CMPE-013/L

Computer Systems and “C” Programming

Maxwell James Dunne

<512<
About: Me

• Undergraduate at UCSC in Computer Engineering and Electrical Engineering.

• Masters from UCSC

• Worked on a mountain lion tracking collar in Graduate School

• Renovated and built new hardware for CMPE118 based off the Uno32
Class Resources

- CANVAS
- Piazza
- Auxiliary Website
  - Storage for lecture videos and slides.
# CMPE-013/L Syllabus

**UNIVERSITY OF CALIFORNIA, SANTA CRUZ**  
**BOARD OF STUDIES IN COMPUTER ENGINEERING**

**CMPE-13/L: COMPUTER SYSTEMS AND C PROGRAMMING**  
**SPRING 2013 LECTURE/LAB CALENDAR**

"Notes to accompany K&R," by Steve Summit available on the class website and also available at:  
http://www.exkimino.com/Linux/class/knotes/top.html [Notes]

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Reading (to be completed before lecture)</th>
<th>Out</th>
<th>Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Intro to C Programming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>04-Apr-2013 Hello World, History, Tools, C Runtime, Heap/Stack</td>
<td>Lab 0: Compiling, Running, and Debugging Due: 10-Apr-13 @ 11:55 PM</td>
<td>Quiz 0 (Intake Quiz)</td>
</tr>
<tr>
<td>2</td>
<td>09-Apr-2013 Comments and Variables</td>
<td>Lab 1: Simple Calculator Due: 17-Apr-13 @ 11:55 PM</td>
<td>Lab 0</td>
</tr>
<tr>
<td>3</td>
<td>11-Apr-2013 Literals and Constants</td>
<td>Lab 2: LED Bounce Due: 24-Apr-13 @ 11:55 PM</td>
<td>Lab 1</td>
</tr>
<tr>
<td>4</td>
<td>16-Apr-2013 Functions, Modules, Scope</td>
<td>Lab 3: Matrix Math Library Due: 02-May-13 @ 11:55 PM</td>
<td>Lab 2</td>
</tr>
<tr>
<td>5</td>
<td>18-Apr-2013 Expressions and Loop Control</td>
<td>Lab 4: RPM Calculator Due: 08-May-13 @ 11:55 PM</td>
<td>Lab 3</td>
</tr>
<tr>
<td>6</td>
<td>23-Apr-2013 Operators and Expressions</td>
<td>Lab 5: Linked List Sort Due: 15-May-13 @ 11:55 PM</td>
<td>Lab 4</td>
</tr>
<tr>
<td>7</td>
<td>25-Apr-2013 Precedence and Associativity</td>
<td>Lab 6: LCD Driver Due: 22-May-13 @ 11:55 PM</td>
<td>Lab 5</td>
</tr>
<tr>
<td>8</td>
<td>30-Apr-2013 Scope and Arrays</td>
<td>Lab 7: BattleBoats Due: 29-May-13 @ 11:55 PM</td>
<td>Lab 6</td>
</tr>
<tr>
<td>9</td>
<td>02-May-2013 Strings and Malloco/Free</td>
<td>Lab 8: Morse Code Decoder Due: 06-Jun-13 @ 11:55 PM</td>
<td>Lab 7</td>
</tr>
<tr>
<td>10</td>
<td>07-May-2013 Pointers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>09-May-2013 Pointers and Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>14-May-2013 Pointers and Functions (cont) and Structs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>16-May-2013 Event Driven Programming</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>21-May-2013 State Machines, Modular C Programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>23-May-2013 Unions and Bitfields</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>28-May-2013 Exercises and Problem Resolution</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CMPE-013/L Piazza

University of California, Santa Cruz - Spring 2013
CMPE 13/L: Computer Systems and C Programming

Description
Computer Systems and C Programming is a class intended to bring you up to speed on programming small and large programs in C. Originally written in 1978, C remains the most popular programming language, and is the most used one in terms of numbers of computer programs written in it.

There are no prerequisites, but you are expected to be at least somewhat familiar with programming and computers (CMPE 12 is recommended).

In this class, we are going to approach C from an embedded paradigm, and all of your programming assignments are going to be on a 16-bit embedded micro, the Microchip PIC24 (or dsPIC33).

You will learn how to program in C, how to write modular code, and some of the tips and tricks when dealing with an embedded micro.

This is a programming class and you will be writing lots of code. Expect to spend at least 15-20 hours outside of class playing with the code to get things to work.

General Information
Lecture Information:
T Th 2:00 - 3:45 PM
CLASS: Physical Sciences 114
LAB: Jack Basson Engineering, 109
Ming Ong Lab, 108

Textbooks:

K&R availability on Amazon and AbeBooks
3/29/13 9:37 PM

It is also available (both new and used) on Amazon:
http://www.amazon.com/Programming-Language-2nd-Edition-Kernighan/dp/0131103628/ref=sr_1_1?

And it looks like quite a bit less on AbeBooks:
http://www.abebooks.com/products/isbn/9780131103627

If you are going to order the book, do so early so you aren't waiting for it to arrive.

--G
#pin
View on Piazza
CMPE-013/L Books


• **[Notes]**: “Notes to accompany K&R,” by Steve Summit available on the class website and at: http://www.eskimo.com/~scs/cclass/krnotes/top.html

CMPE-013/L Optional Books


CMPE-013/L Optional Books


- **[Zyante]**: “Programming in C,” interactive C book online by Frank Vahid and Smita Bakshi (not free) at: [https://zybooks.zyante.com/#/zybook/7VU9pYQ5ue/tableofcontents](https://zybooks.zyante.com/#/zybook/7VU9pYQ5ue/tableofcontents)
CMPE-013/L Attendance

• Highly Recommended
  – Material builds up quickly
  – Videos available after delay, but not the same as being there

• In class quizzes, at the beginning of class at least once a week
  – No excuses accepted post-quiz.

• At least one lab section per week
  – Easier to get through the labs with help
  – TA/ will be available then
Academic Honesty

- Cheating is presenting someone else’s work as your own
- All code turned in will be run against a code-checker
- Anyone caught cheating will immediately fail the class and the lab, and be reported to their college
- Copying each other’s code is never acceptable.

- Don’t do it—not worth it.
CMPE-013/L Grading

• Lecture and Lab are one and the same: CMPE-013 and CMPE-013L will get the same grade, same evaluation.

• In class quizzes (once per week): 30%
• Programming assignments (one per week): 70%

• No midterm, no final
CMPE-013/L Lab Work

- All programs we are using can be loaded onto your own laptop for use at any time—they are all free.
- We’re using microchip’s MPLABX IDE and XC32 compiler
- Running on a microcontroller development board by Digilent, the Uno32 or μ32 (essentially the same board)
- You can buy the this hardware directly from Microchip if you want to after the class is over
Reading Assignment for Today

- [Gonick]: parts 1-3, review your notes from CE-12/L
  - There will be a quiz on this on Thursday
- Check out the Webpage and Piazza Forum
Responsibility

• See CANVAS

on Wednesday
What is a Computer?

- **Computer**
  - A *computer* is a machine that manipulates data based on a list of instructions called *program*.
  - A computer consists of hardware and software.

- **Computer Hardware**
  - *Computer hardware* is the physical part of a computer.
  - A typical computer consists of central processing unit (CPU), main memory and external memory, and input and output devices.
  - A CPU consists of control unit (CU), arithmetic and logic unit (ALU), and registers.
Samples of Computer Hardware

A single board computer and a tiny computer.
The von Neumann Computer Architecture

Both programs and data are stored in the same memory
The Harvard Computer Architecture

 Programs and data are stored in different memory.
Hardware Trends

Moore’s Law (1965):

The number of transistors that can be inexpensively placed on an integrated circuit increases exponentially, doubling approximately every two years.

Based on Moore’s law, every two years, the following approximately double:

- CPU speed at which computers execute their programs.
- The amount of main memory.
- The amount of secondary memory.
Computer Programming Languages

Three types of programming languages

1. **Machine code or machine languages**
   A sequence of 0’s and 1’s giving machine specific instructions
   Example: 00011001

2. **Assembly language**
   Using meaningful symbols to represent machine code.
   Example: add hl,de

Assembler: Assembly code → machine code
Disassembler: machine code → assembly code
Computer Programming Languages

3. High-level languages
   Similar to everyday English and use mathematical notations
   (processed by compilers or interpreters)
   Example of a C statement:

   \[ a = a + 8; \]
Programming Languages

Many differences including:

1. Abstraction type
2. Compiled vs. interpreted
3. Memory management
4. Type system
The C Programming Language

- Procedural
- Compiled
- Manual memory management
- Statically typed
- Small
- Low overhead

4-6 bit
6-pins
Comparison of High-Level Language with Machine Code and Assembly Code

The memory addresses, machine code, and assembly code corresponding to a C statement \( a = a + 8 \) for the Rabbit 3000 8-bit microprocessor.

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Machine code</th>
<th>Assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1EA1</td>
<td>000100010000100000000000000</td>
<td>\texttt{ld de,0x0008}</td>
</tr>
<tr>
<td>0x1EA4</td>
<td>1100010000000000000000</td>
<td>\texttt{ld hl,(sp+0)}</td>
</tr>
<tr>
<td>0x1EA6</td>
<td>000111001</td>
<td>\texttt{add hl,de}</td>
</tr>
<tr>
<td>0x1EA7</td>
<td>1101010000000000000000</td>
<td>\texttt{ld (sp+0),hl}</td>
</tr>
</tbody>
</table>
10 Reasons to Learn C


1. C is one of foundations for modern information technology and computer science.
2. C is the most commonly used programming languages in industry.
3. C is a standardized programming language with international standards.
4. Writing computer programs is essential to solving complex science and engineering problems.
5. Computer programming is becoming a necessary skill for many professions.
6. Computer programming can develop student’s critical thinking capabilities.
7. C is one of the most commonly used programming languages in colleges and universities.
8. C is the language of choice for programming embedded and mechatronic systems with hardware interface.
9. C excels as a model programming language.
10. Once you have learned C, you can pick up other languages without much difficulty by yourself because all other modern languages borrowed heavily from C.
Structured Programming in C

- A disciplined approach to writing programs in C.
- Clear, easy to test and debug, and easy to modify.
History of C

- C
  - Invented by Ritchie based on B, a simplified version of BCPL.
  - Used to develop Unix operating system and Unix commands
  - Most system software such as OS are written in C or C++
  - Replacement for assembly language for hardware interface.
  - By late 1970's C had evolved to "K & R C"

- C Standards
    It is called C89. Some call it C90.
  - 2nd C standard was ratified in 1999, called C99.
Just the Facts

• C was developed in 1972 in order to write the UNIX operating system
• C is more "low level" than other high level languages (good for MCU programming)
• C is supported by compilers for a wide variety of MCU architectures
• C can do almost anything assembly language can do
• C is usually easier and faster for writing code than assembly language
Busting the Myths (1.2)

The truth shall set you free...

- C is not as portable between architectures or compilers as everyone claims
  - ANSI language features **are** portable
  - Processor specific libraries are **not** portable
  - Processor specific code (peripherals, I/O, interrupts, special features) are **not** portable

- C is **not** as efficient as assembly
  - A *good* assembly programmer can *usually* do better than the compiler, no matter what the optimization level – C **will** use more memory
Busting the Myths (2.2)

The truth shall set you free...

- There is **NO SUCH THING** as self documenting code – despite what many C proponents will tell you
  - C makes it possible to write very confusing code – just search the net for obfuscated C code contests...
    (www.ioccc.org)
  - Not every line needs to be commented, but most *blocks* of code should be

- Because of many shortcuts available, C is not always friendly to new users – hence the need for comments!
What we will cover in 13/L (1.3)

- "C" programming
  - Using C in an Embedded Environment
  - Comments
  - Variables, Identifiers and Data Types
  - Literal Constants
  - Symbolic Constants
  - printf() Library Function
  - Operators

- Expressions and Statements
- Making Decisions
- Loops
- Functions
- Multi-File Projects & Storage Class Specifiers
What we will cover in 13/L (2.3)

- "C" programming (con’t)
  - Arrays
  - Data Pointers
  - Function Pointers
  - Structures
  - Unions
  - Bit Fields
  - Enumerations
  - Macros with #define

- Advanced Techniques
  - State Machines
  - Recursion
  - Interrupts
  - Program decomposition
  - Abstraction
  - Scope
  - Static / Dynamic Memory allocation
What we will cover in 13/L (3.3)

- Embedded “C” on a microcontroller
  - Specific issues with uControllers
  - Peripheral usage
  - Reading documentation

- Testing and Debugging
  - Commenting
  - Test harnesses
  - Incremental development
  - Issues with embedded debugging
Key things we’ll enforce (1.2)

- **Well documented code**
  - Self-documenting code does not exist!
  - Good variable names and comments are req’d

- **Clean and clear style**
  - More important to adhere to established guidelines than use the “right” style
  - We’ll use (modified) K&R style guide

- **Modularity and decomposition**
  - Code is segmented by functionality
  - Proper use of .h and .c files
  - Good use of functions for clean implementation
Key things we’ll enforce (2.2)

- **State Machines / Event Driven Programming**
  - Best way to design reactive systems
  - Makes debugging much easier
- **Incremental build and test**
  - Every bit of code has a unit test
  - Unit test is designed along with code block
  - Use of pseudo-code and comments
  - End to end code checks
Concept of Ugly/Beautiful

- Beautiful Dog

- Ugly Dog
Concept of Ugly/Beautiful

- Beautiful Car
- Ugly Car
Concept of Ugly/Beautiful

- Beautiful Building
- Ugly Building
Concept of Ugly/Beautiful

- Beautiful Man
- Ugly Man
Concept of Ugly/Beautiful

- Beautiful Woman
- Ugly Woman
Concept of Ugly/Beautiful

Function: RC_Init

Parameters

RCpins: an unsigned short with a 1 in each position to set the pin as an RC servo pin, should be a bitwise OR of the #define'd RC_PORTXXX pins.

Returns

char: SUCCESS or ERROR

Description

Initializes the RC_Servo subsystem, sets each pin as a digital output, and sets the uptime for each pin at 1.5msec, with a period of 20msec.

Notes: Uses TIMER4 with a rollover.

Author: Gabriel Hugh Elkaim, 2011.12.15 16:42

typedef char C;
typedef long I;
typedef struct a(I t,r,d[3],p[2]) A;
#define P printf
#define R return
#define V1(f) A f(w)A w;
#define V2(f) A f(a,w)A a,w;
#define DO(n,x) {i=0,n=(n);for(;i<n;i++)
I ma(n){R(1)malloc(n*a4);ma(d,s,j)n I d.s3;DO tr(r,d) I d.j{i=1;DO(r,d=+1[j]);R zj}}
A g(t,r,d)I t.d{j=z=(A)ma(5+tr(r,d));i=t;R zj}}
V1(iota){I n=0->p;A z=ga(0,1,8,n);DO(n,z=p[i]=
V2(plus){I r=0->r,d=0->d,n=tr(r,d)A z=ga(0,r)
DO(n,z=p[i]=a->p[i]+w->p[i]);R zj}}
V2(from){I r=0->r,d=0->d,n=tr(r,d)A z=ga(w->p[i])
A z=ga(0,w->p[i]+(n+a->p),n);R zj}
V1(box){A z=ga(1,0,0);z=+p(1)w3R zj}
V2(cat){I a=tr(a,r,a->d),n=tr(w,r,w->d),n=
A z=ga(w->t,1,1);n=m(z+p,a,p,an)jzm(z+p+an)}
V2(find){
V2(ish){I r=0->r,a->d,j=tr(r,a->p),w=tr(r,a->p)
A z=ga(w->t,r,a->p);m(z+p,w,p,w+w=n+n+1)
if(n=0)n=m(z+p+a,p+n,n+1);R zj}
V1(sha){A z=ga(0,1,8,r,w,r);m(z+p,w,d,w->d);R zj}
V1(id){R w3!V1(size){A z=ga(0,0,0);z->p=p=c
p(i){P(0%0,,j)}n){P(0%0,n)}}
pr(w)w3!V1(size){R w3!V1(size){A z=ga(0,0,0);z->p=p=c
if(w=t)DO(n,P(0%0))pr(w->p[i])else DO(n,p[i]});

Maxwell James Dunne

CMPE-013/L. “C” Programming
Concept of Ugly/Beautiful

- Code can be ugly or beautiful
- We will strive to write beautiful code

- While you might not recognize it, yet, you will by the end of this quarter.

— Slow down
Software Architecture

- Often orthogonal to Project Management
- Can be the success or failure of a project
Flow Chart and Pseudo-Code

**Pseudo-C**

- 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

Some standard symbols are used to make algorithms easier to follow. Each step is represented by a specially shaped box. The shape indicates what type of step is to be executed.

There are 4 possibilities:

- **BEGIN OR END**
- **PERFORM A PROCEDURE (ADD, SUBTRACT, ETC)**
- **CONDITIONAL BRANCH**
- **INPUT OR OUTPUT**

Flow Chart and store machines
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;
      isLightOn
  – Correct use of white space and indentation
  – Correct placement of braces { }
Development Tools Data Flow

C Source Files

Assembly Source Files (.asm or .s)

Assembly Source Files (.asm or .s)

Archiver (Librarian)

Object File Libraries (Archives) (.lib or .a)

Linker Script (.lkr or .gld)

Object Files

Assembler

C Compiler

Compiler Driver Program

Linker

Executable

Memory Map

MPLAB® IDE Debug Tool

COFF Debug File

Executable
Development Tools Data Flow

C Source File → Preprocessor → Compiler → Assembly Source File

C Compiler

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

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

Function

Constant Declaration (Text Substitution Macro)

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

Boot loader
Stack/Heap

- Stack grows from bottom “up”

- Heap grows from top “down”
Stack/Heap
Questions?
6:30 - 9:30

CMPE-013/L

git

91+106.50e.ucsc.edu

git

Maxwell James Dunne
Unix/Linux

- Operating System developed by ATT/Bell Labs in 1970’s
- Several variants (BSD, OSX, Solaris, Linux) developed over the years.
- Written in C (and some assembly)
- Unix was designed to be portable, multi-tasking and multi-user in a time-sharing configuration
Basic CLI

- **Command Line Interface**: a means of interacting with a computer where the user issues commands to the program in the form of successive lines of text (command lines).
- Most UNIX commands are using a CLI.
- Commands often have parameters or arguments added to them (e.g.: `ls -a`).
- Contrasts to a GUI (Graphical User Interface).
Basic FileSystems

• How files are stored and organized by the operating system
• All modern OS use a hierarchical file system
  – Starts at a root directory
  – Other directories underneath the root directory
  – Hardware simply looks like another directory (e.g.: /dev).
• Files have several properties that can be specified (e.g: file type, permissions, visibility)
Basic Version Control

- Need to keep track of changes in a collection of documents
- Keep the ability to “roll back” to previous working code
- Coordinate with multiple programmers working on the same code.
- Allows for branch to try something that can then be merged back into development.
Git Theory

- Same theory as all version control systems in that it keeps tracks of changes.
- Except Git differs in a major way.
  - Instead of checking in changes to a server, you 'commit' them locally. Only when you need the changes uploaded to the server do you 'push'.

  - Microsoft is on Git 200GB Repo
Git Basics

- Files are only tracked for changes if added to the repository. Files inside the directory are not tracked by default.
- At any point, a ‘commit’ can be made. This will take a snapshot of all tracked files.
- Once ‘commits’ have been made, they are ‘pushed’ back to the server as a separate operation.
Git Practice

- All commands prefaced with `git`
  - clone
  - status
  - add
  - commit
  - push
  - pull
  - log
Git Commands (clone)

- Copies Repository for the first time. Only used when you do not have a copy of the repository

```
$ git clone git@soe.ucsc.edu:/classes/cmpe013/spring16/MaxSampleStudent.git
Cloning into 'MaxSampleStudent'...
remote: Counting objects: 6681, done.
remote: Compressing objects: 100% (6395/6395), done.
remote: Total 6681 (delta 4526), reused 400 (delta 237)
Receiving objects: 100% (6681/6681), 76.70 MiB | 1.98 MiB/s, done.
Resolving deltas: 100% (4526/4526), done.
Checking connectivity... done.
```
Git Commands (status)

- Give status of the repository. Showing state of files within the repo

```bash
$ git status
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)

  README.txt
nothing added to commit but untracked files present (use "git add" to track)
```

```bash
$ git status
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

  new file:  README.txt
```
Git Commands (add)

- Either adds a new file to the repository or marks a file for commit. This command has no output. Call ‘status’ again to verify operation.
Git Commands (commit)

- Creates a snapshot of all files in the repository at the current time. At any point you can come back to this point so commit early & often.

  $ git commit -m "committing an empty readme for a test"

- Adding the \(-a\) automatically adds all tracked files

  $ git commit -am "committing an empty readme for a test"
Git Commands (push/pull)

- Synchronizes the local repository with the remote server. Push should be used frequently to ensure server copy is updated.
- Once pushed to the server, the files are safe from any and all catastrophes that happen to your system.
- **WARNING:** pull will overwrite your local files. Use with discretion.
- We are not sympathetic to your data loss.
Git Commands (log)

- Gives a history of commits for the repo. Call with argument ‘–n 1’ if only last ID is desired.
Git Usage in Labs

- Work out of your repository.
- If working on personal computer, entire tool chain is available on Windows/Mac/Linux.
- Do **NOT** copy files to your repository to commit them.
Git Workflow

1. Clone the repository or pull if the repository if work has been done in a different location.
2. Work on the lab as normal.
   a) Make commits when milestones are reached with useful commit messages. The messages are important, be descriptive.
   b) MPLABX can make commits directly from the IDE. There is no reason not to commit early and often.
3. Push back to the server when session is done or when you want to make sure you have a backup (you don’t need to push every commit, but if you do, then the most you will ever lose is from the latest commit).

Repeat as necessary.
Commit through MPLABX
Lab Submission

1. Finish working on your lab and commit for the last time.
2. Go to gitlab and grab the commit ID you wish to submit.
3. Submit this string to the associated Google Form.
4. Push your files. Without a push the commit ID submitted is useless.
5. The time you submitted the form counts as your turn in time. You must still push the repository.
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
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

PYC
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.
  
  \[ \begin{align*}
  X &= W + W \\
  Y &= X + X \\
  Z &= Y + Y
  \end{align*} \]

  Print \( Z \) to screen.

  \( x = 2w \)
  \( y = 2x \)
  \( z = 2y = 8w \)

• If interpreting, how many arithmetic operations occur?
  
  3

• 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:
  
  \textit{Print the words}\newline
  \textit{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.
Generic C Code

#include <stdio.h>

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

q = foo(x)
φ = main(x)
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");
    while (1); // Loop forever and never return
}

foo: J foo
int main(void)
{
    while (1) {
        // Read inputs
        // Perform calculations
        // Update outputs
    }
}

q'a.aaaaaqqqqq%
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Value</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>localVar1</td>
<td>int</td>
<td>0x88A</td>
<td>0x0003</td>
<td>3</td>
</tr>
<tr>
<td>localVar2</td>
<td>int</td>
<td>0x888</td>
<td>0x0004</td>
<td>4</td>
</tr>
</tbody>
</table>

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**

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

To Open the Watches Window:

Select: **Windows->Debugging->Watches**
**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.
Closing a Project

2) "Right Click" on the project name then select Close
Fundamentals of C

A Simple C Program

Example

Preprocessor Directives

#include <stdio.h>

#define PI 3.14159

Header File

Constant Declaration (Text Substitution Macro)

Function

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

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
    ```
Quiz on Wednesday due to snafu with classrooms

No Class on Monday

MLK
Comments
Using Block Comments

- Block comments:
  - Begin with `/*` and end with `*/`
  - May 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
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... oops! */
*/
```

- Delimiters don’t match up as intended!
- Dangling delimiter causes compile error
/**
 * @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”);
}
Variables and Data Types
A Simple C Program

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

Definition

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;
```

Maxwell James Dunne
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;
  ```

  Data Memory (RAM)
  ```
  41_{16}
  {'A'}
  5.74532370373175 
  \times 10^{-44}
  ```
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*
Name mangling

foo

foo

31 character

4MB
ANSI C Keywords

auto  double  int  struct
break else long switch
case enum register typedef
c char extern return union
c const float short unsigned
c continue for signed void
c default goto sizeof volatile
c do if static while

code:

• Some compiler implementations may define additional keywords

```asm
"NOP"
```
# 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

Qualified types: **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>± ~10^{-44.85}</td>
<td>± ~10^{38.53}</td>
<td>32</td>
</tr>
<tr>
<td>double</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 XC32 Uses the IEEE-754 Floating Point Format
Variables
How to Declare a Variable

Syntax

\texttt{type \ identifier_1, \ identifier_2, ..., \ 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;
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
  ```
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**

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

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

---

**main.c**

```c
#include "main.h"
#include "main.h"

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

int main(void)
{
    ...
}

Equivalent main.c file without #include
```
Header guards

Realistic example

#include "OledDriver.h"

#include "OledDriver.h"
#include "Oled.h"

main.c
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"
...

#endif // OLED_H
```
Questions?