COSMOS Cluster 6

Embedded Programming and Robotics

Gabriel Hugh Elkaim
Operators
How to Code Arithmetic Expressions

Definition
An arithmetic expression is an expression that contains one or more operands and arithmetic operators.

- Operands may be variables, constants or functions that return a value.
- There are 9 arithmetic operators that may be used:
  - Binary Operators: +, -, *, /, \\
  - Unary Operators: +, -, ++, --
# Operators

**Arithmetic**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>x * y</td>
<td>Product of x and y</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>x / y</td>
<td>Quotient of x and y</td>
</tr>
<tr>
<td>%</td>
<td>Modulo</td>
<td>x % y</td>
<td>Remainder of x divided by y</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>x + y</td>
<td>Sum of x and y</td>
</tr>
<tr>
<td>−</td>
<td>Subtraction</td>
<td>x − y</td>
<td>Difference of x and y</td>
</tr>
<tr>
<td>+ (unary)</td>
<td>Positive</td>
<td>+x</td>
<td>Value of x</td>
</tr>
<tr>
<td>− (unary)</td>
<td>Negative</td>
<td>−x</td>
<td>Negative value of x</td>
</tr>
</tbody>
</table>

**NOTE**

An int divided by an int returns an int:

\[10/3 = 3\]

Use modulo to get the remainder:

\[10\%3 = 1\]
Operators

Division Operator

- If both operands are an integer type, the result will be an integer type (int, char)
- If one or both of the operands is a floating point type, the result will be a floating point type (float, double)

**Example: Integer Divide**

```c
int a = 10;
int b = 4;
float c;
c = a / b;
```

```
c = 2.000000  \times
Because: int / int \Rightarrow int
```

**Example: Floating Point Divide**

```c
int a = 10;
float b = 4.0f;
float c;
c = a / b;
```

```
c = 2.500000  \checkmark
Because: float / int \Rightarrow float
```
Operators
Implicit Type Conversion

- In many expressions, the type of one operand will be temporarily "promoted" to the larger type of the other operand.

Example

```cpp
int x = 10;
float y = 2.0, z;
z = x * y;    // x promoted to float
```

- A smaller data type will be promoted to the largest type in the expression for the duration of the operation.
Operators

Implicit Arithmetic Type Conversion Hierarchy

- long double
- double
- float
- unsigned long long
- long long
- unsigned long
- long
- unsigned int
- int
- unsigned short
- short
- unsigned char
- char

Smaller types converted to largest type in expression

Gabriel Hugh Elkaim
Operators

Arithmetic Expression Implicit Type Conversion

- Example implicit type conversions

Assume x is defined as:
```java
short x = -5;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Implicit Type Conversion</th>
<th>Expression's Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
<td>x is promoted to int</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>x * -2L</td>
<td>x is promoted to long because -2L is a long</td>
<td>long</td>
<td>10</td>
</tr>
<tr>
<td>8/x</td>
<td>x is promoted to int</td>
<td>int</td>
<td>-1</td>
</tr>
<tr>
<td>8%x</td>
<td>x is promoted to int</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>8.0/x</td>
<td>x is promoted to double because 8.0 is a double</td>
<td>double</td>
<td>-1.6</td>
</tr>
</tbody>
</table>
Operators

Applications of the Modulus Operator (%)

- Truncation: $x \mod 2^n$ where $n$ is the desired word width (e.g. 8 for 8 bits: $x \mod 256$)
  - Returns the value of just the lower $n$-bits of $x$
- Can be used to break apart a number in any base into its individual digits

Example

```c
#define MAX_DIGITS 6
long number = 123456;
int i, radix = 10; char digits[MAX_DIGITS];

for (i = 0; i < MAX_DIGITS; i++)
{
    if (number == 0) break;
    digits[i] = (char)(number % radix);
    number /= radix;
}
```
Operators

Arithmetic: Increment and Decrement

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Increment</td>
<td>x++</td>
<td>Use x then increment x by 1</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
<td>x--</td>
<td>Use x then decrement x by 1</td>
</tr>
</tbody>
</table>

**Postfix Example**

```plaintext
x = 5;
y = (x++) + 5;
// y = 10
// x = 6
```

**Prefix Example**

```plaintext
x = 5;
y = (++x) + 5;
// y = 11
// x = 6
```
Operators
How to Code Assignment Statements

Definition
An assignment statement is a statement that assigns a value to a variable.

- Two types of assignment statements
  - Simple assignment
    \[ \text{variable} = \text{expression}; \]
    The expression is evaluated and the result is assigned to the variable
  - Compound assignment
    \[ \text{variable} = \text{variable op expression}; \]
    The variable appears on both sides of the =
### Operators

#### Assignment

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Assignment</td>
<td><code>x = y</code></td>
<td>Assign <code>x</code> the value of <code>y</code></td>
</tr>
<tr>
<td>+=</td>
<td></td>
<td><code>x += y</code></td>
<td><code>x = x + y</code></td>
</tr>
<tr>
<td>-=</td>
<td></td>
<td><code>x -= y</code></td>
<td><code>x = x - y</code></td>
</tr>
<tr>
<td>*=</td>
<td></td>
<td><code>x *= y</code></td>
<td><code>x = x * y</code></td>
</tr>
<tr>
<td>/=</td>
<td></td>
<td><code>x /= y</code></td>
<td><code>x = x / y</code></td>
</tr>
<tr>
<td>%=</td>
<td>Compound</td>
<td><code>x %= y</code></td>
<td><code>x = x % y</code></td>
</tr>
<tr>
<td>&amp;=</td>
<td>Assignment</td>
<td><code>x &amp;= y</code></td>
<td><code>x = x &amp; y</code></td>
</tr>
<tr>
<td>^=</td>
<td></td>
<td><code>x ^= y</code></td>
<td><code>x = x ^ y</code></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td></td>
<td>`x</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td></td>
<td><code>x &lt;&lt;= y</code></td>
<td><code>x = x &lt;&lt; y</code></td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td></td>
<td><code>x &gt;&gt;= y</code></td>
<td><code>x = x &gt;&gt; y</code></td>
</tr>
</tbody>
</table>
Operators

Compound Assignment

• Statements with the same variable on each side of the equals sign:

Example

```
x = x + y;
```

This operation may be thought of as: The new value of \( x \) will be set equal to the current value of \( x \) plus the value of \( y \)

• May use the shortcut assignment operators (compound assignment):

Example

```
x += y;    // Increment x by the value of y
```
Operators
Compound Assignment

Example

```c
int x = 2;  // Initial value of x is 2
x *= 5;     // x = x * 5
```

Before statement is executed: `x = 2`

After statement is executed: `x = 10`

```
x *= 5;
```

Is equivalent to: `x = (x * 5);`

Evaluate right side first: `x = (2 * 5);`

Assign result to x: `x = 10;`
# Operators

## Relational

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result ((FALSE = 0), (TRUE \neq 0))</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td>(x &lt; y)</td>
<td>1 if (x) less than (y), else 0</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to</td>
<td>(x \leq y)</td>
<td>1 if (x) less than or equal to (y), else 0</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td>(x &gt; y)</td>
<td>1 if (x) greater than (y), else 0</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to</td>
<td>(x \geq y)</td>
<td>1 if (x) greater than or equal to (y), else 0</td>
</tr>
<tr>
<td><code>==</code></td>
<td>Equal to</td>
<td>(x == y)</td>
<td>1 if (x) equal to (y), else 0</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Not equal to</td>
<td>(x \neq y)</td>
<td>1 if (x) not equal to (y), else 0</td>
</tr>
</tbody>
</table>

In conditional expressions, **any non-zero value** is interpreted as **TRUE**. A value of 0 is always **FALSE**.
Operators

Difference Between = and ==

Be careful not to confuse = and ==.
They are not interchangeable!

• = is the assignment operator
  \( x = 5 \) assigns the value 5 to the variable \( x \)

• == is the 'equals to' relational operator
  \( x == 5 \) tests whether the value of \( x \) is 5

```python
if (x == 5)
{
  do if value of x is 5
}
```
Operators
Difference Between = and ==

• What happens when the following code is executed?

```c
void main(void)
{
    int x = 2; //Initialize x
    if (x = 5) //If x is 5,
    {
        printf("Hi!"); //...display "Hi!"
    }
}
```
### Operators

#### Logical

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result (FALSE = 0, TRUE ≠ 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>x &amp;&amp; y</td>
<td>1 if both ( x \neq 0 ) and ( y \neq 0 ), else 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
<td>! x</td>
<td>1 if ( x = 0 ), else 0</td>
</tr>
</tbody>
</table>

In conditional expressions, any non-zero value is interpreted as TRUE. A value of 0 is always FALSE.
## Operators

### Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result (for each bit position)</th>
</tr>
</thead>
</table>
| &        | Bitwise AND     | x & y   | 1, if 1 in both x and y  
0, if 0 in x or y or both  |
| | Bitwise OR      | x | y       | 1, if 1 in x or y or both  
0, if 0 in both x and y     |
| ^        | Bitwise XOR     | x ^ y   | 1, if 1 in x or y but not both  
0, if 0 or 1 in both x and y |
| ~        | Bitwise NOT     | ~x      | 1, if 0 in x  
0, if 1 in x  |

- The operation is carried out on each bit of the first operand with each corresponding bit of the second operand.
Operators
Difference Between & and &&

Be careful not to confuse & and &&. They are not interchangeable!

- & is the bitwise AND operator
  \[ \text{0b1010} \& \text{0b1101} \rightarrow \text{0b1000} \]

- && is the logical AND operator
  \[ \text{0b1010} \&\& \text{0b1101} \rightarrow \text{0b0001} \] (TRUE)
  \[ <\text{Non-Zero Value}> \&\& <\text{Non-Zero Value}> \rightarrow 1 \] (TRUE)

\[
\text{if} \ (x \&\& y) \\
\{ \\
\quad \text{do if } x \text{ and } y \text{ are both TRUE (non-zero) }
\}
\]}
Operators
Difference Between & and &&

• What happens when each of these code fragments are executed?

Example 1 – Using A Bitwise AND Operator

```c
char x = 0b1010;
char y = 0b0101;
if (x & y) printf("Hi!");
```

Example 2 – Using A Logical AND Operator

```c
char x = 0b1010;
char y = 0b0101;
if (x && y) printf("Hi!");
```
Operators
Logical Operators and Short Circuit Evaluation

- The evaluation of expressions in a logical operation stops as soon as a TRUE or FALSE result is known.

Example

If we have two expressions being tested in a logical AND operation:

\[ expr1 \&\& expr2 \]

The expressions are evaluated from left to right. If \( expr1 \) is 0 (FALSE), then \( expr2 \) would not be evaluated at all since the overall result is already known to be false.

**Truth Table for AND (&&)**

<table>
<thead>
<tr>
<th>( expr1 )</th>
<th>( expr2 )</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X (0)</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>X (1)</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\( expr2 \) is not evaluated in the first two cases since its value is not relevant to the result.
Operators

Logical Operators and Short Circuit Evaluation

- The danger of short circuit evaluation

Example

```c
if (!(z++) && (c = a + b)) {
    z += 5;
    c += 10;  // Initial value of c may not be correct
}
```

It is perfectly legal in C to logically compare two assignment expressions in this way, though it is not usually good programming practice. A similar problem exists when using function calls in logical operations, which is a very common practice. The second function may never be evaluated.
## Operators

### Shift

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>Shift Left</td>
<td>x &lt;&lt; y</td>
<td>Shift x by y bits to the left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift Right</td>
<td>x &gt;&gt; y</td>
<td>Shift x by y bits to the right</td>
</tr>
</tbody>
</table>

**Shift Left Example:**

\[
x = 5; \quad // \quad x = 0b00000101 = 5
y = x << 2; \quad // \quad y = 0b00010100 = 20
\]

- In both shift left and shift right, the bits that are shifted out are lost.
- For shift left, 0's are shifted in (Zero Fill).
Operators
Shift – Special Cases

• Logical Shift Right (Zero Fill)

If x is **UNSIGNED** (unsigned char in this case):

\[x = 250; \quad // \quad x = 0b11111010 = 250\]

\[y = x >> 2; \quad // \quad y = 0b00111110 = 62\]

• Arithmetic Shift Right (Sign Extend)

If x is **SIGNED** (char in this case):

\[x = -6; \quad // \quad x = 0b11111010 = -6\]

\[y = x >> 2; \quad // \quad y = 0b11111110 = -2\]
Operators

Power of 2 Integer Divide vs. Shift Right

- If you are dividing by a power of 2, it will usually be more efficient to use a right shift instead

\[ y = \frac{x}{2^n} \quad \rightarrow \quad y = x >> n \]

- Works for integers or fixed point values
# Operators

## Memory Addressing

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Address of</td>
<td>&amp;x</td>
<td>Pointer to x</td>
</tr>
<tr>
<td>*</td>
<td>Indirection</td>
<td>*p</td>
<td>The object or function that p points to</td>
</tr>
<tr>
<td>[ ]</td>
<td>Subscripting</td>
<td>x[y]</td>
<td>The y(^{th}) element of array x</td>
</tr>
<tr>
<td>.</td>
<td>Struct / Union Member</td>
<td>x.y</td>
<td>The member named y in the structure or union x</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Struct / Union Member by Reference</td>
<td>p-&gt;y</td>
<td>The member named y in the structure or union that p points to</td>
</tr>
</tbody>
</table>

These operators will be discussed later in the sections on arrays, pointers, structures, and unions. They are included here for reference and completeness.
# Operators

## Other

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Function Call</td>
<td><code>foo(x)</code></td>
<td>Passes control to the function with the specified arguments</td>
</tr>
<tr>
<td><code>sizeof</code></td>
<td>Size of an object or type in bytes</td>
<td><code>sizeof x</code></td>
<td>The number of bytes <code>x</code> occupies in memory</td>
</tr>
<tr>
<td><code>(type)</code></td>
<td>Explicit type cast</td>
<td><code>(short) x</code></td>
<td>Converts the value of <code>x</code> to the specified type</td>
</tr>
<tr>
<td><code>?:</code></td>
<td>Conditional expression</td>
<td><code>x ? y : z</code></td>
<td>The value of <code>y</code> if <code>x</code> is true, else value of <code>z</code></td>
</tr>
<tr>
<td><code>,</code></td>
<td>Sequential evaluation</td>
<td><code>x, y</code></td>
<td>Evaluates <code>x</code> then <code>y</code>, else result is value of <code>y</code></td>
</tr>
</tbody>
</table>
Operators
The Explicit Type Cast Operator

• Earlier, we cast a literal to type float by entering it as: 4.0f

• We can cast the variable instead by using the cast operator: (type)variable

---

**Example: Integer Divide**

```java
int x = 10;
float y;
y = x / 4;
```

**Example: Floating Point Divide**

```java
int x = 10;
float y;
y = (float)x / 4;
```

y = 2.000000  ❌
Because: int / int → int

y = 2.500000  ✓
Because: float / int → float
## Operators

### Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(  )</td>
<td>Parenthesized Expression</td>
<td></td>
</tr>
<tr>
<td>[  ]</td>
<td>Array Subscript</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>.</td>
<td>Structure Member</td>
<td></td>
</tr>
<tr>
<td>-&gt;</td>
<td>Structure Pointer</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>Unary + and − (Positive and Negative Signs)</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>Increment and Decrement</td>
<td></td>
</tr>
<tr>
<td>! ~</td>
<td>Logical NOT and Bitwise Complement</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Dereference (Pointer)</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>&amp;</td>
<td>Address of</td>
<td></td>
</tr>
<tr>
<td>sizeof</td>
<td>Size of Expression or Type</td>
<td></td>
</tr>
<tr>
<td>(type)</td>
<td>Explicit Typecast</td>
<td></td>
</tr>
</tbody>
</table>

*Continued on next slide...*
# Operators

## Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>* / %</td>
<td>Multiply, Divide, and Modulus</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>+ -</td>
<td>Add and Subtract</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>Shift Left and Shift Right</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>&lt; &lt;=</td>
<td>Less Than and Less Than or Equal To</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>&gt; &gt;=</td>
<td>Greater Than and Greater Than or Equal To</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>== !=</td>
<td>Equal To and Not Equal To</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>Conditional Operator</td>
<td>Right-to-Left</td>
</tr>
</tbody>
</table>

*Continued on next slide...*
## Operators

### Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Assignment</td>
<td></td>
</tr>
<tr>
<td>+=  -=</td>
<td>Addition and Subtraction Assignments</td>
<td></td>
</tr>
<tr>
<td>/=  *=</td>
<td>Division and Multiplication Assignments</td>
<td></td>
</tr>
<tr>
<td>%=</td>
<td>Modulus Assignment</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>&lt;&lt;=  &gt;&gt;=</td>
<td>Shift Left and Shift Right Assignments</td>
<td></td>
</tr>
<tr>
<td>&amp;=</td>
<td>=</td>
<td>Bitwise AND and OR Assignments</td>
</tr>
<tr>
<td>^=</td>
<td>Bitwise XOR Assignment</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>Comma Operator</td>
<td>Left-to-Right</td>
</tr>
</tbody>
</table>

- Operators grouped together in a section have the same precedence – conflicts within a section are handled via the rules of associativity.
Operator Notes

• Most operators look just like their normal mathematical notation
• C adds several shortcut operators in the form of compound assignments
• Most C programmers tend to use the shortcut operators

\[
\begin{align*}
x &= x + 5 \\
x &= 4 = 5
\end{align*}
\]
Questions?
AND

\[ \begin{array}{c|cc}
T & F \\
F & F \\
\end{array} \]

\text{OR}

\[ \begin{array}{c|ccc}
T & T & T \\
T & T & T \\
F & T & F \\
\end{array} \]

! (not)

\[ \begin{array}{c|c}
\phi & \text{FALSE} \\
\end{array} \]

\text{not } \phi \quad \text{TRUE}
LAB
BE-113
NO FOOD/DRINK IN LAB
DON'T BLOCK DOORS