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 UL632
CMPE-013 Website

Welcome to CMPE 13/L — First Class is Tuesday 02-Apr-2013 @ 2PM.

Background

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 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. 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 PIC16F84 (or a PIC16F84A). 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.

Acknowledgements

I would like to acknowledge Cyrus Bearechi, for all his help with the course material, organization, and lecture material. Steve Summitt, who taught an Introductory and Intermediate Programming class at the Experimental College at the University of Washington in Seattle, WA, has generously allowed us to use his notes on K&R and other supplementary materials. Microchip Corp. has generously provided slides and software through their academic partner program, and some of the FILE system slides come from Henry Cheng at UC Davis.

Index of class resources

- General Class Information — class and section times, instructor and TA information.
- Lecture Video — Video files of the lectures, and download information for the right codec.
- Handouts — quizzes, quiz solutions, other helpful handouts.
- Piazza — for announcements, general discussion, and help.
- Colors
- Quizzes
- Class Presentation Slides
- Files for in class Examples

Handouts

- General Course Information
- Course Syllabus
- Extended course description
- Essential C handout
# 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.eskimo.com/users/class/notes/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 [Notes]: Introduction [K&amp;R]: Preface and Introduction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>09-Apr-2013 Comments and Variables</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>3</td>
<td>11-Apr-2013 Literals and Constants</td>
<td>Lab 1: Simple Calculator Due: 17-Apr-13 @ 11:55 PM</td>
<td>Lab 0</td>
</tr>
<tr>
<td>4</td>
<td>16-Apr-2013 Expressions and Loop Control</td>
<td>Lab 2: LED Bounce Due: 24-Apr-13 @ 11:55 PM</td>
<td>Lab 1</td>
</tr>
<tr>
<td>5</td>
<td>18-Apr-2013 Functions, Modules, Scope</td>
<td>Lab 3: Matrix Math Library Due: 03-May-13 @ 11:55 PM</td>
<td>Lab 2</td>
</tr>
<tr>
<td>6</td>
<td>23-Apr-2013 Operators and Expressions</td>
<td>Lab 4: RPN Calculator Due: 08-May-13 @ 11:55 PM</td>
<td>Lab 3</td>
</tr>
<tr>
<td>7</td>
<td>25-Apr-2013 Precedence and Associativity</td>
<td>Lab 5: Linked List Sort Due: 15-May-13 @ 11:55 PM</td>
<td>Lab 4</td>
</tr>
<tr>
<td>8</td>
<td>30-Apr-2013 Scope and Arrays</td>
<td>Lab 6: LCD Driver Due: 22-May-13 @ 11:55 PM</td>
<td>Lab 5</td>
</tr>
<tr>
<td>9</td>
<td>02-May-2013 Strings and Memory/Free</td>
<td>Lab 7: Battleships Due: 25-May-13 @ 11:55 PM</td>
<td>Lab 6</td>
</tr>
<tr>
<td>10</td>
<td>07-May-2013 Pointers</td>
<td>Lab 8: Morse Code Decoder Due: 05-Jun-13 @ 11:55 PM</td>
<td>Lab 7</td>
</tr>
<tr>
<td>11</td>
<td>09-May-2013 Pointers and Functions</td>
<td>Lab 9: CRC32 Due: 05-Jun-13 @ 11:55 PM</td>
<td>Lab 8</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</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 Macros and Preprocessor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMPE-013/L: "C" Programming
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 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:
TTh 2:00 - 3:50 PM
CLASS: Physical Sciences 114
LAB: Jack Baskin Engineering, 109
Ming Hng Lab, 108

Textbooks:

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

Optional Textbooks (good references):


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

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

And it looks like quite a bit less on Abe Books:
http://www.abebooks.com/productsisbn/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 Extended Course Description

Computing Systems and C Programming

13L, Computer Systems and C Programming, 3 Introduction to the C programming language as a means for controlling embedded and general computing systems. Continuing the exploration begun in course 12, students move to higher levels of abstraction in the control of complex computer systems. Prerequisites: Courses 12 and 12L. Concurrent enrollment in course 13L required. R. Rugely.

13L Computer Systems and C Programming Laboratory Laboratory sequence in C programming for embedded and general computing systems. Two-hour laboratories per week. Concurrent enrollment in course 13 required. 2 credits. R. Rugely.

Explanation of Prerequisites:

This course follows the "bottom up" approach to teaching programming. The (CMPE12) sequence provides students with an understanding of how a computer or embedded system is designed and programmed at the machine-level and higher-level language levels. Course 12 focuses on the machine level, while Course 12L takes student to a higher level language and extensive training in structured programming habits and tools.

The knowledge of computer memory systems and low-level programming developed in CMPE12 lays the groundwork for understanding C pointers, the classic stumbling block of new programmers who start with C. The course also builds on the exception and interrupt processing knowledge from CMPE12, a key concept for following course CMPE12L.

In addition to cementing students' understanding of computing systems through the continued hardware/software discussion begin in CMPE12/L, this course provides a C-based alternative to the Java course CMPE12A. C is the fundamental software tool for the computer engineer, much as digital logic is fundamental tool.

Students completing CMPE13/CMPE13L will also be able to move directly into CMPE12B, as with transfer students who worked in C on their CMPE12A-articulated course.

Required Skills to Pass the Course

1. Tools for programming, including IDE, compilers, debuggers, and linkers.
2. Knowledge of software design principles, including:
   a. Top-down and bottom-up design
   b. Documentation, inline and external
   c. Coding style
   d. The use of local variables
3. C programming fundamentals:
   a. Data types, declarations, assignments
   b. Functions
   c. Conditional and loop structures
   d. Static and dynamic memory allocation and manipulation
   e. Library and string functions
4. Interfacing of C with embedded systems:
   a. Using C to control hardware
   b. Developing mixed assembly language and C programs
5. Ability to use C to solve programming problems.
6. Ability to manipulate dynamic data structures such as linked lists, multi-dimensional arrays, and structures.

Maxwell James Dunne

CMPE-013/L: "C" Programming
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/krnnotes/top.html

  http://www.soe.ucsc.edu/classes/cmpe013/Spring11/Gonick/
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, 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

Maxwell James Dunne

CMPE-013/L: “C” Programming
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

Senior design
Lab Kit

Uno32

Basic I/O Shield

PICkit3
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 Handout
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.
NVIDIA Autonomous Car

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.

14 nm

CPU 18 months

Maxwell James Dunne
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 $\rightarrow$ machine code
Disassembler: machine code $\rightarrow$ 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

\[ \text{int } c = 3.14 \]

\[ \text{millions 8-bit 8-pin 0.0001 } \]
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>000100010000100000000000000000</td>
<td>ld de,0x0008</td>
</tr>
<tr>
<td>0X1EA4</td>
<td>11000100000000000000000000</td>
<td>ld hl,(sp+0)</td>
</tr>
<tr>
<td>0X1EA6</td>
<td>000111001</td>
<td>add hl,de</td>
</tr>
<tr>
<td>0X1EA7</td>
<td>110101000000000000000000</td>
<td>ld (sp+0),hl</td>
</tr>
</tbody>
</table>

$a = 0 + 8$
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.
--- asm("Noop")

**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**
  - 1\textsuperscript{st} C standard created in 1989 by ANSI, ratified by ISO in 1990. It is called C89. Some call it C90.
  - 2\textsuperscript{nd} 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
  - Randomness
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

16k/32k of RAM $9
128 bytes of RAM $1
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

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,y) {I i=0, n=(n); for(j<n;j++)xi=1;i+=y} ma(n){R(I*m)malloc(n*4)} m((d,s,n)I-d,s3){DO tr(r,d) I-dj;I=1;DO(r,d=10);R zj}
A ga(t, r, d) I-dj;A z=ma(5+tr(r,d));j-t=t, t=R zj; 
V1(iota){I n=w->p[i]; A =ga(a,1, &n); DO(n, z->p[i]= V2(plus){I r=w->r, d=w->d, n=r, d; A =ga(0, 0)
 DO(n, z->p[i]=a->p[i]+w->p[i]); R zj}
V2(from){I r=w->r, d=w->d, n=r, d; A =ga(w->t, r, d); m((z->p,w->p(n+1+a->p), n); R zj; 
V1(box){A =ga(1, 0, 0)+z->p(I)w3 R zj; 
V2(cat){I n=tr(a, r, a->d); m=(w->r, w->d), n=a A =ga(w->t, 1, &n); m=(z->p, a->p, an); m=(z->p+a, n)
V2(find){
V2(rsh){I r=a->r2 a->d:j, n=tr(r, a->p); m=(w->r, A =ga(w->t, r, a->p); m=(z->p, w->p, n=m w->n; n)
 if(n=m) m=(z->p, r, p, n); R zj; 
V1(sha){A =ga(a, 1, &w->r); m=(z->p, w->d, w->r); R zj; 
V1(id){R wj} V1(size){A =ga(a, 0, 0); t->p=w->p? p(i)P(“%d “); j} nl() {P(“%n”)};
pr(w) wj; I r=w->r, d=w->d, n=tr(r, d); DO(r, p(i) if(w->t) DO(n, P(”< “)); pr(w->p[i]) else DO(n, p(i;
C ut[]]=”+<\n”;
A(*w[])(i){=0, plus, from, find, a, r, s, cat}, (*v[])(i){=0, id, size, iota, bo, sha, s, e};

Maxwell James Dunne
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.
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. The "flow" of the algorithm is represented by arrows, and when all the symbols are combined, it's a flow chart.
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
# Preprocessor Directives

```
#include <stdio.h>
```

# Define PI

```
#define PI 3.14159
```

# Header File

```
const 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)
```

# Example

```
A Simple C Program
```

Maxwell James Dunne
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"
  \[ \text{int } \]

- Heap grows from top "down"
Stack/Heap
Questions?
CMPE-013/L

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’.
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

$ 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)

$ 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. This is how the commit ID will be grabbed for turn in. Call with argument ‘–n 1’ if only last ID is desired.
Git Usage in Labs

- Work out of your repository. In the labs your X drive is the same location as your UNIX shell account.
- 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. Call ‘git log –n 1’ to get your last commit ID. This will be a 40-character hexadecimal string.
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.
Lab Submission Verification

1. Move to or create a different directory.
2. Clone your repository to this new directory.
3. Checkout your specific commit using the command below
   - `git checkout CommitID` (the long hexadecimal string)
4. Your repository is now exactly as it will be when graded; check and make sure all files are there in the proper state.