CMPE-013/L

Operators

Maxwell James Dunne
### Operators

#### Arithmetic

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
<td><code>x * y</code></td>
<td>Product of <code>x</code> and <code>y</code></td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
<td><code>x / y</code></td>
<td>Quotient of <code>x</code> and <code>y</code></td>
</tr>
<tr>
<td><code>%</code></td>
<td>Modulo</td>
<td><code>x % y</code></td>
<td>Remainder of <code>x</code> divided by <code>y</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td>Addition</td>
<td><code>x + y</code></td>
<td>Sum of <code>x</code> and <code>y</code></td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
<td><code>x - y</code></td>
<td>Difference of <code>x</code> and <code>y</code></td>
</tr>
<tr>
<td><code>+</code></td>
<td>Positive</td>
<td><code>+x</code></td>
<td>Value of <code>x</code></td>
</tr>
<tr>
<td><code>-</code></td>
<td>Negative</td>
<td><code>-x</code></td>
<td>Negative value of <code>x</code></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
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
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` ×
- Because: `int / int` ➞ `int`

Example: Floating Point Divide

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

- `c = 2.500000` ✓
- Because: `float / int` ➞ `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.
Operators

Arithmetic Expression Implicit Type Conversion

- Example implicit type conversions

Assume `x` is defined as:

```c
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><code>-x</code></td>
<td><code>x</code> is promoted to <code>int</code></td>
<td><code>int</code></td>
<td>5</td>
</tr>
<tr>
<td><code>x * -2L</code></td>
<td><code>x</code> is promoted to <code>long</code> because <code>-2L</code> is a <code>long</code></td>
<td><code>long</code></td>
<td>10</td>
</tr>
<tr>
<td><code>8 / x</code></td>
<td><code>x</code> is promoted to <code>int</code></td>
<td><code>int</code></td>
<td>-1</td>
</tr>
<tr>
<td><code>8 % x</code></td>
<td><code>x</code> is promoted to <code>int</code></td>
<td><code>int</code></td>
<td>3</td>
</tr>
<tr>
<td><code>8.0 / x</code></td>
<td><code>x</code> is promoted to <code>double</code> because <code>8.0</code> is a <code>double</code></td>
<td><code>double</code></td>
<td>-1.6</td>
</tr>
</tbody>
</table>
$\left(\frac{2}{3}\right), 4.8;$

$\left(\frac{2.0}{3.0}\right), 4.8;$

$\frac{2}{3} \quad 0 \cdot 4.8$
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

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

for (i = 0; i < 6; i++) {
    if (number == 0) {
        break;
    }
    digits[i] = (char)(number % radix);
    number = number / radix;
}
```

$123456 \% 10 = 6$

$123456 / 10 = 12345$
## 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>++x</td>
<td></td>
<td></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>--x</td>
<td></td>
<td></td>
<td>Decrement x by 1, then use x</td>
</tr>
</tbody>
</table>

### Postfix Example

\[
x = 5; \\
y = (x++) + 5; \\
// y = 10 \\
// x = 6
\]

### Prefix Example

\[
x = 5; \\
y = (x++) + 5; \\
// y = 11 \\
// x = 6
\]
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
    \[ \text{variable} = \text{variable} \text{ op } \text{expression}; \]
    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><code>=</code></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><code>+=</code></td>
<td></td>
<td><code>x += y</code></td>
<td><code>x = x + y</code></td>
</tr>
<tr>
<td><code>-=</code></td>
<td></td>
<td><code>x -= y</code></td>
<td><code>x = x - y</code></td>
</tr>
<tr>
<td><code>*=</code></td>
<td></td>
<td><code>x *= y</code></td>
<td><code>x = x * y</code></td>
</tr>
<tr>
<td><code>/=</code></td>
<td></td>
<td><code>x /= y</code></td>
<td><code>x = x / y</code></td>
</tr>
<tr>
<td><code>%=</code></td>
<td>Compound</td>
<td><code>x %= y</code></td>
<td><code>x = x % y</code></td>
</tr>
<tr>
<td><code>&amp;=</code></td>
<td>Assignment</td>
<td><code>x &amp;= y</code></td>
<td><code>x = x &amp; y</code></td>
</tr>
<tr>
<td><code>^=</code></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><code>&lt;&lt;=</code></td>
<td></td>
<td><code>x &lt;&lt;= y</code></td>
<td><code>x = x &lt;&lt; y</code></td>
</tr>
<tr>
<td><code>&gt;&gt;=</code></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

```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):

Example

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

Example

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

Compound Assignment

Example

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

```
x = x * 5;
x = x * (5 + q);
x *= 5 + q;
```

Maxwell James Dunne
# 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 equal to</td>
<td>x &lt;= y</td>
<td>1 if x less than or equal to y, else 0</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 equal to</td>
<td>x &gt;= y</td>
<td>1 if x greater than or equal to y, else 0</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

\[
\text{if} \ (x == 5) \ \{ \\
    \text{do if value of } x \text{ is 5} \\
\}\]
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
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>Truth Table for AND (( \land ))</th>
<th>expr1</th>
<th>expr2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE = 0</td>
<td>0</td>
<td>( X(0) )</td>
<td>0</td>
</tr>
<tr>
<td>TRUE = 1</td>
<td>0</td>
<td>( X(1) )</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<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 = x + y) && (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

## Bitwise

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result (for each bit 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 0, if 0 in x or y or both</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1, if 1 in x or y or both 0, if 0 in both x and y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1, if 1 in x or y but not both 0, if 0 or 1 in both x and y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1, if 0 in x 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

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

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

- 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.

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

\[ \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
\end{array} \quad \gg \quad \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
\end{array} \]

- Works for integers or fixed point values.

Maxwell James Dunne

CMPE-013/L: “C” Programming
Operators

Power of 2 Integer Divide vs. Shift in MPLAB® C30

**Example: Divide by 2**

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

```
y = 10
```

```
10:        y = x / 2;
00288  804000  mov.w 0x0800,0x0000
0028A  200022  mov.w #0x2,0x0000
0028C  090011  repeat #17
0028E  D80002  div.sw 0x0000,0x0004
00290  884010  mov.w 0x0000,0x0802
```

**Example: Right Shift by 1**

```
int x = 20;
int y;
y = x >> 1;
```

```
y = 10
```

```
9:        y = x >> 1;
00282  804000  mov.w 0x0800,0x0000
00284  DE8042  asr 0x0000,#1,0x0000
00286  884010  mov.w 0x0000,0x0802
```
# Operators

## Power of 2 Integer Divide vs. Shift in MPLAB® C18

### 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**

---

Maxwell James Dunne
division 32
multiplication single cycle
X / 113

X \* Large constant;

\[
\begin{array}{c|c}
32 & 32 \\
\hline
\end{array}
\]

X / 113
## 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 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 Conditional Operator

Syntax

`(test-expr) ? do-if-true : do-if-false;`

Example

```c
int x = 5;

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

Result:

5 is odd
Operators
The Conditional Operator

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

**Example 1 (most commonly used)**

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

**Example 2 (less often used)**

```
(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: \((\text{\textit{type}})\text{\textit{variable}}\)

**Example: Integer Divide**

```c
int x = 10;
float y;

y = x / 4;
```

\[ y = 2.000000 \quad \times \]

Because: int / int \( \Rightarrow \) int

**Example: Floating Point Divide**

```c
int x = 10;
float y;

y = (float)x / 4;
```

\[ y = 2.500000 \quad \checkmark \]

Because: float / int \( \Rightarrow \) float
Operators
The Conditional Operator

Example

```c
float x = 5;

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

Result:

```
warning: format '\%f' expects type 'double', but argument 2 has type 'float'
```
Operators
The Conditional Operator

Example

```c
float x = 5;

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

Result:

No warnings!
### 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></td>
</tr>
<tr>
<td></td>
<td>Structure Member</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td></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...*
Quiz 1 and 2 after class
# 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><code>+=</code></td>
<td>Addition and Subtraction Assignments</td>
<td></td>
</tr>
<tr>
<td><code>-=</code></td>
<td>Subtraction Assignments</td>
<td></td>
</tr>
<tr>
<td><code>/=</code></td>
<td>Division and Multiplication Assignments</td>
<td></td>
</tr>
<tr>
<td><code>%=</code></td>
<td>Modulus Assignment</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td><code>&lt;&lt;=</code></td>
<td>Shift Left and Shift Right Assignments</td>
<td></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Bitwise AND and OR Assignments</td>
<td></td>
</tr>
<tr>
<td><code>^=</code></td>
<td>Bitwise XOR Assignment</td>
<td></td>
</tr>
<tr>
<td><code>,</code></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.
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 \times c )</td>
<td>( a - (b \times c) )</td>
</tr>
<tr>
<td>( a + ++b )</td>
<td>( a + (++b) )</td>
</tr>
<tr>
<td>( a + ++b \times c )</td>
<td>( a + ((++b) \times 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>( \frac{x}{y} &amp; z )</td>
<td>Left-to-Right</td>
<td>((\frac{x}{y}) &amp; 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.
CMPE-013/L

Loop Structures (cont'd)

Maxwell James Dunne
for

do-while
for Loop

Syntax

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

- `expression_1` initializes a loop count variable once at start of loop (e.g. `i = 0`)
- `expression_2` is the test condition – the loop will continue while this is true (e.g. `i <= 10`)
- `expression_3` is executed at the end of each iteration – usually to modify the loop count variable (e.g. `i++`)
for Loop

Flow Diagram

Syntax

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

Initialize loop variable

```
i = 0
```

Modify loop variable

```
i++
```

Test loop variable for exit condition

```
i < n
```

TRUE

END
for Loop

Example (Code Fragment)

```c
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

**Note**

Infinite Loops

A `for` loop without any expressions will execute indefinitely (can leave loop via `break` statement)

```
for ( ; ; ) {
    ...
}
```
for Loop

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);
}
```
do-while Loop

Syntax

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
do-while Loop

Flow Diagram

Syntax

```
do statement while (expression);
```
**do-while** Loop

Trivial example

Example (Code Fragment)

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

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

Expected Output:

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

1.5 loop
do-while Loop

Useful example

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

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

```c
while (numInputs)
```
break Statement

Syntax

break;

• Causes immediate termination of a loop even if the exit condition hasn't been met

• Also exits from a switch statement
break Statement

Flow Diagram Within a while Loop

Syntax

```
break;
```
**break Statement**

**Example**

```c
int i = 0;

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

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

**Expected Output:**

```
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
```
continue Statement

Syntax

```
continue;
```

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

Flow Diagram Within a `while` Loop

**Syntax**

```
continue;
```

Diagram:

1. **START**
2. **expression?**
   - **TRUE** → **statement** → **END**
   - **FALSE** → **continue** → **statement** → **END**
**continue Statement**

Example

```c
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
```

Iteration 2 does not print.
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
Unit testing
Unit testing

• Testing portions of code in isolation

• Normally testing is per function

• Requires input and expected output to be known a priori

\[ 3 + 4 = 7 \]
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

/0

corner cases

6 hours
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

Example

```
// Declare test constants
testInput ← some input
testExpOutput ← precalculated output

// Calculate result
testActOutput ← function result

// Output test results
if testActOutput equals testExpOutput
   output "Test passed"
else
   output "Test failed!"
```
Unit testing
Trivial example

ExampleLib.c

```c
int AddFive(int x)
{
    return x + 5;
}
```

main.c

```c
#include "ExampleLib.h"

int main(void)
{
    // Declare test constants
    int test1Input = 0;
    int test1ExpOutput = 5;

    // Calculate result
    int test1ActOutput;
    test1ActOutput = AddFive(test1Input);

    // Output test results
    if (test1ActOutput == test1ExpOutput) {
        printf("Test1 passed.\n");
    } else {
        printf("Test1 failed!\n");
    }
}
```
Unit testing

Writing tests

• Write multiple tests
  – At least 1 for every group of inputs
  – Each edge case should have their own test

• Each test should check **one** part of the total functionality
  – One function or logical block of code at a time

Try to break the code you're testing!
Unit testing

Testing framework

• Track how many tests passed/failed
  – Per function
• Track how many functions passed/failed
  – With all tests must pass for the function to pass
• Each test cleanly separated from other tests
  – Both in code and in logic
• Output results
  – Per function/per test results
MatrixEquals(A, B)

return True;

2 tests

True

True

False

#define A_SIZE 3

Multiply
Parameter passing

Pass by value
Pass by reference
Parameter Passing

By Value

- Parameters passed to a function are generally passed by value.
- Values passed to a function are copied into the local parameter variables.
- The original variable that is passed to a function cannot be modified by the function since the function has a duplicate of the variable, not the original.
Parameter Passing

By Value

Example

```c
int a, b, c;

int Foo(int x, int y)
{
    x = x + (y++);
    return x;
}

int main(void)
{
    a = 5;
    b = 10;
    c = Foo(a, b);
}
```

The **value of a** is **copied** into `x`.
The **value of b** is **copied** into `y`.
The function does not change the value of `a` or `b`. 
Parameter Passing

By Value

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Parameter Passing
By Reference

- Parameters can be passed to a function by **reference**
- Entails passing around memory address
- The original variable that is passed to a function **can** be modified by the function since the function knows where the data "lives" in memory
Parameter Passing

By Reference

Example function

```c
int Foo(int x[3])
{
    int z = x[2];
    x[1] = 0;
    return z;
}
```

Example main

```c
int main(void)
{
    int a[3] = {6, 19, -1};
    Foo(a);
    while (1);
}
```
Scope
Scope

Variables Declared Within a Function

- Variables declared within a code block are local to that block.

Example

```c
int x, y, z;

int Foo(int n)
{
    int a;
    ...
    a += n;
}
```

- The `n` refers to the function parameter `n`
- The `a` refers to the `a` declared locally within the function body
Scope
Variables Declared Within a Function

- Variables declared within a **block** are not accessible outside that block

Example

```c
int x;
int Foo(int n)
{
    int a;
    return (a += n);
}
int main(void)
{
    x = Foo(5);
    x = a;  // This will generate an error. a may not be accessed outside of the scope where it was declared.
}
```
Scope
Variables Declared Within a Function

- Variables declared within a block are not accessible outside that block

Example

```c
int x;
int main(void)
{
    {
        int a = 6;  // This will generate an error. a may not be accessed outside of the scope where it was declared.
    }

    x = Foo(5);
    x = a;
}
```
Scope
And the stack

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Scope
And the stack

Example function

```c
int Foo(int x, int y)
{
    int z = x + (y++);
    return z;
}
```

Example main

```c
int main(void)
{
    int a = 6, b = 19;
    Foo(a, b);
    while (1);
}
```
Scope

Global versus Local Variables

Example

```c
int x = 5;

int Foo(int y)
{
    int z = 1;
    return (x + y + z);
}

int main(void)
{
    int a = 2;
    x = foo(a);
    a = foo(x);
}
```

- `x` can be seen by everybody
- `foo`'s local parameter is `y`
- `foo`'s local variable is `z`
- `foo` cannot see `main`'s `a`
- `foo` can see `x`
- `main`'s local variable is `a`
- `main` cannot see `foo`'s `y` or `z`
- `main` can see `x`
### Scope

**Parameters**

- "Overloading" variable names:

<table>
<thead>
<tr>
<th>Declared Locally and Globally</th>
<th>Declared Globally Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>int n;</td>
<td>int n;</td>
</tr>
<tr>
<td>int Foo(int n)</td>
<td>int Foo(int x)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>y += n;</td>
<td>y += n;</td>
</tr>
<tr>
<td>local n hides global n</td>
<td></td>
</tr>
</tbody>
</table>

A locally defined identifier takes precedence
Scope
Parameters

Example

```c
int n;

int Foo(int n)
{
    y += n;
}

int Bar(int n)
{
    z *= n;
}
```

- Different functions may use the same parameter names
- The function will only use its own parameter by that name
Scope
Preprocessor and scoping

Example

```c
#define x 2

void Test(void)
{
    #define x 5
    printf("%d\n", x);
}

void main(void)
{
    printf("%d\n", x);
    Test();
}
```

Result:

```
5
5
```
Storage Class Specifiers
Scope and Lifetime of Variables

• Scope and lifetime of a variable depends on its storage class:
  – Automatic Variables
  – Static Variables
  – External Variables
  – Register Variables

• Scope refers to where in a program a variable may be accessed
  Lifetime refers to how long a variable will exist or retain its value
Storage Class Specifiers
Automatic Variables

• Local variables declared inside a function
  – Created when function called
  – Destroyed when exiting from function
• `auto` keyword *usually* not required – local variables are automatically `auto` *Except when the compiler provides an option to make parameters and locals static by default.*
• Typically created on the stack

```
int Foo(int x, int y)
{
    int a, b;
    ...
```
Storage Class Specifiers

*auto* Keyword with Variables

```c
int Foo(auto int x, auto int y)
{
    ...
}
```

- *auto* is almost never used
- Many books claim it has no use at all
- Some compilers still use *auto* to explicitly specify that a variable should be allocated on the stack when a different method of parameter passing is used by default
Storage Class Specifiers

Static Variables

• Given a permanent address in memory
• Exist for the entire life of the program
  – Created when program starts
  – Destroyed when program ends
• Global variables are always static (cannot be made automatic using auto)

```c
int x; // Global variable is always static

int main(void)
{
    ...
```
Storage Class Specifiers

*static* Keyword with Variables

- A variable declared as *static* inside a function retains its value between function calls (not destroyed when exiting function)
- Function parameters cannot be *static* with some compilers (XC32)

```c
int Foo(int x)
{
    static int a = 0;
    ...
    a += x;
    return a;
}
```

`a` will remember its value from the last time the function was called. If given an initial value, it is only initialized when first created — not during each function call.
Storage Class Specifiers

External Variables

- Variables that are *defined* outside the scope where they are used
- Still need to be *declared* within the scope where they are used
- `extern` keyword used to tell compiler that a variable defined elsewhere will be used within the current scope

External Variable Declaration Syntax:

```
extern type identifier;
```

External Variable Declaration Example:

```
extern int x;
```
Storage Class Specifiers

External Variables

- A variable declared as `extern` within a function is analogous to a function prototype – the variable may be defined outside the function after it is used.

Example

```c
int Foo(int x)
{
    extern int a;
    ...
    return a;
}

int a;
```
A variable *declared* as `extern` outside of any function is used to indicate that the variable is *defined* in another source file – memory only allocated when it's *defined*.
Storage Class Specifiers

Register Variables

- **register** variables are placed in a processor's "hardware registers" for higher speed access than with external RAM
  - Common with loop counters
- Not as important when RAM is integrated into processor package (microcontrollers, ...)
- May be done with PIC®/dsPIC®, but it is architecture/compiler specific...
Storage Class Specifiers

Scope of Functions

• Scope of a function depends on its storage class:
  – Static Functions
  – External Functions

• Scope of a function is either local to the file where it is defined (static) or globally available to any file in a project (external)
Storage Class Specifiers

External Functions

- Functions by default have global scope within a project
- `extern` keyword not required, but function prototype is required in calling file

```c
Main.c

int foo(void);

int main(void)
{
    ...
    x = foo();
}

SomeFileInProject.c

int foo(void)
{
    ...
}
```
Storage Class Specifiers

Static Functions

- If a function is declared as `static`, it will only be available within the file where it was declared (makes it a local function)

```c
Main.c

int foo(void);

int main(void)
{
    ...
    x = foo();
}

SomeFileInProject.c

static int foo(void)
{
    ...
}
```
Storage Class Specifiers

Static Functions

- If a variable is declared as `static`, it will only be available within the file where it was declared.

```c
extern int myVar;

int main(void)
{
    ...  // This line is not executed
    myVar = 6;
}

static int myVar = 0;
```