Convert Temperature

- Want to write a program to convert the temperature from Fahrenheit to Celsius for a range of temperatures.
- Demonstrates a loop and use of literals

The Code

```c
#include <stdio.h>

// print Fahrenheit-Celsius table
for fahr = 0, 20, ..., 300

main()
{
    int fahr, celsius;
    int lower, upper, step;
    lower = 0; // lower limit of temperature table
    upper = 300; // upper limit
    step = 20; // step size

    fahr = lower;
    while (fahr <= upper)
    {
        celsius = 5 * (fahr - 32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```
Careful with types

- celsius = \( \frac{5 \times (\text{fahr} - 32)}{9} \);

Example

```c
unsigned int a;
unsigned int c;
define b 2

void main(void)
{
a = 5;
c = a + b;
printf("a=%d, b=%d, c=%d\n", a, b, c);
}
```

Literal Constants

**Definition**

A literal or a literal constant is a value, such as a number, character or string, which may be assigned to a variable or a constant. It may also be used directly as a function parameter or an operand in an expression.

- Literals
  - Are "hard coded" values
  - May be numbers, characters or strings
  - May be represented in a number of formats (decimal, hexadecimal, binary, character, etc.)
  - Always represent the same value (5 always represents the quantity five)

Constant vs. Literal

**What's the difference?**

- Terms are used interchangeably in most programming literature
- A literal is a constant, but a constant is not a literal
  - #define MAXINT 32767
  - const int MAXINT = 32767;
- For purposes of this presentation:
  - Constants are labels that represent a literal
  - Literals are values, often assigned to symbolic constants and variables
Literal Constants

• Four basic types of literals:
  – Integer
  – Floating Point
  – Character
  – String

• Integer and Floating Point are numeric type constants:
  – Commas and spaces are not allowed
  – Value cannot exceed type bounds
  – May be preceded by a minus sign

Integer Literals

Decimal (Base 10)

• Cannot start with 0 (except for 0 itself)
• Cannot include a decimal point
• Valid Decimal Integers:
  0 5 127 -1021 65535
• Invalid Decimal Integers:
  32,767 25.0 1,024 0552

Integer Literals

Hexadecimal (Base 16)

• Must begin with 0x or 0X (that’s zero-x)
• May include digits 0-9 and A-F / a-f
• Valid Hexadecimal Integers:
  0x 0x1 0x0A2B 0xBEEF
• Invalid Hexadecimal Integers:
  0x5.3 0EA12 0xEG 53h

Integer Literals

Octal (Base 8)

• Must begin with 0 (zero)
• May include digits 0-7
• Valid Octal Integers:
  0 01 012 073125
• Invalid Octal Integers:
  053 0012 000 53h

While Octal is still part of the ANSI specification, almost no one uses it anymore.
**Integer Literals**

**Binary (Base 2)**
- Must begin with `0b` or `0B` (that’s zero-b)
- May include digits 0 and 1
- Valid Binary Integers: `0b 0 0b1 0b010100110000111`
- Invalid Binary Integers: `0b100 01100 0b12 10b`

ANSI C does not specify a format for binary integer literals. However, this notation is supported by most compilers.

**Qualifiers**
- Like variables, literals may be qualified
- A suffix is used to specify the modifier
  - ‘U’ or ‘u’ for unsigned: `25U`
  - ‘L’ or ‘l’ for long: `25L`
  - ‘F’ or ‘f’ for float: `10f` or `10.25F`
- Suffixes may be combined: `0xF5UL`
  - Note: U must precede L
- Numbers without a suffix are assumed to be signed and short

**Floating Point Literals**

**Decimal (Base 10)**
- Like decimal integer literals, but decimal point is allowed
- ‘e’ notation is used to specify exponents (
  $k\times{n \pm n}$)
- Valid Floating Point Literals: `2.56e-5 10.4378 48e8 0.5 10f`
- Invalid Floating Point Literals: `0x5Ae-2 02.41 02.33`

**Character Literals**
- Specified within single quotes (`'`)
- May include any single printable character
- May include any single non-printable character using escape sequences (e.g. `\0` = NUL) (also called digraphs)
- Valid Characters: `a`, `T`, `\n`, `5`, `@`, ` ' ` (space)
- Invalid Characters: `me`, `23`
String Literals

• Specified within double quotes ("")
• May include any printable or non-printable characters (using escape sequences)
• Usually terminated by a null character '\0'
• Valid Strings: "Microchip", "Hi\n", "PIC", "2500", "rob@microchip.com", "He said, \"Hi\""
• Invalid Strings: "He said, \"Hi\"

String Literals

• Strings are a special case of arrays
• If declared without a dimension, the null character is automatically appended to the end of the string:

Example 1 – Wrong Way
```
char color[3] = "RED";
```
Is stored as:
```
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
```

Example 2 – Right Way
```
char color[] = "RED";
```
Is stored as:
```
color[0] = 'R'
color[1] = 'E'
color[2] = 'D'
color[3] = '\0'
```

NOT a complete string – no '\0' at end

String Literals

How to Include Special Characters in Strings

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Character</th>
<th>ASCII Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>BELL (alert)</td>
<td>7</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
<td>8</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal Tab</td>
<td>9</td>
</tr>
<tr>
<td>\n</td>
<td>Newline (Line Feed)</td>
<td>10</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical Tab</td>
<td>11</td>
</tr>
<tr>
<td>\f</td>
<td>Form Feed</td>
<td>12</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage Return</td>
<td>13</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quotation Mark (&quot;)</td>
<td>34</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe/Single Quote (')</td>
<td>39</td>
</tr>
<tr>
<td>?</td>
<td>Question Mark (?)</td>
<td>63</td>
</tr>
<tr>
<td>|</td>
<td>Backslash ()</td>
<td>92</td>
</tr>
<tr>
<td>\0</td>
<td>Null</td>
<td>0</td>
</tr>
</tbody>
</table>

Example
```
char message[] = "Please enter a command…\n"
```

• This string includes a newline character
• Escape sequences may be included in a string like any ordinary character
• The backslash plus the character that follows it are considered a single character and have a single ASCII value
Symbolic Constants

Definition
A constant or a symbolic constant is a label that represents a literal. Anywhere the label is encountered in code, it will be interpreted as the value of the literal it represents.

• Constants
  – Once assigned, never change their value
  – Make development changes easy
  – Eliminate the use of "magic numbers"
  – Two types of constants
    • Text Substitution Labels
    • Variable Constants (!!??)

Text Substitution Labels

• Defines a text substitution label

Syntax
#define label text

- Each instance of label will be replaced with text by the preprocessor unless label is inside a string
- No memory is used in the microcontroller

Example
#define PI 3.14159
#define mol 6.02E23
#define MCU "PIC24FJ128GA010"
#define COEF 2 * PI

Variable Constants (!!!?)

• Some texts on C declare constants like:

Example
const float PI = 3.141593;

• This is not efficient for an embedded system:
  A variable is allocated in program memory, but it cannot be changed due to the const keyword
• This is not the traditional use of const
• In the vast majority of cases, it is better to use #define for constants

#define Gotchas

• Note: a #define directive is NEVER terminated with a semi-colon (;), unless you want that to be part of the text substitution.

Example
#define MyConst 5;
c = MyConst + 3;
c = 5; + 3;
Symbolic Constants

Initializing Variables When Declared

- A constant declared with `const` may not be used to initialize a global or static variable when it is declared (though it may be used to initialize local variables...)

Example

```c
#define CONSTANT1 5
const CONSTANT2 = 10;

int variable1 = CONSTANT1;
int variable2;
// Cannot do: int variable2 = CONSTANT2
```

Exercise 02

Symbolic Constants

- Open the lab Project:

  On the class website
  Examples -> Lab2.zip

  1. Open MPLAB®X and select Open Project Icon (Ctrl + Shift + O)
  2. Open the project listed above.
  3. If you already have a project open in MPLAB X, close it by “right clicking” on the open project and selecting “Close”

Lab Exercise 2

Symbolic Constants

Exercise 02

Symbolic Constants

- Compile and run the code:

  1. Click on the Debug Project button.
  2. If no errors are reported, click on Continue button to start the program.
  3. Click on the Pause button.
Exercise 02
Symbolic Constants

• Expected Results (1):
The UART 1 Output window should show the text that is output by the program, indicating the values of the two symbolic constants in the code.

5

• Expected Results (2):

Only CONSTANT2 can be added to Watches Window
CONSTANT1 cannot be added
CONSTANT2 has address of 0x8CC

• Expected Results (3):

Program Memory only contains CONSTANT2
CONSTANT1 not in Program Memory

Program Memory

CONSTANT2 has a value of 0x00CC

• Conclusions

- Constants make code more readable
- Constants improve maintainability
- #define should be used to define constants
- #define constants use no memory, so they may be used freely
- const should never be used in this context (it has other uses...)
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-SIM

**Syntax**

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

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

**Example**

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

Result:

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

**Conversion Characters for Control String**

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

**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() Useful Format String Examples for Debugging

- Print a 16-bit hexadecimal value with a "0x" prefix and leading zeros if necessary to fill a 4 hex digit value:

```c
printf("Address of x = %#06x\n", x_ptr);
```

- Same as previous, but force hex letters to uppercase while leaving the 'x' in '0x' lowercase:

```c
printf("Address of x = 0x%04X\n", x_ptr);
```

### Lab Exercise 3

printf() Library Function

**Exercise 03**

printf() Library Function

- Open the lab Project:

  1. Open MPLAB®X and select Open Project Icon (Ctrl + Shift + O)
  2. 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 03
printf() Library Function

- Compile and run the code:
  - Click on the Debug Project button.
  - If no errors are reported, click on Continue button to start the program.
  - Wait for the UART1 Output to finish the click on the Pause button.

Expected Results (1):
- The UART1 Output window should show the text that is output by the program by printf(), showing the how values are printed based on the formatting character used in the control string.

Lab 03
printf() Library Function

Expected Results (2):
- Detailed Analysis:

<table>
<thead>
<tr>
<th>Line of Code From Demo Project</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf(&quot;25 as decimal (d): %d\n&quot;, 25)</td>
<td>25</td>
</tr>
<tr>
<td>printf(&quot;'a' as character (c): %c\n&quot;, 'a')</td>
<td>a</td>
</tr>
<tr>
<td>printf(&quot;'a' as decimal (d): %d\n&quot;, 'a')</td>
<td>97</td>
</tr>
<tr>
<td>printf(&quot;2.55 as float (f): %f\n&quot;, 2.55)</td>
<td>2.550000</td>
</tr>
<tr>
<td>printf(&quot;2.55 as decimal (d): %d\n&quot;, 2.55)</td>
<td>16419</td>
</tr>
<tr>
<td>printf(&quot;6.02e23 as exponent (e): %e\n&quot;, 6.02e23)</td>
<td>6.020000e+23</td>
</tr>
<tr>
<td>printf(&quot;6.02e23 as decimal (d): %d\n&quot;, 6.02e23)</td>
<td>26366</td>
</tr>
<tr>
<td>printf(&quot;'Microchip' as string (s): %s\n&quot;, &quot;Microchip&quot;)</td>
<td>Microchip</td>
</tr>
<tr>
<td>printf(&quot;'Microchip' as decimal (d): %d\n&quot;, &quot;Microchip&quot;)</td>
<td>-24058</td>
</tr>
</tbody>
</table>
**Exercise 03**

**Conclusions**

- `printf()` has limited use in embedded applications themselves
- It is very useful as a debugging tool
- It can display data almost any way you want
- Projects that use `printf()` must:
  - Configure a heap (done in MPLAB® X-IDE)
  - Include the `stdio.h` header file

---

**scanf()**

**Standard Library Function**

- Used to read input from the "standard input"
- Normally a keyboard or file
- Often the UART in embedded systems
- **File input** into the SIM in MPLAB-X
- `sscanf()` inputs from string
- `fscanf()` inputs from file
**scanf()**

**Gotchas**

- Ignores blanks and tabs in format string
- Skips over white space (blanks, tabs, newline) as it looks for input
- Returns number of successful conversions

- Arguments **must** be **pointers** to variable types

**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 int, &month is pointer to month
- year  int, &year is pointer to year

**Questions?**