Operators

Subjects:

- Introduction
- Arithmetic operators
- Logical operators
- Compound logical operators

Introduction

Mathematicians create variables and manipulate them in various ways, adding them, multiplying them, and even integrating them. This lecture describes how to declare and define variables. However, it says little about how to use variables to get anything done after you declare them. **This lecture looks at the operations you can perform on variables to do some work**.

Operations require *operators,* such as +, –, =, <, and &. I cover arithmetic, logical, and other types of operators in this lecture.

Arithmetic operators

The set of arithmetic operators breaks down into several groups: **the simple arithmetic operators**, **the assignment operators**, and a set of **special operators unique to programming**. After you digest these, you also need to digest a separate set of logical operators.

Simple operators

You most likely learned in elementary school how to use most of the simple operators. **Table 1** lists them. ***Note: Computers use an asterisk (*), not the* multiplication sign (x), for multiplication**.

Table 1 Simple operators

Operator	*What It Means*
– (unary)	Take the negative of
*	Multiply
/	Divide
+	Add
– (binary)	Subtract
%	Modulo

Arithmetic operators

Most of these operators in the table are ***binary operators*** *because they operate* on two values: one on the left side of the operator and one on the right side. The lone exception is the **unary negative**. However, it's just as straightforward as the others, as shown in this example:

```
int n1 = 5;
int n2 = -n1; // n2 now has the value -5.
```

The value of `-n` is the negative of the value of `n`.

The modulo operator may not be quite as familiar to you as the others. **Modulo is similar to the remainder after division**. Thus, `5 % 3` is `2` (`5 / 3 = 1`, remainder `2`), and `25 % 3` is `1` (`25 / 3 = 8`, remainder `1`). Read it "five modulo three" or simply "five mod three". **Even numbers mod 2 are 0: `6 % 2 = 0` (`6/2 = 3`, remainder 0).**

The arithmetic operators other than modulo are defined for all numeric types.

The modulo operator isn't defined for floating-point numbers because you have no remainder after the division of floating-point values.

Operating orders

The value of some expressions may not be clear. Consider, for example, the following expression:

```
int n = 5 * 3 + 2;
```

C# generally executes common operators from left to right. So, the preceding example assigns the value `17` to the variable `n`.

C# determines the value of `n` in the following example by first dividing `24` by `6` and then dividing the result of that operation by `2` (as opposed to dividing `24` by the ratio `6` over `2`). The result is `2`:

```
int n = 24 / 6 / 2;
```

However, the various operators have a ***hierarchy, or order of precedence***. For example, **multiplication has higher precedence than addition**.

C# scans an expression and performs the operations of higher precedence before those of lower precedence.

Operating orders

Use *parentheses* to make your meaning explicit to human readers of the code as well as to the compiler.

The value of the following expression is clear, regardless of the operators' order of precedence:

```
int n = (7 % 3) * (4 + (6 / 3));
```

Parentheses can override the order of precedence by stating exactly how the compiler is to interpret the expression. To find the first expression to evaluate, C# looks for the innermost parentheses, dividing `6` by `3` to yield `2`:

```
int n = (7 % 3) * (4 + 2); // 6 / 3 = 2
```

Then **C# works its way outward, evaluating each set of parentheses in turn, from innermost to outermost**:

```
int n = 1 * 6; // (4 + 2) = 6
```

So the final result, and the value of `n`, is `6`.

The assignment operator

C# has inherited an interesting concept from C and C++: **Assignment is itself a binary operator**. **The assignment operator has the value of the argument to the right**. The assignment has the same type as both arguments, which must match.

```
n = 5 * 3;
```

In this example, `5 * 3` is `15` and an `int`. The assignment operator stores the `int` on the right into the `int` on the left and returns the value `15`. However, this view of the assignment operator allows the following line:

```
m = n = 5 * 3;
```

Assignments are evaluated in series from right to left. The right-hand assignment stores the value `15` into `n` and returns `15`. The left-hand assignment stores `15` into `m` and returns `15`, which is then dropped, leaving the value of each variable as `15`.
This strange definition for assignment makes the following rather bizarre expressions legal:

```
int n;
int m;
n = m = 2;
```

Avoid chaining assignments because it's less clear to human readers. Anything that can confuse people reading your code (including you) is worth avoiding because confusion breeds errors.

The **increment** operator

Of all the additions that you may perform in programming, **adding 1 to a variable is the most common**:

```
n = n + 1; // Increment n by 1.
```

C# extends the simple operators with a set of operators constructed from other binary operators. For example, `n += 1;` is equivalent to `n = n + 1;`.

An assignment operator exists for just about every binary operator: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`. **Look up *C# language operators in Help for full details* on them**.

Yet even `n += 1` is not good enough. C# provides this even shorter version:

```
++n; // Increment n by 1.
```

All these forms of incrementing a number are equivalent — they all increment n by 1.

C# has **two increment operators**: **++n** and **n++**. The first one, **++n**, is the ***preincrement* operator**, and **n++** is the ***postincrement operator***. *The difference is subtle but* important.

The increment operator

Remember that every expression has a type and a value. In the following code, both `++n` and `n++` are of type `int`:

```
int n;
n = 1;
int p = ++n;
n = 1;
int m = n++;
```

But what are the resulting values of `m` and `p`? (***Hint:*** *The choices are* `1` and `2`.) The value of **`p` is `2`**, and the value of **`m` is `1`**. That is, **the value of the expression `++n` is the value of `n` *after being incremented***, *and* ***the value of*** **the expression `n++` is the value of `n` *before it's incremented***. *Either way, the* resulting value of **`n` itself is `2`**.

C# has equivalent **decrement operators** — **`n--`** and **`--n`**. They work in exactly the same way as the increment operators.

Performing Logical Comparisons

C# provides a set of **logical comparison operators**, as shown in **Table 2**. **These operators are *logical comparisons because they return either* a `true` or a `false` value of type `bool`.**

Table 2 Logical Comparison Operators

Operator	*Operator Is True If*
`a == b`	a has the same value as b
`a > b`	a is greater than b
`a >= b`	a is greater than or equal to b
`a < b`	a is less than b
`a <= b`	a is less than or equal to b
`a != b`	a is not equal to b

Here's an example that involves a logical comparison:

```
int m = 5;
int n = 6;
bool b = m > n;
```

This example assigns the value `false` to the variable `b` because `5` is *not* greater than `6`.

Performing Logical Comparisons

The logical comparisons are defined for all numeric types, including `float`, `double`, `decimal`, and `char`. All the following statements are legal:

```
bool b;
b = 3 > 2; // true
b = 3.0 > 2.0; // true
b = 'a' > 'b'; // false -- Alphabetically, later = greater.
b = 'A' < 'a'; // true -- Upper A is less than lower a.
b = 'A' < 'b'; // true -- All upper are less than all lower.
b = 10M > 12M; // false
```

The comparison operators always produce results of type `bool`. The comparison operators other than == are not valid for variables of type `string` (C# offers other ways to compare strings).

U+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Comparing floating-point numbers

Comparing two **floating-point values** can get dicey, and you need to be careful with these comparisons. Consider the following comparison:

```
float f1;
float f2;
f1 = 10;
f2 = f1 / 3;
bool b1 = (3 * f2) == f1; //b1 is true if (3 * f2) equals f1.
f1 = 9;
f2 = f1 / 3;
bool b2 = (3 * f2) == f1;
```

Notice that both the fifth and eighth lines in the preceding example contain first an **assignment operator (=)** and then a **logical comparison (==)**. These are different operators, so **don't type = when you mean ==**. **C# does the logical comparison and then assigns the result to the variable on the left**.

The only difference between the calculations of `b1` and `b2` is the original value of `f1`. So, what are the values of `b1` and `b2`? The value of `b2` is clearly `true`: `9 / 3` is `3`; `3 * 3` is `9`; and `9` equals `9`.

Comparing floating-point numbers

The value of `b1` isn't obvious: `10 / 3` is `3.333...` and `3.333... * 3` is `9.999...`. **Is `9.999...` equal to 10?** That depends on how clever your processor and compiler are. On a Pentium or later processor, C# isn't smart enough to realize that `b1` should be `true` if the calculations are moved away from the comparison.

You can use the **system absolute value method** to compare `f1` and `f2`:

```
Math.Abs(f1 - 3.0 * f2) < .00001;
// Use whatever level of accuracy.
```

This calculation returns `true` for both cases. You can use the constant `Double.Epsilon` instead of `.00001` to produce the **maximum level of accuracy**. **`Epsilon` is the smallest possible difference between two non equal `double` variables**.

For a self-guided tour of the `System.Math class`, where `Abs` and many other useful mathematical functions live, look for *math in Help.*

Compound logical operations

The `bool` variables have another set of operators defined just for them, as shown in **Table -3**.

Table 3 The Compound Logical Operators

Operator	*Operator Is True If*
`!a`	a is false (also known as the "not" operator).
`a & b`	a and b are true (also known as the "and" operator).
`a | b`	Either a or b or else both are true (also known as a *and/or* b).
`a ^ b`	a is true or b is true but not both (also known as *a xor b*).
`a && b`	a is true and b is true with short-circuit evaluation.
`a || b`	a is true or b is true with short-circuit evaluation.

The `!` operator (`NOT`) is the logical equivalent of the minus sign. For example, `!a` (read "`not a`") is `true` if `a` is `false` and `false` if `a` is `true`.

The next operators in the table are straightforward. First, **`a & b` is `true` only if both `a` and `b` are `true`**. And **`a | b` is `true` if either `a` or `b` is `true` (or both are)**. The exclusive or (xor) operator, or `^`: **an exclusive or is `true` if either `a` or `b` is `true` but not if both `a` and `b` are `true`**.

All three operators produce a logical `bool` value as their result.

Compound logical operations

Consider the following example:

```
bool b = (boolExpression1) & (boolExpression2);
```

In this case, C# evaluates `boolExpression1` and `boolExpression2`. It then looks to see whether they both are `true` before deciding the value of `b`. However, this may be a wasted effort. If one expression is `false`, there's no reason to perform the other. Regardless of the value of the second expression, the result will be `false`. Nevertheless, `&` goes on to evaluate both expressions.

Compound logical operations

The `&&` operator avoids evaluating both expressions unnecessarily, as shown in the following example:

```
bool b = (boolExpression1) && (boolExpression2);
```

In this case, **C# evaluates `boolExpression1`. If it's `false`, then `b` is set to `false` and the program continues on its merry way**. On the other hand, if `boolExpression1` is `true`, then C# evaluates `boolExpression2` and stores the result in `b`. **The `&&` operator uses this *short-circuit evaluation because it* short-circuits around the second Boolean expression, if necessary**.

Most programmers use the **doubled forms** most of the time.

The `||` operator works the same way, as shown in the following expression:

```
bool b = (boolExpression1) || (boolExpression2);
```

If `boolExpression1` is `true`, there's no point in evaluating `boolExpression2` because the result is always `true`.
You can read these operators as **"short-circuit and"** and **"short-circuit or"**.

Matching Expression Types

In calculations, an expression's type is just as important as its value. Consider the following expression:

```
int n;
n = (5 * 5) + 7;
```

The resulting value of `n` is `32`. However, that expression also has an overall type based on the types of its parts. Written in "type language", the preceding expression becomes

```
int [=] (int * int) + int;
```

To evaluate the type of an expression, follow the same pattern you use to evaluate the expression's value. Multiplication takes precedence over addition. An `int` times an `int` is an `int`. Addition comes next. An `int` plus an `int` is an `int`. In this way, you can reduce the preceding expression this way:

```
(int * int) + int
int + int
int
```

Calculating the type of an operation

Most operators come in various flavors. For example, the **multiplication operator comes in the following forms** (the arrow means "produces"):

```
int     * int     ➪ int
uint    * uint    ➪ uint
long    * long    ➪ long
float   * float   ➪ float
decimal * decimal ➪ decimal
double  * double  ➪ double
```

Thus, `2 * 3` uses the `int * int` version of the `*` operator to produce the `int 6`.

Implicit type conversion

The `*` symbol works well for multiplying two `int`s or two `float`s. **But imagine what happens when the left- and right-hand arguments aren't of the same type**. For example, consider what happens in this case:

```
int anInt = 10;
double aDouble = 5.0;
double result = anInt * aDouble;
```

First, **C# doesn't have an `int * double` operation**. C# could just generate an error message and leave it at that; however, it tries to make sense of the programmer's intention. C# has `int * int` and `double * double` versions of multiplication and could convert `aDouble` into its `int` equivalent, but that would involve losing any fractional part of the number (the digits to the right of the decimal point). **Instead, in *implicit promotion, C# converts the `int` `anInt` into a `double` and uses the `double * double` operator***.

An implicit promotion is implicit because C# does it automatically, and it's a promotion because it involves the natural concept of uphill and downhill. The list of multiplication operators is in promotion order from `int` to `double` or from `int` to `decimal` — ***from narrower type to wider type***. ***No* implicit conversion exists between the floating-point types and decimal**. Converting from the more capable type, such as `double`, to a less capable type, such as `int`, is known as a ***demotion***. **Implicit demotions aren't allowed; C# generates an error message.**

Explicit type conversion

Imagine what happens if C# was wrong about implicit conversion and the programmer *wanted to perform integer multiplication?*

You can change the type of any value-type variable by using the *cast operator*. A *cast consists of a type enclosed in parentheses and placed immediately* in front of the variable or expression in question.

Thus the following expression uses the `int * int` operator:

```
int anInt = 10;
double aDouble = 5.0;
int result = anInt * (int)aDouble;
```

The cast of `aDouble` to an `int` is known as an ***explicit demotion or downcast.*** **The conversion is explicit because the programmer explicitly declared her intent.**

You can make an explicit conversion between any two value types, whether it's up or down the promotion ladder. Avoid implicit type conversion. Make any changes in value types explicit by using a cast. Doing so reduces the possibility of error and makes code much easier for humans to read.
C# offers **no type conversion path to or from the `bool` type.**

Assigning types

The same matching of types that you find in conversions applies to the **assignment operator**. Inadvertent type mismatches that generate compiler error messages usually occur in the assignment operator, not at the point of the mismatch.
Consider the following multiplication example:

```
int n1 = 10;
int n2 = 5.0 * n1;
```

The second line in this example generates an error message because of a type mismatch, but the error occurs *at the assignment — not at the multiplication*.

To perform the multiplication, **C# implicitly converts `n1` to a `double`**. C# can then perform `double` multiplication, the result of which is the `double`.

The type of the right-hand and left-hand operators of the assignment operator must match, but the type of the left-hand operator cannot change. Because C# refuses to demote an expression implicitly, the compiler generates the error message

```
Cannot implicitly convert type double to int.
```

Assigning types

C# allows this expression with an explicit cast:

```
int n1 = 10;
int n2 = (int)(5.0 * n1);
```

The parentheses are necessary because the cast operator has very high precedence.

This example works — ***explicit demotion is okay***. *The* `n1` *is* promoted to a `double`, the multiplication is performed, and the `double` result is demoted to an `int`.

Thanks for attention