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
  - diff
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 --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. 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. Go to https://git.soe.ucsc.edu/~git/cgi-bin/git_status.py and log in. This will show the state of your repo on the server and give you your commit ID. This will be a 40-character hexadecimal string. It should match the *git log* response.

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.

6. Turn in a copy of your readme on Canvas. We will grade what is in your repo, not on canvas.
Lab Submission Verification

1. [Move to or create a different directory.]
   **DO NOT DO THIS IN YOUR WORKING REPO.**

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.

5. Delete the newly created repo once you are sure the files are there.
CMPE-013/L

Introduction to “C” Programming

Maxwell James Dunne
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
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.
  
  $$
  \begin{align*}
  X &= W + W \\
  Y &= X + X \\
  Z &= Y + Y
  \end{align*}
  $$
  
  - $X = 2W$
  - $Y = 2X = 4W$
  - $Z = 2Y = 8W$

  Print $Z$ to screen.

- 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?

  $$W \leq 3$$
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:

  Print the words _hello, world_

  blink on LED

• 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.

  2 weeks
#include <stdio.h>

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

    return 0;
}

#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

        // Update outputs
    }
}

Embedded C Code
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

You may find it convenient to alter the columns displayed. “right click” on the column heading

Maxwell James Dunne
Windows used in Examples

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** 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** 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 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);`
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

```
******************************************************************************
* 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
//=============================================
// 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.

```c
/*
  code here  // Comment within a comment
*/
```

Example: Delimited comment within a delimited comment.

```c
/*
  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

```
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)...

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

```plaintext
41_{16}

'A'

5.74532370373175 × 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*
ANSI C Keywords

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

• 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.
- 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 16bits, etc.
Data Type Qualifiers

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>$\pm \sim 10^{-44.85}$</td>
<td>$\pm \sim 10^{38.53}$</td>
<td>32</td>
</tr>
<tr>
<td>double&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>$\pm \sim 10^{-44.85}$</td>
<td>$\pm \sim 10^{38.53}$</td>
<td>32</td>
</tr>
<tr>
<td>long double</td>
<td>$\pm \sim 10^{-323.3}$</td>
<td>$\pm \sim 10^{308.3}$</td>
<td>64</td>
</tr>
</tbody>
</table>

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

Syntax

```
(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

```
int x, y, z;
float warpFactor;
char text_buffer[10];
unsigned index;
```
Variables
How to Declare a Variable

Variables may be declared in a few ways:

**Syntax**

One declaration on a line

```c
type identifier;
```

One declaration on a line with an initial value

```c
type identifier = InitialValue;
```

Multiple declarations of the same type on a line

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

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

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

```c
#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

```
main.h

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

main.c

#include "main.h"

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

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
#include <stdio.h>

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

    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

```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"

...

#endif // OLED_H
```
Questions?