emacs customisation ($HOME/.emacs)

```
;; F5 loads in the .gdbinit file ready for debugging ioquake
(defun my-find-file-debug ()
  "load a file"
  (interactive)
  (find-file (concat (getenv "HOME") "/Sandpit/ioquake-latest/ioquake3/.gdbinit")))
(global-set-key [f5] 'my-find-file-debug)

;; F12 compiles ioquake
(setq compile-command "make")
(defun my-compile ()
  "run the compile command after moving to the correct directory"
  (interactive)
  (find-file (concat (getenv "HOME") "/Sandpit/ioquake-latest/ioquake3/.gdbinit"))
  (compile compile-command))

;; compile ioquake by pressing
(global-set-key [f12] 'my-compile)

;; F8 moves to the next error in the source file.
(global-set-key [f8] 'next-error)
```

emacs customisation (`$HOME/.emacs`)

■
```
;; initialise GNU Modula-2 development functions
(load "~/emacs/gm2-emacs")
```

~/emacs/gm2-emacs

■
```
(setq graft-gcc (getenv "GCCVERSION"))

(setq graft-home (concat (concat (concat (concat "/home/gaius/GM2/graft-" graft-gcc) "/

(setq build-home (concat (concat (concat (concat "/home/gaius/GM2/graft-" graft-gcc) "/

(setq compile-command "make")

(defun my-compile ()
  "run the compile command"
  (interactive)
  (compile compile-command))

(defun my-find-file-debug ()
  "load a file"
  (interactive)
  (find-file (concat build-home "gcc/.gdbinit")))
```

~/emacs/gm2-emacs

```
; (add-hook 'after-save-hook 'my-rebuild)

(defun my-rebuild ()
  "rebuild gcc"
  (interactive)
  (let (source-buffer)
    (setq source-buffer (current-buffer))
    (let (gdbinit)
      (delete-other-windows)
      (split-window)
      (other-window 1)
      (my-find-file-debug)
      (compile compile-command)
      (other-window 1)
      (goto-char (point-max))
      (other-window 1)
      (switch-to-buffer source-buffer))))
```

C preprocessor

- the C language is two languages
    - the language which defines data types, statements, functions and operators
    - there is also the language of the preprocessor (`#include`, `#define`) etc

- we use `#include` to include text within the current source file
    - traditionally these text files are `.h` files or library files, but they can be any text file, including `.c` files
    - however including `.c` files is generally considered bad practice
        - however there are exceptions to this rule, for example including machine generated `.c` files

Example `#include`

- the following is held in file `example.h`

-
```
int example_length (char *s);
```

Example #include

- the following is `main1.c`

```
#include <stdio.h>
#include "example.h"

main ()
{
   printf("my length program counts %d characters\n",
          example_length("hello world"));
}
```

example.c

```c
int example_length (char *s)
{
    int i=0;

    while (s[i] != '\0')
        i++;
    return i;
}
```

#include

- notice that `main1.c` contained two #includes, the first was to include the prototype for `printf`

- also notice that it used <> to delimit the filename
  - special meaning, it searches using an implementation defined rule to find the file

- you can modify the search path of the `gcc` compiler by supplying the `-I` argument

#include

■ you can compile and link these examples into an executible via:

■
```
$ gcc -g example.c main1.c
```

■ and run the program via:

■
```
$ ./a.out
```

Macro substitution

■ the C preprocessor allows text to be substituted, for example

■
```
#define forever while (1)
#include <stdio.h>

main ()
{
    forever {
        printf("hello world\n");
    }
}
```

Macro substituted with arguments

-
```
#define assert(X)    { if (! (X)) \
                        { fprintf(stderr, "assert failed\n"); }}

main ()
{
   int r = write(1, "hello world\n", 12);

   assert(r==12);
}
```

- if you are unsure what is happening try asking the preprocessor what it is doing (use `gcc -E`)

Macro substituted with arguments

```
$ gcc -E main2.c
# 1 "main2.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "main2.c"

main ()
{
   int r = write(1, "hello world\n", 12);

   { if (! (r==12)) { fprintf(stderr, "assert failed\n"); }};
}
```

- use the C preprocessor carefully...
  - do not use the C preprocessor to gain execution speed (by avoiding a function call)
  - the C compiler will probably make better decisions than you..

Special tokens in the C preprocessor

- `__LINE__`, `__DATE__`, and `__FILE__` are all special tokens and are expanded into their obvious meanings, consider

- the `gcc` C compiler also substitutes `__FUNCTION__` (but it does this in the C language component - not during preprocessing)

Special tokens in the C preprocessor

```c
#include <stdio.h>
#include <stdlib.h>
#define assert(X,Y) do_assert(X, Y, __LINE__, __FILE__, __FUNCTION__)

void do_assert (int b, char *m, int line, char *file,
                const char *func)
{
   if (! b) {
      fprintf(stderr, "%s:%d:  %s in function %s (on %s)\n",
                       file, line, m, func, __DATE__);
      exit(1);
   }
}


main ()
{
  int r = write(1, "hello world\n", 11);  /* should be 12 */
  assert(r==12, "write failed");
}
```

gcc -E main3.c | tail -20

```c
void do_assert (int b, char *m, int line, char *file,
                const char *func)
{
   if (! b) {
      fprintf(stderr, "%s:%d:  %s in function %s (on %s)\n",
                      file, line, m, func, "Feb  4 2008");
      exit(1);
   }
}

main ()
{
  int r = write(1, "hello world\n", 11);
  do_assert(r==12, "write failed", 18, "main3.c", __FUNCTION__);
}
```

Watch out for side effects

-
```
#define square(X)   (X)*(X)

main ()
{
    int i=11;

    printf("square of %d is %d\n", i, square(i++));
    printf("square of %d is %d\n", i, square(i++));
```

- why is this wrong?

- never use C preprocessor for speed, unless you know the hardware better than the compiler author..

Correct solution

- ```
  #include <stdio.h>

  int square (int x)
  {
    return x*x;
  }

  main ()
  {
     int i=11;

     printf("square of 11 is %d\n", square(i++));
     printf("square of 12 is %d\n", square(i++));
  }
  ```

- and compile with `gcc -O3 -g main4.c` it will produce great code

Compiler assembler output

■ check this is the case!

■ compile with `gcc -S -g -O3 main4.c`

main4.s

```
.LC0:
    .string  "square of 11 is %d0
.LC1:
    .string  "square of 12 is %d0
    .text
main:
    .loc 1 9 0
    subq  $8, %rsp
    .loc 1 12 0
    movl  $121, %esi
    movl  $.LC0, %edi
    xorl  %eax, %eax
    call  printf
    .loc 1 13 0
    movl  $144, %esi
    movl  $.LC1, %edi
    xorl  %eax, %eax
    .loc 1 14 0
    addq  $8, %rsp
    .loc 1 13 0
    jmp   printf
```

C preprocessor conditionals

- C preprocessor allows the following conditionals
  - `#if defined(MACRONAME)`
    - also abbreviated to `#ifdef MACRONAME`
  - first method is preferable as it can be used with conditional logic

C preprocessor conditionals

```
#define FOO
#define BAR "ok"

#if defined(FOO)
#   include "myfunc1.h"
#elif defined(BAR) && (BAR == "ok")
#   include "myfunc2.h"
#else
#   include "myfunc3.h"
#endif
```

Revisiting the `example_length`

■ goal is to create a prototype header file which allows external access to
`example_length`

■ but also allow local prototype checking

example2.h

```
#if !defined(EXAMPLE_H)
#   define EXAMPLE_H
#   if defined(EXAMPLE_C)
#      define EXTERN
#   else
#      define EXTERN extern
#   endif

EXTERN int example_length (char *s);

#endif
```

example2.c

```
#define EXAMPLE_C
#include "example2.h"

int example_length (char *s)
{
   int i=0;

   while (s[i] != '\0')
      i++;
   return i;
}
```

main5.c

```
#include "example2.h"
#include <stdio.h>

main ()
{
   printf("my length program counts %d characters\n",
          example_length("hello world"));
}
```

Tutorial

■ extend the `example2` module above to include a function `example_reverse` whose prototype is:

■
```
char *example_reverse (char *s)
```

■ and this function must be implemented to create a new string but copy the contents of, `s`, in reverse order
  ■ hint you will need to use malloc

■ extend `main5.c` to test your program

■ finally read about the precedence operator `##`

- finally what does the following code do?
  - check the C preprocessor transformation

Tutorial

```
#include <stdio.h>
#define FOO(X)    #X " and this is something extra"
#define DEBUG(X) ("%s:%d:the value of " #X " is %d\n", \
                  __FILE__, __LINE__, X)
#define BAR(X,Y) X ## Y

main()
{
  int i=9;
  int fred=3;

  printf(FOO(hi) " to do at the end of the" " lab\n");
  printf DEBUG(i);
  printf("value of fred is %d\n", BAR(fr,ed));
}
```

Tutorial

- finally write a macro called `check_malloc` which has the same user prototype as `malloc` but checks that the result is non `NULL`
  - and calls `fprintf(stderr,` etc, if the result is `NULL`

- make it as useful as possible