Chapter 5

Operators and Expressions

Operators and Expressions

Objectives

After completing this unit you will be able to:

- **Use operators correctly in C# programs**
- **Use precedence to write cleaner code**
- **Use the *checked* keyword to control how various arithmetic errors are handled**

Operator Cardinality

- **Unary operators**
  - Example: unary minus.
- **Binary operators**
  - The most common
  - Examples: + - * /
- **C# has one ternary operator.**
  - ?:

Arithmetic Operators

- **The arithmetic operators include the four basic operations of addition, subtraction, multiplication, and division.**
- **We will examine each of these:**

1. in the case where operands are of the same type, and
2. in the case where one operand requires a conversion.

Multiplication

- **The three multiplicative operators in C# are:**
  - multiplication (*)
  - division (/)
  - remainder (%)
- **The only difficulty in multiplication comes from overflow, which is handled differently by the three data types.**
  - Integer multiplication overflow just silently drops bits.
  - Floating-point multiplication overflow results in **Infinity**.
  - Decimal multiplication overflow throws an exception.
- **See the example program *Multiply*.**

Division

- **Integer division always returns an integer result.**
  - The result may be silently truncated.
  - Integer and decimal division by zero throw exceptions.
  - Dividing the largest negative integer by negative one will throw an exception.
- **Floating-point division follows the IEEE 754 rules**
  - Division by zero returns Infinity.
  - Division of zero by anything (including zero) returns NaN.
- **The remainder is an integer result calculated by multiplying the quotient by the divisor and subtracting that from the original number.**
- **Remainder uses integer arithmetic.**

```
X % Y  is X - ((X/Y) * Y)
```

- **See the example program *IntegerDivision*.**

Additive Operators

- **There are six additive operators.**
  - Binary + and -
  - Unary + and -
  - Auto-increment (++) and auto-decrement (--)
- **Integer addition and subtraction may be *checked* or *unchecked*.**
  - In a *checked* context, overflow will generate an exception.
  - In an *unchecked* context, overflow bits are just silently lost.
- **Floating-point addition and subtraction may result in a floating-point number, positive Infinity, negative Infinity, or NaN.**
- **Decimal overflow generates an exception.**
- **Unary minus is equivalent to subtraction from zero.**
- **Unary plus is a no-op.**

Increment and Decrement

- **The increment and decrement operators come in two versions, prefix and postfix.**
- **Prefix:**

```
Y = ++X;
```

Equivalent to:

```
X = X + 1;
Y = X;
```

- **Postfix:**

```
Y = X++;
```

Equivalent to:

```
Y = X;
X = X + 1;
```

- **The increment and decrement operators work on integer, floating point, and decimal types.**

Example: A Small Calculator

```
// Ira.cs
//
// Interactive program to compute the total
// accumulation  in an Individual Retirement
// Account under compound interest.
// Assume that a deposit is made at the end of
// each year and that interest is compounded
// annually.

using System;

public class Ira
{
   public static int Main(string[] args)
   {
      InputWrapper iw = new InputWrapper();
      double amount;  // annual deposit amount
      double rate;    // interest rate
      int years;      // number of years
      double total;   // total accumulation
      amount = iw.getDouble("amount: ");
      rate = iw.getDouble("rate: ");
      years = iw.getInt("years: ");
      total  =  amount *
         (Math.Pow(1 + rate, years) - 1) / rate;
      long total_in_cents =
         (long) Math.Round(total * 100);
      total = total_in_cents /100.0;
      Console.WriteLine("total = {0}", total);
      return 0;
   }
}
```

Relational Operators

- **C# has the usual operators for testing equality, less than, etc.**
- **The result of a relational operation is bool.**
- **Note: The double-equal (==) is used for equality.**
  - Unlike C/C++, using the assignment operator where a relational operator is expected will generate a compile-time error.
  - This completely eliminates one of the old 'gotchas' that was in C/C++.

Operation	Returns true if...
`x == y`	`x equals y`
`x != y`	`x is not equal to y`
`x < y`	`x is less than y`
`x <= y`	`x is less than or equal to y`
`x > y`	`x is greater than y`
`x >= y`	`x is greater than or equal to y`

  - See the sample program **Relational**.

Conditional Logical Operators

- **The conditional logical operators provide the Boolean AND (&&), inclusive OR (||), and NOT (!) operations.**
  - Truth table for AND:

x	y	x && y
false	false	false
false	true	false
true	false	false
true	true	true

  - Truth table for OR:

x	y	x || y
false	false	false
false	true	true
true	false	true
true	true	true

  - Truth table for NOT:

x	! x
false	true
true	false

Short-Circuit Evaluation

- **An important feature of the logical operators is that they are evaluated from left to right.**
- **Evaluation terminates as soon as the answer is known.**
- **This may have some puzzling results if the terms of the expression have side effects.**

```
// ShortCircuit.cs

using System;

public class ShortCircuit
{
   public static int Main(string[] args)
   {
      int x = 4;
      int y = 5;
      Console.WriteLine("x = {0}, y = {1}", x, y);
      bool result = true || (++x == y);
      Console.WriteLine("result = {0}", result);
      Console.WriteLine("x = {0}, y = {1}", x, y);
      result = true && (++x == y);
      Console.WriteLine("result = {0}", result);
      Console.WriteLine("x = {0}, y = {1}", x, y);

      y = ~ x;
      bool a = true;
      bool b = false;
      result = a ^ b;

      return 0;
   }
}
```

Ternary Conditional Operator

- **The ternary operator (?:) is similar to an *if* statement, except that it returns a value.**
- **Ternary form:**

```
expr1 ? expr2 : expr3;
```

  - The first term (expr1) must be **bool**.
  - If expr1 is true, the value of the expression is expr2; otherwise, the value of the expression is expr3.
  - The 2nd and 3rd terms (expr2 and expr3 above) must evaluate to the same type.

```
// AbsoluteValue.cs

using System;

public class AbsoluteValue
{
   public static int Main(string[] args)
   {
      int x = 5;
      int abs = (x < 0) ? -x : x;
      Console.WriteLine("x = {0}, abs = {1}",
         x, abs);
      x = -x;
      abs = (x < 0) ? -x : x;
      Console.WriteLine("x = {0}, abs = {1}",
         x, abs);
      return 0;
   }
}
```

Bitwise Operators

- **Bitwise logical operators are similar to the Boolean operators, except that they are applied to the bits of an integer.**

Operator	Description
~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

- **Bitwise shift operators shift an integer value right or left.**

Operator	Description
<<	Left Shift
>>	Right Shift

Bitwise Logical Operators

- **The truth tables for the bitwise logical operators are similar to the truth tables for the Boolean operators, with a zero being treated as false, and a one treated as true.**
- **The bitwise logical operators do not use short-circuit evaluation.**
- **The exclusive OR, or XOR, (^) is available only in bitwise form.**

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise Shift Operators

- **The bitwise shift Operators take two operands.**
  - The first in the value to be shifted.
  - The second is the number of bit positions to shift by.

```
a = b << n;  // shift n positions left,
          // equivalent to multiplying by 2 n times

a = b >> n;  // shift n positions right and extend
             // sign, equivalent to dividing by 2
             // n times
```

  - See **Shift**

Assignment Operators

- **Normal assignment (=) is the most commonly used operator.**
  - Note that assignment is an expression that returns a value. Evaluating an expression can change a variable as a side effect.

```
int x = 30;
int y = 5;
int z = 1;
x = (y = z++) + 60;
```

  - See the program **Assign**.

- **Compound assignment combines assignment with binary arithmetic operator.**

Description	Operators
Arithmetic	*=, /=, %=, +=, -=
Shift	<<=, >>=
Bitwise	&=, ^=, |=

  - Example:

```
X += 5;
```

  Equivalent to:

```
X = X + 5;
```

Expressions

- **Expressions are built using constants and variables with operators.**
- **The result of one operation can then be used in another.**
- **Operations are performed in *precedence* order.**

Precedence

- **Precedence rules pre-date programming languages, and were developed to simplify the writing of algebraic expressions.**
- **Judicious use of precedence will result in cleaner code. But use parens if you need to!**
- **Precedence order in C# is given in the following table.**

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

Associativity

- **When an operand occurs between two operators at the same precedence level, the *associativity* of the operator controls the order of evaluation.**
- **Most operators associate left-to-right.**
- **The assignment operator (=) and the ternary operator (?:) associated right-to-left.**

Checking

- **C# allows you to control compile and runtime checking of various arithmetic operations (to detect conditions such as multiplication overflow).**
- **The default is to do compile time checking but not runtime checking.**
  - See **Unchecked**.
- **Checking may be applied to an entire module using the command line switch /checked+**
- **A block of code or a specific expression may be checked.**

```
checked // check a block
{
  z = x + 1;
  z = x * y;
}

z = checked (x * y); // check this multiplication
```

  - See **Checked**

Summary

- **We covered the many simple operators in C#.**
- **Some of the operators behave differently with respect to the different data types.**
- **Expressions are built from operators, constants, and variables, and may be used in other expressions.**
- **The programmer may specific the level of checking (how exceptional conditions, such as overflow, are handled).**
  - The default for checking is to perform compile-time checking, but not run-time checking.
- **Use precedence rules to simplify how you write expressions.**