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>*</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:

\[
\frac{10}{3} = 3 \frac{1}{3}
\]

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:
  - foo()?
- 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.

```
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Conversion Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>long double</code></td>
<td>Highest</td>
</tr>
<tr>
<td><code>double</code></td>
<td></td>
</tr>
<tr>
<td><code>float</code></td>
<td></td>
</tr>
<tr>
<td><code>unsigned long long</code></td>
<td></td>
</tr>
<tr>
<td><code>long long</code></td>
<td></td>
</tr>
<tr>
<td><code>unsigned long</code></td>
<td></td>
</tr>
<tr>
<td><code>long</code></td>
<td></td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td></td>
</tr>
<tr>
<td><code>int</code></td>
<td></td>
</tr>
<tr>
<td><code>unsigned short</code></td>
<td></td>
</tr>
<tr>
<td><code>short</code></td>
<td></td>
</tr>
<tr>
<td><code>unsigned char</code></td>
<td></td>
</tr>
<tr>
<td><code>char</code></td>
<td>Lowest</td>
</tr>
</tbody>
</table>
\[243 \cdot 3 = 729\]

\[217 \cdot 39 = 8661\]
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 \times -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
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;
}
```

\( 123456 \mod 10 = 6 \)
\( 123456 \div 10 = 12345 \)
\( 12345 \mod 10 = 5 \)
## 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>Increment</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>--x</td>
<td>Decrement</td>
<td>--x</td>
<td>Decrement x by 1, then use x</td>
</tr>
</tbody>
</table>

### Postfix Example

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

### Prefix Example

```c
x = 5;
y = (++x) + 5;
// y = 11
// x = 6
++x;  // Error
```
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>

---

Maxwell James Dunne

CMPE-013/L: “C” Programming
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 \( y \)
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)
```

# 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><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

```c
x = 5
if (x == 5) {
    do if value of \( x \) is 5
}
```

Maxwell James Dunne
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 = (\neq 0))</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code></td>
<td>Logical AND</td>
<td><code>x &amp;&amp; y</code></td>
<td>1 if both <code>x \neq 0</code> and <code>y \neq 0</code>, else 0</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
<td>Logical OR</td>
</tr>
<tr>
<td><code>!</code></td>
<td>Logical NOT</td>
<td><code>! x</code></td>
<td>1 if <code>x = 0</code>, else 0</td>
</tr>
</tbody>
</table>

`if (x > 3 || x < 2)`

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.

<table>
<thead>
<tr>
<th><code>expr1</code></th>
<th><code>expr2</code></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

```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.
\[ i \times (x \rightarrow y) \in \mathbb{C} \subset \text{foo()} \]

\[ z = \text{foo}(); \]

\[ \text{if } (x \rightarrow z \in \mathbb{C} \& \& z) \]
## 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</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>(One's Complement)</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

Difference Between & and &&

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

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

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

  \[ \textbf{if} \ (x \ \texttt{&&} \ y) \{ \text{do if } x \text{ and } y \text{ are both TRUE (non-zero)} \} \]
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! ");}
```
Matrix Equals
\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 0
\end{bmatrix} = \begin{bmatrix} \cdots \end{bmatrix}
\]

5 - 3 = 2.0000001

FP_DELTA
.000001

(x - y) < FP_DELTA
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:

\[
x = 5; \quad // \quad x = \text{0b}00000101 = 5
\]

\[
y = x << 2; \quad // \quad y = \text{0b}00010100 = 20
\]

- 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):

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

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

• Arithmetic Shift Right (Sign Extend)

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

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

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

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

0 0 0 0 1 0 1 0 >> 0 0 0 0 0 0 1 0 1

10_{10} \quad \text{Right Shift} \quad 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
```

```
00288 804000
0028A 200022
0028C 090011
0028E D80002
00290 884010
mov.w 0x0800,0x0000
mov.w #0x2,0x0004
repeat #17
div.sw 0x0000,0x0004
mov.w 0x0000,0x0802
```

Example: Right Shift by 1

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

```
y = 10
```

```
9:
00282 804000
00284 DE8042
00286 884010
mov.w 0x0800,0x0000
asr 0x0000,#1,0x0000
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
# 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>&lt;-&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>(type)</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>?:</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>,</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

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: \texttt{4.0f}
- We can cast the variable instead by using the cast operator: \texttt{(type)variable}

Example: Integer Divide
\begin{verbatim}
int x = 10;
float y;
y = x / 4;
y = 2.000000 \xmark
\end{verbatim}
Because: \texttt{int / int \rightarrow int}

Example: Floating Point Divide
\begin{verbatim}
int x = 10;
float y;
y = (float)x / 4;
y = 2.500000 \checkmark
\end{verbatim}
Because: \texttt{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("\8f\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>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>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>$a - b * c$</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>$x / y &amp; z$</td>
<td>Left-to-Right</td>
<td>$(x / y) &amp; z$</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 practice to explicitly group elements of an expression.
\[ X = \left( \begin{array} { c c c c c c } \end{array} \right) \]

\[ x \notin \{ e, e^2, e^3, e^4 \} \]
CMPE-013/L

Loop Structures (cont'd)

Maxwell James Dunne
for
do-while
for Loop

Syntax

for (expression₁; expression₂; expression₃)
statement

- expression₁ initializes a loop count variable once at start of loop (e.g. \( i = 0 \))
- expression₂ is the test condition – the loop will continue while this is true (e.g. \( i \leq 10 \))
- expression₃ is executed at the end of each iteration – usually to modify the loop count variable (e.g. \( i++ \))
for Loop

Flow Diagram

Syntax

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

Flow Diagram:

1. **START**
   - Initialize loop variable
   - `i = 0`

2. **expression_1**
   - Modify loop variable
   - `i++`

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

4. **statement**
   - If `TRUE`, go to expression_2
   - If `FALSE`, go to END

5. **END**
   - Repeat the loop

Maxwell James Dunne

CMPE-013/L: “C” Programming
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 \( \text{expression}_1 \) or \( \text{expression}_3 \) are missing, their actions simply disappear
- If \( \text{expression}_2 \) 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)

```c
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: '8c' \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);
```
if (+test) {
    foo()
}
while (+test) {
    foo(),
}
3

do 3
foo()
3 while;
**do-while Loop**

Trivial example

```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**

```c
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**

```plaintext
break;
```

![Flow Diagram](image)
**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
```
while (c) {
    while (c) {
        break;
    }
}
continue Statement

Syntax

```c
continue;
```

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

Flow Diagram Within a `while` Loop

**Syntax**

```
continue;
```

![Flow Diagram](image-url)
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.
for (i = 0; i < 3; i++)
  for (j = 0; j < ?; j++)
    for (r < row)
      for (col1
        for (;;)
257 \times 10 = 2
\text{rand}(\cdot) \times 10 = 0 - 9
\frac{257}{0}
\frac{1}{1} = 1
\frac{1}{2} = 0