2024-09-19

# Operators

## Arithmetic Operators

Variables can be used to do math. All the usual arithmetic operations are available in C#:

| Operation | C# Operator | C# Expression |
| --- | --- | --- |
| Addition | + | myVar + 7 |
| Subtraction | - | myVar - 7 |
| Multiplication | \* | myVar \* 7 |
| Division | / | myVar / 7 |
| Remainder (a.k.a. modulo) | % | myVar % 7 |

Note: the “remainder” or “modulo” operator represents the remainder after doing integer division between its two operands.  
For example, 44 % 7 = 2 because 44/7 = 6 when rounded down, then do 7\*6 to get 42 and 44 - 42 = 2.

## Arithmetic and variables

* The result of an arithmetic expression (like those shown in the table) is a numeric value
  + For example, the C# expression 3 \* 4 has the value 12, which is int data
* A numeric value can be assigned to a variable of the same type, just like a literal: int myVar = 3 \* 4; initializes the variable myVar to contain the value 12
* A numeric-type variable can be used in an arithmetic expression
* When a variable is used in an arithmetic expression, its current value is read, and the math is done on that value
* Example:
* int a = 4;  
  int b = a + 5;  
  a = b \* 2;
  + To execute the second line of the code, the computer will first evaluate the expression on the right side of the = sign. It reads the value of the variable a, which is 4, and then computes the result of 4 + 5, which is 9. Then, it assigns this value to the variable b (remember assignment goes right to left).
  + To execute the third line of code, the computer first evaluates the expression on the right side of the = sign, which means reading the value of b to use in the arithmetic operation. b contains 9, so the expression is 9 \* 2, which evaluates to 18. Then it assigns the value 18 to the variable a, which now contains 18 instead of 4.
* A variable can appear on both sides of the = sign, like this:
* int myVar = 4;  
  myVar = myVar \* 2;
* This looks like a paradox because myVar is assigned to itself, but it has a clear meaning because assignment is evaluated right to left. When executing the second line of code, the computer evaluates the right side of the = before doing the assignment. So it first reads the current (“old”) value of myVar, which is 4, and computes 4 \* 2 to get the value 8. Then, it assigns the new value to myVar, overwriting its old value.

## Compound assignment operators

* The pattern of “compute an expression with a variable, then assign the result to that variable” is common, so there are shortcuts for doing it
* The **compound assignment operators** change the value of a variable by adding, subtracting, etc. from its current value, equivalent to an assignment statement that has the value on both sides:

| Statement | Equivalent |
| --- | --- |
| x += 2; | x = x + 2; |
| x -= 2; | x = x - 2; |
| x \*= 2; | x = x \* 2; |
| x /= 2; | x = x / 2; |
| x %= 2; | x = x % 2; |

## Increment and Decrement Operators

### Increment and decrement basics

* In C#, we have already seen multiple ways to add 1 to a numeric variable:

int myVar = 1;  
myVar = myVar + 1;  
myVar += 1

These two lines of code have the same effect; the += operator is “shorthand” for “add and assign”

* The **increment operator**, ++, is an even shorter way to add 1 to a variable. It can be used in two ways:

myVar++;  
++myVar;

* Writing ++ after the name of the variable is called a **postfix increment**, while writing ++ before the name of the variable is called a **prefix increment**. They both have the same effect on the variable: its value increases by 1.
* Similarly, there are multiple ways to subtract 1 from a numeric variable:

int myVar = 10;  
myVar = myVar - 1;  
myVar -= 1;

* The **decrement operator**, --, is a shortcut for subtracting 1 from a variable, and is used just like the increment operator:

myVar--;  
--myVar;

* To summarize, the increment and decrement operators both have a prefix and postfix version:

|  | Increment | Decrement |
| --- | --- | --- |
| Postfix | myVar++ | myVar-- |
| Prefix | ++myVar | --myVar |

### Difference between prefix and postfix

* The prefix and postfix versions of the increment and decrement operators both have the same effect on the variable: Its value increases or decreases by 1
* The difference between prefix and postfix is whether the “old” or “new” value of the variable is *returned* by the expression
* With postfix increment/decrement, the operator returns the value of the variable, *then* increases/decreases it by 1
* This means the value of the increment/decrement expression is the *old* value of the variable, before it was incremented/decremented
* Consider this example:

int a = 1;  
Console.WriteLine(a++);  
Console.WriteLine(a--);

* The expression a++ returns the current value of a, which is 1, to be used in Console.WriteLine. *Then* it increments a by 1, giving it a new value of 2. Thus, the first Console.WriteLine displays “1” on the screen.
* The expression a-- returns the current value of a, which is 2, to be used in Console.WriteLine, and *then* decrements a by 1. Thus, the second Console.WriteLine displays “2” on the screen.
* With prefix increment/decrement, the operator increases/decreases the value of the variable by 1, *then* returns its value
* This means the value of the increment/decrement expression is the *new* value of the variable, after the increment/decrement
* Consider the same code, but with prefix instead of postfix operators:

int a = 1;  
Console.WriteLine(++a);  
Console.WriteLine(--a);

* The expression ++a increments a by 1, then returns the value of a for use in Console.WriteLine. Thus, the first Console.WriteLine displays “2” on the screen.
* The expression --a decrements a by 1, then returns the value of a for use in Console.WriteLine. Thus, the second Console.WriteLine displays “1” on the screen.

### Using increment/decrement in expressions

* The ++ and -- operators have higher precedence than the other math operators, so if you use them in an expression they will get executed first
* The “result” of the operator, i.e. the value that will be used in the rest of the math expression, depends on whether it is the prefix or postfix increment/decrement operator: The prefix operator returns the variable’s new value, while the postfix operator returns the variable’s old value
* Consider these examples:

int a = 1;  
int b = a++;  
int c = ++a \* 2 + 4;  
int d = a-- + 1;

* The variable b gets the value 1, because a++ returns the “old” value of a (1) and then increments a to 2
* In the expression ++a \* 2 + 4, the operator ++a executes first, and it returns the new value of a, which is 3. Then the multiplication executes (3 \* 2, which is 6), then the addition (6 + 4, which is 10). Thus c gets the value 10.
* In thee expression a-- + 1, the operator a-- executes first, and it returns the *old* value of a, which is 3 (even though a is now 2). Then the addition executes, so d gets the value 4.

## Arithmetic on Mixed Data Types

* The math operators (+, -, \*, /) are defined separately for each data type: There is an int version of + that adds ints, a float version of + that adds floats, etc.
* Each operator expects to get two values of the same type on each side, and produces a result of that same type. For example, 2.25 + 3.25 uses the double version of +, which adds the two double values to produce a double-type result, 5.5.
* Most operators have the same effect regardless of their type, except for /
* The int/short/long version of / does **integer division**, which returns only the quotient and drops the remainder: In the statement int result = 21 / 5;, the variable result gets the value 4, because is 4 with a remainder of 1. If you want the fractional part, you need to use the floating-point version (for float, double, and decimal): double fracDiv = 21.0 / 5.0; will initialize fracDiv to 4.2.

### Implicit conversions in math

* If the two operands/arguments to a math operator are not the same type, they must become the same type – one must be converted
* C# will first try implicit conversion to “promote” a less-precise or smaller value to a more precise, larger type
* Example: with the expression double fracDiv = 21 / 2.4;
  + Operand types are int and double
  + int is smaller/less-precise than double
  + 21 gets implicitly converted to 21.0, a double value
  + Now the operands are both double type, so the double version of the / operator gets executed
  + The result is 8.75, a double value, which gets assigned to the variable fracDiv
* Implicit conversion also happens in assignment statements, which happen *after* the math expression is computed
* Example: with the expression double fraction = 21 / 5;
  + Operand types are int and int
  + Since they match, the int version of / gets executed
  + The result is 4, an int value
  + Now this value is assigned to the variable fraction, which is double type
  + The int value is implicitly converted to the double value 4.0, and fraction is assigned the value 4.0

### Explicit conversions in math

* If the operands are int type, the int version of / will get called, even if you assign the result to a double
* You can “force” floating-point division by explicitly converting one operand to double or float
* Example:
* int numCookies = 21;  
  int numPeople = 6;  
  double share = (double) numCookies / numPeople;
* Without the cast, share would get the value 3.0 because numCookies and numPeople are both int type (just like the fraction example above). With the cast, numCookies is converted to the value 21.0 (a double), which means the operands are no longer the same type. This will cause numPeople to be implicitly converted to double in order to make them match, and the double version of / will get called to evaluate 21.0 / 6.0. The result is 3.5, so share gets assigned 3.5.
* You might also *need* a cast to ensure the operands are the same type, if implicit conversion does not work
* Example:
* decimal price = 3.89;  
  double shares = 47.75;  
  decimal total = price \* (decimal) shares;
* In this code, double cannot be implicitly converted to decimal, and decimal cannot be implicitly converted to double, so the multiplication price \* shares would produce a compile error. We need an explicit cast to decimal to make both operands the same type (decimal).

## Order of Operations

* Math operations in C# follow PEMDAS from math class: Parentheses, Exponents, Multiplication, Division, Addition, Subtraction
  + Multiplication/division are evaluated together, as are addition/subtraction
  + Expressions are evaluated left-to-right
  + Example: int x = 4 = 10 \* 3 - 21 / 2 - (3 + 3);
    - Parentheses: (3 + 3) is evaluated, returns 6
    - Multiplication/Division: 10 \* 3 is evaluated to produce 30, then 21 / 2 is evaluated to produce 10 (left-to-right)
    - Addition/Subtraction: 4 + 30 - 10 - 6 is evaluated, result is 18
* Cast operator is higher priority than all binary operators
  + Example: double share = (double) numCookies / numPeople;
    - Cast operator is evaluated first, converts numCookies to a double
    - Division is evaluated next, but operand types do not match
    - numPeople is implicitly converted to double to make operand types match
    - Then division is evaluated, result is 21.0 / 6.0 = 3.5
* Parentheses always increase priority, even with casts
  + An expression in parentheses gets evaluated before the cast “next to” it
  + Example:
  + int a = 5, b = 4;  
    double result = (double) (a / b);
  + The expression in parentheses gets evaluated first, then the result has the (double) cast applied to it. That means a / b is evaluated to produce 1, since a and b are both int type, and then that result is cast to a double, producing 1.0.