Operators
How to Code Arithmetic Expressions

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
  - A microcontroller register is usually treated as a variable
- There are 9 arithmetic operators that may be used
  - Binary Operators: +, −, *, /, %
  - Unary 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 * 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

Example: Integer Divide

```c
int a = 10;
int b = -4;
float c;
c = a / b;
c = 2.000000
```

Because: int / int = int

Example: Floating Point Divide

```c
int a = 10;
float b = 4.0f;
float c;
c = a / b;
c = 2.50000
```

Because: float / int = float

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)
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

```
char < int < unsigned int < long < unsigned long < long long < unsigned long long
float < double < long double
```

Operators

Arithmetic Expression Implicit Type Conversion

- Example implicit type conversions

Assume `x` is defined as:

```c
short x = -5;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Implicit Type Conversion</th>
<th>Expression's Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-x</td>
<td>x is promoted to int</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>x * -2L</td>
<td>x is promoted to long because -2L is a long</td>
<td>long</td>
<td>10</td>
</tr>
<tr>
<td>8/x</td>
<td>x is promoted to int</td>
<td>int</td>
<td>-1</td>
</tr>
<tr>
<td>8%x</td>
<td>x is promoted to int</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>8.0/x</td>
<td>x is promoted to double because 8.0 is a double</td>
<td>double</td>
<td>-1.6</td>
</tr>
</tbody>
</table>

Operators

Applications of the Modulus Operator (%)

- Truncation: `x % 2^n` where `n` is the desired word width (e.g. 8 for 8 bits: `x % 256`)
  - Returns the value of just the lower `n`-bits of `x`
- Can be used to break apart a number in any base into its individual digits

Example:

```c
#define MAX_DIGITS 6
long number = 123456;
int i, radix = 10, char digits[MAX_DIGITS];

for (i = 0; i < MAX_DIGITS; i++)
{
    if (number == 0) break;
    digits[i] = (char)(number % radix);
    number /= radix;
}
```
Operators

Arithmetic: Increment and Decrement

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

- 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

Definition

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

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

- May use the shortcut assignment operators (compound assignment):
  \( x += y; \) //Increment \( x \) by the value \( y \)
Operators

**Compound Assignment**

Example

```c
int x = 2; //Initial value of x is 2
x *= 5; //x = x * 5
```

Before statement is executed: x = 2

After statement is executed: x = 10

Is equivalent to:

```c
x = (x * 5);
Evaluate right side first: x = (2 * 5);
Assign result to x: x = 10;
```

Operators

**Difference Between = and ==**

- = 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

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

Operators

**Relational**

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

<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 &lt;= 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 &gt;= 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 != y</td>
<td>1 if x not equal to y, else 0</td>
</tr>
</tbody>
</table>

Operators

**Difference Between = and ==**

- What happens when the following code is executed?

```
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 ≠ 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

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>^</td>
<td>Bitwise XOR</td>
<td>x ^ y</td>
<td>1, if 1 in both x and y, 0, if 0 in x or y or both</td>
</tr>
<tr>
<td>~</td>
<td>Bitwise NOT</td>
<td>~x</td>
<td>1, if 0 in x, 0, if 1 in x</td>
</tr>
</tbody>
</table>

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

Operators

Difference Between & and & &

• & is the bitwise AND operator
  0b1010( & 0b1101 → 0b1000
• & & is the logical AND operator
  0b1010 & & 0b1101 → 0b0001 (TRUE)
  <Non-Zero Value> & & <Non-Zero Value> → 1 (TRUE)

  if (x & & y) {
    do if x and y are both TRUE (non-zero)
  }

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

Example 1 – Using A Bitwise AND Operator

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

Example 2 – Using A Logical AND Operator

```c
char x = 0b1010;
char y = 0b0101;
if (x & y) printf("Hi!");
```
Operators
Logical Operators and Short Circuit Evaluation

• The evaluation of expressions in a logical operation stops as soon as a TRUE or FALSE result is known.

Example
If we have two expressions being tested in a logical AND operation:
\( \text{expr1} \land \text{expr2} \)

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>
<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>TRU E = 1</td>
<td>0</td>
<td>X (1)</td>
<td>0</td>
</tr>
<tr>
<td></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>

\( \text{expr2} \) is not evaluated in the first two cases since its value is not relevant to the result.

Example

It is perfectly legal in C to logically compare two assignment expressions in this way, though it is not usually good programming practice. A similar problem exists when using function calls in logical operations, which is a very common practice. The second function may never be evaluated.

Operators
Shift – Special Cases

• Logical Shift Right (Zero Fill)

If \( \text{x} \) is \text{UNSIGNED} (unsigned char in this case):

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

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

• Arithmetic Shift Right (Sign Extend)

If \( \text{x} \) is \text{SIGNED} (char in this case):

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

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

Power of 2 Integer Divide vs. Shift Right

• If you are dividing by a power of 2, it will usually be more efficient to use a right shift instead

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

• Works for integers or fixed point values

Right Shift

\[ 10_{10} \rightarrow \quad 0000001010 \]

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
```

Operators

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

```
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
```

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
```

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

16-Bit Shift on 8-Bit Architecture

These operators will be discussed later in the sections on arrays, pointers, structures, and unions. They are included here for reference and completeness.
### Operators

#### Other

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

### Operators

#### The Conditional Operator

**Syntax**

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

**Example**

```c
int x = 5;

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

**Result:**

5 is odd

### Operators

#### The Explicit Type Cast 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

#### Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parenthesized Expression</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>[ ]</td>
<td>Array Subscript</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>.</td>
<td>Structure Member</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Structure Pointer</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>+ -</td>
<td>Unary + and – (Positive and Negative Signs)</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>++ --</td>
<td>Increment and Decrement</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>! ~</td>
<td>Logical NOT and Bitwise Complement</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>*</td>
<td>Dereference (Pointer)</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>&amp;</td>
<td>Address of</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>sizeof</td>
<td>Size of Expression or Type</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td>(type)</td>
<td>Explicit Typecast</td>
<td>Right-to-Left</td>
</tr>
</tbody>
</table>

Continued on next slide...

#### 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;= &gt;&gt;=</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 and XOR</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Left-to-Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Left-to-Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Right-to-Left</td>
</tr>
</tbody>
</table>

Continued on next slide...

#### Precedence

- Operators grouped together in a section have the same precedence – conflicts within a section are handled via the rules of associativity.

- When expressions contain multiple operators, their precedence determines the order of evaluation.

### Operators

#### Precedence

<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

• If two operators have the same precedence, their associativity determines the order of evaluation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Associativity</th>
<th>Effective Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>x / y % z</td>
<td>Left-to-Right</td>
<td>(x / y) % z</td>
</tr>
<tr>
<td>x = y = z</td>
<td>Right-to-Left</td>
<td>x = (y = z)</td>
</tr>
<tr>
<td>~++x</td>
<td>Right-to-Left</td>
<td>~ (++x)</td>
</tr>
</tbody>
</table>

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

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++;

Solution: Steps 3 and 4

/*###########################################################################
# STEP 3: Use the conditional operator to.../longVariable1 is less than charVariable2.
# Otherwise, set longVariable1 equal to intVariable1
# NOTE: The comments below are broken up into 3 lines, but the code you
# need to write can fit on a single line.
###########################################################################*/

//If charVariable1 < charVariable2, then
longVariable1 = intVariable1; //longVariable1 = intVariable2

/*###########################################################################
# STEP 4: Shift longVariable2 one bit to the right. This can be accomplished
# most easily using the appropriate compound assignment operator.
###########################################################################*/

//Shift longVariable2 one bit to the right
longVariable2 >>= 1;

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
Questions?

Quiz Today

RPN Calculator – need help

2 5 + ← 5
Push

ABT

Local – First in Last Out

Push "a" unstack char
Push "b" char index = -99;
Push "c" char index = -99;

Check List ()
index i;
for (i = 90; i <= 122; i++)"n"
if (strchr[i] = "n")
{ index = -1;*

index = -1;