UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

Arithmetic operators

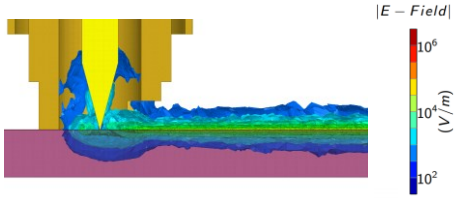
Douglas Wilhelm Harder, M.Math. LEL
Prof. Hiren Patel, Ph.D.
dwharder@uwaterloo.ca hiren.patel@uwaterloo.ca

Outline

- In this lesson, we will:
  - Define binary arithmetic operators
    - Addition, subtraction, multiplication and division
    - Integer division
    - Remainder/modulus operator
    - Upcasting integers to floating-point
  - Order of operations
    - Upcasting and order of operations
  - Unary arithmetic operators
    - Negation and "+"

Arithmetic operations

- Most engineering computations involve simulations of the real world, requiring the application of mathematics and modelling
  - The A380 double-decker jumbo jet was simulated entirely in software prior to being built for the first time…
  - Processors and circuits are simulated using mathematical models
- Here we see a mathematical model of a quantum socket [1]:



Arithmetic operations

- A *binary arithmetic operator* takes two numerical *operands* and returns the result of the operation
  - The operands may be integers or floating-point
- The available binary arithmetic operators are

Operation	Operator	Integers	Floating-point
Addition	+	3 + 5	3.2 + 7.3
Subtraction	–	7 - 6	9.5 - 4.1
Multiplication	×	8*9	1.5*2.7
Division	÷	1/2	4.5/9.6

Note: For clarity, it is usual to place spaces around + and - but no spaces around * or /

Arithmetic operations

- Operands can be literal integers and floating-point numbers, as well as identifiers

  `6.0*width*height` `PI*radius*radius`

- Juxtaposition is never acceptable to represent multiplication

  $2x^2 - 3xy + 4y^2 \longrightarrow$ `2*x*x - 3*x*y + 4*y*y`

- If you entered `2xx - 3xy + 4yy`, this would result in the compiler signalling an error
  - `2xx` is neither a valid integer, floating-point number or identifier

- There is no operator for exponentiation
  - Exponentiation requires a function call to a C++ library

Order of operations

- The compiler uses the same rules you learned from secondary school:
  - Multiplication and division before addition and subtraction
    - In all cases, going from left to right

- Try the following:

```
std::cout << (1 + 2 + 3 + 4 * 5 * 6 + 7 + 8 + 9);
std::cout << (1 * 2 * 3 + 4 * 5 * 6 + 7 + 8 + 9);
std::cout << (1 * 2 * 3 * 4 * 5 + 6 + 7 + 8 + 9);
std::cout << (1 * 2 * 3 * 4 * 5 - 6 * 7 + 8 * 9);
```


Order of operations

- Parentheses can be used to enforce the order in which operations are performed

- Common mistakes include

`k/m*n`	when they mean	`k/(m*n)` or `k/m/n`
`k/m+n`	when they mean	`k/(m + n)`
`k+m/n`	when they mean	`(k + m)/n`

Integer division

- In C++, the result of an arithmetic operation on integers must produce an integer
  - This is a problem for division

```
std::cout <<    (1/2);     // outputs  0
std::cout <<    (7/3);     // outputs  2
std::cout <<  (-11/4);     // outputs -2
std::cout << (-175/-13);   // outputs 13
```

- The result is the quotient discarding any remainder

$$\frac{175}{13} = 13 + \cancel{\frac{6}{13}} \qquad \frac{534}{15} = 35 + \cancel{\frac{3}{5}}$$

Order of operations

- Here are some further examples that depend on integer division:

```
std::cout << (1 / 2 + 3 * 4 + 5 * 6 * 7 - 8 * 9);
std::cout << (1 + 2 * 3 * 4 / 5 * 6 * 7 * 8 / 9);
std::cout << (1 * 2 + 3 + 4 * 5 * 6 / 7 * 8 + 9);
```

- For example:

```
(1 / 2) + (3 * 4) + (5 * 6 * 7) - (8 * 9)
   0    +   12    +     210     -   72 = 150
```


Integer remainder

- To find the remainder of a division, use the *modulus* operator
  - Also called the *remainder* operator

```
std::cout <<    (1 % 2);     // outputs  1
std::cout << std::endl;
std::cout <<    (7 % 3);     // outputs  1
std::cout << std::endl;
std::cout <<  (-11 %  4);    // outputs -3
std::cout << std::endl;
std::cout << (-175 % -13);   // outputs -6
std::cout << std::endl;
```

- For any integers `m` and `n`, it is always true that
  `(n/m)*m + (n % m)` equals `n`

Integer remainder

- Let's take a closer look at:
  `(n/m)*m + (n % m)`
- Don't we know from mathematics that as long as $m \neq 0$,

$$\frac{n}{m} \cdot m = n \quad ?$$

- C++ evaluates one expression at a time
  - If the compiler sees `(7/3)*3`,
    - It first will have `(7/3)` calculated, which evaluates to `2`
    - It then proceeds to calculate `2*3` which is `6`

Spacing around operators

- In C++, you can put any amount of whitespace between operators and their operands:

```
std::cout << ((n/m)*m + (n % m));
std::cout << ((n/m)*m+(n%m));
```

```
std::cout                    <<
  (                          (     n
/   m    )*   m              +
 (     n%          m                ))
                          ;
```

- We recommend:
  - Putting one space between operands and `+`, `-` and `%`
  - Juxtaposing operands with `*` and `/` operands
- Forcing your self soon makes it habitual
  - You will not even think about it when you type…

Upcasting

- Suppose the compiler sees:

```
3.2/2
```

- Does it use floating-point division, or integer division?
  - The only way for this to make sense is for the compiler to *interpret* the 2 as a floating-point number
  - This process is called *upcasting*
    - Literals are upcast by the compiler

Order of operations and upcasting

- Again, C++ is very exact when upcasting occurs:
  - Only when one operand is a floating-point number and the other is an integer is the integer upcast to a floating-point number
- What is the output of each of the following? Why?

```
std::cout << (10.0 + (1 / 2) * 3.0);
std::cout << (10.0 +  1 / 2 * 3.0);
std::cout << (10.0 +  1 / (2 / 3.0));
```


Unary operators

- There are two unary operators for arithmetic:
  - Unary negation operator changes the sign of what follows:

```
std::cout << -(1 + 2 + 3);
std::cout << -(2*3*4);
std::cout << -(1 + 2*3);
```

  - Unary neutral operator leaves the sign unchanged:

```
std::cout << +(1 + 2 - 5);
std::cout << +(-2*3*4);
std::cout << +(1 - 2*3);
```


Arithmetic expression

- If all of the variables are `int`, the result of these is an `int`:

```
35
a
a + b + c + d + 1
12*(a + b)*(1 - b)
(a + b + c)/10;
(a - 1)*a*(a + 1)
-a + b
+c
```

- If all of the variables are `double`, the result of these is a `double`:

```
35.0
a
a + b + c + d + 1.3
12.5*(a + b)*(1.0 - b)
(a + b + c)/10.5;
(a – 1.7)*a*(a + 1.7)
-a + b
+c
```

Arithmetic expression

- We can now make the following statements:
  - An integer arithmetic expression will always evaluate to an integer
- We can make an identical description of floating-point arithmetic expressions

Summary

- Following this presentation, you now:
  - Understand the binary arithmetic operators in C++
    - Addition, subtraction, multiplication and division
  - The effect of integer division and the remainder operator
  - Upcasting integers to floating-point
  - Understand the order of operations and upcasting
  - The two unary arithmetic operators

References

[1] Thomas McConkey, a simulation of a 6 GHz microwave signal transmitting through a coaxial pogo pin onto a micro-coplanar waveguide transmission line of a thin film superconducting aluminium (i.e., a quantum socket). Developed with the Ansys software HFSS.

[2] Wikipedia, https://en.wikipedia.org/wiki/Operators_in_C_and_C++#Arithmetic_operators

[3] cplusplus.com tutorial, http://www.cplusplus.com/doc/tutorial/operators/

[4] C++ reference, https://en.cppreference.com/w/cpp/language/operator_arithmetic

Acknowledgments

Proof read by Dr. Thomas McConkey and Charlie Liu.

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.