CHAPTER 4
Operators

An *operator* is a symbol (such as =, +, or >) that causes C# to take an action. That action might be an assignment of a value to a variable, the addition of two values, a comparison of two values, and so forth.

In the previous chapter, you saw the assignment operator used. The single equals sign (=) is used to assign a value to a variable; in this case, the value 15 to the variable `myVariable`:

```
myVariable = 15;
```

C# has many different operators that you'll learn about in this chapter. There's a full set of mathematical operators, and a related set of operators just for incrementing and decrementing in integral values by one, which actually are quite useful for controlling loops, as you'll see in Chapter 5. There are also operators available for comparing two values, which are used in the branching statements, as I'll demonstrate in the next chapter.

The Assignment Operator (=)

The assignment operator causes the operand on the left side of the operator to have its value changed to whatever is on the right side of the operator. The following expression assigns the value 15 to myVariable:

```
myVariable = 15;
```

The assignment operator also allows you to *chain* assignments, assigning the same value to multiple variables, as follows:

```
myOtherVariable = myVariable = 15;
```

The previous statement assigns 15 to `myVariable`, and then also assigns the value (15) to `myOtherVariable`. This works because the statement:

```
myVariable = 15;
```

is an expression; it evaluates to the value assigned. That is, the expression:

```
myVariable = 15;
```

itself evaluates to 15, and it is this value (15) that is then assigned to `myOtherVariable`.

> It is important not to confuse the assignment operator (=) with the equality, or equals, operator (==), which has two equals signs and is described later in the chapter. The assignment operator does not test for equality; it assigns a value.

Mathematical Operators

C# uses five mathematical operators: four for standard calculations and one to return the remainder when dividing integers. The following sections consider the use of these operators.

Simple Arithmetical Operators (+, -, *, /)

C# offers four operators for simple arithmetic: the addition (+), subtraction (-), multiplication (*), and division (/) operators work as you might expect, with the possible exception of integer division.

When you divide two integers, C# divides like a child in the third grade: it throws away any fractional remainder. Thus, dividing 17 by 4 returns a value of 4 (with C# discarding the remainder of 1).

This limitation is specific to *integer* division. If you do not want the fractional part thrown away, you can use one of the types that support decimal values, such as float or double. Division between two floats (using the / operator) returns a decimal answer. Integer and floating-point division is illustrated in Example 4-1.

Example 4-1. Integer and float division

```
using System;
public class Tester
{
 public static void Main( )
 {
 int smallInt = 5;
 int largeInt = 12;
 int intQuotient;
 intQuotient = largeInt / smallInt;
 Console.WriteLine("Dividing integers. {0} / {1} = {2}",
 largeInt, smallInt, intQuotient);

 float smallFloat = 5;
 float largeFloat = 12;
 float FloatQuotient;
 FloatQuotient = largeFloat / smallFloat;
```

Example 4-1. Integer and float division (continued)

```
 Console.WriteLine("Dividing floats. {0} / {1} = {2}",
 largeFloat, smallFloat, FloatQuotient);

 }
}
Output:
Dividing integers. 12 / 5 = 2
Dividing floats. 12 / 5 = 2.4
```

The modulus Operator (%)

C# provides a special operator, modulus (%), to retrieve the remainder from integer division. For example, the statement `17%4` returns `1` (the remainder after integer division).

You read that statement as, “Seventeen modulo four equals 1.”

Example 4-2 demonstrates the effect of division on integers, floats, doubles, and decimals.

Example 4-2. Modulus operator

```
using System;
class Values
{
   static void Main( )
   {
      int firstInt, secondInt;
      float firstFloat, secondFloat;
      double firstDouble, secondDouble;
      decimal firstDecimal, secondDecimal;

      firstInt = 17;
      secondInt = 4;
      firstFloat = 17;
      secondFloat = 4;
      firstDouble = 17;
      secondDouble = 4;
      firstDecimal = 17;
      secondDecimal = 4;
      Console.WriteLine( "Integer:\t{0}\nfloat:\t\t{1}",
      firstInt / secondInt, firstFloat / secondFloat );
      Console.WriteLine( "double:\t\t{0}\ndecimal:\t{1}",
      firstDouble / secondDouble, firstDecimal / secondDecimal );
      Console.WriteLine( "\nRemainder(modulus) from integer division:\t{0}",
      firstInt % secondInt );

   }
}
```

Writeline Control Characters

Consider this line from Example 4-2:

```
Console.WriteLine("Integer:\t{0}\nfloat:\t\t{1}\n",
 firstInt/secondInt, firstFloat/secondFloat);
```

It begins with a call to `Console.Writeline( )`, passing in this partial string:

```
"Integer:\t{0}\n
```

This will print the characters `Integer:` followed by a tab (`\t`), the first parameter (`{0}`), and a newline character (`\n`).

The next string snippet:

```
float:\t\t{1}\n
```

is very similar. It prints `float:`, followed by two tabs (to ensure alignment), the contents of the second parameter (`{1}`), and then another newline. Notice the subsequent line, as well:

```
Console.WriteLine( "\nRemainder(modulus) from integer
division:\t{0}",  firstInt%secondInt);
```

This time, the string begins with a newline character, which causes a line to be skipped just before the string `Modulus`: is printed. You can see this effect in the output.

The output looks like this:

```
Integer:            4
float:              4.25
double:             4.25
decimal:            4.25

Remainder(modulus) from integer division:        1
```

The modulus operator is more than a curiosity; it greatly simplifies finding every nth value, as you'll see in Chapter 5.

Increment and Decrement Operators

A common requirement is to add a value to a variable, subtract a value from a variable, or otherwise change the mathematical value, and then to assign that new value back to the original variable. C# provides several operators for these calculations.

Calculate and Reassign Operators

Suppose you want to increment the `mySalary` variable by 5,000. You can do this by writing:

```
mySalary = mySalary + 5000;
```

In simple arithmetic, this would make no sense, but in C#, this line means, "Add 5,000 to the value in mySalary, and assign the sum back to mySalary." Thus, after this operation completes, mySalary will have been incremented by 5,000. You can perform this kind of assignment with any mathematical operator:

```
mySalary = mySalary * 5000;
mySalary = mySalary - 5000;
```

and so forth.

The need to perform this kind of manipulation is so common that C# includes special operators for self-assignment. Among these operators are `+=`, `-=`, `*=`, `/=`, and `%=`, which, respectively, combine addition, subtraction, multiplication, division, and modulus, with self-assignment. Thus, you can write the previous examples as:

```
mySalary += 5000;
mySalary *= 5000;
mySalary -= 5000;
```

These three instructions, respectively, increment mySalary by 5,000, multiply mySalary by 5,000, and subtract 5,000 from the mySalary variable.

Increment or Decrement by 1

Because incrementing and decrementing by exactly 1 is a very common need, C# provides two additional special operators for these purposes: increment (`++`) and decrement (`--`).

Thus, if you want to increment the variable `myAge` by 1, you can write:

```
myAge++;
```

This is equivalent to writing:

```
myAge += 1;
```

The Prefix and Postfix Operators

To complicate matters further, you might want to increment a variable and assign the results to a second variable:

```
resultingValue = originalValue++;
```

The question arises: do you want to assign before you increment the value or after? In other words, if `originalValue` starts out with the value 10, do you want to end with both `resultingValue` and `originalValue` equal to 11, or do you want `resultingValue` to be equal to 10 (the original value) and `originalValue` to be equal to 11?

C# offers two specialized ways to use the increment and decrement operators: prefix and postfix. The way you use the `++` operator determines the order in which the increment/decrement and assignment take place. The semantics of the prefix increment operator is "increment the original value and then assign the incremented value to result" while the semantics of the postfix increment operator is "assign the original value to result, and then increment original."

To use the prefix operator to increment, place the ++ symbol before the variable name; to use the postfix operator to increment, place the ++ symbol after the variable name:

```
result = ++original; // prefix
result = original++; // postfix
```

It is important to understand the different effects of prefix and postfix, as illustrated in Example 4-3. Note the output.

Example 4-3. Prefix and postfix operators

```
using System;
class Values
{
   static void Main( )
   {
      int original = 10;
      int result;

      // increment then assign
      result = ++original;
      Console.WriteLine( "After prefix: {0}, {1}", original, result );

      // assign then increment
      result = original++;
      Console.WriteLine( "After postfix: {0}, {1}", original, result );
   }
}
```

The output looks like this:

```
After prefix: 11, 11
After postfix: 12, 11
```

The prefix and postfix operators can be applied, with the same logic, to the decrement operators, as shown in Example 4-4. Again, note the output.

Example 4-4. Decrementing prefix and postfix

```
using System;
class Values
{
   static void Main( )
   {
      int original = 10;
      int result;

      // increment then assign
      result = --original;
      Console.WriteLine( "After prefix: {0}, {1}", original,
      result );
```

Example 4-4. Decrementing prefix and postfix (continued)

```
      // assign then increment
      result = original--;
      Console.WriteLine( "After postfix: {0}, {1}",
      original, result );
   }
}
```

The output looks like this:

```
    After prefix: 9, 9
    After postfix: 8, 9
```

Relational Operators

Relational operators compare two values and then return a Boolean value (true or false). The greater than operator (>), for example, returns true if the value on the left of the operator is greater than the value on the right. Thus, 5>2 returns the value true, while 2>5 returns the value false.

The relational operators for C# are shown in Table 4-1. This table assumes two variables: bigValue and smallValue, in which bigValue has been assigned the value 100, and smallValue the value 50.

Table 4-1. C# relational operators (assumes bigValue = 100 and smallValue = 50)

Name	Operator	Given this statement	The expression evaluates to
Equals	==	bigValue == 100 bigValue == 80	True False
Not Equals	!=	bigValue != 100 bigValue != 80	False True
Greater than	>	bigValue > smallValue	True
Greater than or equal to	>=	bigValue >= smallValue smallValue >= bigValue	True False
Less than	<	bigValue < smallValue	False
Less than or equal to	<=	smallValue <= bigValue bigValue <= smallValue	True False

Each of these relational operators acts as you might expect. Notice that most of these operators are composed of two characters. For example, the greater than or equal to operator (>=) is created with the greater than symbol (>) and the equals sign (=). Notice also that the equals operator is created with two equals signs (==) because the single equals sign alone (=) is reserved for the assignment operator.

It is not uncommon to confuse the assignment operator (=) with the equals operator (==). Just remember that the latter has two equals signs, and the former only one.

The C# equals operator (==) tests for equality between the objects on either side of the operator. This operator evaluates to a Boolean value (true or false). Thus, the statement:

```
myX == 5;
```

evaluates to true if and only if the `myX` variable has a value of 5.

Use of Logical Operators with Conditionals

As you program, you'll often want to test whether a condition is true; for example, using the `if` statement, which you'll see in the next chapter. Often you will want to test whether two conditions are both true, only one is true, or neither is true. C# provides a set of logical operators for this, shown in Table 4-2.

Table 4-2. Logical operators

Name	Operator	Given this statement	The expression evaluates to	Logic
And	&&	`(x == 3) && (y == 7)`	False	Both must be true.
Or	||	`(x == 3) || (y == 7)`	True	Either or both must be true.
Not	!	`! (x == 3)`	True	Expression must be false.

The examples in this table assume two variables, `x` and `y`, in which `x` has the value 5 and `y` has the value 7.

The *and* operator tests whether two statements are both true. The first line in Table 4-2 includes an example that illustrates the use of the and operator:

```
(x == 3) && (y == 7)
```

The entire expression evaluates false because one side (`x == 3`) is false. (Remember that x has the value 5 and y has the value 7.)

With the *or* operator, either or both sides must be true; the expression is false only if both sides are false. So, in the case of the example in Table 4-2:

```
(x == 3) || (y == 7)
```

the entire expression evaluates true because one side (`y==7`) is true.

With a *not* operator, the statement is true if the expression is false, and vice versa. So, in the accompanying example:

```
! (x == 3)
```

the entire expression is true because the tested expression (x==3) is false. (The logic is: “it is true that it is not true that x is equal to 3.”)

The Conditional Operator

Although most operators are unary (they require one term, such as myValue++) or binary (they require two terms, such as a+b), there is one *ternary* operator, which requires three terms: the conditional operator (?:):

```
cond-expr ? expression1 : expression2
```

This operator evaluates a *conditional* expression (an expression that returns a value of type bool) and then invokes either *expression1* if the value returned from the conditional expression is true, or *expression2* if the value returned is false. The logic is: “if this is true, do the first; otherwise do the second.” Example 4-5 illustrates this concept.

Example 4-5. The ternary operator

```
using System;
class Values
{
   static void Main( )
   {
      int valueOne = 10;
      int valueTwo = 20;

      int maxValue = valueOne > valueTwo ? valueOne : valueTwo;

      Console.WriteLine( "ValueOne: {0}, valueTwo: {1}, maxValue: {2}",
      valueOne, valueTwo, maxValue );

   }
}
```

The output looks like this:

```
ValueOne: 10, valueTwo: 20, maxValue: 20
```

In Example 4-5, the ternary operator is being used to test whether valueOne is greater than valueTwo. If so, the value of valueOne is assigned to the integer variable maxValue; otherwise, the value of valueTwo is assigned to maxValue.

Operator Precedence

The compiler must know the order in which to evaluate a series of operators. For example, if I write:

```
myVariable = 5 + 7 * 3;
```

there are three operators for the compiler to evaluate (=, +, and *). It could, for example, operate left to right, which would assign the value 5 to myVariable, then add 7 to the 5 (12) and multiply by 3 (36)—but of course, then it would throw that 36 away. This is clearly not what is intended.

The rules of precedence tell the compiler which operators to evaluate first. As is the case in algebra, multiplication has higher precedence than addition, so 5+7*3 is equal to 26 rather than 36. Both addition and multiplication have higher precedence than assignment, so the compiler will do the math and then assign the result (26) to myVariable only after the math is completed.

In C#, parentheses are also used to change the order of precedence much as they are in algebra. Thus, you can change the result by writing:

```
myVariable = (5+7) * 3;
```

Grouping the elements of the assignment in this way causes the compiler to add 5+7, multiply the result by 3, and then assign that value (36) to myVariable.

Table 4-3 summarizes operator precedence in C#, using x and y as possible terms to be operated upon.[*]

Table 4-3. Precedence

Category	Operators
Primary	(x) x.y x->y f(x) a[x] x++ x-- new typeof sizeof checked unchecked stackalloc
Unary	+ - ! ~ ++x --x (T)x *x &x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >= is as
Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	?:
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

[*] This table includes operators that are so esoteric as to be beyond the scope of this book. For a fuller explanation of each, please see *Programming C#*, Fourth Edition, by Jesse Liberty (O'Reilly, 2005).

The operators are listed in precedence order according to the category in which they fit. That is, the primary operators (such as x++) are evaluated before the unary operators (such as !). Multiplication is evaluated before addition.

In some complex equations, you might need to nest parentheses to ensure the proper order of operations. For example, assume I want to know how many seconds my family wastes each morning. The adults spend 20 minutes over coffee each morning and 10 minutes reading the newspaper. The children waste 30 minutes dawdling and 10 minutes arguing.

Here's my algorithm:

```
(((minDrinkingCoffee + minReadingNewspaper )* numAdults ) +
((minDawdling + minArguing) * numChildren)) * secondsPerMinute.
```

An *algorithm* is a well-defined series of steps to accomplish a task.

Although this works, it is hard to read and hard to get right. It's much easier to use interim variables:

```
wastedByEachAdult = minDrinkingCoffee + minReadingNewspaper;
wastedByAllAdults = wastedByEachAdult * numAdults;
wastedByEachKid = minDawdling + minArguing;
wastedByAllKids = wastedByEachKid * numChildren;
wastedByFamily = wastedByAllAdults + wastedByAllKids;
totalSeconds = wastedByFamily * 60;
```

The latter example uses many more interim variables, but it is far easier to read, understand, and (most importantly) debug. As you step through this program in your debugger, you can see the interim values and make sure they are correct. See Chapter 9 for more information.

Summary

- An operator is a symbol that causes C# to take an action.
- The assignment operator (=) assigns a value to an object or variable.
- C# includes four simple arithmetic operators: +, -, *, and /, and numerous variations such as +=, which increments a variable on the left side of the operator by the value on the right side.
- When you divide integers, C# discards any fractional remainder.
- The modulus operator (%) returns the remainder from integer division.
- C# includes numerous special operators such as the self-increment (++) and self-decrement (--) operators.

- To increment a value before assigning it, you use the prefix operator (++x); to increment the value after assigning it, use the postfix operator (x++).
- The relational operators compare two values and return a Boolean. These operators are often used in conditional statements.
- The conditional operator (`?:`) is the one ternary operator found in C#. It invokes the expression to the left of the colon if the tested condition evaluates true, and the expression to the right of the colon if the tested condition evaluates false.
- The compiler evaluates operators according to a series of precedence rules, and parentheses have the "highest" precedence.
- It is good programming practice to use parentheses to make your order of precedence explicit if there may be any ambiguity.

Quiz

Question 4-1. What is the output of these operations?

```
4 * 8
(4 + 8) / (4 - 2)
4 + 8 / 4 - 2
```

Question 4-2. Set x = 25 and y = 5. What do these expressions evaluate to?

```
(x >= y)
(x >= y * 5)
(x == y)
(x = y)
(x >= y) && (y <= x)
```

Question 4-3. Describe the difference between the prefix and postfix operators.

Question 4-4. Arrange these operators in order of precedence:

```
&
!=
?:
&&
++
```

Exercises

Exercise 4-1. Write a program that assigns the value 25 to variable `x`, and 5 to variable `y`. Output the sum, difference, product, quotient, and modulus of `x` and `y`.

Exercise 4-2. What will be the output of the following method?

```
static void Main( )
{
   int varA = 5;
   int varB = ++varA;
   int varC = varB++;
   Console.WriteLine( "A: {0}, B: {1}, C: {2}", varA, varB, varC );
}
```

Exercise 4-3. Write a program that demonstrates the difference between the prefix and postfix operators.