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

ECE 150 *Fundamentals of Programming*

Arithmetic operators

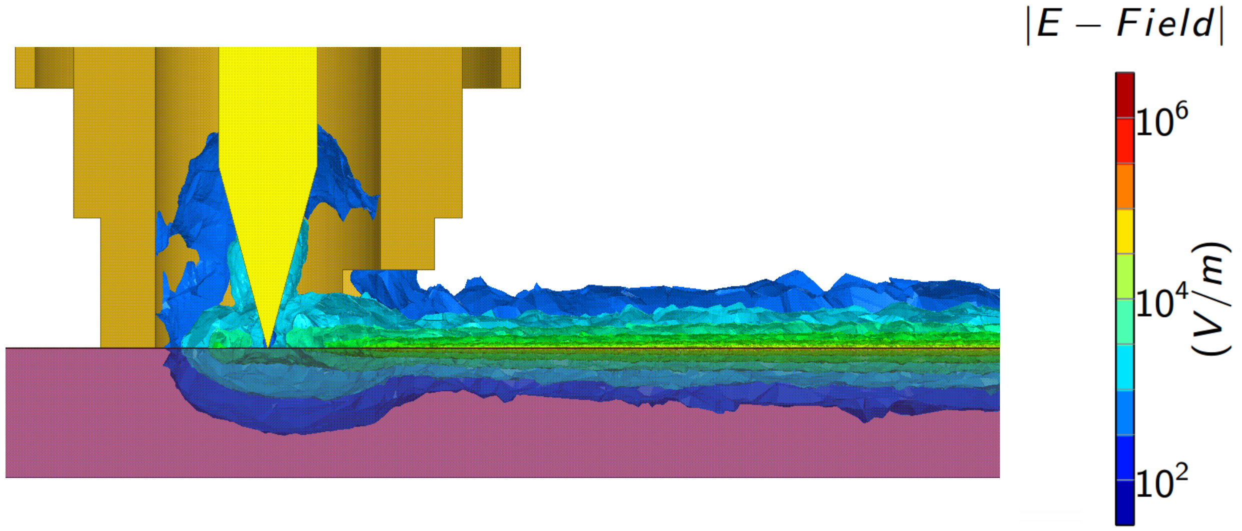
Douglas Wilhelm Harder, M.Math., LEL
Prof. Hiren Patel, Ph.D., P.Eng.
Prof. Werner Dietl, Ph.D.

Outline

- In this lesson, we will:
  - Define binary arithmetic operators
    - Addition, subtraction, multiplication and division
    - Integer division
    - Remainder/modulus operator
    - Conversion of integers to floating-point
  - Look at order of operations
    - Standard conversions and order of operations
  - Consider initialization of and assignment to local variables
  - Describe the unary arithmetic operators
    - Negation and “+”

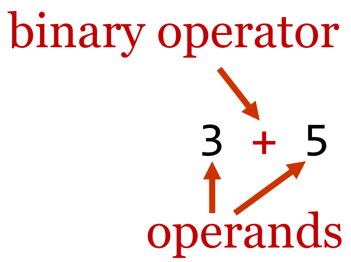
Arithmetic operations

- 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 evaluates that arithmetic operation
  - The operands may be integers or floating-point
  - They may be literals or local variables



- 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`
Remainder	n/a	5%3	-

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

Arithmetic expression

- An *arithmetic expression* is any combination of operands combined with arithmetic operators

```
7.5
x*x + 1
6.0*width*height
pi*radius*radius
n*(n + 1)/2.0
n*n*(n + 1)*(n + 1)/4.0;
```

- Arithmetic expressions can be:
  - Printed
  - Used to initialize local variables
  - Assigned to local variables

Arithmetic operations

- Juxtaposition is never acceptable to represent multiplication

$$2x^2 - 3xy + 4y^2 \longrightarrow \texttt{2.0*x*x - 3.0*x*y + 4.0*y*y}$$

- 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
  - More on this later…

Order of operations

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

```
 x + y/2.0  +   z/a /b *c
(x + y/2.0) + ((z/a)/b)*c
```

  - Parentheses can be used either
    - To enforce a different order of operations
    - To clarify your intended order of operations

Order of operations

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    std::cout << (1 + 2 + 3 + 4 * 5 * 6 + 7 + 8 + 9) << std::endl;
    std::cout << (1 * 2 * 3 + 4 * 5 * 6 + 7 + 8 + 9) << std::endl;
    std::cout << (1 * 2 * 3 * 4 * 5 + 6 + 7 + 8 + 9) << std::endl;
    std::cout << (1 * 2 * 3 * 4 * 5 - 6 * 7 + 8 * 9) << std::endl;

    return 0;
}
```

Order of operations

- It is paramount to remember that parentheses can be used either to emphasize or 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`

- In two cases, spacing would help you see what is going on:

```
k/m+n            k/m + n
k+m/n              k + m/n
```

Initialization

- Arithmetic expressions can be used to initialize a local variable:

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    double x{};
    std::cout << "Enter a value of x: ";
    std::cin >> x;

    double y{ x*x - 2.0*x + 1.0 };

    std::cout << " 2"                        << std::endl;
    std::cout << "x  - 2x + 1 = " << y << std::endl;

    return 0;
}
```

Output:

```
Enter a value of x: 6.52
 2
x  - 2x + 1 = 30.4704
```

Assignment

- Another example of initialization:

```
// Function definitions
int main() {
    double radius{};
    std::cout << "Enter the radius of a sphere: ";
    std::cin >> radius;

    double pi{3.1415926535897932}; // 17 digits of precision
    double volume{ 4.0/3.0*pi*radius*radius*radius };
    std::cout << "The volume of the sphere is " << volume << std::endl;

    double area{ 4.0*pi*radius*radius };
    std::cout << "The surface area is "<< area << std::endl;

    return 0;
}
```

Output:

```
Enter a radius of a sphere: 1.5
The volume of the sphere is 14.1372
The surface area is 28.2743
```

Assignment

- This also works, but is less pleasing…

```
// Function definitions
int main() {
    double radius{};
    std::cout << "Enter the radius of a sphere: ";
    std::cin >> radius;

    double pi{3.1415926535897932};
    std::cout << "The volume of the sphere is "
              << (4.0/3.0*pi*radius*radius*radius) << std::endl;

    std::cout << "The surface area is "<< (4.0*pi*radius*radius)
              << std::endl;

    return 0;
}
```

Assignment

- The result can also be assigned to a local variable:

```
int main() {
    double x{};
    std::cout << "Enter a value of x: ";
    std::cin >> x;

    double y{};
    std::cout << "Enter an approximation of sqrt(x): ";
    std::cin >> y;

    y = (y + x/y)/2.0;
    std::cout << "A better approximation of sqrt(" << x << ") = "
              << y << std::endl;

    y = (y + x/y)/2.0;
    std::cout << "An even better approximation of sqrt(" << x << ") = "
              << y << std::endl;

    return 0;
}
```

Assignment

- Executing this program:

```
Enter a value of x: 3.5
Enter an approximation of sqrt(x):  1.8
A better approximation of sqrt(3.5) = 1.87222
A even better approximation of sqrt(3.5) = 1.87083
```

$$\sqrt{3.5} \approx 1.8708286933869707$$

```
y = (y + x/y)/2.0;
std::cout << "A better approximation of sqrt(" << x << ") = "
          << y << std::endl;

y = (y + x/y)/2.0;
std::cout << "A even better approximation of sqrt(" << x << ") = "
          << y << std::endl;
```

Assignment

- The left-hand side of an assignment operator must be a local variable
  - You cannot assign to an arithmetic expression

```
2.0*x + 1.0 = y - 1.0;
x = (y - 2.0)/2.0;
```

  - Again, we emphasize, always read the above as
    "*x* is assigned the value of *y* minus two all divided by two."

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)  << std::endl;      // outputs  0
std::cout <<    (7/3)  << std::endl;      // outputs  2
std::cout <<  (-11/4)  << std::endl;      // outputs -2
std::cout <<  (188/13) << std::endl;      // outputs 14
```

- The result is the quotient discarding any remainder

$$\frac{188}{13} = 14 + \cancel{\frac{6}{13}} \qquad \frac{534}{15} = 35 + \cancel{\frac{3}{5}}$$

Order of operations

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

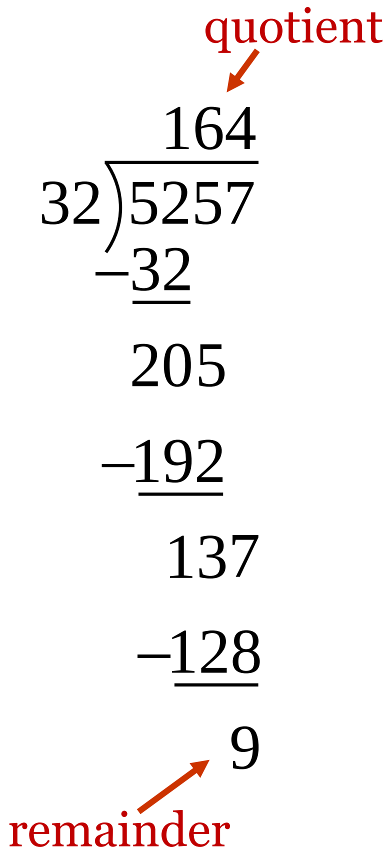
```
std::cout << (1 / 2 + 3 * 4 + 5 * 6 * 7 - 8 * 9) << std::endl;
std::cout << (1 + 2 * 3 * 4 / 5 * 6 * 7 * 8 / 9) << std::endl;
std::cout << (1 * 2 + 3 + 4 * 5 * 6 / 7 * 8 + 9) << std::endl;
```

- For example:

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

Integer remainder

- Recall that in long division, you find the quotient and the remainder



$$5257 = 164 \cdot 32 + 9$$

$$\frac{5257}{32} = 164 + \frac{9}{32}$$

Integer remainder

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

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

- 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 operation 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…

Standard conversions

- Suppose the compiler sees:

```
3.5/2
```

- 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 a *standard conversion*
    - Conversion of literals is performed by the compiler

Order of operations and conversions

- Again, C++ is very exact when standard conversions occurs:
  - Only when one operand is a floating-point number and the other is an integer is the integer converted to a floating-point number

- What is the output of each of the following? Why?

```
std::cout << (10.0 + 3.0/(9/2))   << std::endl;
std::cout << (10.0 + 3.0/9*2)     << std::endl;
std::cout << (10.0 +   3/(9/2.0)) << std::endl;
```

Unary operators

- A unary operator has only one operand
  - For example, from secondary school, the "!" is a unitary operator
    - It only takes one operands, e.g., 5!
- There are two unary operators for arithmetic:
  - Unary negation operator changes the sign of what follows:

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

  - Unary *neutral* operator '+' leaves the sign unchanged:

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

Standard conversions

- If all of the operands are integers, the result will be an integer:

```
35
3 + 6 + 4 + 7 + 1
12*(3 + 6)*(1 - 4)
(5 + 3 + 7)/10;
(56 - 1)*3*(4 + 1)
-243 + 6
+23
```

- If even one operand is a float, the result of will be a float:

```
35.0
3 + 7 + 2.9 + 7 + 1.3
12.5*(5 + 2)*(1.0 - 6)
(3 + 2 + 1)/10.5;
(6 – 1.7)*4*(8.9 + 1.7)
-6.4 + 3
+2.7
```

Arithmetic expression

- We can now make the following statements:
  - An integer arithmetic expression will always evaluate to an integer
  - A floating-point arithmetic expression will always evaluate to a float
  - A mixed arithmetic expression will always evaluate to a float

Summary

- Following this presentation, you now:
  - Understand the binary arithmetic operators in C++
    - Addition, subtraction, multiplication and division
  - Know that the result can:
    - Initialize a local variable
    - Be assigned to a local variable
  - The effect of integer division and the remainder operator
  - Conversion of integers to floating-point
  - Understand the order of operations and standard conversions
  - Are aware of 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