Comments

**Definition**

Comments are used to document a program's functionality and to explain what a particular block or line of code does. Comments are ignored by the compiler, so you can type anything you want into them.

- Two kinds of comments may be used:
  - Block Comment
    ```c
    /* This is a comment */
    ```
  - Single Line Comment
    ```c
    // This is also a comment
    ```

Comments are used to document a program's functionality and to explain what a particular block or line of code does. Comments are ignored by the compiler, so you can type anything you want into them.

**Using Block Comments**

- Block comments:
  - Begin with /* and end with */
  - May span multiple lines

```c
#include <stdio.h>

/* Function: main() */
int main(void)
{
    printf("Hello, world!\n");  // Display "Hello, world!"
}
```
Comments
Using Single Line Comments

- Single line comments:
  - Begin with // and run to the end of the line
  - May not span multiple lines

```
/**
 * Program: hello.c
 * Author: R. Ostapiuk
 */
#include <stdio.h>

// Function: main()
int main(void)
{
    printf("Hello, world!\n"); // Display "Hello, world!"
}
```

Comments
Nesting Comments

- Block comments may not be nested within other delimited comments
- Single line comments may be nested

Example: Single line comment within a delimited comment.
```
/*
 * code here  // Comment within a comment
 */
```

Example: Delimited comment within a delimited comment.
```
/*
 * code here  /* Comment within a comment */
 * code here  /* Comment within a... oops! */
 */
```

Dangling delimiter causes compile error
Delimiters don't match up as intended!
Comments

Best Practices

/***************************************************************************/
* Program: hello.c
* Author: R. Ostapiuk
***************************************************************************/
#include <stdio.h>

/***************************************************************************/
* Function: main()
***************************************************************************/
int main(void)
{
    int i; // Loop count variable
    char *p; // Pointer to text string
    printf("Hello, world!\n"); // Display "Hello, world!"
}
Example

```c
#include <stdio.h>
#define PI 3.14159

int main(void)
{
    float radius, area;
    // Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = %f", area);
}
```

Variables

A variable is a name that represents one or more memory locations used to hold program data.

- A variable may be thought of as a container that can hold data used in a program.
Variables

• Variables are names for storage locations in memory

```c
int warp_factor;
char first_letter;
float length;
```

Variables

• Variable declarations consist of a unique identifier (name)...

```c
int warp_factor;
char first_letter;
float length;
```
Variables

• ...and a **data type**
  - Determines size
  - Determines how values are interpreted

```
int warp_factor;
char first_letter;
float length;
```

![Diagram showing memory allocation for variables]

**Identifiers**

• **Names** given to program elements:
  - **Variables**, **Functions**, **Arrays**, **Other elements**

```
#include <stdio.h>
#define PI 3.14159
int main(void)
{
    float radius, area;
    //Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = \%f", area);
}
```
### Identifiers

- Valid characters in identifiers:

  ![Identifier Diagram]

  - First Character
    - '_' (underscore)
    - 'A' to 'Z'
    - 'a' to 'z'
  - Remaining Characters
    - '_' (underscore)
    - 'A' to 'Z'
    - 'a' to 'z'
    - '0' to '9'

- Case sensitive!

- Only first 31 characters significant*

---

### ANSI C Keywords

<table>
<thead>
<tr>
<th>auto</th>
<th>double</th>
<th>int</th>
<th>struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>else</td>
<td>long</td>
<td>switch</td>
</tr>
<tr>
<td>case</td>
<td>enum</td>
<td>register</td>
<td>typedef</td>
</tr>
<tr>
<td>char</td>
<td>extern</td>
<td>return</td>
<td>union</td>
</tr>
<tr>
<td>const</td>
<td>float</td>
<td>short</td>
<td>unsigned</td>
</tr>
<tr>
<td>continue</td>
<td>for</td>
<td>signed</td>
<td>void</td>
</tr>
<tr>
<td>default</td>
<td>goto</td>
<td>sizeof</td>
<td>volatile</td>
</tr>
<tr>
<td>do</td>
<td>if</td>
<td>static</td>
<td>while</td>
</tr>
</tbody>
</table>

- Some compiler implementations may define additional keywords
### Data Types

**Fundamental Types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>single character</td>
<td>8</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
<td>16</td>
</tr>
<tr>
<td>float</td>
<td>single precision floating point number</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>double precision floating point number</td>
<td>64</td>
</tr>
</tbody>
</table>

The size of an `int` varies from compiler to compiler.
- MPLAB-X XC16 `int` is 16-bits

If you need precise length variable types, use `stdint.h`
- `uint8_t` is unsigned 8 bits
- `int16_t` is signed 16 bits, etc.

### Data Type Qualifiers

**Modified Integer Types**

Qualifiers: `unsigned`, `signed`, `short` and `long`

<table>
<thead>
<tr>
<th>Qualified Type</th>
<th>Min</th>
<th>Max</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsigned char</code></td>
<td>0</td>
<td>255</td>
<td>8</td>
</tr>
<tr>
<td><code>char, signed char</code></td>
<td>-128</td>
<td>127</td>
<td>8</td>
</tr>
<tr>
<td><code>unsigned short int</code></td>
<td>0</td>
<td>65535</td>
<td>16</td>
</tr>
<tr>
<td><code>short int, signed short int</code></td>
<td>-32768</td>
<td>32767</td>
<td>16</td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td>0</td>
<td>65535</td>
<td>16</td>
</tr>
<tr>
<td><code>int, signed int</code></td>
<td>-32768</td>
<td>32767</td>
<td>16</td>
</tr>
<tr>
<td><code>unsigned long int</code></td>
<td>0</td>
<td>$2^{32} - 1$</td>
<td>32</td>
</tr>
<tr>
<td><code>long int, signed long int</code></td>
<td>$-2^{31}$</td>
<td>$2^{31} - 1$</td>
<td>32</td>
</tr>
<tr>
<td><code>unsigned long long int</code></td>
<td>0</td>
<td>$2^{64} - 1$</td>
<td>64</td>
</tr>
<tr>
<td><code>long long int</code>, <code>signed long long int</code></td>
<td>$-2^{63}$</td>
<td>$2^{63} - 1$</td>
<td>64</td>
</tr>
</tbody>
</table>
Data Type Qualifiers

Modified Floating Point Types

<table>
<thead>
<tr>
<th>Qualified Type</th>
<th>Absolute Min</th>
<th>Absolute Max</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>± ~10^{44.85}</td>
<td>± ~10^{38.53}</td>
<td>32</td>
</tr>
<tr>
<td>double^{(1)}</td>
<td>± ~10^{44.85}</td>
<td>± ~10^{38.53}</td>
<td>32</td>
</tr>
<tr>
<td>long double</td>
<td>± 10^{323.3}</td>
<td>± 10^{308.3}</td>
<td>64</td>
</tr>
</tbody>
</table>

MPLAB-X XC16: ^{(1)}double is equivalent to long double if
-fno-short-double is used

MPLAB-X XC16 Uses the IEEE-754 Floating Point Format

Variables

How to Declare a Variable

Syntax

\[ \text{type identifier}_1, \text{identifier}_2, \ldots, \text{identifier}_n; \]

- A variable must be declared before it can be used
- The compiler needs to know how much space to allocate and how the values should be handled

Example

\[
\begin{align*}
\text{int } & x, y, z; \\
\text{float } & \text{warpFactor;} \\
\text{char } & \text{text_buffer[10];} \\
\text{unsigned } & \text{index;} \\
\end{align*}
\]
Variables

How to Declare a Variable

Variables may be declared in a few ways:

### Syntax

- **One declaration on a line**

  ```
  type identifier;
  ```

- **One declaration on a line with an initial value**

  ```
  type identifier = InitialValue;
  ```

- **Multiple declarations of the same type on a line**

  ```
  type identifier1, identifier2, identifier3;
  ```

- **Multiple declarations of the same type on a line with initial values**

  ```
  type identifier1 = Value1, identifier2 = Value2;
  ```

### Examples

- ```
    unsigned int x;
    unsigned y = 12;
    int a, b, c;
    long int myVar = 0x12345678;
    long z;
    char first = 'a', second, third = 'c';
    float big_number = 6.02e+23;
  ```

It is customary for variable names to be spelled using "camel case", where the initial letter is lower case. If the name is made up of multiple words, all words after the first will start with an upper case letter (e.g. `myLongVarName`).
Variables
How to Declare a Variable

• Sometimes, variables (and other program elements) are declared in a separate file called a header file

• Header file names customarily end in .h

• Header files are associated with a program through the #include directive

#include Directive

• Three ways to use the #include directive:

Syntax

```
#include <file.h>
Look for file in the compiler search path
The compiler search path usually includes the compiler’s directory and all of its subdirectories.
For example: C:\Program Files\Microchip\MPLABX\XC16\*. *

#include "file.h"
Look for file in project directory only

#include "c:\MyProject\file.h"
Use specific path to find include file
```
The contents of main.h are effectively pasted into main.c starting at the #include directive’s line.

After the preprocessor runs, this is how the compiler sees the main.c file:

- The contents of the header file aren’t actually copied to your main source file, but it will behave as if they were copied.
Exercise 1

Variables and Data Types
Exercise 01
Variables and Data Types

• Open the lab Project:

- On the class website
- Examples -> Lab1.zip

1. 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 01
Variables and Data Types

• Compile and run the code:

2. Click on the **Debug Project** button.

3. If no errors are reported, click on **Continue** button to start the program.

4. Click on the **Pause** button.
Exercise 01
Variables and Data Types

• Expected Results (1):

The UART 1 Output window should show the text that is output by the program, indicating the sizes of C’s data types in bytes.

• Expected Results (2):

The Variables window (alt + shift + 1) shows the values and addresses of the variables.
Exercise 01
Variables and Data Types

• What does the code do?

Example lines of code from the demo program:

```c
#define CONSTANT1 50

int intVariable;

intVariable = CONSTANT1;

printf("\nAn integer variable requires %d bytes.", sizeof(int));

while(1);
```
Exercise 01
Conclusions

• Variables must be **declared before used**
• Variables must have a **data type**
• Data type determines memory use
• Most efficient data types:
  – int on 16-bit architectures
  – char on 8-bit architectures (if int is 16-bit)
• Don't use float/double unless you really need them

//

Questions?
.h "header" - Information required to use the module
"API"

\* Public function prototypes

\* \* "code" - Implementation

\* \* Private
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

#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 = 5 * (fahr - 32) / 9;

Be careful w/ container size

Example

```
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);
}
```

A Simple C Program

Literal Constants
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)