Software Architecture

- Often orthogonal to Project Management
- Can be the success or failure of a project

alt + shift + f
Flow Chart and Pseudo-Code

- Pseudo-code is plain English that explains in coarse steps what the code should do
  - Not syntax specific
  - Forms the basis for your top-level comment
  - Hides details of programming language
Style Examples

- Adherence to a specific style
  - Variables in camelCase with leading lower case
  - Functions in CamelCase with leading Upper case
  - `#define` (literal constants) are in UPPERCASE
    - Exception: macros can look like functions if they act like one
  - Variable and Function names are descriptive
    - Eg: `backingUpState = TRUE;`
  - Correct use of white space and indentation
  - Correct placement of braces `{}`
Development Tools Data Flow

C Source File → Preprocessor → Compiler → Assembly Source File

C Header File
Compilation

- **Source Code Analysis**
  - “front end”
  - parses programs to identify its pieces
  - variables, expressions, statements, functions, etc.
  - depends on language (not on target machine)

- **Code Generation**
  - “back end”
  - generates machine code from analyzed source
  - may optimize machine code to make it run more efficiently
  - dependent on target architecture

- **Symbol Table**
  - map between symbolic names and items
  - like assembler, but more kinds of information
Fundamentals of C

A Simple C Program

Example

Preprocessor Directives

\#include <stdio.h>

\#define PI 3.14159

Header File

Constant Declaration (Text Substitution Macro)

int main(void)
{
    float radius, area;

    //Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = %f", area);
}

Function

Variable Declarations

Comment
C Runtime Environment (1.2)

• C Runtime is the “backend” of C:
  – Allocates space for stack
  – Initialize stack pointer
  – Allocates space for heap
  – Copies values from Flash/ROM to variables in RAM that were declared with initial values
  – Clear uninitialized RAM
  – Disable all interrupts
  – Call main() function (where your code starts)
C Runtime Environment (2.2)

- Runtime environment setup code is automatically linked into your application by XC32 compiler.
- Code comes from:
  - XC32: crt0.s / crt0.o (crt = C RunTime)
- User modifiable if absolutely necessary.
Stack/Heap

- Stack grows from bottom “up”

- Heap grows from top “down”
Stack/Heap
Questions?
• Lab kits being distributed
  – Come to Friday’s section if at all possible to pick it up
• Reload the lab manuals
• Check piazza
• Keep your git safe
  • Unit.xic
C: A High Level Programming Language

- Gives symbolic names to values
  - Don’t need to know which register or memory location
- Provides abstraction of underlying hardware
  - operations do not depend on instruction set
  - example: can write “a = b * c”, even though underlying hardware may not have a multiply instruction
C: A High Level Programming Language

- Provides expressiveness
  - use meaningful symbols that convey meaning
  - simple expressions for common control patterns (if-then-else)
- Enhances code readability
- Safeguards against bugs
  - can enforce rules or conditions at compile-time or run-time

< 60 2
Compilation vs. Interpretation

- Different ways of translating high-level language

- **Interpretation**
  - interpreter = program that executes program statements
  - generally one line/command at a time
  - limited processing
  - easy to debug, make changes, view intermediate results
  - languages: BASIC, LISP, Perl, Java, Matlab, Python
Compilation vs. Interpretation

• Compilation
  – translates statements into machine language
  – does not execute, but creates executable program
  – performs optimization over multiple statements
  – change requires recompilation
    • can be harder to debug, since executed code may be different
  – languages: C, C++, Fortran, Pascal, Ada
Compilation vs. Interpretation

- Consider the following algorithm:
  
  Get W from the keyboard.
  
  \[ X = W + W \]
  
  \[ Y = X + X \]
  
  \[ Z = Y + Y \]
  
  Print Z to screen.

  \[ Z = 8 W \]

- If interpreting, how many arithmetic operations occur?

- If compiling, we can analyze the entire program and possibly reduce the number of operations. Can we simplify the above algorithm to use a single arithmetic operation?
Compilation

• **Source Code Analysis**
  – “front end”
  – parses programs to identify its pieces
  – variables, expressions, statements, functions, etc.
  – depends on language (not on target machine)

• **Code Generation**
  – “back end”
  – generates machine code from analyzed source
  – may optimize machine code to make it run more efficiently
  – very dependent on target machine

• **Symbol Table**
  – map between symbolic names and items
  – like assembler, but more kinds of information
“Hello World”

- The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:
  
  \[ \text{Print the words} \quad \text{[hello, world]} \]

- This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went.

- With these mechanical details mastered, everything else is comparatively easy.
```c
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n"); // Uses the I/O library to print
    return 0;
}
```
Embedded C Code

```c
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");

    while (1); // Loop forever and never return
}
```
int main(void)
{
    while (1) {
        // Read inputs
        // Perform calculations
        printf("Hello world");
        // Update outputs
    }
}
Setting up the IDE
Configuring the Simulator

Set the Debug simulator to wait at the beginning of the main() function.
Resetting MPLAB®X windows

As you will see MPLABX has numerous adjustable windows. New MPLABX users can get a little confused about where and how the set the windows.

If you get confused

**Windows -> Reset Window**

Restores MPLABX Windows back to their original locations
Opening a Project

Select the Open Project button
Opening a Project

1) Navigate to the Project Directory

2) Select the Project

3) Select Open Project
Opening a Project

Project will Open in MPLAB X
Building a Project

To build the project and send it to the Debugger select the **Debug Project Button**
Building a Project

Simulation ready to start

Successful Build

Control Buttons Appear
Running the Simulation

To run the project push the Continue button
Pausing the Simulation

To pause execution of the simulation hit the Pause button.
Windows used in Examples

Variables Window

**Variable Window** displays a particular set of program variables.

To Open the Variables window:

Select:
Windows->Debugging->Variables
Windows used in Examples

Variables Window

**Variable Window** displays several columns of data

![Variables Window](image)

You may find it convenient to alter the columns displayed.

“right click” on the column heading
Windows used in Examples

**UART1 Output**

**UART1 Output Window** prints out text from C programs

To clear this window:

Right click *inside* of the window then select *Clear*.
Windows used in Examples

Watches Window

Watches Window is similar to the Variables window but displays a different set of data.

To Open the Watches Window:

Select:
Windows->Debugging->Watches
Windows used in Examples

Watches Window

Watches Window needs to be ‘told’ what data to watch

“Right click” while in the Watches Window to add or delete watches

** Column configuration is identical to Variables Window
Closing a Project

1. Stop the simulation by pushing the Finish Debugger Session button.


CMPE-013/L: “C” Programming
Closing a Project

2) "Right Click" on the project name then select Close
Preprocessor Directives

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

Header File

```
int main(void)
{
    float radius, area;

    //Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = \%f", area);
}
```

Constant Declaration (Text Substitution Macro)

Variable Declarations

Comment
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
    /* This is a comment */
  
  – Single Line Comment
    // This is also a comment
Comments
Using Block Comments

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

```c
/***************************************************************************/
* Program: hello.c
* Author:  R. Ostapiuk
***************************************************************************/
#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

```c
#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 comment...
     oops! */
     */

Delimiters don’t match up as intended!
Dangling delimiter causes compile error
Comments
Best Practices/Doxygen

/**
 * @file
 * @author R. Ostapiuk
 * @section DESCRIPTION
 * This is an example Hello World program
 */

#include <stdio.h>

/**
 * Main, the entrypoint for this C program.
 * @return A success code, where non-zero values indicate failure
 */

int main(void)
{
    int i;            // Loop counter variable
    char *p;          // Pointer to text string

    // Display greeting
    printf("Hello, world!\n");
}
Variable Declarations
# Variables and Data Types

## A Simple C Program

```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);
}
```
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.

```c
int myVariable;
myVariable = 5;
```
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)...

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

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

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

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

Example of Identifiers in a Program

```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);
}
```
Identifiers

- Valid characters in identifiers:
  - 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

---


CMPE-013/L: “C” Programming
# 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.

- XC16 int as 16-bits
- XC32 defines int as 32-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>unsigned char</td>
<td>0</td>
<td>255</td>
<td>8</td>
</tr>
<tr>
<td>char, signed char</td>
<td>-128</td>
<td>127</td>
<td>8</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0</td>
<td>65535</td>
<td>16</td>
</tr>
<tr>
<td>short int, signed short int</td>
<td>-32768</td>
<td>32767</td>
<td>16</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>65535</td>
<td>16</td>
</tr>
<tr>
<td>int, signed int</td>
<td>-32768</td>
<td>32767</td>
<td>16</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0</td>
<td>$2^{32}-1$</td>
<td>32</td>
</tr>
<tr>
<td>long int, signed long int</td>
<td>$-2^{31}$</td>
<td>$2^{31}-1$</td>
<td>32</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>0</td>
<td>$2^{64}-1$</td>
<td>64</td>
</tr>
<tr>
<td>long long int, signed long long int</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>$\pm 10^{-44.85}$</td>
<td>$\pm 10^{38.53}$</td>
<td>32</td>
</tr>
<tr>
<td>double$^{(1)}$</td>
<td>$\pm 10^{-44.85}$</td>
<td>$\pm 10^{38.53}$</td>
<td>32</td>
</tr>
<tr>
<td>long double</td>
<td>$\pm 10^{-323.3}$</td>
<td>$\pm 10^{308.3}$</td>
<td>64</td>
</tr>
</tbody>
</table>

MPLAB-X XC32 Uses the IEEE-754 Floating Point Format
Variables
How to Declare a Variable

Syntax

\texttt{type\ identifier_1,\ identifier_2,\ldots,\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{verbatim}
int x, y, z;    // int y=0, z=0;
float warpFactor;
char text_buffer[10];
unsigned index;
\end{verbatim}
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 identifier_1, identifier_2, identifier_3;
```

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

```
type identifier_1 = Value_1, identifier_2 = Value_2;
```
Variables
How to Declare a Variable

Examples

```c
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
```
#include Directive

main.h Header File and main.c Source File

#include "main.h"

int main(void)
{
    a = 5;
    b = 2;
    c = a+b;
}

unsigned int a;
unsigned int b;
unsigned int c;

The contents of main.h are effectively pasted into main.c starting at the #include directive’s line
#include Directive

Equivalent main.c File

- 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

```c
unsigned int a;
unsigned int b;
unsigned int c;

int main(void)
{
    a = 5;
    b = 2;
    c = a+b;
}
```

Equivalent main.c file without #include
Header Guards
Duplicate #includes

main.h

unsigned int a;
unsigned int b;
unsigned int c;

main.c

#include "main.h"
#include "main.h"

int main(void)
{
    a = 5;
    b = 2;
    c = a + b;
}

The contents of main.h are effectively pasted twice into main.c starting at the #include directive’s line
Header guards
Equivalent main.c File

- Duplicate declarations will occur.
- Which will give compilation errors as there cannot exist multiple declarations of the same variable in the same scope.

```c
#include <stdio.h>

int main(void)
{
    unsigned int a;
    unsigned int b;
    unsigned int c;

    ...

    Equivalent main.c file without include
}
```
Header guards
Realistic example

```c
#include "OledDriver.h"

#include "Oled.h"

main.c
```

```c
#include "OledDriver.h"
#include "Oled.h"
```
Header guards
How do you write/use them

- Declare a macro when a header file is processed.
- Check for that macro before including the code.

```c
#ifndef OLED_H
#define OLED_H

#include "OledDriver.h"

#define foo
...
#endif // OLED_H
```
Questions?