Strings
How to Initialize a String at Declaration
Character arrays may be initialized with string literals:

**Syntax**

```c
char arrayName[] = "Microchip";
```

- Array size is not required
- Size automatically determined by length of string
- NULL character '\0' is automatically appended

**Example**

```c
char str1[] = "Microchip"; // 10 chars "Microchip\0"
char str2[6] = "Hello";    // 6 chars "Hello\0"

// Alternative string declaration
char str3[] = {'P', 'I', 'C', '\0'};
```
Strings

How to Initialize a String in Code

In code, strings must be initialized element by element:

Syntax

```c
arrayName[0] = char1;
arrayName[1] = char2;
... 
arrayName[n] = '\0';
```

- Null character '\0' must be appended manually

Example

```c
str[0] = 'H';
str[1] = 'e';
str[2] = 'l';
str[3] = 'l';
str[4] = 'o';
str[5] = '\0';
```

Strings

Comparing Strings

- Strings cannot be compared using relational operators (==, !=, etc.)
- Must use standard C library string manipulation functions
  
  ```c
  char str[] = "Hello";
  if (!strcmp(str, "Hello")) {
      printf("The string is \"%s\".\n", str);
  }
  ```

  `strcmp()` returns 0 if strings equal
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>$x \times y$</td>
<td>Product of $x$ and $y$</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>$x \div y$</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

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: +, -, ++, --

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

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

```c
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

Implicit Arithmetic Type Conversion Hierarchy

Smaller types converted to largest type in expression

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

Arithmetic Expression Implicit Type Conversion

• Example implicit type conversions

Assume \( x \) is defined as:
\[
\text{short } x = -5;
\]

<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>( 8.0 / 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: \( x \% 2^n \) where \( n \) is the desired word width
  (e.g. 8 for 8 bits: \( x \% 256\))
  – 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

\[
\text{long number = 123456;}
\text{int i, radix = 10;}
\text{char digits[6];}
\text{for (i = 0; i < 6; ++i) {}
\text{  if (number == 0) {}
\text{    break;}
\text{  }\}
\text{  digits[i] = (char)(number \% radix);}
\text{  number = number / radix;}
\text{}}
\]
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></td>
<td>++x</td>
<td>Increment (x) by 1, then use (x)</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
<td>x--</td>
<td>Use (x) then decrement (x) by 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--x</td>
<td>Decrement (x) by 1, then use (x)</td>
</tr>
</tbody>
</table>

Postfix Example

\[
\begin{align*}
& x = 5; \\
& y = (x++) + 5; \\
& // y = 10 \\
& // x = 6
\end{align*}
\]

Prefix Example

\[
\begin{align*}
& x = 5; \\
& y = (++x) + 5; \\
& // y = 11 \\
& // x = 6
\end{align*}
\]

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>x = y</td>
<td>Assign x the value of y</td>
</tr>
<tr>
<td>+=</td>
<td></td>
<td>x += y</td>
<td>x = x + y</td>
</tr>
<tr>
<td>-=</td>
<td></td>
<td>x -= y</td>
<td>x = x - y</td>
</tr>
<tr>
<td>*=</td>
<td></td>
<td>x *= y</td>
<td>x = x * y</td>
</tr>
<tr>
<td>/=</td>
<td></td>
<td>x /= y</td>
<td>x = x / y</td>
</tr>
<tr>
<td>%=</td>
<td>Compound</td>
<td>x %= y</td>
<td>x = x % y</td>
</tr>
<tr>
<td>&amp;=</td>
<td>Assignment</td>
<td>x &amp;= y</td>
<td>x = x &amp; y</td>
</tr>
<tr>
<td>^=</td>
<td></td>
<td>x ^= y</td>
<td>x = x ^ y</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td></td>
<td>x &lt;&lt;= y</td>
<td>x = x &lt;&lt; y</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td></td>
<td>x &gt;&gt;= y</td>
<td>x = x &gt;&gt; y</td>
</tr>
</tbody>
</table>

---

### Operators

#### Compound Assignment

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

  ```c
  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):

  ```c
  x += y; // Increment x by the value y
  ```
Operators

Compound Assignment

Example

```c
int x = 2;
x *= 5;
```

```
x = 10
```

Example

```c
int x = 2, y = 6;
x *= (5 - y);
```
Operators

Relational

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

```c
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!"
    }
}
```

Logical Operators

<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 ≠ 0 and y ≠ 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
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:

\[ expr1 \land expr2 \]

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.

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

Truth Table for AND (&&)
FALSE = 0
TRUE = 1

 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

If \( z = 0 \), then \( c \) will not be evaluated

\[
\textbf{if} \ ( (z = x + y) \land (c = a + b)) \\
\{ \\
\textbf{z +=} 5; \\
\textbf{c +=} 10; \textit{Initial value of c may not be correct} \\
\}
\]

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

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

- The operation is carried out on each bit of the first operand with each corresponding bit of the second operand.

Operators

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
  \[ \text{0b1010} && \text{0b1101} \Rightarrow \text{0b0001} \text{ (TRUE)} \]

  ```c
  if (x && y) {
    do if x and y 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

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:

```c
x = 5;  // x = 0b00000101 = 5
y = x << 2;  // 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

• 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

\[
\begin{align*}
  y &= x / 2^n \\
  y &= x >> n
\end{align*}
\]

- Works for integers or fixed point values

\[
\begin{array}{ccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 \quad \text{Right Shift} \quad \begin{array}{ccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{array}
\end{array}
\]
Operators

Power of 2 Integer Divide vs. Shift in MPLAB®

Example: Divide by 2

```c
int x = 20;
int y;
y = x / 2;  // y = 10
```

Example: Right Shift by 1

```c
int x = 20;
int y;
y = x >> 1;  // y = 10
```

16-Bit Shift on 8-Bit Architecture
## 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><code>&amp;x</code></td>
<td>Pointer to x</td>
</tr>
<tr>
<td>*</td>
<td>Indirection</td>
<td><code>*p</code></td>
<td>The object or function that p points to</td>
</tr>
<tr>
<td>[]</td>
<td>Subscripting</td>
<td><code>x[y]</code></td>
<td>The y(^{th}) element of array x</td>
</tr>
<tr>
<td>.</td>
<td>Struct / Union Member</td>
<td><code>x.y</code></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><code>p-&gt;y</code></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 pointers, structures, and unions. They are included here for reference and completeness.

### 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 x 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 x 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 y if x is true, else value of z</td>
</tr>
<tr>
<td><code>:</code></td>
<td>Sequential evaluation</td>
<td><code>x, y</code></td>
<td>Evaluates x then y, else result is value of y</td>
</tr>
</tbody>
</table>
Operators

The Conditional Operator

- The conditional operator may be used to conditionally assign a value to a variable

**Example 1 (most commonly used)**

```c
int x = 5;
(x % 2 != 0) ? printf("%d is odd\n", x) : printf("%d is even\n", x);
```

Result:

5 is odd

**Example 2 (less often used)**

```c
(condition) ? (x = a):(x = b);
```

In both cases:

- `x = a` if condition is true
- `x = b` if condition is false
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

<table>
<thead>
<tr>
<th>Example: Integer Divide</th>
<th>Example: Floating Point Divide</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x = 10;</td>
<td>int x = 10;</td>
</tr>
<tr>
<td>float y;</td>
<td>float y;</td>
</tr>
<tr>
<td>y = x / 4;</td>
<td>y = (float)x / 4;</td>
</tr>
<tr>
<td>y = 2.000000 ×</td>
<td>y = 2.500000 ✓</td>
</tr>
<tr>
<td></td>
<td>Because: int / int = int</td>
</tr>
<tr>
<td></td>
<td>Because: float / int = float</td>
</tr>
</tbody>
</table>

Operators
The Conditional Operator

<table>
<thead>
<tr>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>float x = 5;</td>
</tr>
<tr>
<td>printf(&quot;%f\n&quot;, x);</td>
</tr>
</tbody>
</table>

Result:
warning: format ‘%f’ expects type ‘double’, but argument 2 has type ‘float’
Example

```c
float x = 5;

printf("%f\n", (double)x);
```

Result:

No warnings!

---

Operators

The Conditional Operator

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>( )</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>( )</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;= &gt; &gt;=</td>
<td>Less Than and Less Than or Equal To 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 and OR Assignments</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise XOR Assignment</td>
</tr>
<tr>
<td>&amp;&amp;</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 grouped together in a section have the same precedence – conflicts within a section are handled via the rules of associativity.
Operators

Precedence

• When expressions contain multiple operators, their precedence determines the order of evaluation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effective Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>a - b * c</td>
<td>a - (b * c)</td>
</tr>
<tr>
<td>a + ++b</td>
<td>a + (++b)</td>
</tr>
<tr>
<td>a + ++b * c</td>
<td>a + ((++b) * c)</td>
</tr>
</tbody>
</table>

If functions are used in an expression, there is no set order of evaluation for the functions themselves.

e.g. \( x = f() + g() \)
There is no way to know if \( f() \) or \( g() \) will be evaluated first.

Operators

Associativity

• If two operators have the same precedence, their associativity determines the order of evaluation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Associativity</th>
<th>Effective Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x / y \ % z )</td>
<td>Left-to-Right</td>
<td>((x / y) \ % z)</td>
</tr>
<tr>
<td>( x = y = z )</td>
<td>Right-to-Left</td>
<td>(x = (y = z))</td>
</tr>
<tr>
<td>( \sim++x )</td>
<td>Right-to-Left</td>
<td>(\sim (++x))</td>
</tr>
</tbody>
</table>

• You can rely on these rules, but it is good programming practice to explicitly group elements of an expression
Loop Structures (cont'd)

for
do-while

Syntax

for (expression\textsubscript{1}; expression\textsubscript{2}; expression\textsubscript{3})

statement

- expression\textsubscript{1} initializes a loop count variable once at start of loop (e.g. \texttt{i = 0})
- expression\textsubscript{2} is the test condition – the loop will continue while this is true (e.g. \texttt{i <= 10})
- expression\textsubscript{3} is executed at the end of each iteration – usually to modify the loop count variable (e.g. \texttt{i++})
**for Loop**

**Syntax**

```
for (expression_1; expression_2; expression_3)
  statement
```

**Flow Diagram**

- **Initialize loop variable**
  
  - `i = 0`
  
  - `expression_1`

- **Modify loop variable**
  
  - `expression_3`
  
  - `i++`

- **Test loop variable for exit condition**
  
  - `i < n`
  
  - `expression_2`

- **Loop iteration**

**Example (Code Fragment)**

```
int i;
int i;

for (i = 0; i < 5; i++) {
    printf("Loop iteration %d\n", i);
}
```

**Expected Output:**

```
Loop iteration 0
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
```
**for Loop**

- Any or all of the three expressions may be left blank (semi-colons must remain)
- If $expression_1$ or $expression_3$ are missing, their actions simply disappear
- If $expression_2$ is missing, it is assumed to always be true

---

**Example (Code Fragment)**

```c
FILE *f = fopen("myfile.txt", "r");
char c;
for (c = getc(f); c != EOF; c = getc(f)) {
    printf("Char: '%c'\n", c);
}  // Got - END OF FILE
```
**do-while Loop**

**Syntax**

```c
do statement while (expression);
```

- `statement` is executed and then `expression` is evaluated to determine whether or not to execute `statement` again.
- `statement` will always execute at least once, even if the expression is false when the loop starts.

---

**Flow Diagram**

![Flow Diagram](image)

- **START**
- **statement**
- **expression?**
  - **TRUE**
  - **FALSE**
- **END**
### do-while Loop

#### Trivial example

**Code Fragment**

```c
int i = 0;  // Loop counter initialized outside of loop

do {
    printf("Loop iteration %d\n", ++i);
} while (i < 5);  // Condition checked at end of loop iterations
```

**Expected Output:**

```
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
Loop iteration 5
```

---

#### Useful example

**Code Fragment**

```c
int numInputs;
float input1, input2;

do {
    printf("Enter two numbers: \n");
    numInputs = scanf("%f %f", &input1, &input2);
    fflush(stdin);
} while (numInputs != 2);
```
**break Statement**

Syntax

```c
break;
```

- Causes immediate termination of a loop even if the exit condition hasn't been met
- Also exits from a `switch` statement

Flow Diagram Within a `while` Loop

START

expression?

TRUE

statement

FALSE

break

statement

END
### break Statement

**Example**

```c
int i = 0;

while (i < 10) {
    ++i;
    if (i == 5) {
        break;
    }
    printf("Loop iteration %d\n", i);
}
```

**Expected Output:**
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4

Exit from the loop when i = 5.
Iteration 6-9 will not be executed.

### continue Statement

**Syntax**

```c
continue;
```

- Causes program to finish current iteration and begin the next loop
**continue Statement**

**Syntax**
```
continue;
```

**Flow Diagram Within a while Loop**

**Example (Code Fragment)**
```
int i = 0;

while (i < 6) {
    ++i;
    if (i == 2) {
        continue;
    }
    printf("Loop iteration %d\n", i);
}
```

Expected Output:
- Loop iteration 1
- Loop iteration 3
- Loop iteration 4
- Loop iteration 5

*Skip remaining iteration when i = 2. Iteration 2 will not be completed.*
Unit testing

- Testing portions of code in isolation
- Normally testing is per function
- Requires input and expected output to be known a priori
Unit testing

Rationale

- Find problems early
  - Before integration
- Simplify testing by only testing small, segmented portions of code
- Test functionality that may not be exposed otherwise
- Find documentation errors

Unit testing

Preparing

- The most important question:
  "How am I going to test this?"
- Break code into clean functions with:
  - Clear input
  - Clear output
  - No/minimal side effects
Unit testing
Testing architecture

- Known input
- Expected output
- Output matches
- Actual output
- f()
- Print failure
  - no
- Print success
  - yes

Print failure
Print success

Quiz Today!!
(1) Lab 3 - Matrix Math

START IT NOW!!

(2) Drop deadline is on MONDAY