Strings

How to Initialize a String at Declaration

Character arrays may be initialized with string literals:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>char arrayName[] = &quot;Microchip&quot;;</td>
<td>char str1[] = &quot;Microchip&quot;; // 10 chars &quot;Microchip0&quot;</td>
</tr>
<tr>
<td></td>
<td>char str2[6] = &quot;Hello&quot;; // 6 chars &quot;Hello\0&quot;</td>
</tr>
<tr>
<td>// Alternative string declaration</td>
<td>char str3[] = {'M', 'i', 'c', 'r', 'o', 'h', 'i', 'p', '\0'};</td>
</tr>
</tbody>
</table>

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

Example

```c
char str[] = "Microchip";
char str2[6] = "Hello";
// Alternative string declaration
char str3[] = {'M', 'i', 'c', 'r', 'o', 'h', 'i', 'p', '\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")==0) {
    printf("The string is "\n", str);
}
```

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>+ (post)</td>
<td>Positive</td>
<td>+x</td>
<td>Value of x</td>
</tr>
<tr>
<td>- (post)</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

Implicit Type Conversion

- In many expressions, the type of one operand will be temporarily "promoted" to the larger type of the other operand

Example

```
int x = 10;
float y = 2.0; z;
x = 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

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>int</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>x * 2L</td>
<td>long-2L</td>
<td>long-2L</td>
<td>long</td>
</tr>
<tr>
<td>8 / x</td>
<td>int</td>
<td>int</td>
<td>-1</td>
</tr>
<tr>
<td>8 % x</td>
<td>int</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>8.0 / x</td>
<td>double-8.0double</td>
<td>double-8.0double</td>
<td>double</td>
</tr>
</tbody>
</table>

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

```
int a = 10;
int b = 4;
float c;

c = a / b;
c = 2.000000 ×
Because int / int = int
```

Example: Floating Point Divide

```
int a = 10;
float b = 4.0f;
float c;
c = a / b;
c = 2.500000 ✓
Because: float / int = float
```

Operators

Implicit Arithmetic Type Conversion Hierarchy

<table>
<thead>
<tr>
<th>long double</th>
<th>double</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>long int</td>
<td>signed long</td>
<td>long long</td>
</tr>
<tr>
<td>unsigned long</td>
<td>long long</td>
<td>long long</td>
</tr>
<tr>
<td>signed long</td>
<td>unsigned long</td>
<td>unsigned long</td>
</tr>
<tr>
<td>signed int</td>
<td>signed short</td>
<td>signed short</td>
</tr>
<tr>
<td>unsigned int</td>
<td>unsigned short</td>
<td>unsigned char</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: \( x \% 2^8 \))
  - 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

```
long number = 123456;
int i radius = 10;
char digits[i] = {'
for (i = 0; i < 6; i++) {
  if (number == 0) {
    break;
  }
  digits[i] = (char) (number % radius);
  number = number / radius;
}
```
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 then use x</td>
</tr>
<tr>
<td>--</td>
<td>Decrement</td>
<td>x--</td>
<td>Use x then 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

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></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:

```
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):

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

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

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

Relational

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
<th>Result (FALSE = 0, TRUE = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td><code>x &lt; y</code></td>
<td>1 if x is less than y, else 0</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to</td>
<td><code>x &lt;= y</code></td>
<td>1 if x is less than or equal to y, else 0</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td><code>x &gt; y</code></td>
<td>1 if x is greater than y, else 0</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to</td>
<td><code>x &gt;= y</code></td>
<td>1 if x is greater than or equal to y, else 0</td>
</tr>
<tr>
<td><code>==</code></td>
<td>Equal to</td>
<td><code>x == y</code></td>
<td>1 if x is equal to y, else 0</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Not equal to</td>
<td><code>x != y</code></td>
<td>1 if x is 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 ===

- What happens when the following code is executed?

```
void main(void)
{
    int x = 2; // Initialize x
    if (x == 5) {
        printf("Hi!"); // display "Hi!"
    }
}
```

Operators

Logical

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

```
if we have two expressions being tested in a logical AND operation:
expr1 && expr2
The expressions are evaluated from left to right. If expr2 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 (&amp;&amp;)</th>
<th>expr1</th>
<th>expr2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TRUE</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TRUE</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

The expression 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

```
if x = 0, then c will not be evaluated
if ((c = x + y) && (c = a + b))
{
    x = 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 |
| |           |         |                               |
| |           |         |                               |
| ^        | Bitwise XOR | `x ^ y` | 1, if 1 in `x` or `y` but not both  
0, if 0 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.

### 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)`

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

### 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>

- 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)

**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!");}
```

### Logical Shift Right (Zero Fill)

If `x` is `UNSIGNED` (unsigned char in this case):

- `x = 250; // x = 0b11111010 = 250`
- `y = x >> 2; // y = 0b00111110 = 62`

### Arithmetic Shift Right (Sign Extend)

If `x` is `SIGNED` (char in this case):

- `x = -6; // x = 0b11111010 = -6`
- `y = x >> 2; // y = 0b11111110 = -2`

**Operators**

- Logical Shift Right (Zero Fill)
- Arithmetic Shift Right (Sign Extend)

**Operators**

- Power of 2 Integer Divide vs. Shift Right

```c
y = x / 2^n
```

```c
0 0 0 0 1 0 1 0 >> 0 0 0 0 0 1 0 1
```

- Works for integers or fixed point values
Operators

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

Example: Divide by 2

```c
int x = 20;
int y;
y = x / 2;
y = 10
```

Example: Right Shift by 1

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

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>a &amp; x</td>
<td>Pointer to x</td>
</tr>
<tr>
<td>*</td>
<td>Indirection</td>
<td>*p</td>
<td>The object or function that p points to</td>
</tr>
<tr>
<td>[]</td>
<td>Subscripting</td>
<td>x[y]</td>
<td>The yth element of array x</td>
</tr>
<tr>
<td>-&gt;</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>

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

Operators

The Conditional Operator

Syntax:

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

Example

```c
int x = 5;

(x < 1 || x > 3) ?
    printf("4d is odd\n","x") :
    printf("4d 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

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

Example 1 (most commonly used)

```c
(x < 1 || x > 3) ? (x = a) : (x = b);
```

In both cases:

- x = a if condition is true
- x = b if condition is false

Example 2 (less often used)

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

Result: `2.500000`

Because: `int / int` → `int`

Example: Floating Point Divide

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

Result: `2.500000`

Because: `float / int` → `float`

---

Operators

The Conditional Operator

- Example:
  ```c
  float x = 5;
  printf("%f\n", (double)x);
  ```

Result:

- No warnings!

---

Operators

Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>()</code></td>
<td>Parenthesized Expression</td>
<td>High</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td><code>[]</code></td>
<td>Array Subscript</td>
<td>Medium</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td><code>.</code></td>
<td>Structure Member</td>
<td>Medium</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>Structure Pointer</td>
<td>Medium</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Unary + and - (Positive and Negative Signs)</td>
<td>Medium</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td><code>++</code></td>
<td>Increment and Decrement</td>
<td>Medium</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Logical OR and Bitwise Complement</td>
<td>Medium</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>Dereference (Pointer)</td>
<td>Medium</td>
<td>Right-to-Left</td>
</tr>
<tr>
<td><code>sizeof</code></td>
<td>Size of Expression or Type</td>
<td>Medium</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td><code>(type)</code></td>
<td>Explicit Typecast</td>
<td>Low</td>
<td>Right-to-Left</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) + 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 % 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 = ++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 Loop**

`for` loop

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

**Syntax**

```
for (expression1; expression2; expression3)
    statement
```

Flow Diagram

```
START
expression1
expression2
expression3
```

- `START` - Initial loop variable
- `expression1` - Modify loop variable
- `expression2` - Test loop variable for exit condition
- `expression3` - Exit condition
- `statement` - TRUE
- `expression1` - FALSE

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 `expression1` or `expression2` are missing, their actions simply disappear
- If `expression3` is missing, it is assumed to always be true

**Example (Code Fragment)**

```c
FILE *f = fopen("myfile.txt", "r");
char c;
for (c = getc(f); c != EOF; c = getc(f)) {
    printf("Char: %c\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

**Example (Code Fragment)**

```c
int i = 0;
do {
    printf("Loop iteration: \n", ++i);
} while (i < 5);
```

**Expected Output:**

```
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
Loop iteration 5
```
**break Statement**

- Causes immediate termination of a loop even if the exit condition hasn’t been met.
- Also exits from a `switch` statement.

```c
int i = 0;
while (i < 10) {
    ++i;
    if (i == 5) {
        break;
    }
    printf("Loop iteration %d
", i);
}
```

**Expected Output:**
- Loop iteration 1
- Loop iteration 2
- Loop iteration 3
- Loop iteration 4

**continue Statement**

- Causes program to finish current iteration and begin the next loop.

```c
int i = 0;
while (i < 6) {
    ++i;
    if (i == 2) {
        continue;
    }
    printf("Loop iteration %d
", i);
}
```

**Expected Output:**
- Loop iteration 1
- Loop iteration 3
- Loop iteration 4
- Loop iteration 5
- Iteration 2 does not print
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

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