Know your tools

- "a bad workman blames his tools",
                                        Cambridge Idioms Dictionary

- we will examine:
  - `emacs, etags, grep, diff, patch, gcc, gm2, cvs, gdb, svn`

- although in this lecture we will only cover `emacs` and `gdb`
  - and revise our knowledge of C pointers

For the GNU/Linux game developer GDB is the BFG

- get to know this tool!

emacs

- GNU Emacs is an extensible, customisable text editor-and more

- at its core is an interpreter for Emacs Lisp, a dialect of the Lisp programming language with extensions to support text editing

- features of GNU Emacs include:
  - content-sensitive editing modes
  - highly customisable, using Emacs Lisp code or a graphical interface
  - can run a shell, ssh session, read news, read mail, run `gdb`
  - all the above are editing sessions
  - learn how to navigate it once, use it in a multitude of ways

Minimal number of key commands for emacs

- deliberately kept short!

- ^c means control key is pressed and kept down while the c key is also pressed. After which both are released.

- M-x means press the meta key (the <alt> key) and then press the x key and then release both.

- M-x can also be achieved by pressing the <esc> key, releasing it and then pressing x and releasing it.

- choose which ever seems most natural

emacs keys

-
```
Keys      meaning
==================
^x^c      exit emacs
^x2       split screens horizontally into two
^xo       move cursor into other window
^x^f      load in a new file
^x^s      save current buffer
^xs       save all buffers
^s        search forward
^r        search reverse
^k        cut rest of line into kill buffer
^y        yank the last kill buffer (paste it into the cu
^<space>  mark the current position
^w        kill all text between current position and last
M-x       move to the execute-extended-command line
^g        stop emacs from doing something
^xb       change buffer (press tab to see all available b
```

emacs function keys

-
```
f5    debug doom3
f8    goto next compile error
f11   full screen (toggle)
f12   recompile doom3
```

- can be customised by changing $HOME/.emacs

Further emacs information

- emacs homepage ⟨http://www.gnu.org/software/emacs⟩

- the best way to learn how to use emacs is by reading the built-in documentation

- to do this, start emacs and then use the commands:
  - Interactive beginners' tutorial - to start this from within emacs, type ^ht
    - this is an extremely well written tutorial - well worth the reading effort
  - List of Frequently Asked Questions, type ^h^f

C Pointers and arrays revisited

- a pointer is a variable that contains an address of a (normally different) variable

- arrays and pointers are closely related in C

- we can declare an array of integers by:

```
int a[10];
```

- and we can declare a pointer to an integer, by:

```
int *b;
```

Initialising a pointer

- we can make b point to the start of the array, by:

```
int *b = (int *)&a;
```

- to set the first element of the array to 999 we can either use the pointer or the array variable

Initialising a pointer

```
#include <stdio.h>

int main ()
{
    int a[10];
    int *b = (int *)&a;

    a[0] = 111;
    printf("the first element of the array has been set t
            a[0]);
    *b = 999;
    printf("the value of the first element is now %d\n",
    return 0;
}
```

Initialising a pointer

- we can assign 777 to the second element of the array by the following code:

```
#include <stdio.h>

int main ()
{
    int a[10];
    int *b = (int *)&a;

    b++;
    *b = 777;
    printf("the second element of the array has been set
            a[1]);
    return 0;
}
```

- notice that we moved to the second element on the array by: b++

Initialising a pointer

- we could have also written the code like this:

-
```
#include <stdio.h>

int main ()
{
    int a[10];
    int *b = (int *)&a[1];

    *b = 777;
    printf("the second element of the array has been set
            a[1]);
    return 0;
}
```

Initialising a pointer

- or like this:

-
```
#include <stdio.h>

int main ()
{
    int a[10];
    int *b = ((int *)&a)+1;

    *b = 777;
    printf("the second element of the array has been set
            a[1]);
    return 0;
}
```

Initialising a pointer

- the addition of 1 to a pointer means increment the address value in the pointer variable by:
  sizeof(*b) bytes

- avoid arithmetic on pointers if at all possible

Interchanging pointers and arrays

- we can also set the third element of the array to 444 by:

-
```
#include <stdio.h>

int main ()
{
    int a[10];
    int *b = (int *)&a;

    b[3] = 444;
    printf("the second element of the array has been set
            b[3]);
    return 0;
}
```

- notice how we are treating b as an array, although we declared it as a pointer

Interchanging pointers and arrays

- clearer than adding, 3, to a pointer, and the same code is generated by the compiler

- use the debugger to print out values, or set values

- compile the previous example using

```
$ gcc -g pointer2.c
```

- then we can run the debugger as follows

Interchanging pointers and arrays

-
```
$ gdb ./a.out
GNU gdb 6.4.90-debian
Copyright etc...
(gdb) break main
Breakpoint 1 at 0x400480: file pointer2.c, line 6.
(gdb) run
Starting program: /home/gaius/text/Southwales/gaius/c/a.o
Breakpoint 1, main () at pointer2.c:6
6        int *b = (int *)&a;
(gdb) step
8        b[3] = 444;
(gdb) ptype b
type = int *
(gdb) step
9        printf("the second element of the array has been
step
the second element of the array has been set to 444
11   }
```

Interchanging pointers and arrays

-
```
(gdb) set *b=999
(gdb) print b[0]
$2 = 999
(gdb) print b[3]
$3 = 444
(gdb) set *(b+3)=777
(gdb) print b[3]
$4 = 777
(gdb) quit
```

structs and pointers

- recall a `struct` can be define a linked list like this:

-
```
struct list {
   struct list *right;
   struct list *left;
   char        ch;
}
```

- here we declare a `list` structure which has 3 fields
  - `right`, `left`, and `ch`
  - `right` and `left` are also pointers to a `list` structure and `ch` is a character

Initialising a pointer to a struct

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

struct list {
   struct list *right;
   struct list *left;
   char        ch;
};

int main ()
{
    struct list *h = (struct list *)malloc (sizeof (struc

    h->right = NULL;
    h->left = NULL;
    h->ch = '\0';

    return 0;
}
```

prototype for malloc

-
```
extern void *malloc (unsigned int nBytes);
```

- which means the function `malloc` takes one parameter, the number of bytes requested
  - and returns an address to the start of a memory block which can be used to contain `nBytes` of information

- remember a generic pointer can be defined by the construct `void *`

Implementing a program to create a linked list of characters

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

const char *myString = "hello world";

struct list {
  struct list *left;
  struct list *right;
  char         ch;
};

int main ()
{
  /* unfinished */

  return 0;
}
```

Implementing a program to create a linked list of characters

- fragment of implementation

-
```
struct list *head = NULL;

/* need to complete function add */

int main ()
{
  int n = strlen (myString);
  int i;

  for (i=0; i<n; i++) {
    add(a[i]);
  }
  return 0;
}
```

Implementing function add (which contains one deliberate mistake)

```
void add (char ch)
{
  struct list *e = (struct list *)malloc (sizeof (struct
  if (e == NULL) {
    perror("trying to add an element to the list");
    exit(1);
  }
  if (head == NULL) {
    head = e;
    e->right = e;
    e->left = e;
    e->ch = ch;
  }
  else {
    /* add e to the end of the list */
    e->right = head;
    e->left  = head->left;
    head->left->right = e;
    head->left = e;
  }
}
```

Function main

```
int main ()
{
  int n = strlen (myString);
  struct list *f;
  int i;

  for (i=0; i<n; i++) {
    add(myString[i]);
  }
  if (head != NULL) {
    f = head;
    do {
      printf("char %c\n", f->ch);
      f = f->right;
    } while (f != head);
  }
  return 0;
}
```

Tutorial

- firstly use the debugger and find the bug in `add`

- secondly can you rewrite functions `add` and `main` so that you always keep a dummy head element and therefore you can reduce the `head==NULL` tests
  - the lines of code will reduce and there will be no need for an `else` statement