



# COP290– Compilation etc.

---

# *Text Editors*



- Crucial tools for using Unix
- Two main editors (easy flamewar topic):
  - emacs
  - vi
- Great features in both:
  - Syntax highlighting
  - Brace matching
  - Sophisticated text manipulation/movement
  - Scriptable
  - ...

# *Text Editors: vi*



- If it's unix, vi is installed
  - vim, <http://www.vim.org>
- Simple and small
- Modal
  - Command: move, perform commands
  - Insert
  - Others (replace, selection, more)
- Built in help, “: help”

# *Text Editors: emacs*



- Kitchen-sink powerful
- Configurable, extensible, complicated
- emacs and xemacs
- emacsclient
- Tutorial: C-h t

# *gcc the Gnu Compiler Collection*



- Frontends for:
  - C: gcc
  - C++: g++
  - More (ada, java, objective-c, fortran, ...)
- Backends for:
  - x86, ia-64, ppc, m68k, alpha, hppa, mips, sparc, mmix, pdp-11, vax, ...

# *Compiling a simple program*



- `g++ hello.C -o hello`
  - `g++ -c hello.C`
  - `g++ hello.o -o hello`
- `gcc myprog.C -o myprog -lm -IX11`
- `g++ file1.C file2.C -o myprog`

# *Preprocessor*



1. Preprocessor Features
2. Including Files
3. Symbolic Constants
4. Macros
5. Conditional Compilation

# *Preprocessor Features*



- The preprocessor is part of the compiler
  - it modifies the C program *before* the program is compiled
- The modifications involve text *substitution*
  - text in the program is replaced by other text

*continued*





- Substitution is actually based on replacing C *tokens* by other text.

- Examples of tokens in a line of C code

- the code:

```
x = inc(foo);
```

- the tokens:

```
x      =      inc      (      foo      )      ;
```

# *Main Kinds of Substitutions*



- File Inclusion
- Symbolic Constants
- Macros
- Conditional Compilation

# *Seeing the Substitution*



- It is possible to execute only the preprocessor part of `gcc`:

```
$ gcc -E examp.c
```

- This call to `gcc` will output the modified version of `examp.c`
  - can be used for checking that the right substitutions are occurring

# Including Files (`#include`)



- Some examples:

```
#include <stdio.h>      /* from /usr/include */
```

```
#include "mydefs.h"     /* from current dir */
```

```
#include <sys/file.h> /* from /usr/include/sys */
```

Each `#include` line in the program will be *replaced* by the contents of the named header file.

# *What's in a header file?*



- Headers are C code (so *have a look*).
- Typical code:
  - symbolic constant definitions (`#define` lines)
  - `typedef` declarations
  - `extern` declarations (see later)
  - function prototypes
- You will write your own header files when coding large, multiple file programs
  - see later

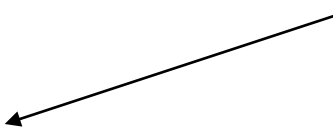
# Symbolic Constants (#define)



- Some examples:

```
#define TRUE 1
#define FALSE 0
#define SIZE 10
#define MAX (SIZE*2 + 1)
```

a constant  
defined using  
another constant



- Common mistake – adding a ‘;’

```
# define SIZE 10;
    while (x < SIZE) ...
```

becomes

```
while (x < 10;) ...
```

**The preprocessor  
replaces SIZE  
by 10;**

# Macros



- Some examples:

```
#define sum(x, y)      ((x) + (y))
#define cube(x)        ((x) * (x) * (x))
#define print(str)     { printf("%s\n", str); }
```

the macro  
body

- Use:

## Translation:

```
c = sum(a, b); c = ((a) + (b));
y = cube(a);      y = ((a) * (a) * (a));
print(argv[1]);   { printf("%s\n", argv[1]); }
```

# *Rules you Must Follow*



- Don't place ';'s at the end of a macro body.
- Bracket every parameter in the macro body.
- Bracket the entire macro body.
- Avoid side-effects in macro calls.
- Add braces around multiple statements in a macro body; have a ';' after every statement.

When in doubt: add some brackets



# *Macros are not Functions*



- A macro 'call' is textually *replaced* by the macro body inside the preprocessor.
- There is *no overhead* of a function call at run-time.
- Macro replacement causes the *size* of the program to increase.
- *Lots* of tricky errors are possible.

# *Multi-line Macros*



- Big macros use ‘\’ to run over multiple lines:

```
#define swap_int(s1, s2) { int temp; \
                           temp = s1; \
                           s1 = s2;  \
                           s2 = temp; }
```

# *Some Common Errors*



1. Including space after the macro name.
2. Missing parameter brackets.
3. Missing macro body brackets.
4. Including side effects in macros.

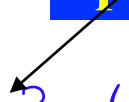
# *Space after macro name*



- Example:

```
#define abs (x) ((x) > 0) ? (x) : (- (x))  
/* WRONG */
```

space after abs



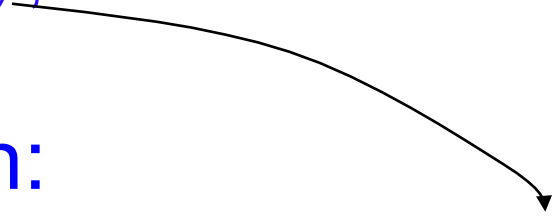
- Use:

```
x = abs(2);
```

- Expansion:

replaced by

```
x = (x) ((x) > 0) ? (x) : (- (x)) (2);
```



# *Undefining*



- Cancel a definition:

```
#undef name
```

- applies from this line in the text downwards

- Can redefine later:

```
#define name 10
```

# *Conditional Compilation*



- Specify which blocks of code to compile
  - some parts of the program can be *ignored* by the compiler
- Used for debugging and portability.

# *If Tests*



- General format:

```
#if expression
... /* code to be included if
      expression is true */
#endif
```

- Expressions are C-like, but can only refer to symbolic constants and/or macros.

# Examples



```
#define MAX 9
:
#if MAX < 10
    ....          /* code to include */
#endif
```

---

```
#define X      5
#define Y      7
#define DEBUG   2
:
#if X < Y && DEBUG == 1
    ...          /* code to include */
#endif
```



# *Leaving Out Code*



- Example:

```
#if 0
    ...           /* lines of code to
                   be left out */
#endif
```

- Useful for commenting out lines that contain lots of comments
  - cannot nest comments in C

# *Multi-way Branches*



- **Format:**

```
#if expr1
...
#elif expr2
...
#elif expr3
...
:
#else
...
#endif
```



# Definition Test

- `#ifdef name`      Test if `name` defined  
  `#ifndef name`      Test if `name` *not* defined

- Used for debugging:

```
#define DEBUG
:
#ifdef DEBUG
    printf("Some debug message\n");
#endif
```

*continued*



- Or:

```
#define DEBUG 1
:
#if DEBUG == 1
    printf("Some debug message\n");
#endif
```

- In this version, we must supply a value for `DEBUG`.

*continued*



- Prevent multiple includes:

```
#ifndef INCLUDED_TYPES
#define INCLUDED_TYPES
#include "types.h"
#endif
```

**only include types.h  
if INCLUDED\_TYPES  
isn't already defined**

- Implement version differences:

```
#ifdef UNIX
char *version = "UNIX version";
#else
char *version = "DOS version";
#endif
```

# *Creating Symbolic Constants*



- Can use `#define`

or

- Create (and initialise) on the command line:

```
$ gcc -DDEBUG -o examp examp.c  
/* define DEBUG */
```

```
$ gcc -DMAX=2 -o foo foo.c  
/* define MAX with value 2 */
```