### Operators

#### Other

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>()</code></td>
<td>Function Call</td>
<td><code>foo(x)</code></td>
<td>Passes control to the function with the specified arguments</td>
</tr>
<tr>
<td><code>sizeof</code></td>
<td>Size of an object or type in bytes</td>
<td><code>sizeof x</code></td>
<td>The number of bytes <code>x</code> occupies in memory</td>
</tr>
<tr>
<td><code>(type)</code></td>
<td>Explicit type cast</td>
<td><code>(short) x</code></td>
<td>Converts the value of <code>x</code> to the specified type</td>
</tr>
<tr>
<td><code>?:</code></td>
<td>Conditional expression</td>
<td><code>x ? y : z</code></td>
<td>The value of <code>y</code> if <code>x</code> is true, else value of <code>z</code></td>
</tr>
<tr>
<td><code>;</code></td>
<td>Sequential evaluation</td>
<td><code>x, y</code></td>
<td>Evaluates <code>x</code> then <code>y</code>, else result is value of <code>y</code></td>
</tr>
</tbody>
</table>

---

### Operators

#### The Conditional Operator

**Syntax**

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

**Example**

```c
int x = 5;

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

**Result:**

```
5 is odd
```
Operators
The Conditional Operator

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

Example 1 (most commonly used)

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

Example 2 (less often used)

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

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

\[
\begin{align*}
\text{int} & \ x = 10; \\
\text{float} & \ y; \\
y & = x / 4;
\end{align*}
\]

\(y = 2.000000\) ❌
Because: \texttt{int / int ➞ int}

Example: Floating Point Divide

\[
\begin{align*}
\text{int} & \ x = 10; \\
\text{float} & \ y; \\
y & = (\text{float})x / 4;
\end{align*}
\]

\(y = 2.500000\) ✓
Because: \texttt{float / int ➞ float}
## 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...)

<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. For example:

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

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><code>x / y % z</code></td>
<td>Left-to-Right</td>
<td><code>(x / y) % z</code></td>
</tr>
<tr>
<td><code>x = y = z</code></td>
<td>Right-to-Left</td>
<td><code>x = (y = z)</code></td>
</tr>
<tr>
<td><code>~++x</code></td>
<td>Right-to-Left</td>
<td><code>~(++x)</code></td>
</tr>
</tbody>
</table>

- You can rely on these rules, but it is good programming practice to explicitly group elements of an expression

Lab Exercise 4

Operators
Exercise 04

Operators

• Open the lab Project:

On the class website
Examples -> Lab4.zip

Open MPLAB®X and select Open Project Icon (Ctrl + Shift + O)
Open the Project listed above.

If you already have a project open in MPLAB X, close it by “right clicking” on the open project and selecting “Close”
Exercise 04
Operators

Solution: Steps 1 and 2

/*###########################################################################
# STEP 1: Add charVariable1 to charVariable2 and store the result in
#          charVariable1. This may be done in two ways. One uses the
#          ordinary addition operator, the other uses a compound assignment
#          operator. Write two lines of code to perform this operation
#          twice - once for each of the two methods.
#          Don't forget to end each statement with a semi-colon!
###########################################################################*/

//Add using addition operator
charVariable1 = charVariable1 + charVariable2;

//Add using compound assignment operator
charVariable1 += charVariable2;

/*###########################################################################
# STEP 2: Increment charVariable1. There are several ways this could be
#          done. Use the one that requires the least amount of typing.
###########################################################################*/

//Increment charVariable1
charVariable1++;
Exercise 04

Operators

Solution: Step 5

/*###########################################################################
# STEP 5: Perform the operation (longVariable2 AND 0x30) and store the result
# back in longVariable2. Once again, the easiest way to do this is
# to use the appropriate compound assignment operator that will
# perform an equivalent operation to the one in the comment below.
###########################################################################*/

//longVariable2 = longVariable2 & 0x30
longVariable2 &= 0x30;

Exercise 04

Conclusions

• Most operators look just like their normal mathematical notation
• C adds several shortcut operators in the form of compound assignments
• Most C programmers tend to use the shortcut operators

x = x + 5;
x += 5;
Questions?

```c
if (head > buflen)
    head = 0;
else
    if (head == nhead)
        head = 0;
```
Lab 2 - Banana

```
main.c - Banana

int led0 = 0;
int led1 = 0;
int led2 = 0;
int led3 = 0;
int led4 = 0;
int led5 = 0;
int led6 = 0;

// Initializer
InitLEDs:
    Set TRISA register to 0x00 for active LED; I. any other axis
    Set LAT register to 0x00 for active LED.
    Record 1 save mask of enabled LEDs
    ~TRISA

SetLED:
    Check input against mask (~TRISA)
    If I in each bit of LAT, I went bit
    If I not in any other axis.

GetLED:
    Read LAT.
    Compare it to my mask (~TRISA)
```

TRISA = 0x00FFFA
```c
InitAddr (0x15);
SelAddr (0x32);
x = GetAddr();
SelAddr (0x1F);
-> x = GetAddr (); "0x44"
```

Diagram:

```
main() f
ISR

MODULE LEVEL VARIABLES
"Static"

TRM

```

```
xp rp
PL

```

```
0x FDD4

```

```
r
```

```
0010 0010
D4, D5
```

```
D4, D6, D8, D10
1010
```
CMPE-013/L

Functions

Gabriel Hugh Elkaim
Spring 2013
Functions

Program Structure

```c
main()
{
    ...
    eat();
    ...
    drink();
    ...
    be_merry();
    return;
}

be_merry()
{
    ...
    return;
}

drink()
{
    ...
    be_merry()
    return;
}
eat()
{
    ...
    return;
}
```

Functions

What is a function?

Definition

Functions are self contained program segments designed to perform a specific, well defined task.

- All C programs have one or more functions
- The `main()` function is required
- Functions can accept parameters from the code that calls them
- Functions usually return a single value
- Functions help to organize a program into logical, manageable segments
Functions

Remember Algebra Class?

• Functions in C are conceptually like an algebraic function from math class...

\[ f(x) = x^2 + 4x + 3 \]

• If you pass a value of 7 to the function: \( f(7) \), the value 7 gets "copied" into \( x \) and used everywhere that \( x \) exists within the function definition: \( f(7) = 7^2 + 4\times7 + 3 = 80 \)

Syntax

\[
\begin{align*}
\text{Header:} & \quad \text{Data type of return } \text{expression} \\
& \quad \text{Name} \\
& \quad \{ \text{identifier} (\text{type}_{1} \text{arg}_{1}, \ldots, \text{type}_{n} \text{arg}_{n}) \} \\
& \quad \{ \text{declarations statements code} \} \\
& \quad \text{return expression;} \\
\text{Body:} & \quad \text{Parameter List (optional)} \\
& \quad \text{Return Value (optional)}
\end{align*}
\]
Functions
Function Definitions: Syntax Examples

Example

```c
int maximum(int x, int y)
{
    int z;
    z = (x >= y) ? x : y;
    return z;
}
```

Example – A more efficient version

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

Syntax

```
(type) identifier(type1 arg1, ..., typeN argN)
{
    declarations
    statements
    return expression
}
```

- A function's `type` must match the type of data in the return `expression`
Functions
Function Definitions: Return Data Type

• A function may have multiple return statements, but only one will be executed and they must all be of the same type

Example
```c
int bigger(int a, int b)
{
    if (a > b)
        return 1;
    else
        return 0;
}
```

Functions
Function Definitions: Return Data Type

• The function type is `void` if:
  – The `return` statement has no expression
  – The `return` statement is not present at all

• This is sometimes called a procedure function since nothing is returned

Example
```c
void identifier(type_1 arg_1, ..., type_n arg_n)
{
    declarations
    statements
    return;
    return; may be omitted if nothing is being returned
}
```
Functions
Function Definitions: Parameters

- A function's parameters are declared just like ordinary variables, but in a comma delimited list inside the parentheses
- The parameter names are only valid inside the function (local to the function)

Syntax

```c
type identifier (type1 arg1, ... , type n arg n )
{
    declarations
    statements
    return expression;
}
```

Functions
Function Definitions: Parameters

- Parameter list may mix data types
  - int foo(int x, float y, char z)
- Parameters of the same type must be declared separately – in other words:
  - int maximum(int x, y) will not work
  - int maximum(int x, int y) is correct

Example

```c
int maximum(int x, int y)
{
    return ((x >= y) ? x : y);
}
```
Functions
Function Definitions: Parameters

• If no parameters are required, use the keyword `void` in place of the parameter list when defining the function

Example

```c
void identifier()
{
    declarations
    statements
    return expression;
}
```

Functions
How to Call / Invoke a Function

Function Call Syntax

- No parameters and no return value
  ```c
  foo();
  ```
- No parameters, but with a return value
  ```c
  x = foo();
  ```
- With parameters, but no return value
  ```c
  foo(a, b);
  ```
- With parameters and a return value
  ```c
  x = foo(a, b);
  ```
Functions

Function Prototypes

• Just like variables, a function must be declared before it may be used
• Declaration must occur before main() or other functions that use it
• Declaration may take two forms:
  – The entire function definition
  – Just a function prototype – the function definition itself may then be placed anywhere in the program

Functions

Function Prototypes

• Function prototypes may be take on two different formats:
  – An exact copy of the function header:
    
    Example – Function Prototype 1

    ```
    int maximum(int x, int y);
    ```

  – Like the function header, but without the parameter names – only the types need be present for each parameter:
    
    Example – Function Prototype 2

    ```
    int maximum(int, int);
    ```
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()`

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 defined after it is used in `main()`
Functions
Passing Parameters by Value

• Parameters passed to a function are passed by value
• Values passed to a function are copied into the local parameter variables
• The original variable that is passed to a function cannot be modified by the function since only a copy of its value was passed

Example
Passing Parameters by Value

```c
int a, b, c;

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

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

The value of `a` is copied into `x`.
The value of `b` is copied into `y`.
The function does not change the value of `a` or `b`. 
Functions
Recursion

• A function can call itself repeatedly
• Useful for iterative computations (each action stated in terms of previous result)
• Example: Factorials (5! = 5 * 4 * 3 * 2 * 1)

```c
long int factorial(int n)
{
    if (n <= 1)
        return(1);
    else
        return(n * factorial(n - 1));
}
```