CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne

Spring 2015
Announcements

- Lab 2 is out
- X Drive issues
Functions

Declaration and Use: Example 1

Example 1

```c
int a = 5, b = 10, c;

int Maximum(int x, int y)
{
    return ((x >= y) ? x : y);
}

int main(void)
{
    c = Maximum(a, b);
    printf("The max is %d\n", c);
}
```

Function is declared and defined before it is used in main()
Functions

Declaration and Use: Example 2

Example 2

```c
int a = 5, b = 10, c;

int Maximum(int x, int y);

int main(void)
{
    c = Maximum(a, b);
    printf("The max is %d\n", c);
}

int Maximum(int x, int y)
{
    return ((x >= y) ? x : y);
}
```

Function is *declared* with prototype before use in `main()`

Function is *defined* after it is used in `main()`
`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 window in MPLAB X
printf()
Standard Library Function

Syntax

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

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

Example

```
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:
  
  ```c
  printf("ASCII = %d", 'a');
  ```
  
  will output: `ASCII = 97`

- A more problematic string:
  
  ```c
  printf("Value = %d", 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

```
printf("a");
```

Output:

[Blank space]
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
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/long double
### `printf()` Format specifiers

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

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
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:

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

1. Specifies that 1 decimal place will be output

2. 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);
```
\texttt{scanf()}\textsuperscript{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 \textbf{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
**scanf()**

Standard Library Function

Example

```c
int a, b;
printf("Input a and b\n");
scanf("%d %d", &a, &b);
printf("a=%d\nb=%d", a, b); ← "3140 56\n"
scanf("%d %d", &a, &b);
printf("a=%d\nb=%d", a, b); ← "77 -3\n"
```
The input buffer

\[
\begin{array}{cccccc}
3 & 1 & 4 & 0 & 5 & 6 \\
\end{array}
\]

\[
\text{scanf( ">%d %d", &a, &b) }
\]

\[
\text{a = 3140, b = 56 }
\]
`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
Enter
```

```
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

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

- **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");
}
```

```
1
2
1 2
1 2
1 2
1 2
1 2
1 2
```
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**

```
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

type arrayName[size] = {item₁, ..., itemₙ};

• The items must all match the type of the array

Example

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

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:

```
type arrayName[size1]...[size_n];
```

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

Example

```
int a[10][10];  // 10x10 array for 100 integers

float b[10][10][10];  // 10x10x10 array for 1000 floats
```

32000
Arrays

Initializing Multidimensional Arrays at Declaration

Arrays may be initialized with lists within a list:

**Syntax**

\[
\text{type arrayName}[\text{size}_0]...[\text{size}_n] = \\
\{\{\text{item},...\text{item}\},
\vdots
\{\text{item},...\text{item}\}\};
\]

**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}};
```

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,0</td>
</tr>
<tr>
<td>1</td>
<td>1,0</td>
</tr>
<tr>
<td>2</td>
<td>2,0</td>
</tr>
</tbody>
</table>

```

<table>
<thead>
<tr>
<th>y</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>
```

Maxwell James Dunne – Spring 2015
Arrays

Visualizing 3-Dimensional Arrays

\[
\text{int } a[2][2][2] = \begin{cases}
\{\{0, 1\}, \{2, 3\}\}, \\
\{\{4, 5\}, \{6, 7\}\}\end{cases};
\]

Plane, Row, Column

\[
\begin{array}{c}
\begin{array}{c}
0 \\
1
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
0 \\
1
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
0 \\
1
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
0 \\
1
\end{array}
\end{array}
\]

Maxwell James Dunne – Spring 2015

CMPE-013/L: “C” Programming
/***********************
 * 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; // fix
    }

    while ( 1 );
}
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

Maxwell James Dunne – Spring 2015
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

```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 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**

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

- Null character '\0' must be appended manually

**Example**

```
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 / 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: +, -, ++, --
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  X
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

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

Arithmetic Expression Implicit Type Conversion

- Example implicit type conversions

Assume $x$ is defined as:

```
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;
}
```
# Operators

## Arithmetic: Increment and Decrement

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

### Postfix Example

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

### Prefix Example

```
x = 5;
y = (++x) + 5;
// y = 11
// x = 6
```
Operators
How to Code Assignment Statements

Definition
An assignment statement is a statement that assigns a value to a variable.

• Two types of assignment statements
  – Simple assignment
    \[ \text{variable} = \text{expression}; \]
The expression is evaluated and the result is assigned to the variable
  – Compound assignment
    \[ \text{variable} = \text{variable op expression}; \]
The variable appears on both sides of the =
# Operators

## Assignment

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

Compound Assignment

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

  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

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

Compound Assignment

Example

```c
int x = 2;

x *= 5;

x = x * 5;

x = 10;
```
Operators
Compound Assignment

Example

```c
int x = 2, y = 6;

x *= 5 - y;

x = x * (5 - y)

x = x * (5 - 6)
x = -2
```
# Operators

## Relational

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result (FALSE = 0, TRUE ≠ 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>(x &lt; y)</td>
<td>1 if (x) less than (y), else 0</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>(x \leq y)</td>
<td>1 if (x) less than or equal to (y), else 0</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>(x &gt; y)</td>
<td>1 if (x) greater than (y), else 0</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>(x \geq y)</td>
<td>1 if (x) greater than or equal to (y), else 0</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>(x == y)</td>
<td>1 if (x) equal to (y), else 0</td>
</tr>
<tr>
<td>!=</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

  \[
  \text{if } (x == 5) \{ \\
  \text{do if value of } x \text{ is 5} \\
  \}
  \]
Operators
Difference Between = and ==

• What happens when the following code is executed?

Example

```c
void main(void)
{
    int x = 2;           // Initialize x
    if (x = 5) {         // If x is 5...
        printf("Hi!");   // ...display "Hi!"
    }
}
```
# Operators

## Logical

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result (FALSE = 0, TRUE ≠ 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; &amp;</td>
<td>Logical AND</td>
<td>x &amp; &amp; y</td>
<td>1 if both (x \neq 0) and (y \neq 0), else 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
<td>! x</td>
<td>1 if (x = 0), else 0</td>
</tr>
</tbody>
</table>

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

- The evaluation of expressions in a logical operation stops as soon as a true or false result is known

Example

If we have two expressions being tested in a logical AND operation:

\[ expr1 \land expr2 \]

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

<table>
<thead>
<tr>
<th>Truth Table for AND (( \land ))</th>
<th>( expr1 )</th>
<th>( expr2 )</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE = 0</td>
<td>0</td>
<td>X (0)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>X (1)</td>
<td>0</td>
</tr>
<tr>
<td>TRUE = 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\( expr2 \) is not evaluated in the first two cases since its value is not relevant to the result.
Operators
Logical Operators and Short Circuit Evaluation

• The danger of short circuit evaluation

Example

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>
</table>
| &        | Bitwise AND    | \(x \& y\) | 1, if 1 in both \(x\) and \(y\)  
          |                |         | 0, if 0 in \(x\) or \(y\) or both              |
| l        | Bitwise OR     | \(x \mid y\) | 1, if 1 in \(x\) or \(y\) or both  
          |                |         | 0, if 0 in both \(x\) and \(y\)                |
| ^        | Bitwise XOR    | \(x ^ y\) | 1, if 1 in \(x\) or \(y\) but not both  
          |                |         | 0, if 0 or 1 in both \(x\) and \(y\)          |
| ~        | Bitwise NOT    | \(~x\)   | 1, if 0 in \(x\)  
          |                |         | 0, if 1 in \(x\)                               |

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

Difference Between & and &&

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

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

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

  \[
  \text{if} \ (x && y) \ { \}
  \]

  do if \ x \ and \ y \ are \ both \ TRUE \ (non-zero) \]
Operators

Difference Between & and &&

- What happens when each of these code fragments are executed?

Example 1 – Using A Bitwise AND Operator

```c
char x = 0b1010;
char y = 0b0101;
if (x & y) {printf("Hi!");}
```

Example 2 – Using A Logical AND Operator

```c
char x = 0b1010;
char y = 0b0101;
if (x && y) {printf("Hi!");}
```
# Operators

## Shift

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;&lt;</code></td>
<td>Shift Left</td>
<td><code>x &lt;&lt; y</code></td>
<td>Shift <code>x</code> by <code>y</code> bits to the left</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Shift Right</td>
<td><code>x &gt;&gt; y</code></td>
<td>Shift <code>x</code> by <code>y</code> bits to the right</td>
</tr>
</tbody>
</table>

**Shift Left Example:**

```plaintext
x = 5;  // x = 0b00000101 = 5
y = x << 2;  // y = 0b00010100 = 20
```

- In both shift left and shift right, the bits that are shifted out are lost
- For shift left, 0's are shifted in (Zero Fill)
Operators
Shift – Special Cases

• Logical Shift Right (Zero Fill)

If \( x \) is **UNSGNED** (unsigned \texttt{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 **SGNED** (\texttt{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.

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

- 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
```

Assembly code for divide:

10:
00288 804000
0028A 200022
0028C 090011
0028E D80002
00290 884010

Assembly code for shift:

9:
00282 804000
00284 DE8042
00286 884010

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:
```
0132 C08C MOVFF 0x8c, 0x8a
0134 F08A NOP
0136 C08D MOVFF 0x8d, 0x8b
0138 F08B NOP
013A 0E02 MOVLB 0x2
013C 6E0D MOVWF 0xd, ACCESS
013E 6A0E CLRF 0xe, 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:
```
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 RRCLF 0x8b, F, BANKED
0130 338A RRCLF 0x8b, F, BANKED
```

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><code>&amp;</code></td>
<td>Address of</td>
<td><code>&amp;x</code></td>
<td>Pointer to <code>x</code></td>
</tr>
<tr>
<td><code>*</code></td>
<td>Indirection</td>
<td><code>*p</code></td>
<td>The object or function that <code>p</code> points to</td>
</tr>
<tr>
<td><code>[ ]</code></td>
<td>Subscripting</td>
<td><code>x[y]</code></td>
<td>The <code>y</code>th element of array <code>x</code></td>
</tr>
<tr>
<td><code>.</code></td>
<td>Struct / Union Member</td>
<td><code>x.y</code></td>
<td>The member named <code>y</code> in the structure or union <code>x</code></td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>Struct / Union Member by Reference</td>
<td><code>p-&gt;y</code></td>
<td>The member named <code>y</code> in the structure or union that <code>p</code> 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]

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

Example 2 [less often used]

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

In both cases: x = a if condition is true
x = b if condition is false
Operators
The Explicit Type Cast Operator

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

```
Example: Integer Divide

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

\textcolor{red}{y = 2.000000} \times

Because: int / int \Rightarrow int
```

```
Example: Floating Point Divide

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

\textcolor{green}{y = 2.500000} \checkmark

Because: float / int \Rightarrow float
```
Operators
The Conditional Operator

Example

```c
float x = 5;

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

Result:

```
warning: format '%f' expects type 'double', but argument 2 has type 'float'
```

Operators
The Conditional Operator

Example

```c
float x = 5;

printf("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>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>~++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
for Loop

Syntax

for (expression$_1$; expression$_2$; expression$_3$) statement

- expression$_1$ initializes a loop count variable once at start of loop (e.g. $i = 0$)
- expression$_2$ is the test condition – the loop will continue while this is true (e.g. $i <= 10$)
- expression$_3$ is executed at the end of each iteration – usually to modify the loop count variable (e.g. $i++$)
**for Loop**

Flow Diagram

**Syntax**

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

![Flow diagram](image)

- **Initialize loop variable**
  - \( i = 0 \)

- **Modify loop variable**
  - \( i++ \)

- **Test loop variable for exit condition**
  - \( i < n \)
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)

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.


CMPE-013/L: “C” Programming
do-while Loop

Flow Diagram

Syntax

```c
do statement while (expression);
```
**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);  // Loop counter incremented manually inside loop

Expected Output:

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

Condition checked at end of loop iterations
do-while Loop

Useful example

Example (Code Fragment)

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

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

**Syntax**

```python
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;
```

```
START

expression?

TRUE

statement

FALSE

break

statement

END
```
**break** Statement

Example

```c
int i = 0;

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

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

Expected Output:

- Loop iteration 1
- Loop iteration 2
- Loop iteration 3
- Loop iteration 4
**continue Statement**

**Syntax**

```
continue;
```

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

Flow Diagram Within a `while` Loop

**Syntax**

`continue;`

START

expression?

TRUE

statement

FALSE

continue

statement

END
**continue Statement**

**Example**

```c
int i = 0;

while (i < 6) {
    ++i;
    if (i == 2) {
        continue;  // Skip remaining iteration when i = 2. Iteration 2 will not be completed.
    }
    printf("Loop iteration %d\n", i);
}
```

**Expected Output:**

```
Loop iteration 1
Loop iteration 3             // Iteration 2 does not print
Loop iteration 4
Loop iteration 5
```
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

Output matches

f()

Actual output

Print failure

Print success

no

yes