

# Introduction to the Command line

- in this lecture series we will be using GNU/Linux to develop our understanding of systems programming in C
- one of the beauties of GNU/Linux is that you can do all your development from the command line or alternatively from a graphical interface
  - on GNU/Linux the command line interface is *extremely* powerful
  - once learnt it will last you a lifetime
  - different GUI's and IDE's come and go

## GNU/Linux at Southwales

- you can create yourself a username at Southwales by [clicking here](http://mcgreg.comp.glam.ac.uk/login.html)  
(`http://mcgreg.comp.glam.ac.uk/login.html`)
  - the same link can be used to change your password, or reset your password if you forget it
  
- all second floor laboratories are dual boot and will boot into Fedora Core or Windows
  
- you can also access `mcgreg.comp.glam.ac.uk` (the GNU/Linux fileservers) via `ssh` on the command line in Fedora Core or
  
- alternatively via `putty` under Windows
  - both programs can give you remote command line access to the GNU/Linux server

## GNU/Linux at Southwales

- hence you *should* be able to use `putty` under Windows in J1 (24 hour access lab) and also from halls of residence

## Introduction to the Command line

- make sure that you have a working username and password under GNU/Linux and login to the server or the Fedora Core client
- if you are using Windows, either reboot the machine and start Fedora Core or download `putty` and remotely log into `mcgreg.comp.glam.ac.uk`
- you can also use the `telnet` program under Windows to access `mcgreg.comp.glam.ac.uk`
  - if you are using `telnet` or `putty` maximise the window
- if you are using Fedora Core then open up a `terminal` window and maximise it

## Introduction to the command line

- the first command to be aware of is `man`. To examine what this does type:
- ```
$ man man
```
- when you have read enough, type 'q'

## Introduction to the command line

- to find out whether a command exists for a particular function, type

- ```
$ man -k directory
```

- this command tells you all the commands which are associated with directories

- you can filter the search by:

- ```
$ man -k directory | grep list
```

## Critical general purpose command line programs

- `cd` change directory
- `pwd` print working directory
- `cp` copy a file
- `mv` rename a file (move)
- `cat` display contents of a file

## Critical general purpose command line programs

- `less` display contents of a file, a page at a time
- `grep` print lines matching a pattern
- all programs can be combined using the pipe operator
- for example

```
$ man -k directory | less
```

## Critical development command line programs

- `gcc` the C GNU compiler
- `gdb` the GNU debugger
- `emacs` the GNU editor

## Minimal introduction to emacs

- to start editing the file `tiny.c` with `emacs` editor type:

- ```
$ emacs tiny.c
```

- critical key commands
  - this editor can be controlled from the keyboard (without the mouse)
- use cursor keys, page up, page down, to move around the text

## Minimal introduction to emacs

- in this section of the notes the notation  $\text{^C}$  means press the control key and then press the C key, finally release both keys
- type  $\text{^X^S}$  to save your file
- type  $\text{^X^C}$  to quit emacs

## Creating a simple C program under GNU/Linux

- using emacs create the following file (called `tiny.c`)
- the contents of this `tiny.c` should be as follows

```
#include <stdio.h>

main()
{
    int i;

    for (i=1; i<=12; i++) {
        printf("%d x 8 = %d\n", i, i*8);
    }
}
```

## Minimal introduction to GCC

- gcc is the GNU C compiler
- now exit emacs and compiler `tiny.c`, you can compile and link `tiny.c` like this:

```
$ gcc -g tiny.c
```

- this generates a file `a.out` which can be run, from the command line by typing:

```
$ ./a.out
```

## Minimal introduction to GDB

- gdb is the GNU debugger, which can be useful to both
  - debug your program
  - understand how a program works

## Minimal introduction to GDB

- for example, suppose we wanted to understand which lines of code are executed in your `tiny.c` program, you could

```
$ gdb a.out  
GNU gdb 6.4.90-debian  
Copyright (C) 2006 Free Software Foundation, Inc.  
GDB is free software, etc  
(gdb) break main  
Breakpoint 1 at 0x8048365: file tiny.c, line 7.  
(gdb) run
```

## Minimal introduction to GDB

```
Breakpoint 1, main () at tiny.c:7
7      for (i=1; i<=12; i++) {
(gdb) step
8      printf("%d x 8 = %d\n", i, i*8);
(gdb) step
1 x 8 = 8
7      for (i=1; i<=12; i++) {
(gdb) step
8      printf("%d x 8 = %d\n", i, i*8);
(gdb) print i
$1 = 2
(gdb) step
2 x 8 = 16
7      for (i=1; i<=12; i++) {
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

## Extending tiny.c to use a function

- use emacs to modify the `tiny.c` program (to include a mistake)

```
#include <stdio.h>

int mult (int i)
{
    return i*9;
}

main()
{
    int i;

    for (i=1; i<=12; i++) {
        printf("%d x 8 = %d\n", i, mult(i));
    }
}
```

## Extending tiny.c to use a function

- now recompile the program by:

- ```
$ gcc -g tiny.c
```

## Extending tiny.c to use a function

- and run the program, as before

```
$ ./a.out
1 x 8 = 9
2 x 8 = 18
3 x 8 = 27
4 x 8 = 36
5 x 8 = 45
6 x 8 = 54
7 x 8 = 63
8 x 8 = 72
9 x 8 = 81
10 x 8 = 90
11 x 8 = 99
12 x 8 = 108
```

## Extending tiny.c to use a function

- we can single step the program to find out where the mistake occurred

```
$ gdb a.out  
GNU gdb 6.4.90-debian  
Copyright (C) 2006 Free Software Foundation, Inc.  
GDB is free software, etc  
(gdb) break main  
Breakpoint 1 at 0x8048365: file tiny.c, line 7.  
(gdb) run
```

## Extending tiny.c to use a function

```
(gdb) run
Starting program: a.out

Breakpoint 1, main () at tiny2.c:12
12     for (i=1; i<=12; i++) {
(gdb) step
13         printf("%d x 8 = %d\n", i, mult(i));
(gdb) step
mult (i=1) at tiny2.c:5
5         return i*9;
(gdb) fin
Run till exit from #0  mult (i=1) at tiny2.c:5
0x08048388 in main () at tiny2.c:13
13         printf("%d x 8 = %d\n", i, mult(i));
Value returned is $1 = 9
```

at this point we see our mistake, the function has returned 9

## Extending tiny.c to use a function

- we can see this again if we continue around the `for` loop

```
(gdb) step
1 x 8 = 9
12     for (i=1; i<=12; i++) {
(gdb) step
13         printf("%d x 8 = %d\n", i, mult(i));
(gdb) step
mult (i=2) at tiny2.c:5
5         return i*9;
```

## Extending tiny.c to use a function

```
(gdb) print i
$1 = 2
(gdb) up
#1 0x08048388 in main () at tiny2.c:13
13      printf("%d x 8 = %d\n", i, mult(i));
(gdb) print i
$2 = 2
(gdb) down
#0 mult (i=2) at tiny2.c:5
5      return i*9;
```

## Using gdb from within emacs

- you can run `gdb` from within `emacs` and have `emacs` perform source file correspondence
- *if* you wish to do this then it would be sensible to create a file `.gdbinit` and populate it with

```
#  
# this file is the gdb start up script and  
# you can place any gdb commands in here  
#  
break main  
run
```

- this file is read by `gdb` when `gdb` is executed

## Using gdb from within emacs

- now at the command line, you can type:

```
$ emacs
```

- now type: **<alt>xgdb<enter>** within emacs
- now you can enter the gdb commands **step next print fin** and **quit** and emacs will track the source file and line number in an alternate window

# Tutorial

- work through these lecture notes, trying each example in turn