gcc macro

coly li

referenced from gcc manual

definition

macro is a block of code with a given name. when the name is used, it is replaced by the content of macro (that block of code).

two kinds of macro
object-like ------- data object
function-like ------- function calls

object-like macro

commonly used as symbolic names or numbers
#define BUFFER_SIZE 1024foo = (char *) malloc (BUFFER_SIZE);macro is named in uppercase makes code easire to read, use "#define" to define a macro:#define MACRO_NAME tokens

sequence

gcc processes macro in sequence, which means:

```
foo = X
#define X=4
bar = X
```

the result will be:

```
foo = X
bar = 4
```

X before the #define will not be affected

sequence 2

see this example:

```
#define    TABLESIZE    BUFSIZE
#define    BUFSIZE      16
```

NOTE, when TABLESIZE is defined, BUFSIZE is not defined yet.
In this case, TABLESIZE is just a BUFSIZE, even BUFSIZE is not used, TABLESIZE can be expended by gcc preprocessor.

If BUFSIZE is defined before TABLESIZE is referenced in code, it will not be problem to gcc.

NOTE: Even BUFSIZE is defined as 16 following TABLESIZE, if there is a

```
#undef    BUFSIZE
#define    BUFSIZE    8
```

then the followed TABLESIZE will be expended to 8 (other than 16).

self-referential macro

When a macro name appears in its definition, it is called self-referential macro. in most cases, should avoid using self-referential macro.
In gcc, macro will be expended as greedily as possible, but self-referential macro is an exception. GCC only expends self-referential macro once.
Example:

```
#define x     (4 + y)
#define y     (2 * x)
```

GCC will expend them into

```
x ==>  (4 + y) ==> (4 + (2 * x))
y ==>  (2 * x) ==> (2 * (4 + y))
```

Is it confused ?

function-like macro

if a pair of parentheses immediately follows the macro name, it's function-like macro.

```
#define lang_init()  c_init()
lang_init() ==> c_init()
```

If only uses macro name without the parentheses, it will not be expend as a function-like macro.

```
extern void foo()
#define foo() XXXXXXX /* the local version */
... ...
foo();                          /* macro version gets called */
funptr = foo; funptr();  /* external version gets called */
```

NOTE: "#define foo () bar()" foo will be treaded as object-like macro as "() bar ()"

macro arguments

For a function-like macro, the parentheses balance is mandatory.

```
#define    min(x, y)     ((x) < (y) ? (x) : (y))
```

Leaving arguments as empty is not an error for gcc reprocessor.

```
    min(, b)        ==> ((   ) < (b) ? (   ) : (b))
    min(a, )        ==> ((a  ) < ( ) ? (a  ) : ( ))
    min(,)          ==> ((   ) < ( ) ? (   ) : ( ))
    min((,),)       ==> (((,)) < ( ) ? ((,)) : ( ))
```

NOTE: can not leave all arguments as empty, for multiple arguments, at least a comma is needed.
NOTE: only parentheses balance is mandatory, no such requirement to square brackets (braces)

stringification

sometimes, one may want to convert macro arguments to string.
when a macro parameter is used with a leading '#', gcc preprocess replace it with the literal text of the actual argument, converted to a string constant.

You can not stringify a string with surrouding strings combined, but you can write a series surrounding string constants and stringified arguments. e.g.

```
#define WAR_IF(EXP) \
        do { if(EXP) fprintf(stderr, "Warning: " #EXP "\n";)} while(0)
WAR_IF(x==0) ==>
        do {if (x==0) fprintf(stderr, "Warning: " "x==0" "\n")} while
(0)
```

stringification 2

gcc preprocessor backslash-escapes the quotes surrounding the embedded string constants, and all backslashes inside string or character constant.

str to stringify	result
foo\n	foo\n
"foo\n"	\"foo\\n\"
"\n"	\"\\n\"
'\n'	\'\\n\'
\n	\n
"foo"\n"bar"\n	\"foo\"\n\"bar\"\n

By this rule, preprocessor can stringify proper content of string constant

stringification 3

there is no way to convert a macro argument to string constant. If you do want to make it, use 2 level macros

```
#define xstr(s)    str(s)
#define str(s)     #s
#define foo         4
str(foo) ==> "foo"
xstr(foo) ==> xstr(4) ==> str(4) ==> "4"
```

NOTE: it only works when foo is a macro. If foo is a varaible(e. g. int foo=4), both xstr() and str() always stringify it into "foo".

concatenation

merging 2 tokens into 1 is called token pasting or token concatenation. '##' preprocessing operator performs token pasting.

example 1:
 NAME ## _command ==> NAME_command
example 2:
#define NAME name
NAME ## _command ==> name_command

concatenation 2

concatenation expends macro before concatenating.
stringification does not expends macro before stringifying.

sometimes, ## is over used,
#define xstr(s) str(s)
#define str(s) #s
#define foo   4
#define bar   5
e.g. If you want 45, the following is over used,
        char *str=xstr(foo) ## xstr(bar);
gcc will complain a stray ##. Remove the unnecessary ##,
        char *str=xstr(foo)xstr(bar)
it works.

concatenation 3

the merged token should be valid, e.g. merging 'x' and '*' will be an invalid result, even whethere there is white space between 'x' and '*' is undefined.

If the argument is empty, ## has no effect.

and # ?

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

    struct command commands[] =
    {
      COMMAND (quit),
      COMMAND (help),
      ...
    };
```

variadic macros

A macro can be used to accept a variable number of arguments. Here is an example:
#define eprintf(...) printf(stderr, __VA_ARGS__)

this kind of macro is called variadic. When the macro is invoked, all tokens in its argument list, including commas, become variable argument. This sequence of tokens replaces the identifier __VA_ARGS__ where it appears.
eprintf("%s:%d:", __FILE__, __LINE__)
  ==> printf(stderr, "%s:%d:", __FILE__, __LINE__)

variable argument is completely macro-expended before it is inserted into the macro expansion.

variadic macros 2

An extension without __VA_ARGS__, ... follows args immediately:
#define eprintf(args...) fprintf(stderr, args)

the named arguments can also be listed with variable arguments:
#define eprintf(fmt, ...) fprintf(stderr, fmt, __VA_ARGS__)

#ifdef DEBUG
#define BUG1(fmt, args...) do{printk(fmt, args);}while(0)
#define BUG2(fmt, ...) do{printk(fmt, __VA_ARGS__);}while(0)
#else
#define BUG1(fmt, args...)
#define BUG2(fmt, ...)

variadic macros 3

If variadic variable is empty, there is a remaining comma and now variable followed. the expanded format is unacceptable by C99 compiler.

GCC extension, allow __VA_ARGS__ part to be empty, the remaining comma is allowed.

#define eprintf(fmt, ...) fprintf(stderr, fmt, ##__VA_ARGS__)
In above example, if __VA_ARGS__ is empy, the comma just before ## will be deleted by gcc.

For long time, gcc supports variable arguments by:
#define eprintf(fmt, args...) fprintf(stderr, fmt , ##args)
NOTE: in this case, there must be a blank between the comma (before ##) and what ever before the comma.

predefined macros

__FILE__, __LINE__, __DATE__, __TIME__
__STDC__, __STDC_VERSION__, __STDC_HOSTED__
__cplusplus__
__OBJC__
__ASSEMBLER__

common predefined macros, too many, not listed here.

system predefined macros

undefining macros

If a macro ceases to be useful, it can be undefined by #undef directly. #undef is only a argument, the name of macro, otherwise error will be complained. If the argument is not macro, #undef has no effect.

```
#define FOO 4
x = FOO;        ==> x = 4;
#undef FOO
x = FOO;        ==> x = FOO;
```

redefining macros

If a macro is redefined, the new definition should be effectively same to the old one. Effectively same is,

- Both are the same type of macro (object- or function-like).
- All the tokens of the replacement list are the same.
- If there are any parameters, they are the same.Whitespace appears in the same places in both. It need not be exactly the same amount of whitespace, though. Remember that comments count as whitespace.

same	different
#define FOUR (2 + 2)	#define FOUR (2 + 2)
#define FOUR (2 + 2)	#define FOUR (2+2)
#define FOUR (2 /* two */ + 2)	#define FOUR (2 * 2)

If the definition is different, gcc will use the new difinition with a warning for redefinition conflict.

macro pitfall ---

misnesting

Example 1:

```
#define twice(x) (2*(x))
#define call_with_1(x) x(1)
call_with_1 (twice)
    ==> twice(1)
    ==> (2*(1))
```

Example 2:

```
#define strange(file) fprintf (file, "%s %d",
...
strange(stderr) p, 35)
    ==> fprintf (stderr, "%s %d", p, 35)
```

Example 1 might be useful, example 2 is confused and should be avoided.

macro pitfall ----

Operator Precedence Problems

```
#define ceil_div(x, y) (x + y - 1) / y

a = ceil_div (b & c, sizeof (int));
    ==> a = (b & c + sizeof (int) - 1) / sizeof (int);
```

This does not do what is intended. The operator-precedence rules of C make it equivalent to this:

```
    a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

What we want is this:

```
    a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as

```
    #define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

provides the desired result.

USING parentheses explicitly.

macro pitfall ---

Swallowing the Semicolon

do {...} while (0);

macro pitfall ---

Duplication of Side Effects

First implementation:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
next = min(x + y, foo(z))
        ==> ((x + y) < (foo(z)) ? (x + y) : (foo(z)))
```

In this implementation, there are 2 major issues:

1) foo(z) gets called twice, which may result different value each time.

2) x + y may not be able to compare with foo(z), data type incompatible.

macro pitfall ---

Duplication of Side Effects (2)

Improved implementation with gcc typeof() extension:

```
#define min(X, Y)   \
        ({ typeof (X) x_ = (X); typeof (Y) y_ = (Y); \
        (x_ < y_) ? x_ : y_; })
```

advantage:

-X and Y only get referenced once

disadvantage:

- If X and Y is not same data types, compiler may automatically upgrad data type for one of them. Sometimes, this is what programer does not want to.

macro pitfall ---

Duplication of Side Effects (3)

The further Improved implementation with gcc typeof() extension:

```
#define min(X, Y)   \
        ({ typeof (X) x_ = (X); typeof (Y) y_ = (Y); \
          (void) (&x_ == &y_); \
          (x_ < y_) ? x_ : y_; })
```

advantage:

- If X and Y are different data type, there will be an error in compiling time.

disadvantage:

-slow ??

macro pitfall ---

Newlines in Arguments

Some time, the error line number may not be the location where the real problem is.