[linux内核中的min、max函数](http://blog.csdn.net/snowbabybw/article/details/6120780)

这些天为了整理一下前段时间看ldd3时所学的驱动知识，所以就去看了看usb驱动。不看不知道，一看吓一跳，里面有很多语法我发现用的太好了，不像我们平时那样写代码。里面写的代码真是太好了。然而要理解到里面的东西还是不容易的。

今天看见usb\_stor\_access\_xfer\_buf函数的时候，想看看里面的min 函数是不是按照我们写的那么写。但是一看傻啦，还有些东东我还没用过，于是赶紧google一下，那到底下面代码是什么样的呢？大家可以到、include/linux/kernel.h 里面去找。不相找事吧，那好我贴在这里。

/\*  
 \* min()/max() macros that also do  
 \* strict type-checking.. See the  
 \* "unnecessary" pointer comparison.  
 \*/  
#define min(x,y) ({ /  
 typeof(x) \_x = (x); /  
 typeof(y) \_y = (y); /  
 (void) (&\_x == &\_y);  /  
 \_x < \_y ? \_x : \_y; })

#define max(x,y) ({ /  
 typeof(x) \_x = (x); /  
 typeof(y) \_y = (y); /  
 (void) (&\_x == &\_y);  /  
 \_x > \_y ? \_x : \_y; })

当我google 了以后发现很多人都看过这个函数，看来我是out了。这都是在大学欠下的账啊，正所谓：“出来混，迟早都要还的”，还好今天就开始学了。

下面看一位兄弟的理解，他讲的很好。

<http://blog.chinaunix.net/u/24474/showart_1101969.html>

**min宏定义探究**

|  |  |
| --- | --- |
| |  | | --- | | 在GCC的文档中建议使用如下的min宏定义： 引用：#define min(X,Y)  / (\_\_extension\_\_  / ({  /    typeof(X) \_\_x=(X), \_\_y=(Y);   /    (\_\_x<\_\_y)?\_\_x:\_\_y;  / }) / )   本文讨论了这样作法的意义。     1、传统的min带来的副作用     2、GCC中的({statement list})的扩展     3、typeof(expression)     4、\_\_extension\_\_的含义     5、使用typeof和({})实现min，避免了副作用  附录1、旧版本的的GCC中的的解决方法 附录2、C++中使用template的解决方法  1、传统的min带来的副作用  min通常被定义成这样的宏： #define min(X,Y) ((X) < (Y) ? (X) : (Y))  这种定义会带来一些副作用，看下面的例子：  int x = 1, y = 2; int main() {    printf("min=%d/n", min(x++, y++));    printf("x = %d, y = %d/n", x, y); }  执行完min(x++、y++)，我们期望x的值为2，y的值为3。  但是，实际的结果是，执行完mini(x++、y++)后，x的值为3，y的值为3，原因在于宏展开后x++被执行了两次：  引用： int x = 1, y = 2; int main() {    printf("min=%d/n", x++ < y++ ? x++ : y++);    printf("x = %d, y = %d/n", x, y); }  2、GCC中的({statement list})的扩展 ({statement list})是一个表达式，逗号表达式类似，但是功能更强，({与})中可以包含有多条语句(可以是变量定义、复杂的控制语句)，该表达式的值为statement list中最后一条语句的值，举例：  int main() {     int result = ({      int i, sum = 0;      for (i = 1; i <= 100; i++)           sum+= i;      sum;     })      printf("result=%d/n", result); }  运行结果： result=5050  3、typeof(expression) typeof(expression)用来获取expression的类型，举例：  int main() {    int integer;    typeof(100) i;  /\* 表达式100的类型是int，定义了int型变量i \*/    typeof(integer) j; /\* 表达式integer的类型是int，定义了int型变量j \*/     i = 1;    j = 2; }  4、\_\_extension\_\_的含义 GCC引入了很多标准C中的没有的扩展，如({和)}，GCC提供了pednatic选项用于检测程序是否使用了GCC的扩展，当使用pedantic选项编译如下程序时  int main() {     int result = ({      int i, sum = 0;      for (i = 1; i <= 100; i++)           sum+= i;      sum;     })     printf("result=%d/n", result); }  编译器发出警告： $ cc -pedantic test.c test.c: 在函数 ‘main’ 中： test.c:9: 警告：ISO C 不允许在表达式中使用花括号组  编译器提醒程序员，这段C程序使用了不符合ISO C标准的语法，如果使用其他的编译器(非GCC)编译这段代码有能会出错。在所有使用GNU 扩展关键字的表达式之前加\_\_extension\_\_ 关键字后，使用pedantic选项编译时，编译器就不再发出警告信息:  int main() {     int result = \_\_extension\_\_({       int i, sum = 0;       for (i = 1; i <= 100; i++)         sum+= i;       sum;     })      printf("result=%d/n", result); }  $ cc -pedantic test.c $ 编译成功！  5、使用typeof和({})实现min，避免了副作用  #define min(X,Y) / ({  / typeof(X) \_\_x=(X), \_\_y=(Y);  / (\_\_x<\_\_y)?\_\_x:\_\_y;  / })  使用传统的min会出现问题的例子：  int x = 1, y = 2;; int main() {    printf("min=%d/n", min(x++, y++));    printf("x = %d, y = %d/n", x, y); }  它被扩展为 引用： int x = 1, y = 2;; int main() {    printf("min=%d/n", ({        typeof(x) \_\_x = (x++), \_\_y = (y++);  /\* 定义了两个整形变量 \*/        (\_\_x<\_\_y)?\_\_x:\_\_y;     })    );    printf("x = %d, y = %d/n", x, y); }   在执行min(x++, y++)期间，x++和y++只执行了一次，因而结果是正确的。  附录1、旧版本的的GCC中的的解决方法 旧版本的GCC提供了两个内置的运算操作符：<?和>?， <?返回两个操作数中较小的一个，>?返回两个操作数中较大的一个，使用这两个操作符定义的min如下:  #define min(x, y) ((x) <? (y)) #define max(x, y) ((x) >? (y)) 但是新版本的GCC文档中宣称：现在这两个运算操作符已经过时了，建议大家不要使用。  附录2、C++中使用template的解决方法 template <class type> type min(type a, type b) {      return a < b ? a : b; }   add linux kernel min, max define: include/linux/kernel.h  /\*  \* min()/max() macros that also do  \* strict type-checking.. See the  \* "unnecessary" pointer comparison.  \*/ #define min(x,y) ({ /         typeof(x) \_x = (x);     /         typeof(y) \_y = (y);     /         (void) (&\_x == &\_y);            /         \_x < \_y ? \_x : \_y; })  #define max(x,y) ({ /         typeof(x) \_x = (x);     /         typeof(y) \_y = (y);     /         (void) (&\_x == &\_y);            /         \_x > \_y ? \_x : \_y; }) | |
| Min和Max宏：  　　 　　/\* 　　\* min()/max() macros that also do 　　\* strict type-checking.. See the 　　\* "unnecessary" pointer comparison. 　　\*/ 　　#define min(x,y) ({ typeof(x) \_x = (x); typeof(y) \_y = (y); (void) (&\_x == &\_y); \_x < \_y ? \_x : \_y; }) 　　#define max(x,y) ({ typeof(x) \_x = (x); typeof(y) \_y = (y); (void) (&\_x == &\_y); \_x > \_y ? \_x : \_y; }) 　　/\* 　　\* ..and if you can't take the strict 　　\* types, you can specify one yourself. 　　\* 　　\* Or not use min/max at all, of course. 　　\*/ 　　#define min\_t(type,x,y) ({ type \_\_x = (x); type \_\_y = (y); \_\_x < \_\_y ? \_\_x: \_\_y; }) 　　#define max\_t(type,x,y) ({ type \_\_x = (x); type \_\_y = (y); \_\_x > \_\_y ? \_\_x: \_\_y; }) 　　不是感觉跟我们用的有些不一样啊:  　　(void) (&\_x == &\_y);  　　(void) (&\_x == &\_y)这句话本身都执行程序来讲完全是一句废话，它的作用在于，本身我们无法做这样的操作typeof(\_x)==typeof(\_y)，所以故意判断他们2个的地址指针是否相等，显然是不可能相等，但是如果\_x和\_y的类型不一样，其指针类型也会不一样，2个不一样的指针类型进行比较操作，会抛出一个编译警告。也就是说char \*p; int \*q; 然后p==q;，这个判断因为一个是char\*一个是int\*，会在编译时产生一个warning。巧妙就巧妙在这里。  　　由于内核是很多开发着一起开发的，其中还有一些其他的实现，就跟我们平常用的一样：  　　#define min(a,b) (((a) < (b)) ? (a) : (b))  　　试想：  　　min(++a,++b) ==> ((++a)<(++b))?(++a):(++b)  　　是不是就有问题了，传入的参数被加了两次。 |

好了，我就是把别人的东西收集了一下，到现在你明白里面的GCC中的({statement list})的扩展了吗，应该基本可以理解了。接着努力，路漫漫其修远兮