CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
printf()  
Standard Library Function

- Used to write text to the "standard output"
- Normally a computer monitor or printer
- Often the UART in embedded systems
- SIM Uart1/2 window in MPLAB X
**printf()**
Standard Library Function

**Syntax**

```c
printf(ControlString, arg1, ..., argn);
```

- Everything printed verbatim within string except `%d's which are replaced by the argument values from the list

**Example**

```c
int a = 5, b = 10;
printf("a = %d\n b = %d\n", a, b);
```

Result:

```
a = 5
b = 10
```

NOTE: the 'd' in `%d is the conversion character.
(See next slide for details)
printf()
Gotchas

• The value displayed is interpreted entirely by the formatting string:
  printf("ASCII = 97", 'a');
  will output: ASCII = 97

A more problematic string:
  printf("Value = 6.02e23", 6.02e23);
  will output: Value = 26366

• Incorrect results may be displayed if the format type doesn't match the actual data type of the argument
`printf()`

Output buffer

- `printf()` operates on lines of text.
- Output text may not be transmit until a newline is sent.

Example

```c
printf("a");
while(1);
```

Output:
printf()  
Output buffer

- printf() operates on lines of text.
- Output stored in a buffer until a newline triggers transmission.

Example

```c
printf("a\n");
```

Output:

"a\n"
printf()
The output buffer

stdio.h

output buffer

UART
printf()
Format specifiers

%[flags][width][.precision][size]type

• Flags – Special printing options
• Width – The minimum size (in chars) of the output
• Precision – Field width
• Size – Convert from base types to longer/shorter types
• Type – The base variable type
printf()

Format specifiers

`%[flags][width][.precision][size]type`

- **Flags** – Special printing options
  - `'-'` -> Left justify
  - `'0'` -> Pad with zeros
  - `'+'` -> Output `+'` for positive values
  - `' '` -> Don’t output a sign symbol
  - `'#'` -> Prefix integer value based on output type
printf()  
Format specifiers

%[flags][width][.precision][size]type

- **Width** – The minimum size (in chars) of the output
  - Output is padded
  - ‘0’ flag specifies padding with ‘0’s instead of ‘ ‘s

0-18
printf()

Format specifiers

%[flags][width][.precision][size]type

- **Precision** – Field width
  - For integers, minimum number of digits
  - For floats, number of fractional digits/significant figures
  - For strings, number of characters
`printf()`

Format specifiers

`%[flags][width][.precision][size]type`

- **Size** – Convert from base types to longer/shorter types
  - `‘h’` -> Converts to short
  - `‘l’` -> Converts to long/double
  - `‘ll’` -> Converts to long long/double
printf()
Format specifiers

<table>
<thead>
<tr>
<th>Conversion Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Single character</td>
</tr>
<tr>
<td>s</td>
<td>String (all characters until '\0')</td>
</tr>
<tr>
<td>d</td>
<td>Signed decimal integer</td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal integer</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hexadecimal integer with lowercase digits (1a5e)</td>
</tr>
<tr>
<td>X</td>
<td>As x, but with uppercase digits (e.g. 1A5E)</td>
</tr>
<tr>
<td>f</td>
<td>Signed decimal value (floating point)</td>
</tr>
<tr>
<td>e/E</td>
<td>Signed decimal with exponent (e.g. 1.26e-5)</td>
</tr>
<tr>
<td>p</td>
<td>A pointer value indicating a memory address</td>
</tr>
<tr>
<td>g/G</td>
<td>As e or f, but depends on size and precision of value</td>
</tr>
<tr>
<td>g</td>
<td>Prints ‘%’</td>
</tr>
</tbody>
</table>
printf()  
Format String Examples

• Print a hexadecimal:

```c
printf("0x%06x\n", x);
```

0  Any unused spaces will be filled with zeros

6  Specifies that 6 characters must be output (including 0x prefix) *excluding*

```
871587  0x0d4ca3
```
printf()  
Format String Examples

• Printing a double:

```c
printf("f = 806.3f\n", f);
```

0  Any unused spaces will be filled with zeros
6  Specifies that 6 characters must be output
.3  Specifies that 3 decimal places will be output

3.3  1234

03.300  23300
printf() Format String Examples

- Printing a double:

```c
printf("%.1f\n", percentCorrect);
```

- .1 Specifies that 1 decimal place will be output

- %% Outputs a literal ‘%’

97.322 -> 97.38
printf()  
Format String Examples

• Printing a double:

```c
printf("8.1f88\n", (double)percentCorrect);
```

.1  Specifies that 1 decimal place will be output

%%  Outputs a literal ‘%’

97.322  97.38
**scanf()**

Standard Library Function

- Used to read input from the "standard input"
- Normally a keyboard or file
- Often the UART in embedded systems
- Input file in the simulator
- Entire family of functions:
  - `sscanf()` reads from a string
  - `fscanf()` reads from a file
**scanf()**
Standard Library Function

**Syntax**

```c
int scanf(FormatString, arg1, ..., argn);
```

- The format string tells `scanf` what kind of input.
- `arg1` through `argn` are **POINTERS** to variable of the right type.

**Example**

```c
int a, b;
printf("Input a and b\n\n");
scanf("%d %d", &a, &b);
printf("a=%d\nb=%d", a, b);
```
**scanf()**

Gotchas

- Ignores blanks and tabs in format string
- Skips over white space (blanks, tabs, newline) as it looks for input
- Returns number of successful conversions
- Arguments *must* be pointers to variable types
- Arguments not processed in the input will be left in the input buffer.
`scanf()`
The input buffer

`stdio.h`

- Output buffer
- Input buffer
- UART
```c
int a, b;
printf("Input a and b\n");
scanf("%d %d", &a, &b);  // 3140 56
printf("a=%d\n\nb=%d", a, b);  // "3140 56\n"
scanf("%d %d", &a, &b);
printf("a=%d\n\nb=%d", a, b);  // "77 -3\n"
```
**scanf()**

The input buffer

```
3 1 4 0 5 6 
```

```
scanf("%d %d", &a, &b)
```

```
3 1 4 0 5 6 
```

```
a = 3140, b = 56
```
`scanf()`

The input buffer

```
\n7 7 - 3 \n```

```
scanf("%d %d", \&a, \&b)
```

```
Nothing!
```
`scanf()`

The input buffer

```
3 1 4 0 5 6 \n
scanf("%d %d%c", &a, &b, &c)
```

```
3 1 4 0 5 6 \n
a = 3140, b = 56
```
scanf()  
Format specifiers

%[*][width][modifier]type

- * – Ignores this field
- Width – The maximum number of characters to match
- Modifier – Convert from base types to longer/shorter types
- Type – The base variable type
**scanf()**

**Examples**

- Read input line with date in the format:
  - 25/12/2012

```c
scanf("%d/%d/%d", &day, &month, &year);
```

- `day` int, &day is pointer to day
- `month` int, &month is pointer to month
- `year` int, &year is pointer to year
scanf()  

Examples  

• Read input line with date in the format:  
  – 25 Dec 2012  

\[
\text{scanf} \left( \text{"8d 8s 8d"} \right, \ &\text{&day, month, &year} \right);  
\]

day int, &day is pointer to day

month char[20], is a string for putting the month into, does not need "&" because name of array is already a pointer

year int, &year is pointer to year
scanf()

Return value

Example

```c
int a, b;
char c;
while (scanf("%d %d%c", &a, &b, &c) != 3) {
    printf("Please enter an integer pair!\n");
}
```
Arrays are variables that can store many items of the same type. The individual items known as elements, are stored sequentially and are uniquely identified by the array index (sometimes called a subscript).

- Arrays:
  - May contain any number of elements
  - Elements must be of the same type
  - The index is zero based
  - Array size (number of elements) must be specified at declaration
Arrays

How to Create an Array

Arrays are declared much like ordinary variables:

Syntax

```c
type arrayName[size];
```

- **size** refers to the number of elements
- **size** can be a constant OR specified at runtime (c99)

Example

```c
int a[10];

char s[25];

char str[x];
```
Arrays

How to Initialize an Array at Declaration

Arrays may be initialized with a list when declared:

Syntax

\[
\text{type arrayName[size]} = \{item_1, \ldots, item_n\};
\]

- The items must all match the type of the array

Example

\[
\text{int } a[5] = \{10, 20, 30, 40, 50\};
\]

\[
\text{char } b[5] = \{'a', 'b', 'c', 'd', 'e'\};
\]
Arrays

How to Use an Array

Arrays are accessed like variables, but with an index:

Syntax

```
arrayName[index]
```

- `index` may be a variable or a constant
- The first element in the array has an index of 0
- C does not provide any bounds checking

Example

```
int i, a[10];  // An array that can hold 10 integers

for(i = 0; i < 10; i++) {
    a[i] = 0;  // Initialize all array elements to 0
}

a[4] = 42;  // Set fifth element to 42
```
Arrays

Creating Multidimensional Arrays

Add additional dimensions to an array declaration:

Syntax

\[ \text{type arrayName}[\text{size}_1] \ldots [\text{size}_n]; \]

- Arrays may have any number of dimensions
- Three dimensions tend to be the largest used in common practice

Example

\[
\begin{align*}
\text{int a}[10][10]; & \quad \text{// 10x10 array for 100 integers} \\
\text{float b}[10][10][10]; & \quad \text{// 10x10x10 array for 1000 floats}
\end{align*}
\]
Arrays

Initializing Multidimensional Arrays at Declaration

Arrays may be initialized with lists within a list:

**Syntax**

```c
type arrayName[size_0]...[size_n] =
    {
        {item,...,item},
        ...
        {item,...,item}
    };
```

**Example**

```c
char a[3][3] = {
    {'X', '0', 'X'},
    {'0', '0', 'X'},
    {'X', 'X', '0'}
};

int b[2][2][2] = {
    {{0, 1}, {2, 3}},
    {{4, 5}, {6, 7}}
};
```
# Arrays

**Visualizing 2-Dimensional Arrays**

```c
int a[3][3] = {
    {0, 1, 2},
    {3, 4, 5},
    {6, 7, 8}
};
```

### Array Access Example:

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>1</td>
<td>3, 4, 5</td>
</tr>
<tr>
<td>2</td>
<td>6, 7, 8</td>
</tr>
</tbody>
</table>

- `a[0][0] = 0;`  
- `a[0][1] = 1;`  
- `a[0][2] = 2;`  
- `a[1][0] = 3;`  
- `a[1][1] = 4;`  
- `a[1][2] = 5;`  
- `a[2][0] = 6;`  
- `a[2][1] = 7;`  
- `a[2][2] = 8;`
Arrays

Visualizing 3-Dimensional Arrays

```c
int a[2][2][2] = {{ {0, 1}, {2, 3} },
                   { {4, 5}, {6, 7} } };```

**Plane, Row, Column**

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Plane 0**

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Plane 1**

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Maxwell James Dunne
/******************  
* Print out 0 to 90 in increments of 10  
*******************/

int main(void)
{
    int i = 0;
    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    while (i < 10)
    {
        a[i] *= 10;
        printf("%d\n", a[i]);
        ++i;
    }

    while (1);
}
CMPE-013/L

Strings

Maxwell James Dunne
**Strings**

Character Arrays and Strings

**Definition**

Strings are arrays of char whose last element is a null character \'\0\' with an ASCII value of 0. C has no native string data type, so strings must always be treated as character arrays.

- Strings:
  - Are enclosed in double quotes "string"
  - Are terminated by a null character \'\0\'
  - Must be manipulated as arrays of characters (treated element by element)
  - May be initialized with a string literal
Strings

Creating a String Character Array

Strings are created like any other array of char:

Syntax

```c
char arrayName[length];
```

- `length` must be one larger than the length of the string to accommodate the terminating null character '\0'
- A `char` array with `n` elements holds strings with `n-1 char`

Example

```c
char str1[10]; // Holds 9 characters plus '\0'
char str2[6];  // Holds 5 characters plus '\0'
```
Strings
How to Initialize a String at Declaration

Character arrays may be initialized with string literals:

**Syntax**

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

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

**Example**

```
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

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

- Null character '\0' must be appended manually

Example

```plaintext
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);
}
```
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:

\[
10 / 3 = 3
\]

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: +, -, ++, --

\[ a + 1 \]
\[ a + 4 \]
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 = 2.0000000000000000000
```

Example: Floating Point Divide

```c
int a = 10;
float b = 4.0f;
float c;
c = a / b;
c = 2.5000000000000000000
```

Because: int / int ⇒ int

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

long double

double

float

unsigned long long

long long

unsigned long

long

unsigned int

int

unsigned short

short

unsigned char

char

Smaller types converted to largest type in expression
Operators

Arithmetic Expression Implicit Type Conversion

- Example implicit type conversions

Assume \( x \) is defined as:

\[
\text{short } x = -5;
\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Implicit Type Conversion</th>
<th>Expression's Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( -x )</td>
<td>( x ) is promoted to ( \text{int} )</td>
<td>( \text{int} )</td>
<td>5</td>
</tr>
<tr>
<td>( x \times -2L )</td>
<td>( x ) is promoted to ( \text{long} ) because (-2L) is a long</td>
<td>( \text{long} )</td>
<td>10</td>
</tr>
<tr>
<td>( 8 \div x )</td>
<td>( x ) is promoted to ( \text{int} )</td>
<td>( \text{int} )</td>
<td>-1</td>
</tr>
<tr>
<td>( 8 \mod x )</td>
<td>( x ) is promoted to ( \text{int} )</td>
<td>( \text{int} )</td>
<td>3</td>
</tr>
<tr>
<td>( 8.0 \div x )</td>
<td>( x ) is promoted to ( \text{double} ) because ( 8.0 ) is a double</td>
<td>( \text{double} )</td>
<td>-1.6</td>
</tr>
</tbody>
</table>
\[ \text{Input: } x = 3 \]
\[ \text{Input: } y = 4 \]

\[ \frac{8.0}{\left( \frac{\text{Input}}{3.0} \right)} = \frac{8.0}{\frac{3.0}{4}} \]
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 \( \mod 10 = 6 \)
1234 56 / 10 = 12349
i++;  

j = i % 11;
Operators

Arithmetic: Increment and Decrement

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

**Postfix Example**

```
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.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>An assignment statement is a statement that assigns a value to a variable.</td>
<td><code>variable = expression;</code></td>
</tr>
</tbody>
</table>

- Two types of assignment statements
  - Simple assignment
    ```c
    variable = expression;
    ```
    The expression is evaluated and the result is assigned to the variable
  - Compound assignment
    ```c
    variable = variable op expression;
    ```
    The variable appears on both sides of the =
## Operators

### Assignment

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

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

  **Example**

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

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

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

  **Example**

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

Compound Assignment

Example

```c
int x = 2;

x *= 5;

x = x * 5;
```
Operators
Compound Assignment

Example

```
int x = 2, y = 6;
x *= 5 - y;
x = (x * (5 - y));
x = x * y;
x = x - y - 5;
```
<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 &lt;= 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 &gt;= 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 != y</td>
<td>1 if x not equal to y, else 0</td>
</tr>
</tbody>
</table>

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

Difference Between \(=\) and \(==\)

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

- \(=\) is the assignment operator
  \(x = 5\) assigns the value 5 to the variable \(x\)
- \(==\) is the 'equals to' relational operator
  \(x == 5\) tests whether the value of \(x\) is 5

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

• What happens when the following code is executed?

Example

```c
void main(void)
{
    int x = 2;  // Initialize x
    if (x = 5) {
        printf("Hi!");  // 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 ≠ 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:

\[ \text{expr1} \&\& \text{expr2} \]

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

<table>
<thead>
<tr>
<th>Truth Table for AND (&amp;&amp;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{expr1}</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

\( \text{expr2} \) 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.
if (check1() & check2())
    False
## 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></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 (One's Complement)</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0, if 1 in x</td>
</tr>
</tbody>
</table>

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

Difference Between & and &&

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

• & is the bitwise AND operator
  \texttt{ob1010 \& ob1101 \rightarrow ob1000}

• && is the logical AND operator
  \texttt{ob1010 \&\& ob1101 \rightarrow ob0001} \texttt{(TRUE)}
  \texttt{<Non-Zero Value> \&\& <Non-Zero Value> \rightarrow 1} \texttt{(TRUE)}

\textbf{if} \ (x \ \&\& \ y) \ \{ \\
  \textit{do if} \ x \textit{and} \ y \textit{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:

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

- 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 **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.

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

- Works for integers or fixed point values.
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
```

Example: Right Shift by 1

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

10:
00288 804000 mov.w 0x0800,0x0000
0028A 200022 mov.w #0x2,0x0004
0028C 090011 repeat #17
0028E D80002 div.sw 0x0000,0x0004
00290 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
```

10:

| 10 | C08C | MOVFF 0x8c, 0x8a |
| 132 |      |                 |
|     | F08A | NOP             |
| 134 |      |                 |
|     | C08D | MOVFF 0x8d, 0x8b |
| 136 |      |                 |
|     | F08B | NOP             |
| 138 |      |                 |
|     | 0E02 | MOVLW 0x2      |
| 13A |      |                 |
|     | 6E0D | MOVWF 0xd, ACCESS |
| 13C |      |                 |
|     | 6A0E | CLR 0xe, ACCESS |
| 13E |      |                 |
|     | C08A | MOVFF 0x8a, 0x8 |
| 140 |      |                 |
|     | F008 | NOP             |
| 142 |      |                 |
|     | C08B | MOVFF 0x8b, 0x9 |
| 144 |      |                 |
|     | F009 | NOP             |
| 146 |      |                 |
|     | EC6B | CALL 0xd6, 0   |
| 148 |      |                 |
|     | F000 | NOP             |
| 14A |      |                 |
|     | C008 | MOVFF 0x8, 0x8a |
| 14C |      |                 |
|     | F08A | NOP             |
| 14E |      |                 |
|     | C009 | MOVFF 0x9, 0x8b |
| 150 |      |                 |
|     | F08B | NOP             |
| 152 |      |                 |

Example: Right Shift by 1

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

9:

| 9  | C08C | MOVFF 0x8c, 0x8a |
| 122 |      |                 |
|     | F08A | NOP             |
| 124 |      |                 |
|     | C08D | MOVFF 0x8d, 0x8b |
| 126 |      |                 |
|     | F08B | NOP             |
| 128 |      |                 |
|     | 0100 | MOVLW 0        |
| 12A |      |                 |
|     | 90D8 | BCF 0xfd8, 0 , ACCESS |
| 12C |      |                 |
|     | 338B | RRCF 0x8b, F, BANKED |
| 12E |      |                 |
|     | 338A | RRCF 0x8a, F, BANKED |
| 130 |      |                 |

16-Bit Shift on 8-Bit Architecture

Maxwell James Dunne
CMPE-013/L: “C” Programming
# 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 \textit{x}</td>
</tr>
<tr>
<td>*</td>
<td>Indirection</td>
<td>*p</td>
<td>The object or function that \textit{p} points to</td>
</tr>
<tr>
<td>[ ]</td>
<td>Subscripting</td>
<td>\textit{x}[y]</td>
<td>The \textit{y}th element of array \textit{x}</td>
</tr>
<tr>
<td>.</td>
<td>Struct / Union Member</td>
<td>\textit{x}.\textit{y}</td>
<td>The member named \textit{y} in the structure or union \textit{x}</td>
</tr>
<tr>
<td>\rightarrow</td>
<td>Struct / Union Member by Reference</td>
<td>\textit{p}→\textit{y}</td>
<td>The member named \textit{y} in the structure or union that \textit{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>,</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)

\[ x = (\text{condition}) \ ? \ a : b; \]

Example 2 (less often used)

\[ (\text{condition}) \ ? \ (x = a):(x = b); \]

In both cases:
\[ x = a \text{ if condition is true} \]
\[ x = b \text{ if condition is false} \]
Operators
The Explicit Type Cast Operator

- Earlier, we cast a literal to type float by entering it as: `4.0f`
- We can cast the variable instead by using the cast operator: `(type)variable`

Example: Integer Divide

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

- `y = 2.000000` ✗
  - Because: `int / int` ➞ `int`

Example: Floating Point Divide

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

- `y = 2.500000` ✓
  - Because: `float / int` ➞ `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></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><code>sizeof</code></td>
<td>Size of Expression or Type</td>
<td></td>
</tr>
<tr>
<td><code>(type)</code></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;&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.

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>$x \div y &amp; z$</td>
<td>Left-to-Right</td>
<td>$(x \div y) &amp; z$</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.