Getting around the UNIX shell

- For this course, you should familiarise yourself with the UNIX shell.
- There are many tutorials on the web.
  - There is a good guide here: http://www.ee.surrey.ac.uk/Teaching/Unix/
- Also, look at the manual pages of commands that you use
  - Most commands have a manual page (*e.g.* `man ls`)
- You can find man pages for C library functions by looking under section 3 (library functions) (*e.g.* `man 3 printf`)

Systems Programming

- We will be learning systems programming in C
  - C is the language of choice for systems programming, since it is only slightly abstracted from the system's native programming language: assembly (which in truth is a slight abstraction of machine code)
  - Specifically, there is no hidden memory management within C, so we have to understand how to manually address and allocate memory.
  - C is not object-oriented, as is not the computer. OOP, as in Java, is simply a convenient way to *think* about programming: ultimately it becomes machine code on the host computer.
- In these practical lectures we will study actual code, which I will try to comment heavily, since diving in is the best way to understand.

How to Think about C

- In order to better get a grasp on C, we need to think about how the computer actually works:
  - The computer churns away, executing simple (machine code) instructions stored in subsequent memory addresses that, for example, manipulate the contents of CPU registers (*e.g.* adding, dividing, *etc.*)
  - It would be too restrictive if the computer could execute addresses only sequentially, so the computer will also allow for logical branching instructions (*e.g.* if the value of register X is greater than Y, jump to address A then begin executing subsequent instructions).
  - The computer has no notion of our program's variables and their types (*e.g.* string, float, Person, functions, *etc.*): they are simply distinct memory addresses (or CPU registers) from which it can load and store data as one or more bytes, or execute bytes as machine code.

How to Think about C

- Our process is simply a sequence of machine code instructions (generated by the compiler to correspond with our high-level C program) that will be loaded into memory then executed when launched from the OS.
- What we think of as a function call (*e.g.* `my_function(arg1, arg2)`) will actually be compiled down into a kind of `jump` instruction, that causes the CPU to jump to the compiled function body code, then jump back from the function upon a call to `return`.
- Arguments will often be passed to the function using a convenience known as the *stack*, which is simply a region of the processes' memory that has been setup to hold temporary (*i.e.* local-to-function) variable data.

Building A C Program

- Put simply, we take the C code that we write in our favourite text editor, pass it to the C compiler, and out pops some machine code that can be loaded and run on CPU (*i.e.* an *executable*)
  - Here, we will use the widely-used open source compiler *gcc*.
- Compile the file `hello_world.c`:
  - `gcc hello_world.c -o hello_world`
- Run the program:
  - `./hello_world`

The C API

- Many C library functions (*e.g.* for opening files, network communication, *etc.*) have been standardised to simplify the porting of code from one system to another (*e.g.* Linux to Windows)
- The GNU C library API is a great place to learn about the libraries, with APIs and examples of their use.
  - http://www.gnu.org/software/libc/manual/

Compilation Flags

- It is possible to write ambiguous (*i.e.* sloppy) code in C, so when compiling we will actually put the compiler into a strict warning mode with the following flags:
  - `-Wall -Werror`
- Also we will make use of modern C conventions, using the flags:
  - `-D _GNU_SOURCE`
- This will be important when we assess your code, since you should ensure there are no warnings or errors under these strict modes.
- An example of these all together:
  - `gcc -Wall -Werror -D _GNU_SOURCE hello_world.c -o hello_world`