C Programming under Linux

P2T Course, Martinmas 2003–4
C Lecture 6

Dr Ralf Kaiser

Room 514, Department of Physics and Astronomy

University of Glasgow

`r.kaiser@physics.gla.ac.uk`

Summary

- The C Preprocessor
- Bit Operations

```
http://www.physics.gla.ac.uk/~kaiser/
```

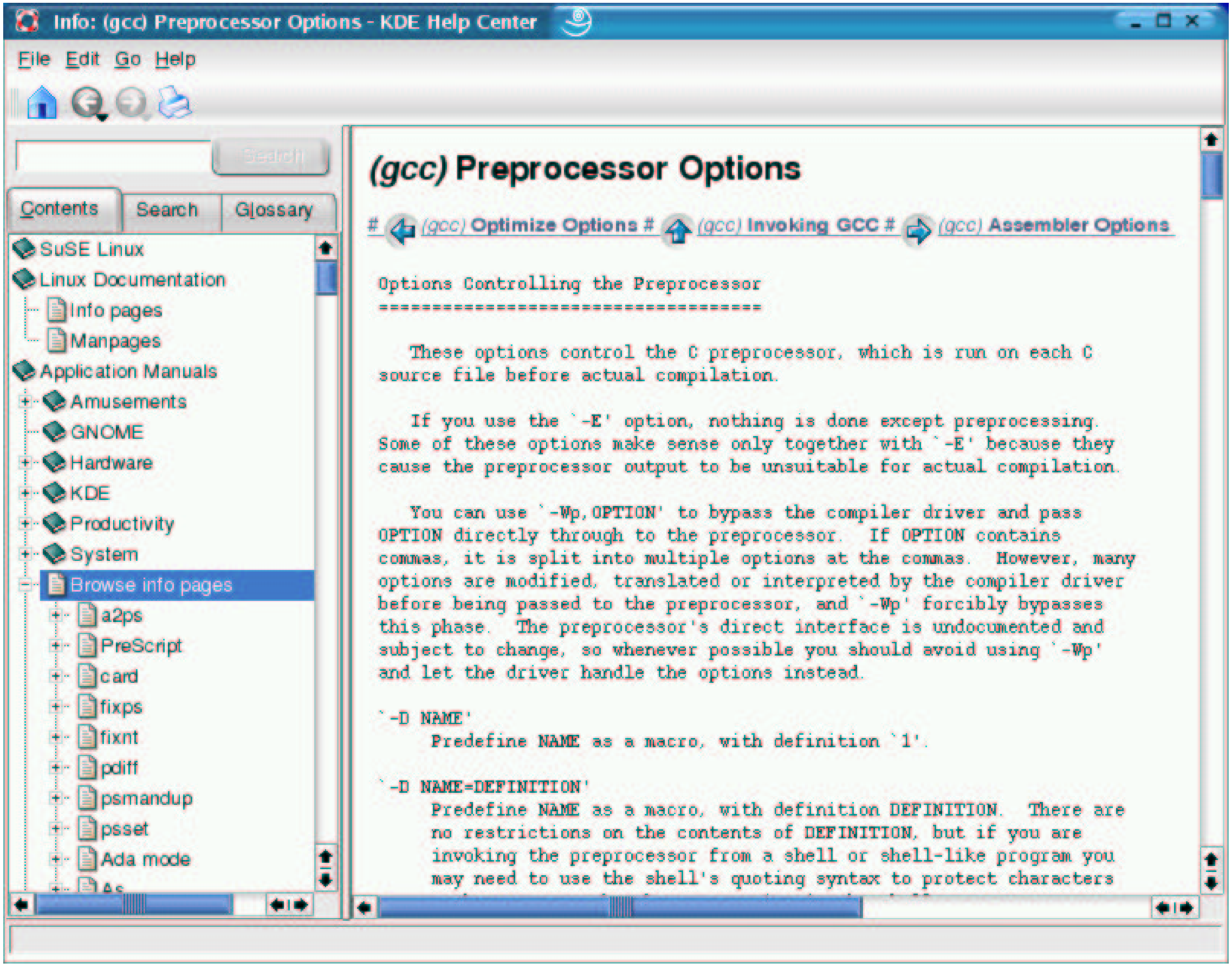
The C Preprocessor

- The C preprocessor is a separate program called by `gcc` before the compiler and it does just that: it acts on the source code and alters it before it is compiled.
- We have already (without pointing it out) made use of the preprocessor. Preprocessor instructions are those that start with a hash (#) in the first column, like `#include <stdio.h>`.
- The syntax of the preprocessor is completely different from the syntax of C; it has no understanding at all of C constructs.
- In fact, it is one of the most common errors of new programmers to try to use C constructs in a preprocessor directive.
- Where C has in principle a free format, the preprocessor does not: The hash mark (#) always must be in the first column.
- A preprocessor directive ends at the end-of-line, not with a semi-colon. A line may be continued by putting a backslash (\) at the end.

`gcc` Preprocessor Options

- `gcc -E program.c`
  puts program.c only through the preprocessor and sends the results to `stdout`. This can be invaluable to track down errors that appear to be caused by preprocessor instructions. Under Linux the output can be re-routed to a file via `gcc -E program.c > program_pp.c` or to a display tool like `less` via `gcc -E program.c | less`.
- `gcc -DNAME program.c`
  allows to define a constant `NAME` on the command line. If there are conditional compilation options this saves you from having to edit the file every time.
- `gcc -DNAME=value program.c`
  the constant `NAME` can also get a value assigned to it.
- `gcc -o FILE program.c`
  writes the compiled code into `FILE` instead of `a.out`, the C default.

gcc Preprocessor Options



(gcc) Preprocessor Options

(gcc) Optimize Options # (gcc) Invoking GCC # (gcc) Assembler Options

```
Options Controlling the Preprocessor
====================================

   These options control the C preprocessor, which is run on each C
source file before actual compilation.

   If you use the `-E' option, nothing is done except preprocessing.
Some of these options make sense only together with `-E' because they
cause the preprocessor output to be unsuitable for actual compilation.

   You can use `-Wp,OPTION' to bypass the compiler driver and pass
OPTION directly through to the preprocessor.  If OPTION contains
commas, it is split into multiple options at the commas.  However, many
options are modified, translated or interpreted by the compiler driver
before being passed to the preprocessor, and `-Wp' forcibly bypasses
this phase.  The preprocessor's direct interface is undocumented and
subject to change, so whenever possible you should avoid using `-Wp'
and let the driver handle the options instead.

`-D NAME'
     Predefine NAME as a macro, with definition `1'.

`-D NAME=DEFINITION'
     Predefine NAME as a macro, with definition DEFINITION.  There are
     no restrictions on the contents of DEFINITION, but if you are
     invoking the preprocessor from a shell or shell-like program you
     may need to use the shell's quoting syntax to protect characters
```

The `#include` Instruction

- The `#include` directive allows the program to source code from another file. The syntax is simply

```
#include file-name
```

- Files that are included in other programs are called header files and it is customary to give them the ending .h.
- If the file name is in angle brackets (<>), like

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

  the file is a standard header file. Under Linux, these files are located in `/usr/include`. Standard include files define data structures and macros used by library routines. E.g. `printf` is such a library routine.
- Local include files may be specified by using double quotes (" ") around the file name, e.g. `#include "defs.h"`. Absolute pathnames should be avoided in this case, because it makes the code less portable. (Well, that's generally true...)

The `#include` Instruction cont.

- Anything can be put into a header file. However, good programming practice allows only definitions and function prototypes.
- Include files may be nested, i.e. they may contain `#include` instructions themselves. This may lead to problems, if two include files (`one.h, two.h`) each include the same third file (`three.h`). In this case the contents of `three.h` would be included twice before the compiler is called. Depending on the contents of `three.h` this may lead to fatal errors.
- A way around this problem is to build a check into `three.h` to see if it has been included already. This can be done using the instruction `#ifndef symbol` which is true if `symbol`is not defined:

```
#ifndef _THREE_H_INCLUDED_
......
#define _THREE_H_INCLUDED_
#endif /* _THREE_H_INCLUDED _ */
```

The `#define` Instruction

- The general form of a simple `#define` statement is

  ```
  #define name substitute-text
  ```

  where `name` can be any valid C identifier and `substitute-text` can be anything.

- The preprocessor will then take any occurrence of `name` in the source code and replace it with `substitute-text` before handing it over to the compiler.

- The `#define` instruction can be used to define constants. For example, the constant PI can be defined by

  ```
  #define PI 3.1415926
  ```

- By convention, constants defined in this way are given names in upper case letters to distinguish them from variables, which are given names in lower case letters.

- `#define` can also be used to define macros, i.e. statements or groups of statements

`#define` vs `const`

- The keyword `const` is relatively new; older code only uses `#define` to define constants.
- However, `const` has several advantages:
  C checks the syntax of `const` statements immediately and `const` uses C syntax and follows normal C scope rules.
- So the two ways to define the same constant look like this:

```
#define MAX 10        /* Define a value using the preprocessor */

const int MAX = 10;   /* Define a C constant integer  */
```

- `#define` can only define simple constants, while `const` can define almost any type of C constant, including things like structure classes. (That we don't know yet and will learn about a bit later.)

The `#define` Instruction - Example 1

Simple `while` loop illustrating the use of a `#define` statement.

```
 1 #define BIG_NUMBER 10 ** 10
 2
 3 main()
 4 {
 5     /* index for our calculations */
 6     int   index;
 7     index = 0;
 8     /* syntax error on next line */
 9     while (index < BIG_NUMBER) {
10         index = index * 8;
11    }
12     return (0);
13 }
```

- Line 9 expands to

  ```
  while (index < 10 ** 10)
  ```

  and '**' is a FORTRAN operator that is illegal in C
- The preprocessor does not check for correct C syntax.

Output (trying to compile):

```
kaiser@npl03:~> make -k
gcc -g -Wall -D__USE_FIXED_PROTOTYPES__ -ansi -o big big.c
big.c:4: warning: return type defaults to ‘int’
big.c: In function ‘main’:
big.c:9: invalid type argument of ‘unary *’
make: *** [big] Error 1
```

The `#define` Instruction - Example 2

`if` construction illustrating the use of a `#define` statement.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define DIE \
 5   printf("Fatal Error:Abort\n");exit(8);
 6
 7 int main() {
 8     /* a random value for testing */
 9    int value;
10    value = 1;
11    if (value < 0)
12        DIE
13    printf("We did not die\n");
14    return (0);
15 }
```

Output: Nothing. Program exits.

- Properly indented, line 11 and 12 expand to

  ```
  if (value < 0)
      printf("Fatal Error:Abort\n");
  exit(8);
  ```

  the program exits always.
- The cure is to put curly braces ({})around all multistatement macros.

Parameterised Macros

- So far we have only discussed simple `define` statements or macros. But macros also can take parameters. For example, the following computes the square of a number:

```
#define SQR(x) ((x) * (x))
```

- When used, the macro will replace x by the text of the following argument:

  `SQR(5)` expands to `((5) * (5))`

- Always put parentheses () around the parameters of a macro, otherwise unexpected problems may occur.
- Note that there must be no space between the macro name and the parentheses that enclose the parameter.

Parameterised Macro - Example

Illustrating possible problems with a parameterised macro (`sqr.c`, correct in `sqr-i.c`).

```
#include <stdio.h>
#define SQR(x) (x * x)

int main()
{
    int counter;    /* counter for loop */

    for (counter = 0; counter < 5; ++counter)
    {
        printf("x %d, x squared %d\n",
            counter+1, SQR(counter+1));
    }
    return (0);
}
```

Output:

```
x 1, x squared 1
x 2, x squared 3
x 3, x squared 5
x 4, x squared 7
x 5, x squared 9
```

- `gcc -E` shows that `SQR(counter+1)` was expanded to `(counter+1 * counter+1)`.
- With `x` being `counter+1` this actually becomes `x-1 + x = 2x-1`.
- The problem are the missing parentheses in the macro definition.

Conditional Compilation

- The preprocessor allows conditional compilation, i.e. sections of the source code are marked and if they are compiled or not depends on whether a condition is fulfilled.
- This is typically done through a combination of the `#define` instruction with `#ifdef`, `#else`, `#ifndef` and `#endif`.
- For example, to print a message indicating that we are dealing with a debugging version of the code or with the production version could be done by switching a variable `DEBUG` on with `#define DEBUG` or off with `#undef DEBUG` and preprocessor code like

```
#ifdef DEBUG
    printf("Test version. Debugging is on.\n");
#else DEBUG
    printf("Production version\n");
#endif /* DEBUG */
```

- This feature is of great use for the portability of C code to different machines.

Bits and Bytes Revisited

- A bit is the smallest unit of information, represented by the values 0 and 1. At the machine level it corresponds to on/off, high/low, charged/discharged.
- Bit manipulations are used to control the machine at the lowest level, closest to the hardware. They are needed for low-level coding, like writing device drivers, pixel-level graphic programming or custom made data acquisition systems. In nuclear or particle physics experiments this will typically be needed somewhere.
- Eight bits form a byte. One byte can be represented by the C data type `char`.
- Instead of binary numbers, hexadecimal numbers can be used to represent bits and bytes. In this case each hexadecimal number represents 4 bits, or two hexadecimal numbers one byte.

Binary and Hexadecimal Numbers Revisited

Hexadecimal	Binary	Hexadecimal	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

- Remember that there is a difference between the number and it's representation. By changing from one system to another the number stays the same, only the representation changes.

Bitwise Operators

Bit operators, or bitwise operators, allow to work on individual bits. They work on any integer or character data type.

Operator	**Meaning**
&	bitwise and
|	bitwise or
^	bitwise exclusive or
~	complement
«	shift left
»	shift right

The Bitwise And Operator (&)

The bitwise and operator & compares two bits. If they are both 1 the result is 1, otherwise it is 0:

Bit1	**Bit2**	**Bit1&Bit2**
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```
printf("%x & %x = %x\n", 0x45, 0x71, (0x45 & 0x71));
```

outputs `45 & 71 = 41`.

This is because:

	0x45	binary 01000101
&	0x71	binary 01110001
=	0x41	binary 01000001

The Bitwise Or Operator

The inclusive or (or simply the or) operator (|) compares its two operands and if one or the other bit is 1, the result is 1.

Bit1	Bit2	Bit1&Bit2
0	0	0
0	1	1
1	0	1
1	1	1

Example:

	0x45	binary 01000101
&	0x71	binary 01110001
=	0x75	binary 01110101

It is tempting to assume that there is a simple rule for (&) and (|) with hexadecimal operands, i.e. that the result of the bitwise and & is the combination of the smaller digits, the result of the bitwise or | is the combination of the larger digits. However, this is unfortunately not true.

Bitwise Operators and Hexadecimal Numbers

Bitwise Operators and Hexadecimal Numbers (band.c).

```c
int main()
{
  int i, j;

  printf("  &");
  for (j=0; j<16; j++)
{
  printf("%3d", j);
}
  printf("\n\n");
  for (i=0; i<16; i++)
    {
      printf("%3d", i);
      for (j=0; j<16; j++)
{
  printf("%3d", i&j);
}
      printf("\n");
    }
  printf("\n\n");
  return(0);
}
```

Output:

&	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
2	0	0	2	2	0	0	2	2	0	0	2	2	0	0	2	2
3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
4	0	0	0	0	4	4	4	4	0	0	0	0	4	4	4	4
5	0	1	0	1	4	5	4	5	0	1	0	1	4	5	4	5
6	0	0	2	2	4	4	6	6	0	0	2	2	4	4	6	6
7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
8	0	0	0	0	0	0	0	0	8	8	8	8	8	8	8	8
9	0	1	0	1	0	1	0	1	8	9	8	9	8	9	8	9
10	0	0	2	2	0	0	2	2	8	8	10	10	8	8	10	10
11	0	1	2	3	0	1	2	3	8	9	10	11	8	9	10	11
12	0	0	0	0	4	4	4	4	8	8	8	8	12	12	12	12
13	0	1	0	1	4	5	4	5	8	9	8	9	12	13	12	13
14	0	0	2	2	4	4	6	6	8	8	10	10	12	12	14	14
15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The Bitwise Exclusive Or Operator

The exclusive or operator ^ (also know as xor) compares its two operands and is 1 if one bit is 1, but not both.

Bit1	**Bit2**	**Bit1^Bit2**
0	0	0
0	1	1
1	0	1
1	1	0

Example:

	0x45	binary 01000101
^	0x71	binary 01110001
=	0x34	binary 00110100

The Ones Complement Operator ($\sim$)

The ones complement or not operator $\sim$ (also know as xor) is a unary operator that returns the inverse of its operand.

Bit	$\sim$**Bit**
0	1
1	0

Example:

c=	0x45	binary 01000101
$\sim$c=	0xBA	binary 10111010

The Left and Right Shift Operators ($<<$,$>>$)

The left-shift operator « moves the data to the left by a specified number of bits. Any bits that are shifted out on the left side disappear; new bits coming in from the right are zeros. The right-shift » does the same in the other direction. For example:

	hexadecimal	**binary**	**decimal**
	c=0x1C	00011100	28
c«1	c=0x38	00111000	56
c»2	c=0x07	00000111	7

Shifting left/right by one bit is the same as multiplying/dividing by 2. `q = i >> 2` is the same as `q = i / 4`. And shifting is much faster than division.

This might give you the idea to use this trick to speed up your code. Thankfully you don't have to do this, because your compiler is smart enough to do just this for you. So don't.

Registers

- Registers can be seen as a special kind of computer memory. They have two basic functions: data storage and data movement.
- This means that they are used to hold data that are being manipulated, that are about to be send somewhere, or just received from somewhere, or they indicate e.g. a status.
- Registers in principal can have any number of bits, but 8 bit registers (1 byte) are typical.
- In registers bits can be shifted left and right, therefore one also finds the term shift register.
- As an example, take the following status register, where bit 6 is the `DONE` bit, indicating that an operation, e.g. a data transfer, is done.

7	6	5	4	3	2	1	0
ERR	DONE	BUSY	TRIG				

Setting, Clearing and Testing Bits

- Now let's assume that we have a variable `status` of type `char`, where bit 6 is the `DONE` bit. The hexadecimal form of only bit 6 being set is 0x40. We want to be able to set this bit, test if it is set and clear it.
- To set a bit we use the bitwise or operator |:

  `status = status|0x40` or `status |= 0x40`

- To test if a bit is set we use the bitwise and operator &. This is also know as masking out the bit:

  `if ((status & 0x40) != 0)` or `if (status & 0x40)`

- To clear a bit we create a mask that has all bits set except the one we want to clear. This is done using the not operator ∼. Then the variable/register is anded with the mask to clear the bit:

  `status = status & ~0x40` or `status &= ~0x40`