CS 449 – Lecture 1

Dr. Jack Lange

This class is based on UNIX

- Specifically Linux
- The user interface will be the command line
  - Command line == UNIX shell environment
- When you login to a UNIX system you get a text prompt
  - At that prompt you type commands
  - Commands == Programs

Basic shell commands

- ls – list contents of a directory
- cd – change the current directory you are in
- pwd – print the current directory you are in

Shell interaction vs. scripting

- Normal shell environments are interactive
  - You type a command, the command executes
- Shell scripts allow batching commands
  - Type a set of commands in an executable text file
  - The text file becomes a **new** command
  - Commands executed line by line
- A shell script always starts with a special first line
  #!*<path to shell>*
  - That line tells the system which shell to use to execute the commands below the first line
  - E.g. "#!/bin/bash"
- Example:
  - test.sh

More commands

- `echo` – print argument to the screen
  - E.g. echo "hello world"
- `cat` – reads and writes whole file contents
  - E.g. cat <*file*>
- `more`/`less` - Reads a file one screenful at a time
  - E.g. `more <file>`
- `mv` – Moves a file to a new location
- `cp` – Copies a file to a new location
- `rm` – deletes a file
- `rmdir` – deletes a directory

Editing files

- Number of different programs that you can use

- pico/nano
  - Easiest for beginners
  - What I used in highschool and beginning of college
- vi/vim
  - Very powerful, a bit obtuse
  - This is what my graduate students use
- emacs
  - Written in Lisp
  - Functionality includes everything and the kitchen sink
  - This is what I use now

- Recommendation:
  - Pico/nano is more than enough for this class
  - Once you start doing serious programming, its time to move to something more powerful
  - Sublime is OK, but don't use eclipse (or any other integrated IDE)

What makes a file executable?

- In UNIX: "Everything is a file"
- For a file to execute it needs to be marked as special
  - Someone needs to grant it permission to execute
  - Every file has a set of permissions that define who can do what with it
- File permissions
  - Set of bits that flag specific permissions for certain users
  - <type (1 flag)><user (3 flags)><group (3 flags)><everyone (3 flags)>
    - E.g. drwxr-xr-x
    - this is a directory that the owner can modify, but everyone else can only read
  - Type
    - ./- = regular file
    - d = directory
  - r = reads allowed
  - w = writes allowed
  - x = execution allowed, or directory can be accessed

Finding commands/Getting Help

- The most important skill in computer science:
  - Knowing how to use Google effectively
- Man pages – `man <command>`
  - Documentation usually included on every UNIX system
  - Includes information on every command and command option available on the system
  - Includes documentation on the available C library APIs
  - These are an incredibly important resource
    - I use them regularly to remind myself about library APIs

The origin of C

- C is an old language
  - Designed to be an abstraction layer for assembly
- C interacts directly with hardware
  - C operations directly correspond to hardware instructions
    - E.g. adding two numbers (a + b), compiles into a single assembly instruction
  - Provides direct access to hardware memory

C vs. Java

- This class uses C
  - Very different from Java
- In Java: the lowest level resource is an object
- In C: the lowest level resource is memory
- C provides multiple ways to "name", group, and access memory
  - "This memory over here is named 'x'. "
  - "These 4 bytes are called 'y'."

What is memory?

- **Memory is a giant array of bytes (8 bits)**
- Each byte is referenced via an index/offset
  - The base index/offset always starts at 0
  - The index of a byte is its "address"
    - Byte with address 9 == memory[9]
  - Hardware interacts with memory in the same way
- How big is the array?
  - Determined by the machines **address size**.
  - Not the amount of memory installed in the system

Memory cont'd

- Memory holds **everything** associated with a program
  - Code, data, variables, libraries, etc…
- C allows direct access to memory
- **So**, in C you can access every part of the program in the same way
  - In memory there is no difference between code and data
  - You can treat code like data, or data like code

Naming Memory

- C allows you to assign names to memory locations
  - The program associates a variable name with a RAM location
- Locations are chosen by the compiler
  - But, we can find out…
- Example:

```
int x = 8;
&x;   <-- Tells us the address of x
          (i.e. where it is in memory)
```

Variables

- Variables name a **region** of memory, not a single byte location
- Variables have types other than raw bytes
  - char, int, short, float, etc…
- A type defines what the value of a variable **means**, and what value it holds
  - Types can take up more than one byte
    - More than a single entry in the memory array
  - The number of bytes required to represent a variable type is chosen by the compiler
    - There are some loose rules, but this is a source of pain
- `int sizeof();`
  - Built in compiler function that returns the size of a variable's type

Type encodings

- Remember: C interacts directly with memory
  - So all data is stored directly as raw bits (not objects)
- `char y = 10;`
  `==>> 1 byte of value 0001010`
- Signed vs unsigned
  - Most primitive types can be signed (+/-) or unsigned (+)
  - Unsigned: Raw binary value
  - Signed: 2's complement
    - "Flip the bits and add 1"
    - 1 in the highest order bit location means value is negative

Standard primitive types (x86_64)

```
char           1 byte  -128 to 127
unsigned char  1 byte  0 to 255
short          2 bytes -32768 to 32767
unsigned short 2 bytes 0 to 65535
int            4 bytes -2147483648 to 2147483647
unsigned int   4 bytes 0 to 4294967295
long           8 bytes -9223372036854775808 to
                        9223372036854775807
unsigned long  8 bytes 0 to 18446744073709551615
float          4 bytes 1.175494e-38 to 3.402823e+38
double         8 bytes 2.225074e-308 to 1.797693e+308
pointer        8 bytes
```

Important: These sizes are not always true!!!
It depends on the compiler and underlying architecture

Hello World

```
/* hello-1.c
     Illustrates:  printf, fflush, stdin, stdout
     Compile with: gcc hello-1.c –o hello-1
     Execute with: ./hello-1
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char * argv[])
{
     char x = 0;

     printf("Hello World! Type a character and hit return\n");

     x = getchar();

     printf("x = %c\n", x);

     return 0;
}
```