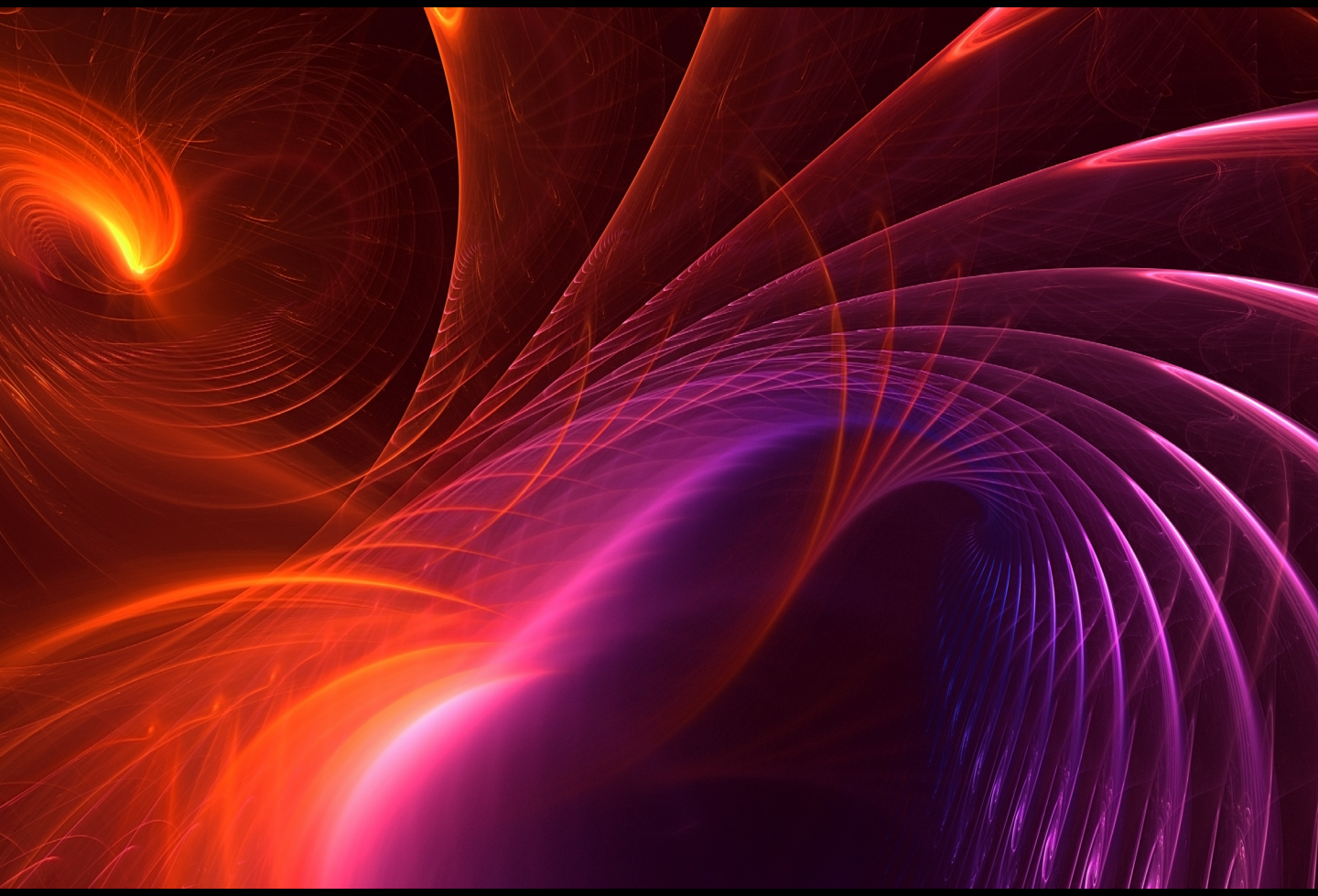
C# 13 Programming Essentials



.NET 9 Edition

Payload publishing

C# 13 Programming Essentials

.NET 9 Edition

C# 13 Programming Essentials - .NET 9 Edition

ISBN-13: 978-1-965764-01-5

Rev: 1.0

https://www.payloadbooks.com

Chapter 8

8. C# Operators and Expressions

This chapter covers using operators to create expressions when programming in C#, including arithmetic and assignment operators. Other topics covered include operator precedence, logical operators, and the ternary operator.

In the previous chapter, we used variables and constants in C# and described the different variable and constant types. However, being able to create constants and variables is only part of the story. The next step is using these variables and constants in C# code. The primary method for working with the data stored in constants and variables is in the form of expressions. This chapter will examine C# expressions and operators in detail.

8.1 What is an expression?

The most fundamental expression consists of an operator, two operands, and an assignment. The following is an example of an expression:

```
int theResult = 1 + 2;
```

In the above example, the (+) operator adds two operands (1 and 2) together. The assignment operator (=) subsequently assigns the result of the addition to an integer variable named *theResult*. The operands could have easily been variables or constants (or a mixture of each) instead of the actual numerical values used in the example.

In the remainder of this chapter, we will examine the various operators available in C#.

8.2 The basic assignment operator

We have already looked at the most basic of assignment operators, the = operator. This assignment operator assigns the result of an expression to a variable. In essence, the = assignment operator takes two operands. The left-hand operand is the variable to which a value is to be assigned, and the right-hand operand is the value to be assigned. The right-hand operand is, more often than not, an expression that performs some arithmetic or logical evaluation. The following examples are all valid uses of the assignment operator:

```
int x;
int y = 15;
int z = 5;

x = 10;  // Assigns the value 10 to a variable named x
x = y + z; // Assigns the result of variable y added to variable z to variable x
x = y;   // Assigns the value of variable y to variable x
```

Assignment operators may also be chained to assign the same value to multiple variables. For example, the following code assigns the value 20 to the *x*, *y*, and *z* variables:

```
int x, y, z;

x = y = z = 20;
```

8.3 C# arithmetic operators

C# provides a range of operators to create mathematical expressions. These operators primarily fall into the category of binary operators in that they take two operands. The exception is the unary negative operator (-), which indicates that a value is negative rather than positive. This contrasts with the subtraction operator (-), which takes two operands (i.e., one value to be subtracted from another). For example:

```
int y = -10; // Unary - operator used to assign -10 to a variable named y
int z = 2;

int x = y - z; // Subtraction operator. Subtracts z from y
```

The following table lists the primary C# arithmetic operators:

Operator	Description
– (unary)	Negates the value of a variable or expression
*	Multiplication
/	Division
+	Addition
–	Subtraction
%	Modulo

Table 8-1

Note that multiple operators may be used in a single expression, for example:

```
int y = 10; // Unary - operator used to assign -10 to a variable named y
int z = 2;

int x = y * 10 + z - 5 / 4;
```

While the above code is perfectly valid, it is essential to be aware that C# does not evaluate the expression from left to right or right to left but instead in an order specified by the precedence of the various operators that conform to basic mathematical principles. Operator precedence is an important topic to understand since it impacts the result of a calculation and will be covered in detail in the next section.

8.4 C# operator precedence

C# uses the same operator order concept as in basic mathematics. For example, we probably all learned from our school days that the answer to the following calculation is 210, not 300:

```
int x;

x = 10 + 20 * 10;
Console.WriteLine(x);
```

When the code runs, the generated result will be 210. This is a direct result of operator precedence. C# knows the same rules we learned at school that tell it which order operators should be evaluated in an expression. For example, C# correctly considers the multiplication operator (*) to be of higher precedence than the addition (+) operator.

Fortunately, the precedence built into C# can be overridden by surrounding the lower priority section of an expression with parentheses (another common concept in basic mathematics). For example:

```
int x;

x = (10 + 20) * 10;
```

In the above example, the expression fragment enclosed in parentheses is evaluated before the higher precedence multiplication, resulting in a value of 300.

The following table outlines the C# operator precedence order from highest precedence to lowest:

Precedence	Operators
Highest	+, -, !, ~, ++x, –x, (T)x
	* / %
	+ –
	<< >>
	< > <= >= is as
	== !=
	&
	^
	|
	&&
	||
	:?
Lowest	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Table 8-2

It should come as no surprise that the assignment operators have the lowest precedence since you would only want to assign the result of an expression once that expression has been thoroughly evaluated. Don't worry about memorizing the above table. Most programmers use parentheses to evaluate their expressions in the desired order.

8.5 Compound assignment operators

C# provides several operators to combine an assignment with a mathematical or logical operation. These are primarily used when performing an evaluation where the result is to be stored in one of the operands. For example, one might write an expression as follows:

```
x = x + y;
```

The above expression adds the value contained in variable *x* to the value contained in variable *y* and stores the result in variable *x*. This can be simplified using the addition compound assignment operator (+=):

```
int x = 10;
int y = 20;

x += y;
```

The above expression performs the same task as x = x + y but saves the programmer some typing. This is yet another feature that C# has inherited from the C programming language. Numerous compound assignment operators are available in C#. The most frequently used are outlined in the following table:

Operator	Description
x += y	Add x to y and place the result in x
x -= y	Subtract y from x and place the result in x
x *= y	Multiply x by y and place the result in x
x /= y	Divide x by y and place the result in x
x %= y	Perform Modulo on x and y and place the result in x
x &= y	Assign to x the result of logical AND operation on x and y
x |= y	Assign to x the result of logical OR operation on x and y
x ^= y	Assign to x the result of logical Exclusive OR on x and y

Table 8-3

8.6 Increment and decrement operators

Another helpful shortcut involves the C# increment and decrement operators. As with the compound assignment operators described in the previous section, consider the following C# code fragment:

```
x = x + 1; // Increase value of variable x by 1
x = x - 1; // Decrease value of variable y by 1
```

These expressions increment and decrement the value of *x* by 1. Instead of using this approach, it is quicker to use the ++ and -- operators. The following examples perform the same tasks as the examples above:

```
x++; // Increment x by 1
x--; // Decrement x by 1
```

These operators can be placed either before or after the variable name. If the operator is placed before the variable name, the increment or decrement is performed before any other operations are performed on the variable. For example, in the following code, *x* is incremented before it is assigned to *y*, leaving *y* with a value of 10:

```
int x = 9;
int y;

y = ++x;
```

In the following example, the value of *x* (9) is assigned to variable *y* before the decrement is performed. Consequently, after the expression is evaluated, the value of *y* will be 9, and the value of *x* will be 8:

```
int x = 9;
int y;

y = x--;
```

8.7 Comparison operators

In addition to mathematical and assignment operators, C# includes a set of logical operators that help perform comparisons. These operators all return a Boolean (bool) true or false result depending on the comparison result and are binary in that they work with two operands.

Comparison operators are most frequently used in constructing program control flow. For example, an *if* statement may be built based on whether one value matches another:

```
int x = 9;
```

```
int y = 9;

if (x == y)
      Console.WriteLine("x is equal to y");
```

Output:

```
x is equal to y
```

The result of a comparison may also be stored in a bool variable. For example, the following code will result in a true value being stored in the variable named *result*:

```
bool result;
int x = 10;
int y = 20;

result = x < y;
```

Clearly, 10 is less than 20, resulting in a true evaluation of the x < y expression. The following table lists the full set of C# comparison operators:

Operator	Description
x == y	Returns true if x is equal to y
x > y	Returns true if x is greater than y
x >= y	Returns true if x is greater than or equal to y
x < y	Returns true if x is less than y
x <= y	Returns true if x is less than or equal to y
x != y	Returns true if x is not equal to y

Table 8-4

8.8 Boolean logical operators

Another set of operators that return Boolean true and false values is the C# Boolean logical operators. These operators both return Boolean results and take Boolean values as operands. The key operators are NOT (!), AND (&&), OR (||), and XOR (^).

The NOT (!) operator inverts the current value of a Boolean variable or the result of an expression. For example, if a variable named *flag* is currently true, prefixing the variable with a ! character will invert the value to false:

```
bool flag = true; //variable is true
bool secondFlag;

secondFlag = !flag; // secondFlag set to false
```

The OR (||) operator returns true if one of its two operands evaluates to true. Otherwise, it returns false. For example, the following code evaluates to true because at least one of the expressions on either side of the OR operator is true:

```
if ((10 < 20) || (20 < 10))
      Console.WriteLine("Expression is true");
```

Output:

```
Expression is true
```

The AND (&&) operator returns true only if both operands evaluate to true. The following example will return false because only one of the two operand expressions evaluates to true:

```
int x = 10;
int y = 20;

if ((x < 20) && (y < 10))
      Console.WriteLine("Expression is true");
else
      Console.WriteLine("Expression is false");
```

Output:

```
Expression is false
```

The XOR (^) operator returns true if one (and only one) of the two operands evaluates to true. For example, the following example will return true since only one operator evaluates to be true:

```
int x = 10;
int y = 20;

if ((x < 20) ^ (y < 10))
      Console.WriteLine("Expression is true");
else
      Console.WriteLine("Expression is false");
```

Output:

```
Expression is true
```

If both operands were evaluated to be true or both were false the expression would return false.

8.9 Range and index operators

The C# range (..) and index from end (^) operators allow you to declare value ranges, which are invaluable when working with collections such as arrays. The chapter titled *"Accessing and Sorting C# Array Elements"* will cover both of these operators.

8.10 The ternary operator

The C# ternary operator provides a shortcut way of making decisions. The syntax of the ternary operator is as follows:

```
[condition] ? [true expression] : [false expression]
```

The way this works is that [condition] is replaced with an expression that will return either true or false. The expression that replaces the [true expression] is evaluated if the result is true. Conversely, the [false expression] is evaluated if the result is false. Let's see this in action:

```
int x = 10;
int y = 20;

Console.WriteLine( x > y ? x : y );
```

Output:

```
20
```

The true and false expressions above simply output the largest value. In practice, this can be any valid expression. The following modification, for example, specifies a string value to be displayed for each outcome:

```
int x = 10;
int y = 20;

Console.WriteLine(x > y ? "x is larger" : "y is larger");
```

Output:

```
y is larger
```

8.11 Null-coalescing operators

The null-coalescing operator (??) allows a default value to be used if an operand has a null value. The syntax for using this operator is as follows:

```
<operand> ?? <default operand>
```

If the left operand is not null, then the operand's value is returned; otherwise, the expression returns the default operand value.

The following example will output text that reads "Welcome back, Customer" because the *customerName* variable is set to null:

```
string customerName = null;
string recipient = customerName ?? "Customer";

Console.WriteLine($"Welcome back, {recipient}");
```

Output:

```
Welcome back, Customer
```

C# also includes the null-coalescing assignment operator (??=), the syntax for which is as follows:

```
<operand1> ??= <operand2>
```

In this case, the value of operand2 will be assigned to operand1 only if *operand1* is null. Otherwise, *operand1* will remain unchanged.

In the following example, the value initially assigned to *customerName* remains unchanged since it does not contain a null value:

```
string customerName = "David";
customerName ??= "Customer";

Console.WriteLine($"Welcome back, {customerName}");
```

Output:

```
Welcome back, David
```

8.12 Bitwise Operators

As previously discussed, computer processors work in binary. These are essentially streams of ones and zeros, each one referred to as a bit. Bits are formed into groups of 8 to form bytes. As such, it is not surprising that we, as programmers, will occasionally end up working at this level in our code. To facilitate this requirement, C# provides a range of *bit operators*.

Those familiar with bitwise operators in other languages such as C, C++, and Java will find nothing new in this