### Arithmetic Operators

**Topics**
- Arithmetic Operators
- Operator Precedence
- Evaluating Arithmetic Expressions
- In-class Project
- Incremental Programming

**Reading**
- Section 2.5

### Arithmetic Operators in C

<table>
<thead>
<tr>
<th>Name</th>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>num1 + num2</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>initial - spent</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>fathoms * 6</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>sum / count</td>
</tr>
<tr>
<td>Modulus</td>
<td>%</td>
<td>m % n</td>
</tr>
</tbody>
</table>

### Division

- If both operands of a division expression are integers, you will get an integer answer. The fractional portion is thrown away.
- Examples:
  - $17 \div 5 = 3$
  - $4 \div 3 = 1$
  - $35 \div 9 = 3$
Division (con’t)
- Division where at least one operand is a floating point number will produce a floating point answer.
- Examples: 17.0 / 5 = 3.4
  - 4 / 3.2 = 1.25
  - 35.2 / 9.1 = 3.86813
- What happens? The integer operand is temporarily converted to a floating point, then the division is performed.

Division By Zero
- Division by zero is mathematically undefined.
- If you allow division by zero in a program, it will cause a **fatal error**. Your program will terminate execution and give an error message.
- Non-fatal errors do not cause program termination, just produce incorrect results.

Modulus
- The expression $m \% n$ yields the integer remainder after $m$ is divided by $n$.
- Modulus is an integer operation -- both operands MUST be integers.
- Examples: 17 % 5 = 2
  - 6 % 3 = 0
  - 9 % 2 = 1
  - 5 % 8 = 5
Uses for Modulus

- Used to determine if an integer value is even or odd
  
  \[ 5 \% 2 = 1 \text{ odd} \quad 4 \% 2 = 0 \text{ even} \]

If you take the modulus by 2 of an integer, a result of 1 means the number is odd and a result of 0 means the number is even.

- The Euclid’s GCD Algorithm (done earlier)

Arithmetic Operators

Rules of Operator Precedence

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Precedence &amp; Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Evaluated first. If nested (embedded), innermost first. If on same level, left to right.</td>
</tr>
<tr>
<td>* / %</td>
<td>Evaluated second. If there are several, evaluated left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>Evaluated third. If there are several, evaluated left to right.</td>
</tr>
<tr>
<td>=</td>
<td>Evaluated last, right to left.</td>
</tr>
</tbody>
</table>

Using Parentheses

- Use parentheses to change the order in which an expression is evaluated.

- \[ a + b \times c \] Would multiply \( b \times c \) first, then add \( a \) to the result.

- If you really want the sum of \( a \) and \( b \) to be multiplied by \( c \), use parentheses to force the evaluation to be done in the order you want. \[ (a + b) \times c \]

- Also use parentheses to clarify a complex expression.
Practice With Evaluating Expressions

Given integer variables \(a\), \(b\), \(c\), \(d\), and \(e\), where \(a = 1\), \(b = 2\), \(c = 3\), \(d = 4\), evaluate the following expressions:

\[
\begin{align*}
& a + b - c + d \\
& a * b / c \\
& 1 + a * b \% c \\
& a + d \% b - c \\
& e = b = d + c / b - a 
\end{align*}
\]

A Sample Project

- Let’s write a program that computes and displays the volume and surface area of a cube.
- Procedure:
  - Use the pseudocode that we developed in “Algorithms, Part 3 of 3”
  - Convert the algorithm to code
  - Clean up the code (spacing, indentation, commenting)

The Box - Pseudocode

Display “Enter the height: “
Read <height>
While (<height> <= 0 )
  Display “The height must be > 0”
  Display “Enter the height: “
  Read <height>
End_while
The Box - Pseudocode (con’t)

Display “Enter the width: “
Read <width>
While (<width> <= 0 )
    Display “The width must be > 0”
    Display “Enter the width: “
    Read <width>
End_while

The Box - Pseudocode (con’t)

Display “Enter the depth: “
Read <depth>
While (<depth> <= 0 )
    Display “The depth must be > 0”
    Display “Enter the depth: “
    Read <depth>
End_while

The Box - Pseudocode (con’t)

<volume> = <height> X <width> X <depth>
<surface1> = <height> X <width>
<surface2> = <width> X <depth>
<surface3> = <height> X <depth>
<surface area> = 2 X (<surface1> + <surface2> + <surface3>)
The Box - Pseudocode (con’t)

- Display “Height = “, <height>
- Display “Width = “, <width>
- Display “Depth = “, <depth>
- Display “Volume = “, <volume>
- Display “Surface Area = “, <surface area>

Good Programming Practice

- It is best not to take the “big bang” approach to coding.
- Use an incremental approach by writing your code in incomplete, yet working, pieces.
- For example, for your projects,
  - Don’t write the whole program at once.
  - Just write enough to display the user prompt on the screen.
  - Get that part working first (compile and run).
  - Next, write the part that gets the value from the user, and then just print it out.

Good Programming Practice

- Get that working (compile and run).
- Next, change the code so that you use the value in a calculation and print out the answer.
- Get that working (compile and run).
- Continue this process until you have the final version.
- Get the final version working.

Always have a working version of your program!
Using the Incremental Approach

- Let’s think about how we could have developed the volume and surface area program incrementally.