**C语言宏定义##连接符和#符的使用**

C语言中如何使用宏C（和C++）中的宏（Macro）属于编译器预处理的范畴，属于编译期概念（而非运行期概念）。下面对常遇到的宏的使用问题做了简单总结。

**关于#和##**

在C语言的宏中，#的功能是将其后面的宏参数进行字符串化操作（Stringfication），简单说就是在对它所引用的**宏变量** 通过替换后在其左右各加上一个双引号。比如下面代码中的宏：

#define WARN\_IF(EXP) do{ if (EXP) fprintf(stderr, "Warning: " #EXP "/n"); } while(0)

那么实际使用中会出现下面所示的替换过程：

WARN\_IF (divider == 0);

**被替换为**

do {

if (divider == 0)

fprintf(stderr, "Warning" "divider == 0" "/n");

} while(0);

这样每次divider（除数）为0的时候便会在标准错误流上输出一个提示信息。

而##被称为连接符（concatenator），用来将两个Token连接为一个Token。注意这里连接的对象是Token就行，而*不一定* 是宏的变量。比如你要做一个菜单项命令名和函数指针组成的结构体的数组，并且希望在函数名和菜单项命令名之间有直观的、名字上的关系。那么下面的代码就非常实用：

struct command

{

char \* name;

void (\*function) (void);

};

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

// 然后你就用一些预先定义好的命令来方便的初始化一个command结构的数组了：

struct command commands[] = {

COMMAND(quit),

COMMAND(help),

...

}

COMMAND宏在这里充当一个代码生成器的作用，这样可以在一定程度上减少代码密度，间接地也可以减少不留心所造成的错误。我们还可以n个##符号连接 n+1个Token，这个特性也是#符号所不具备的。比如：

#define LINK\_MULTIPLE(a,b,c,d) a##\_##b##\_##c##\_##d

typedef struct \_record\_type LINK\_MULTIPLE(name,company,position,salary);

// 这里这个语句将展开为：

// typedef struct \_record\_type name\_company\_position\_salary;

**关于...的使用**

...在C宏中称为Variadic Macro，也就是变参宏。比如：

#define myprintf(templt,...) fprintf(stderr,templt,\_\_VA\_ARGS\_\_)

// 或者

#define myprintf(templt,args...) fprintf(stderr,templt,args)

第一个宏中由于没有对变参起名，我们用默认的宏\_\_VA\_ARGS\_\_来替代它。第二个宏 中，我们显式地命名变参为args，那么我们在宏定义中就可以用args来代指变参了。同C语言的stdcall一样，变参必须作为参数表的最有一项出 现。当上面的宏中我们只能提供第一个参数templt时，C标准要求我们必须写成：

myprintf(templt,);

的形式。这时的替换过程为：

myprintf("Error!/n",);

**替换为：**

fprintf(stderr,"Error!/n",);

这是一个语法错误，不能正常编译。这个问题一般有两个解决方法。首先，GNU CPP提供的解决方法允许上面的宏调用写成：

myprintf(templt);

而它将会被通过替换变成：

fprintf(stderr,"Error!/n",);

很明显，这里仍然会产生编译错误（非本例的某些情况下不会产生编译错误）。除了这种方式外，c99和GNU CPP都支持下面的宏定义方式：

#define myprintf(templt, ...) fprintf(stderr,templt, ##\_\_VAR\_ARGS\_\_)

这时，##这个连接符号充当的作用就是当\_\_VAR\_ARGS\_\_为空的时候，消除前面的那个逗号。那么此时的翻译过程如下：

myprintf(templt);

**被转化为：**

fprintf(stderr,templt);

这样如果templt合法，将不会产生编译错误。 这里列出了一些宏使用中容易出错的地方，以及合适的使用方式。

**错误的嵌套－Misnesting**

宏的定义不一定要有完整的、配对的括号，但是为了避免出错并且提高可读性，最好避免这样使用。

**由操作符优先级引起的问题－Operator Precedence Problem**

由于宏只是简单的替换，宏的参数如果是复合结构，那么通过替换之后可能由于各个参数之间的操作符优先级高于单个参数内部各部分之间相互作用的操作符优先级，如果我们不用括号保护各个宏参数，可能会产生预想不到的情形。比如：

#define ceil\_div(x, y) (x + y - 1) / y

那么

a = ceil\_div( b & c, sizeof(int) );

将被转化为：

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

// 由于+/-的优先级高于&的优先级，那么上面式子等同于：

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

这显然不是调用者的初衷。为了避免这种情况发生，应当多写几个括号：

#define ceil\_div(x, y) (((x) + (y) - 1) / (y))

**消除多余的分号－Semicolon Swallowing**

通常情况下，为了使函数模样的宏在表面上看起来像一个通常的C语言调用一样，通常情况下我们在宏的后面加上一个分号，比如下面的带参宏：

MY\_MACRO(x);

但是如果是下面的情况：

#define MY\_MACRO(x) { /\* line 1 \*/ /\* line 2 \*/ /\* line 3 \*/ }

//...

if (condition())

MY\_MACRO(a);

else

{...}

这样会由于多出的那个分号产生编译错误。为了避免这种情况出现同时保持MY\_MACRO(x);的这种写法，我们需要把宏定义为这种形式：

#define MY\_MACRO(x) do {

/\* line 1 \*/ /\* line 2 \*/ /\* line 3 \*/ } while(0)

这样只要保证总是使用分号，就不会有任何问题。

**Duplication of Side Effects**

这里的Side Effect是指宏在展开的时候对其参数可能进行多次Evaluation（也就是取值），但是如果这个宏参数是一个函数，那么就有可能被调用多次从而达到不一致的结果，甚至会发生更严重的错误。比如：

#define min(X,Y) ((X) > (Y) ? (Y) : (X))

//...

c = min(a,foo(b));

这时foo()函数就被调用了两次。为了解决这个潜在的问题，我们应当这样写min(X,Y)这个宏：

#define min(X,Y) ({ typeof (X) x\_ = (X); typeof (Y) y\_ = (Y); (x\_ < y\_) ? x\_ : y\_; })

({...})的作用是将内部的几条语句中最后一条的值返回，它也允许在内部声明变量（因为它通过大括号组成了一个局部Scope）。