CMPE-013/L

Introduction to “C” Programming

Bryant Wenborg Mairs

Spring 2014
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] = char_1;
arrayName[1] = char_2;
...;
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
- `strcmp()` returns 0 if strings equal

**Example**

```c
char str[] = "Hello";

if (!strcmp(str, "Hello")) {
    printf("The string is \"%s\".\n", str);
}
```
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 \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 \mod 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
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

Implicit Arithmetic Type Conversion Hierarchy

- float
- double
- long double
- 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
Operators
Arithmetic Expression Implicit Type Conversion

• Example implicit type conversions

Assume x is defined as:

```
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: $x \% 2^n$ 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;
}
```
## 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

\[
x = 5;
y = (++x) + 5;
\]
\[
// y = 11
// x = 6
\]
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><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;amp;= y</code></td>
<td><code>x = x &amp;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

\[ x = x + y; \]

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

\[ x += y; \quad // \text{Increment } x \text{ by the value } y \]
Example

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

```c
int x = 2, y = 6;
x *= 5 - y;
```
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!"
    }
}
```
### 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 && 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.
• The danger of short circuit evaluation

Example

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

```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></td>
</tr>
<tr>
<td>l</td>
<td>Bitwise OR</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
<td>x ^ y</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></td>
</tr>
<tr>
<td>~</td>
<td>Bitwise NOT (One's Complement)</td>
<td>~x</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
  $0b1010 \& 0b1101 \rightarrow 0b1000$

• && is the logical AND operator
  $0b1010 \&\& 0b1101 \rightarrow 0b0001$ (TRUE)

  $<\text{Non-Zero Value}> \&\& <\text{Non-Zero Value}> \rightarrow 1$ (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
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

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

```
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

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:

\[\begin{align*}
  x &= 5; &// & x &= 0b00000101 &= 5 \\
  y &= x << 2; &// & y &= 0b00010100 &= 20
\end{align*}\]

- 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 **UN SIGNED** (unsigned char in this case):

\[
x = 250; \quad \text{// } x = 0b11111010 = 250
\]

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

• Arithmetic Shift Right (Sign Extend)

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

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

\[
y = x >> 2; \quad \text{// } y = 0b11111110 = -2
\]
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

\[ y = x / 2^n \]
\[ y = x >> n \]

0 0 0 0 1 0 1 0

\( 10_{10} \)  \( \text{Right Shift} \)  \( 5_{10} \)

- Works for integers or fixed point values
Operators

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

**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
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10:</td>
<td>y = x / 2;</td>
<td>y = 10</td>
</tr>
<tr>
<td>00288</td>
<td>804000</td>
<td>mov.w 0x0800,0x0000</td>
</tr>
<tr>
<td>0028A</td>
<td>200022</td>
<td>mov.w #0x2,0x0004</td>
</tr>
<tr>
<td>0028C</td>
<td>090011</td>
<td>repeat #17</td>
</tr>
<tr>
<td>0028E</td>
<td>D80002</td>
<td>div.sw 0x0000,0x0004</td>
</tr>
<tr>
<td>00290</td>
<td>884010</td>
<td>mov.w 0x0000,0x0802</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9:</td>
<td>y = x &gt;&gt; 1;</td>
<td>y = 10</td>
</tr>
<tr>
<td>00282</td>
<td>804000</td>
<td>mov.w 0x0800,0x0000</td>
</tr>
<tr>
<td>00284</td>
<td>DE8042</td>
<td>asr 0x0000,#1,0x0000</td>
</tr>
<tr>
<td>00286</td>
<td>884010</td>
<td>mov.w 0x0000,0x0802</td>
</tr>
</tbody>
</table>
Operators

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

Example: Divide by 2

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

```
10:          y = x / 2;
0132 C08C   MOVFF 0x8c, 0x8a
0134 F08A   NOP
0136 C08D   MOVFF 0x8d, 0x8b
0138 F08B   NOP
013A 0E02   MOVLW 0x2
013C 6E0D   MOVWF 0xd, ACCESS
0140 C08A   MOVFF 0x8a, 0x8
0142 F008   NOP
0144 C08B   MOVFF 0x8b, 0x9
0146 F009   NOP
0148 EC6B   CALL 0xd6, 0
014A F000   NOP
014C C008   MOVFF 0x8, 0x8a
014E F08A   NOP
0150 C009   MOVFF 0x9, 0x8b
0152 F08B   NOP
```

Example: Right Shift by 1

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

```
9:            y = x >> 1;
0122 C08C   MOVFF 0x8c, 0x8a
0124 F08A   NOP
0126 C08D   MOVFF 0x8d, 0x8b
0128 F08B   NOP
012A 0100   MOVLB 0
012C 90D8   BCF 0xfd8, 0, ACCESS
012E 338B   RRCF 0x8b, F, BANKED
0130 338A   RRCF 0x8a, F, BANKED
```

16-Bit Shift on 8-Bit Architecture
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>foo(x)</td>
<td>Passes control to the function with the specified arguments</td>
</tr>
<tr>
<td>sizeof</td>
<td>Size of an object or type in bytes</td>
<td>sizeof x</td>
<td>The number of bytes x occupies in memory</td>
</tr>
<tr>
<td>(type)</td>
<td>Explicit type cast</td>
<td>(short) x</td>
<td>Converts the value of x to the specified type</td>
</tr>
<tr>
<td>?:</td>
<td>Conditional expression</td>
<td>x ? y : z</td>
<td>The value of y if x is true, else value of z</td>
</tr>
<tr>
<td>,</td>
<td>Sequential evaluation</td>
<td>x, y</td>
<td>Evaluates x then y, else result is value of y</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: \((\text{type})\text{variable}\)

Example: Integer Divide

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

\(y = 2.000000\) \(\times\) Because: \text{int} / \text{int} \Rightarrow \text{int}

Example: Floating Point Divide

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

\(y = 2.500000\) \(\checkmark\) Because: \text{float} / \text{int} \Rightarrow \text{float}
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>Left-to-Right</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>Right-to-Left</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;=</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.
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.

\[ x = f() + g() \]

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>~++x</td>
<td>Right-to-Left</td>
<td>~ (++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

< best loop
for Loop

Syntax

\[
\text{for (expression}_1; \text{ expression}_2; \text{ expression}_3) \\
\text{statement}
\]

- \text{expression}_1 initializes a loop count variable once at start of loop (e.g. \( i = 0 \))
- \text{expression}_2 is the test condition – the loop will continue while this is true (e.g. \( i \leq 10 \))
- \text{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

\[ \text{for } (\text{expression}_1; \text{expression}_2; \text{expression}_3) \]
\[ \text{statement} \]

START

Initialize loop variable

\[ i = 0 \]

expression_1

Test loop variable for exit condition

\[ i < n \]

expression_2?

TRUE

statement

Modify loop variable

FALSE

END

i++

Initialize loop variable

i = 0

Test loop variable for exit condition

i < n

expression_2?

TRUE

statement

Modify loop variable

FALSE

END

i++
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

Infinite Loops

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

```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.
do-while Loop

Syntax:

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

Flow Diagram:

START

statement

expression?

TRUE

statement

FALSE

END

START
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);
} 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
```
do-while Loop

Useful example

```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

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;
```

START

expression?

TRUE

statement

break

FALSE

statement

END
Example (Code Fragment)

```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**

```
continue;
```

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

Flow Diagram Within a **while** Loop

**Syntax**

```c
continue;
```

```
expression?

START

TRUE

statement

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
```

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

Iteration 2 does not print
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
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

Expected output

Known input

Actual output

Output matches

Print failure

Print success

Print failure

Print success

f()