Lecture 5 — Expressions & Operators

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Acknowledgments: W.D. Bishop

Expressions and Operators

Expressions are a statement that is evaluated.

Evaluation of a mathematical formula is an example.

Expressions can produce a temporary result or modify a variable.

An expression is built up using operators and operands.

Operators vs. Operands

Operators perform some operation (action).

Operands are the elements on which the operation is performed.

Let's examine a very simple mathematical expression: $2 + 4$.

Operators in this expression: $+$
Operands in this expression: $2, 4$

Expressions

Expressions can be more complicated: $(3 + 2) \times 5 - 1$

In programming, mathematical expressions are not the only kind.

We've already seen one kind expression frequently: assignment.

```
int i = 0;
```

Operators in this expression: `=`
Operands in this expression: `i`, `0`

Operator Categories

Operators fall into a number of different categories.

We will examine some, but not all, of them today:

1. Assignment operators
2. Arithmetic operators
3. Relational operators
4. Logical operators
5. Bitwise operators

Assignment Operator

You have already seen the assignment operator in a previous lecture.

`x = 5;` is an assignment statement.
Read aloud, this is “x is assigned five.”;

Arithmetic Operators

There are five standard arithmetic operators:

Operator	Definition
+	Adds two operands
–	Subtracts two operands
*	Multiplies two operands
/	Divides two operands
%	Computes the remainder after division

Master of the Fifth Math

The % operator is called “modulus”.

You’ve calculated the remainder already in school lots of times, even if you don’t remember it.

In elementary school math, before the concept of decimals was introduced, what did you do with an expression like $22 \div 6$?

Answer: 3 remainder 4.

In code: `22 % 6` will give a result of `4`.

Arithmetic Operators

Arithmetic operators work just as you would expect from standard math rules: $2 + 2$ will produce a result of 4.

As in algebra, we can use variables in arithmetic, like `x + 4`.

The result of this expression will depend on the value of `x` at the time the expression is evaluated.

Of course, we will probably want to do something with that result, like store it in a variable.

Danger: Integer Division

There is a danger with integer division: loss of precision.

Example: expression that divides 29 / 5 (even indirectly, where 29 and 5 are stored variables), your result will not be as expected.

The integer result of this outcome will be 5 (not rounded to 6).
This occurs even if the result will be stored in a `double` variable.

The result is truncated. Sometimes, this may be what you want.

To prevent it, make one of the types explicitly a `double`.
Change the equation to 29 / 5.0d

Terminology Note

The operators we have seen so far are binary: they take two operands.

The term for an operator that takes one operand is unary.

The term for an operator that takes three operands is ternary.

Unary Operators: Minus

A popular unary operator is the – (minus) operator.

When applied to a number, it inverts the sign of that number.
Mathematically equivalent to multiplying by -1.

Example:

```
int x = 9;
int y = -x;
```

This sets the value of `y` to be –9.

Unary Operators: Plus

For symmetry, there is also a unary + (plus) operator.

It does nothing useful. It has no effect on the value of the operand.

Example:

```
int x = -9;
int y = +x;
```

This sets the value of `y` to be `-9`. Yes, still negative.

Increment and Decrement Operators

Unary operators exist to increment & decrement numeric types.

The Increment operator (++) increments a variable by 1.

The Decrement operator (--) decrements a variable by 1.

Either operator may precede an operand (i.e., prefix notation): `++x;`
or follow an operand (i.e., postfix notation): `x++;`

When used as standalone statements, both notations are equivalent.

If used inside an expression, precedence rules apply.
We'll talk about these rules later.

Incrementing & Decrementing Variables

Some examples of using the increment and decrement operators:

```
int a = 5; // Declares a variable a of type int and sets it to 5
int b = 3; // Declares a variable b of type int and sets it to 3
int c = -4;  // Declares a variable c of type int and sets it to -4

a++; // Increments variable a (i.e., a is now 6)
b--; // Decrements variable b (i.e., b is now 2)
++c; // Increments variable c (i.e., c is now -3)
c++; // Increments variable c (i.e., c is now -2)
```

Additional Assignment Operators

As notational convenience, the language offers a number of "shortcut" operators that do something and an assignment at once.

A very common one is the += operator.

The statement `k += 7;` is equivalent to `k = k + 7;`

Yes, this does mean that these statements are all equivalent:

```
x = x + 1;
x += 1;
x++;
++x;
```

Additional Assignment Operators

A partial list of the assignment operators of C#:

Operator	Definition
=	Assigns the result to the variable
+=	Adds the result to the variable
-=	Subtracts the result from the variable
*=	Multiplies the variable by the result
/=	Divides the variable by the result
%=	Assigns the remainder to the variable

Operator Precedence

Recall from math class that a mathematical expression has a defined order of operations.

You may have learned the acronym BEDMAS.

- **B** - Brackets
- **E** - Exponents
- **D** - Division
- **M** - Multiplication
- **A** - Addition
- **S** - Subtraction

The things highest in the list are done first, then move down.

Operator Precedence

Purely mathematically, consider $10 \times 12 + 2$.

Follow the BEDMAS order: do the multiplication first, then addition.
$10 \times 12 = 120$, $120 + 2 = 122$.

If we fail to follow the order, we get the wrong answer:
$12 + 2 = 14$, $10 \times 14 = 140$.

To force the evaluation in a specific order, use brackets:
Write $10 \times (12 + 2)$ and the answer 140 is correct.

This principle applies in code: operations have a precedence order.

Operator Precedence

Sadly, there is no convenient acronym for the C# order of precedence.
The precedence rules in programming languages are too complex.

Basic BEDMAS still applies, so elements in brackets are done first.

Therefore: always use brackets to make the order of operations clear.
Even if it might seem redundant.

Precedence Example

What is the value of `x` after this statement is executed?

```
int x = 4 + 1 * 5 - 3++;
```

Answer:

1. First, 3++ is evaluated. `x = 4 + 1 * 5 - 4;`
2. Next, 1 * 5 is evaluated. `x = 4 + 5 - 4;`
3. Next, 4 + 5 is evaluated. `x = 9 - 4;`
4. Finally, 9 - 4 is evaluated. `x = 5;`

Is this confusing? The lack of brackets makes the order non-obvious.

Precedence Example: Using Brackets

Now we will use brackets to force precedence:

```
int x = (4 + 1) * 5 - 3++;
```

Now the addition of (4 + 1) is evaluated first.

This changes the value of `x` to 21.

Brackets were used here to change the order.
Adding more brackets is good practice to ensure clarity.

Type Promotion

Up until now, when we have added or assigned variables, we have ensured that they are of the same type.

It's possible to write a statement in which we perform some operation using types that are not the same.

If we have two different types, there are two possible outcomes: an error, or promotion.

Type Promotion

Type Promotion: the compiler converts a value of one type to another.

Example: `double x = 3;`
The literal 3 is an integer (`int` type).

The compiler takes 3 and converts it to `double` to do the assignment.

Type Promotion

Type promotion may also take place automatically. Consider this code:

```
int x = 2;
int y = 3;
double z = x + y;
```

When executing this instruction, the `x + y` expression is evaluated and produces an intermediate result of 5.

The intermediate result is an integer.

Before it is stored in the variable `z` it is converted to `double`.

C# Type Promotion Rules

The simplified version of the type promotion rules in C# are:

1. If either operand is a `double`, the result is a `double`
2. Otherwise, if either operand is a `long`, the result is a `long`
3. Otherwise, the result is an `int`.

Example: If we add a `long` to an `int`, the result is `long`.

Example 2: If we add a `long` to a `double`, the result is a `double`.

Explicit Type Conversion

Is it possible to be explicit about the conversion of a type?

Yes, through a process called casting.

A cast tells the compiler explicitly to convert one type to another.

Casting takes place, usually, during an assignment statement.
Consider this example where we will cast `a` to `double`.

```
int a = 7;
double b = (double) a;
```

A cast is, incidentally, another example of a unary operator.

Explicit Type Conversion

Automatically, the compiler will happily do type promotion: convert a variable type to a "bigger" type (e.g., `int` to `double`).

It's possible, but sometimes dangerous, to convert to a "smaller" type.
e.g., `double` to `int`.

Why is it dangerous? We might lose some data (or get a bogus value).
Example: the value 70 000 cannot be stored in a `ushort`.
The maximum value of a `ushort` is 65535.

To do so without an error or warning from the compiler, it is usually necessary to make it an explicit conversion.

Explicit Type Conversion

Consider this example where we'll convert a `double` to `int`:

```
double e = 7.9;
int f = (int) e;
```

What value is stored in the variable `f`?

Integer Division Problem

Earlier we saw that when dividing 29 / 5 we get the result of 5, not 6.

We saw the solution of making a literal a double (5.0d).

If the division is of two variables, cast one of the operands instead.
The compiler will promote the other variable.

```
double price = rate / (double) modifier;
```

Other Kinds of Operators

In the next lecture, we'll examine some other kinds of operators:

Logical and Bitwise operators.