Introduction to Low-Level Programming

- Review Syllabus
- Class Grades Server
  - grades.cs.umd.edu
- Program #1A on the web page later today
  - due two weeks from today
  - to help you get used to the language and environment
- Discussion Sections
  - new material (outside of lecture)
  - hands-on practice and projects
  - quizzes
- Reading
  - Chapters 1 & 2 (by Tuesday)
    - Skip 1.1.4

What is “low-level” Programming

- Possible Definitions
  - Low-level languages are closer to the hardware than are high-level programming languages, which are closer to human languages. (www.webopedia.com)
  - Computer languages are classed a low-level if each instruction specifies only one operation of the computer, or high-level if each instruction may specify a complex combination of operations. (onlinedictionary.datasegment.com)
  - Programming where details and intricacies of the hardware are visible
- The C Language
  - lower than Java (need to manage memory)
  - higher level than assembly

Why Study Low-level Programming?

- Provides Understanding of How Things Work
  - Compilers, Processors, Data Structures
- Allows access to Hardware when required
  - Writing device drivers
  - Creating Operating Systems
- You are in Control
  - Can be faster
    - If you are careful and good
  - Can be slower
    - If you rewrite low level routines where faster ones exist
  - Can be Dangerous – easier to have software that
    - Crashes program (or OS)
    - Fails in odd ways (memory corruption)

The “C” Programming Language

- Widely Used for System Programming
  - Writing Operating Systems and Compilers
- “Middle Aged” Language – created in late 1970’s
  - Reflects lessons learned from Fortran, COBOL
- Goals
  - Designed to allow low-level programming
  - Small language
    - Syntax is relatively simple
    - Libraries provided but optionally used
  - Assumes you know what you are doing
- Major Differences from JAVA
  - Explicit Memory Allocation and Deallocation
  - Procedural rather than object oriented
  - Can compile directly to machine code (rather than byte codes)

What Does a C Program Look Like?

```
#include <stdio.h>
int main(void)
{
   printf("Hello World\n");
   return 0;
}
```

- This Code
  - Includes a library file so you can use the functions defined there
  - Defines the main function
- First thing executed is the main function
- Returning from main function terminates the program

Compilation Stages

- Preprocessor
  - makes sure the parts each have the parts they need
  - begin with a # (pound sign)
  - do not end in a ;
- Translation
  - makes sure individual portions are consistent within themselves
  - creates an object file
- Linking
  - brings those parts together
  - makes sure the parts are consistent with each other
  - creates an executable file

Preprocessor

- function prototypes
  - Define name, parameters, and return types
  - Sort of like how methods are defined within class definitions
  - example

```
int  Funct1(int x, int y);
```

- #include
  - Provides ability to include definitions from other files
  - Generally only include prototypes from external files
    - Actual code is kept in a separate file
    - Files are generally compiled individually to object code
    - And then compiled together to make executable code
- Two ways to use
  - #include <stdio.h>
  - #include “swap.h”

Formatting Your program

- Comments
  - Must be surrounded by /* and */
  - May span multiple lines
    - Common bug is to forget to close a comment
    - Can not be nested
- Vertical White Space
  - Blank lines before a new function definition
  - Blank lines within a function to separate different tasks
- Horizontal White Space
  - Indentation
  - Generally corresponds to nesting level

Basic C Syntax

- Variable Declarations
  - Format:
    - type variableName1, variableName2;
  - Examples
    - int a;
    - float x, y;
    - int m = 8;
  - Must be before any executable code
  - May be global (whole file) or specific to a function
- Looping and Conditionals
  - while, do, for
  - if, switch
- Function Calls
  - FunctName(argument1, argument2, …)
- Block of code
  - { }

printf Function

- Definition
  - included with #include <stdio.h>
  - then just use it because it has already been defined for you
- Syntax:
  - printf(formatString);
  - printf(formatString, variable1, variable2, variabe3, ..);
- Within Format String:
  - Defines constants to print, or format for variables:
    - %d Print an integer in decimal
    - %x Print an integer in hex
    - %f Print a floating point value
    - %c Print a character
    - %s Print a character string
    - \n Print a newline
- Example assuming int a; and a has the value 23
  - code: printf("%d is smaller than %d\n", a, a+5);
  - output: 23 is smaller than 28

Passing Arguments to Functions

- Comma separated list of arguments
- All arguments are passed by value
- May pass literal values
- Compiler will check parameter usage:
  - Match a given call to its prototype
  - Match a function definition to its prototype
  - Pitfall: Critical to have only one prototype for each function
    - Put in a common file (.h)
    - the .h file is then included in and shared by all files that use and/or define that function
- Which of these is legal assuming the current scope contains:

int a, b, c; and int Func1(int x, int y);

  - a = Func1(1, 2);
  - c = Func1(a, b+2);
  - printf("%d",Func1(a + b, a));
  - c = Func1(a * 3, 4) + b;
  - Func1(a, Func1(1,2));

Using Linux

- Logging in
  - Can login at console or remotely (ssh)
- Basic operations (command line)
  - ls — list directory
  - pwd — print current directory
  - mkdir <dir> — create a new file
  - rmdir <dir>
  - cp <f1> <f2> — copy file <f1> into file <f2>
  - rm <f1> — delete file <f1>
  - logout — end session
- Section on Monday will allow you to test and practice using UNIX

Compiling & Running a Program

- Programs must be compiled to execute
- Compiler is invoked as:
  - gcc <options> <source files>
  - Common Options:
    - -g Enable Debugging
    - -Wall Warn about common errors
    - -o <filename> Filename for the executable
    - -c Only compile to object code
    - -fprofile-arcs Count how often statements run
    - -ftest-coverage Test program coverage
  - A simple command line:
    - gcc –g –Wall –o prog1 prog1.c tools.c
- Make
  - covers this for you