as x = x \* y;  Console.WriteLine(x); // 8 |

Най-често използваните комбинирани оператори за присвояване са += (добавя стойността на операнд2 към операнд1), -= (изважда стойността на операнда в дясно от стойността на тази в ляво). Други комбинирани оператори за присвояване са \*=, /= и %=.

Следващият пример дава по-добра представа как работят комби­нираните оператори за присвояване:

|  |
| --- |
| int x = 6;  int y = 4;  Console.WriteLine(y \*= 2); // 8  int z = y = 3; // y=3 and z=3  Console.WriteLine(z); // 3  Console.WriteLine(x |= 1); // 7  Console.WriteLine(x += 3); // 10  Console.WriteLine(x /= 2); // 5 |

В примера първо създаваме променливите x и y и им присвояваме стойностите 6 и 4. На следващият ред принтираме на конзолата y, след като сме му присвоили нова стойност посредством оператора \*= и лите­рала 2. Резултатът от операцията е 8. По нататък в примера прилагаме други съставни оператори за присвояване и извеждаме получения резултат на конзолата.

### Условен оператор ?:

Условният оператор ?: използва **булевата стойност от един израз**, за да определи кой от други два израза да бъде пресметнат и върнат като резултат. Операторът работи над 3 операнда и за това се нарича тернарен. Символът "?" се поставя между първия и втория операнд, а ":" се поставя между втория и третия операнд. **Първият операнд** (или израз) трябва да е от булев тип, а **другите два операнда** трябва да са от един и същ тип, например числа или стрингове.

Синтаксисът на оператора ?: е следният:

|  |
| --- |
| операнд1 ? операнд2 : операнд3 |

Той работи така: ако операнд1 има стойност true, операторът връща като резултат операнд2. Иначе (ако операнд1 има стойност false), операторът връща резултат операнд3.

По време на изпълнение се пресмята **стойността на първия аргумент**. Ако той има стойност true, тогава се пресмята **втория** (среден) аргумент и той се връща като резултат. Обаче, ако пресметнатият резултат от първия аргумент е false, то тогава се пресмята **третият** (последният) аргумент и той се връща като резултат.

#### Условен оператор ?: – пример

Ето един пример за употребата на оператора "?:":

|  |
| --- |
| int a = 6;  int b = 4;  Console.WriteLine(a > b ? "a>b" : "b<=a"); // a>b  int num = a == b ? 1 : -1; // num will have value -1 |

### Други оператори

Досега разгледахме аритметичните оператори, логическите и побитовите оператори, оператора за конкатенация на символни низове, а също и условния оператор ?:. Освен тях в C# има още няколко оператора, на които си струва да обърнем внимание:

* Операторът за достъп "." (точка) се използва за достъп до член на променли­вите или методите на даден клас или обект. Пример за използването на оператора точка:

|  |
| --- |
| Console.WriteLine(DateTime.Now); // Prints the date + time |

* Квадратни скоби [] се използват за достъп до елементите на масив по индекс и затова се нарича още **индексатор**. Индексатори се ползват още за достъп до символите в даден стринг. Пример:

|  |
| --- |
| int[] arr = { 1, 2, 3 };  Console.WriteLine(arr[0]); // 1  string str = "Hello";  Console.WriteLine(str[1]); // e |

* Скоби () се използват за предефиниране приоритета на изпълнение на изразите и операторите. Вече видяхме как работят скобите.
* Операторът за преобразуване на типове (type) се използва за преобразуване на променлива от един тип в друг. Ще се запознаем с него в детайли в секцията "[Преобразуване на типовете](#_Преобразуване_на_типовете)".
* Операторът as също се използва за преобразуване на типове, но при невалидност на преобразуването връща null, а не изключение.
* Операторът new се използва за създаването и инициализирането на нови обекти. Ще се запознаем в детайли с него в главата "[Създаване и използване на обекти](#_Глава_11._Създаване)".
* Операторът is се използва за проверка дали даден обект е съвместим с даден тип.
* Операторът ?? е подобен на условния оператор ?:. Разликата е, че той се поставя между два операнда и връща левия операнд само ако той няма стойност null, в противен случай връща десния. Пример:

|  |
| --- |
| int? a = 5;  Console.WriteLine(a ?? -1); // 5  string name = null;  Console.WriteLine(name ?? "(no name)"); // (no name) |

#### Други оператори – примери

Ето няколко примера за операторите, които разгледахме в тази секция:

|  |
| --- |
| int a = 6;  int b = 3;  Console.WriteLine(a + b / 2); // 7  Console.WriteLine((a + b) / 2); // 4  string s = "Beer";  Console.WriteLine(s is string); // True  string notNullString = s;  string nullString = null;  Console.WriteLine(nullString ?? "Unspecified"); // Unspecified  Console.WriteLine(notNullString ?? "Specified"); // Beer |

## Преобразуване на типовете

По принцип операторите работят върху аргументи от **еднакъв тип данни**. Въпреки това в C# има голямо разнообразие от типове данни, от които можем да избираме най-подходящия за определена цел. За да извър­шим операция върху променливи от два различни типа данни, ни се налага да **преобразуваме** двата типа към един и същ. Преобразуването на типовете (typecasting) бива явно и неявно (implicit typecasting и explicit typecasting).

Всички изрази в езика C# имат тип. Този тип може да бъде изведен от структурата на израза и типовете, променливите и литералите, използвани в него. Възможно е да се напише израз, който е с неподходящ тип за конкретния контекст. В някои случаи това ще доведе до **грешка при компилацията** на програмата, но в други контекстът може да **приеме тип, който е сходен** или свързан с типа на израза. В този случай програмата извършва скрито **преобразуване на типове**.

Специфично преобразуване от тип S към тип T позволя на израза от тип S да се третира като израз от тип Т по време на изпълнението на прог­рамата. В някои случай това ще изисква проверка на валидността на преобразуването. Ето няколко примера:

* Преобразуване от тип object към тип string ще изисква проверка по време на изпълнение, за да потвърди, че стойността е наистина инстанция от тип string.
* Преобразуване от тип string към object не изисква проверка. Типът string е наследник на типа object и може да бъде преобразуван към базовия си клас без опасност от грешка или загуба на данни. На наследяването ще се спрем в детайли в главата "[Принципи на обектно-ориентираното програмиране](#_Глава_20._Принципи)".
* Преобразуване от тип int към long може да се извърши без проверка по време на изпълнението, защото няма опасност от загуба на данни, тъй като множеството от стойности на типа long е надмножество на стойностите на типа int.
* Преобразуване от тип double към long изисква преобразуване от 64-битова плаваща стойност към 64-битова целочислена. В зависимост от стойността, може да се получи **загуба на данни** и поради това е необходимо изрично преобразуване на типовете.

В C# не всички типове могат да бъдат преобразувани във всички други, а само към някои определени. За удобство ще групираме някой от възмож­ните преобразувания в C# според вида им в две категории:

* скрито (неявно) преобразуване;
* изрично (явно) преобразуване;
* преобразуване от и към string.

### Неявно (implicit) преобразуване на типове

Неявното (скритото) преобразуване на типове е възможно единствено, когато **няма възможност от загуба на данни** при преобразуването, т.е. когато конвертираме от тип с по-малък обхват към тип с по-голям обхват (например от int към long). За да направим неявно преобразуване, не е нужно да използваме какъвто и да е оператор и затова такова преобра­зуване се нарича още скрито (implicit). Преобразу­ването става автома­тично от компилатора, когато присвояваме стойност от по-малък обхват в променлива с по-голям обхват или когато в израза има няколко типа с различен обхват. Тогава преобразуването става към типа с най-голям обхват.

#### Неявно преобразуване на типове – пример

Ето един пример за неявно (implicit) преобразуване на типове:

|  |
| --- |
| int myInt = 5;  Console.WriteLine(myInt); // 5  long myLong = myInt;  Console.WriteLine(myLong); // 5  Console.WriteLine(myLong + myInt); // 10 |

В примера създаваме променлива myInt от тип int и присвояваме стойност 5. По-надолу създаваме променлива myLong от тип long и зада­ваме стойността, съдържаща се в myInt. Стойността запазена в myLong, автоматично се конвертира от тип int към тип long. Накрая в примера извеждаме резултата от събирането на двете променливи. Понеже променливите са от различен тип, те автоматично се преобразуват към типа с по-голям обхват, тоест към long и върнатият резултат, който се отпечатва на конзолата, отново е long. Всъщност подаденият параметър на метода Console.WriteLine() e от тип long, но вътре в метода той отново ще бъде конвертиран, този път към тип string, за да може да бъде отпечатан на конзолата. Това преобразувание се извършва чрез метода Long.ToString().

#### Възможни неявни преобразования

Ето някои от възможните неявни (implicit) преобразувания на примитивни типове в C#:

* sbyte → short, int, long, float, double, decimal;
* byte → short, ushort, int, uint, long, ulong, float, double, decimal;
* short → int, long, float, double, decimal;
* ushort → int, uint, long, ulong, float, double, decimal;
* char → ushort, int, uint, long, ulong, float, double, decimal (въпреки че char е символен тип, в някои случаи той може да се разглежда като число и има поведение на числов тип, дори може да участва в числови изрази);
* uint → long, ulong, float, double, decimal;
* int → long, float, double, decimal;
* long → float, double, decimal;
* ulong → float, double, decimal;
* float → double.

При преобразуването на типове от по-малък обхват към по-голям няма загуба на данни. Числовата стойност остава същата след преобразу­ването. Както във всяко правило, и тук има малко изключение. Когато преобразуваме тип int към тип float (32-битови стойности), разликата е, че int използва всичките си битове за представяне на едно целочислено число, докато float използва част от битовете си за представянето на плаващата запетая. Оттук следва, че е възможно при преобразуване от int към float да има загуба на точност, поради закръгляне. Същото се отнася и за преобразуването на 64-битовия long към 64-битовия double.

### Изрично (explicit) преобразуване на типове

Изричното преобразуване на типове (explicit typecasting) се използва винаги, когато **има вероятност за загуба на данни**. Когато конвертираме тип с плаваща запетая към цело­числен тип, винаги има загуба на данни, идваща от премахването на дробната част и е задължително използването на изрично преобразуване (например double към long). За да направим такова конвертиране, е нужно изрично да използваме оператора за преобразуване на данни (type). Възможно е да има загуба на данни също, когато конвертираме от тип с по-голям обхват към тип с по-малък (double към float или long към int).

#### Изрично преобразуване на типове – пример

Следният пример илюстрира употребата на изрично конвертиране на типовете и загуба на данни, която може да настъпи в някои случаи:

|  |
| --- |
| double myDouble = 5.1d;  Console.WriteLine(myDouble); // 5.1  long myLong = (long)myDouble;  Console.WriteLine(myLong); // 5    myDouble = 5e9d; // 5 \* 10^9  Console.WriteLine(myDouble); // 5000000000    int myInt = (int)myDouble;  Console.WriteLine(myInt); // -2147483648  Console.WriteLine(int.MinValue); // -2147483648 |

На първия ред от примера присвояваме стойността 5.1 на променливата myDouble. След като я преобразуваме (изрично), посредством оператора (long) към тип long и изкараме на конзолата променливата myLong, виждаме, че променливата е изгубила дробната си част, защото long e целочислен тип. След това присвояваме на реалната променлива с двойна точност myDouble стойност 5 милиарда. Накрая конвертираме myDouble към int посредством оператора (int) и отпечатваме променли­вата **myInt**. Резултатът e същия, както и когато отпечатаме int.MinValue, защото myDouble съдържа в себе си по-голяма стойност от обхвата на int.

|  |  |
| --- | --- |
|  | Не винаги е възможно да се предвиди каква ще бъде стойността на дадена промен­лива след препълване на обхвата й! Затова използвайте достатъчно големи типове и внимавайте при преминаване към "по-малък" тип. |

#### Загуба на данни при преобразуване на типовете

Ще дадем още един пример за загуба на данни при преобразуване на типове:

|  |
| --- |
| long myLong = long.MaxValue;  int myInt = (int)myLong;  Console.WriteLine(myLong); // 9223372036854775807  Console.WriteLine(myInt); // -1 |

Операторът за преобразуване може да се използва и при **неявно преобра­зуване по желание**. Това допринася за четимостта на кода, намалява шанса за грешки и се счита за добра практика от много програмисти.

Ето още няколко примера за преобразуване на типове:

|  |
| --- |
| float heightInMeters = 1.74f; // Explicit conversion  double maxHeight = heightInMeters; // Implicit  double minHeight = (double)heightInMeters; // Explicit  float actualHeight = (float)maxHeight; // Explicit  float maxHeightFloat = maxHeight; // Compilation error! |

В примера на последния ред имаме израз, който ще генерира грешка при компилирането. Това е така, защото се опитваме да конвертираме неявно от тип double към тип float, от което може да има загуба на данни. C# е строго типизиран език за програмиране и не позволява такъв вид прис­вояване на стойности.

#### Прихващане на грешки при преобразуване на типовете

Понякога е удобно вместо да получаваме грешен резултат при евентуално препълване при преминаване от по-голям към по-малък тип, да получим **уведомление** за проблема. Това става чрез ключовата дума checked, която включва уведомлението за препълване при целочислените типове:

|  |
| --- |
| double d = 5e9d; // 5 \* 10^9  Console.WriteLine(d); // 5000000000  int i = checked((int)d); // System.OverflowException  Console.WriteLine(i); |

При изпълнението на горния фрагмент от код се получава изключение (т.е. уведомление за грешка) OverflowException. Повече за изключени­ята и средствата за тяхното прихващане и обработка можете да прочетете в главата "[Обработка на изключения](#_Глава_12._Обработка)".

#### Възможни изрични преобразувания

Явните (изрични) преобразувания между числовите типове в езика C# са възможни между всяка двойка от следните типове:

|  |
| --- |
| sbyte, byte, short, ushort, char, int, uint, long, ulong, float, double, decimal |

При тези преобразувания **могат да се изгубят** както данни за големината на числото, така и информация за неговата точност (precision).

Забележете, че преобразуването към string и от string не е възможно да се извършва чрез преобразуване на типовете (typecasting).

### Преобразуване към символен низ

При необходимост можем да преобразуваме към низ, всеки отделен тип, включително и стойността null. Преобразуването на символни низове става **автоматично** винаги, когато използваме оператора за конкатенация (+) и някой от аргументите не е от тип низ. В този случай аргументът се преоб­разува към низ и операторът връща нов низ представляващ конкатена­цията на двата низа.

Друг начин да преобразуваме различни обекти към тип символен низ е като извикаме метода ТoString() на съответната променлива или стойност. Той е валиден за всички типове данни в .NET Framework. Дори извикването 3.ToString() е напълно валидно в C# и като резултат ще се върне низа "3".

#### Преобразуване към символен низ – пример

Нека разгледаме няколко примера за преобразуване на различни типове данни към символен низ:

|  |
| --- |
| int a = 5;  int b = 7;  string sum = "Sum=" + (a + b);  Console.WriteLine(sum);  String incorrect = "Sum=" + a + b;  Console.WriteLine(incorrect);  Console.WriteLine(  "Perimeter = " + 2 \* (a + b) + ". Area = " + (a \* b) + "."); |

Резултатът от изпълнението на примера е следният:

|  |
| --- |
| Sum=12  Sum=57  Perimeter = 24. Area = 35. |

От резултата се вижда, че долепването на число към символен низ връща като резултат символния низ, следван от текстовото представяне на числото. Забележете, че операторът "+" за залепване на низове може да предизвика неприятен ефект при събиране на числа, защото има **еднакъв приоритет** с оператора "+" за събиране. Освен ако изрично не променим приоритета на операциите чрез поставяне на скоби, те винаги се изпъл­няват отляво надясно.

Повече подробности по въпроса как да преобразуваме от и към string ще разгледаме в главата "[Вход и изход от конзолата](#_Глава_4._Вход)".

## Изрази

Голяма част от работата на една програма е пресмятането на изрази. Изразите представляват поредици от оператори, литерали и променливи, които се изчисляват до определена стойност от някакъв тип (число, сим­волен низ, обект или друг тип). Ето няколко примера за изрази:

|  |
| --- |
| int r = (150-20) / 2 + 5;  // Expression for calculation of the surface of the circle  double surface = Math.PI \* r \* r;  // Expression for calculation of the perimeter of the circle  double perimeter = 2 \* Math.PI \* r;  Console.WriteLine(r);  Console.WriteLine(surface);  Console.WriteLine(perimeter); |

В примера са дефинирани три израза. Първият израз пресмята радиуса на дадена окръжност. Вторият пресмята площта на окръжността, а послед­ният намира периметъра й. Ето какъв е резултатът при изпълнението на горния програмен фрагмент:

|  |
| --- |
| 70  15393,80400259  439,822971502571 |

Изчисляването на израз може да има и странични действия, защото изразът може да съдържа вградени оператори за присвояване, увелича­ване или намаляване на стойност (increment, decrement) и извикване на методи. Ето пример за такъв страничен ефект:

|  |
| --- |
| int a = 5;  int b = ++a;  Console.WriteLine(a); // 6  Console.WriteLine(b); // 6 |

При съставянето на изрази трябва да се имат предвид типовете данни и поведението на използваните оператори. Пренебрегването на тези особе­ности може да доведе до неочаквани резултати. Ето един прост пример:

|  |
| --- |
| double d = 1 / 2;  Console.WriteLine(d); // 0, not 0.5  double half = (double)1 / 2;  Console.WriteLine(half); // 0.5 |

В примера се използва израз, който разделя две цели числа и присвоява резултата на променлива от тип double. Резултатът за някои може да е неочакван, но това е защото игнорират факта, че операторът "/" за цели числа работи целочислено и резултатът е цяло число, получено чрез **отрязване на дробната част**.

От примера се вижда още, че ако искаме да извършим деление с резултат дробно число, е необходимо да преобразуваме до float или double поне един от операндите. При този сценарий делението вече не е целочислено и резултатът е коректен.

Друг интересен пример е **делението на 0**. Повечето програмисти си мислят, че делението на 0 е невалидна операция и предизвиква грешка по време на изпълнение (exception), но това всъщност е вярно само за целочисленото деление на 0. Ето един пример, който показва, че при нецелочислено деление на 0 се получава резултат Infinity или NaN:

|  |
| --- |
| int num = 1;  double denum = 0; // The value is 0.0 (real number)  int zeroInt = (int) denum; // The value is 0 (integer number)  Console.WriteLine(num / denum); // Infinity  Console.WriteLine(denum / denum); // NaN  Console.WriteLine(zeroInt / zeroInt); // DivideByZeroException |

При работата с изрази е важно да се **използват скоби** винаги, когато има и най-малко **съмнение за приоритетите** на използваните операции. Ето един пример, който показва колко са полезни скобите:

|  |
| --- |
| double incorrect = (double)((1 + 2) / 4);  Console.WriteLine(incorrect); // 0  double correct = ((double)(1 + 2)) / 4;  Console.WriteLine(correct); // 0.75  Console.WriteLine(**"2 + 3 = "** + 2 + 3); // 2 + 3 = 23  Console.WriteLine(**"2 + 3 = "** + (2 + 3)); // 2 + 3 = 5 |

## Упражнения

1. Напишете израз, който да проверява дали дадено цяло число е четно или нечетно.
2. Напишете булев израз, който да проверява дали дадено цяло число се дели на 5 и на 7 без остатък.
3. Напишете израз, който да проверява дали третата цифра (отдясно на ляво) на дадено цяло число е 7.
4. Напишете израз, който да проверява дали третият бит на дадено число е 1 или 0.
5. Напишете израз, който изчислява площта на трапец по дадени a, b и h.
6. Напишете програма, която за подадени от потребителя дължина и височина на право­ъгълник пресмята и отпечатва на конзолата неговия периметър и лице.
7. Силата на гравитационното поле на Луната е приблизително 17% от това на Земята. Напишете програма, която да изчислява тежестта на човек на Луната, по дадената тежест на Земята.
8. Напишете програма, която проверява дали дадена точка О (x, y) е вътре в окръжността К ((0,0), 5). Пояснение: точката (0,0) е център на окръжността, а радиусът й е 5.
9. Напишете програма, която проверява дали дадена точка О (x, y) е вътре в окръжността К ((0,0), 5) и едновременно с това извън право­ъгълника ((-1, 1), (5, 5)). Пояснение: правоъгълникът е зададен чрез координатите на долния си ляв и горния си десен ъгъл.
10. Напишете програма, която приема за вход четирицифрено число във формат *abcd* (например числото 2011) и след това извършва следните действия върху него:
    * Пресмята сбора от цифрите на числото (за нашия пример 2+0+1+1 = 4).
    * Разпечатва на конзолата цифрите в обратен ред: *dcba* (за нашия пример резултатът е 1102).
    * Поставя последната цифра на първо място: *dabc* (за нашия пример резултатът е 1201).
    * Разменя мястото на втората и третата цифра: *acbd* (за нашия пример резултатът е 2101).
11. Дадено е число n и позиция p. Напишете поредица от операции, които да отпечатат стойността на бита на позиция p от числото n (0 или 1). Пример: n=35, p=5 -> 1. Още един пример: n=35, p=6 -> 0.
12. Напишете булев израз, който проверява дали битът на позиция p на цялото число v има стойност 1. Пример v=5, p=1 -> false.
13. Дадено е число n, стойност v (v = 0 или 1) и позиция p. Напишете поредица от операции, които да променят стойността на n, така че битът на позиция p да има стойност v. Пример n=35, p=5, v=0 -> n=3. Още един пример: n=35, p=2, v=1 -> n=39.
14. Напишете програма, която проверява дали дадено число n (1 < n < 100) е просто (т.е. се дели без остатък само на себе си и на единица).
15. \* Напишете програма, която разменя стойностите на битовете на позиции 3, 4 и 5 с битовете на позиции 24, 25 и 26 на дадено цяло положително число.
16. \* Напишете програма, която разменя битовете на позиции {p, p+1, …, p+k-1) с битовете на позиции {q, q+1, …, q+k-1} на дадено цяло положително число.

## Решения и упътвания

1. Вземете остатъка от деленето на числото на 2 и проверете дали е 0 или 1 (съответно числото е четно или нечетно). Използвайте оператора % за пресмятане на остатък от целочислено деление.
2. Ползвайте логическо "И" (оператора &&) и операцията % за остатък при деление. Можете да решите задачата и чрез само една проверка – за деление на 35 (помислете защо).
3. Разделете числото на 100 и го запишете в нова променлива. Нея разделете на 10 и вземете остатъка. Остатъкът от делението на 10 е третата цифра от първоначалното число. Проверете равна ли е на 7.
4. Използвайте побитово "И" върху числото и число, което има 1 само в третия си бит (т.е. числото 8, ако броенето на битовете започне от 0). Ако върнатият резултат е различен от 0, то третия бит е 1.
5. Формула за лице на трапец: S = ((a + b) / 2) \* h.
6. Потърсете в Интернет как се въвеждат цели числа от конзолата и използвайте формулата за лице на правоъгълник. Ако се затруднявате погледнете упътването на следващата задача.
7. Използвайте следния код, за да прочетете число от конзолата, след което го умножете по 0.17 и го отпечатайте:

|  |
| --- |
| Console.Write("Enter number: ");  int number = Convert.ToInt32(Console.ReadLine()); |

1. Използвайте питагоровата теорема a2 + b2 = c2. За да е точката вътре в кръга, то x\*x + y\*y следва да е по-малко или равно на 25.
2. Използвайте кода от предходната задача и добавете проверка за правоъгълника. Една точка е вътре в даден правоъгълник със стени успоредни на координатните оси, когато е вдясно от лявата му стена, вляво от дясната му стена, надолу от горната му стена и нагоре от долната му стена.
3. За да вземете отделните цифри на числото, можете да го делите на 10 и да взимате остатъка при деление на 10 последователно 4 пъти.
4. Ползвайте побитови операции:

|  |
| --- |
| int n = 35; // 00100011  int p = 6;  int i = 1; // 00000001  int mask = i << p; // Move the 1st bit left by p positions  // If i & mask are positive then the p-th bit of n is 1  Console.WriteLine((n & mask) != 0 ? 1 : 0); |

1. Задачата е аналогична на предната.
2. Ползвайте побитови операции, по аналогия със задача 11. Можете да нулирате бита на позиция p в числото n по следния начин:

|  |
| --- |
| n = n & (~(1 << p)); |

Можете да установите в единица бита на позиция p в числото n по следния начин:

|  |
| --- |
| n = n | (1 << p); |

Помислете как можете да комбинирате тези две упътвания.

1. Прочетете за цикли в Интернет. Използвайте цикъл и проверете числото за делимост на всички числа от 1 до корен квадратен от числото. В конкретната задача, тъй като ограничението е само до 100, можете предварително да намерите простите числа от 1 до 100 и да направите проверки дали даденото число n e равно на някое от тях.
2. За решението на тази задача използвайте комбинация от задачите за взимане и установяване на бит на определена позиция.
3. Използвайте предната задача и прочетете в интернет как се работи с цикли и масиви (в които да запишете битовете)