Linux内核必须使用GNU的GCC编译器来编译，而GCC提供了很多的C语言扩展，这些扩展对优化、目标代码布局、更安全的检查等提供了很强的支持。因此，内核代码所使用的C语法并不完全符合ANSI C标准，实际上，只要有可能，内核开发者总是要用到GCC提供的C语言扩展部分。所以特意找了几个常用的特性总结下。

**1、语句内嵌表达式（statement-embedded expression）**

在gnu c 中，用括号将复合语句括起来也形成了表达式。他允许你在一个表达式内使用循环，跳转和局部变量。

一个复合语句是用大括号{}括起来的一组语句。在包含语句的表达式这种结构中，再用括号( )将大括号括起来

例如：

({ int y = foo (); int z;

  if (y > 0) z = y;

  else z = - y;

  z; }

)

就是一个合法表达式，用于计算foo( )函数返回值的绝对值。

在上面的复合语句中，最后的一句必须是一个以分号结尾的表达式。这个表达式代表了整个结构的值。

如果你在大括号里的最后一句用的是其他的语句，则整个结构的返回类型为void，即没有合法的返回值。

这种特性使得宏定义变得更加安全（因为每个操作数都只被计算一次，例如++运算）。例如计算最大值通常在c语言中被定义为这样的宏：

#define max(a,b) ((a) > (b) ? (a) : (b))

但是其中的a和b可能会被计算两次，如果操作数带有副作用，则会产生错误的结果。

在gnu c中，如果你知道了操作数的类型（假设为int），你可以这样安全的定义宏：

#define maxint(a,b) \

({int \_a = (a), \_b = (b); \_a > \_b ? \_a : \_b; })

内核中做法：

#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; })

使用语句表达式只计算参数一次，避免了可能的错误。

语句内嵌在常量表达式（例如枚举类型），位域尺寸或静态变量初始化中是不允许的。

如果你不知道操作数的类型，你也可以使用typeof来获得类型。

典型用法用下单链表pop操作：

#define \_$slist\_pop(H,L)\

({\

typeof((H)->L) \_p$ = (H)->L;\

if( \_p$){ (H)->L = \_p$->next; \_p$->next = NULL;} \_p$;\

})

使用 typeof 获得类型、最后一个 \_p$ 用于返回值。

**2、可变参数宏 \_\_VA\_ARGS\_\_ 常用： ##\_\_VA\_ARGS\_\_**

举例如下：

#define err\_log(fmt, ...) fprintf (stderr, fmt, \_\_VA\_ARGS\_\_)

其中的"…"表示可变参数，实际调用时，它们会替代宏体里的\_\_VA\_ARGS\_\_。

但在fprintf的参数列表中最后的逗号后面没有参数。在编译时就会报错。

此时可如此定义：

#define err\_log(fmt, ...) fprintf(stderr, fmt, ##\_\_VA\_ARGS\_\_)

"##"主要针对参数为空的情况。既然称为可变参数，那传递空参数也是可以的。如果没有使用"##"，传递空参数时，比如：

err\_log("A null message");

宏展开后，其中的字符串后面会多个多余的逗号，而"##"则会使预处理器去掉这个多余的逗号。

还有如下的宏方便用于调试：

\_\_FILE\_\_ 宏在预编译时会替换成当前的源文件名

\_\_LINE\_\_宏在预编译时会替换成当前的行号

\_\_FUNCTION\_\_宏在预编译时会替换成当前的函数名称

**3、GCC 中零长数组**

GCC 中允许使用零长数组，把它作为结构体的最后一个元素非常有用.

struct line {

    int length;

    char contents[0];

};

struct line \*thisline = (struct line \*) malloc (sizeof (struct line) + this\_length);

thisline->length = this\_length;

零长数组在有固定头部的可变对象上非常适用，我们可以根据对象的大小动态地去分配结构体的大小。

作为零长度数组的原始实现的神奇之处，sizeof被赋值为0。sizeof (struct line) = 4B

**4、标号元素**

在标准C里，数组或结构变量的初始化值必须以固定的顺序出现，而在GCC中，通过指定索引或结构域名，

则允许初始化值以任意顺序出现。

指定数组索引的方法是在初始化值前写"[INDEX] ="，还可以使用"[FIRST ... LAST] ="的形式指定一个范围

int platform\_intr\_list[ACPI\_MAX\_PLATFORM\_INTERRUPTS] = {

   [0 ... ACPI\_MAX\_PLATFORM\_INTERRUPTS - 1] = -1

};

将数组platform\_intr\_list的任何元素都初始化为-1

对于结构初始化，比如：

const struct file\_operations ext2\_file\_operations = {

      .llseek     = generic\_file\_llseek,

      .read       = do\_sync\_read,

      .write      = do\_sync\_write,

}

将结构ext2\_file\_operations的元素llseek初始化为generic\_file\_llseek，当结构体的定义变化导致元素的偏移位置改变时，仍然可以确保已知元素的正确性。

**5、特殊属性（\_\_attribute\_\_）**

GCC允许声明函数、变量和类型的特殊属性，以便指示编译器进行特定方面的优化和更仔细的代码检查。使用方式为在声明后加上：

attribute\_\_ (( ATTRIBUTE ))

其中ATTRIBUTE是属性的说明，多个说明之间以逗号分隔.

//=========变参调用的编译检查宏==========

#ifndef \_AS\_PRINTF

#if defined(\_\_GNUC\_\_)

# define \_$PRINTF(m,n) \_\_attribute\_\_ ((\_\_format\_\_ (\_\_printf\_\_, (m), (n))))

# define \_$SCANF(m,n) \_\_attribute\_\_ ((\_\_format\_\_ (\_\_scanf\_\_, (m), (n))))

#else

# define \_$PRINTF(m,n)

# define \_$SCANF(m,n)

#endif

#endif

1.声明变长数组

在VC中,对于数组的声明,只能是常量,而不能够是变量.而在gcc中如下代码却是可以成功运行的

代码:

#include

int main()

{

int i, j;

scanf("%d%d", &i, &j);

int a[i][j];

printf("%d\n", sizeof(a));

return 0;

}

在这里, 可以在运行的时候再决定这个数组的大小为多少, 就避免了使用malloc在堆中分配这种方法.

2.为数组赋值

如果声明了一个100000长度的整形数组,需要将这个数组的最后一位赋为10,而其余位全部赋为0,如果在VC中,则需要使用循环的方式将前99999赋为0,然后再将最后一位赋为10,而使用gcc则可以很好地解决这个问题. 如下:

代码:

#include

int main()

{

int i;

int array[100000] = {[99999] = 10};

for (i = 0; i < 100000; i++)

printf("%d \n", array[i]);

return 0;

}

此时,除了最后一位为10,其余位全为0.

3.在C程序中在main()函数之前运行的函数

通过gcc的函数的属性,可以设置某些函数在main()函数之前运行.如:

代码:

#include

void first() \_\_attribute\_\_((constructor));

void first()

{

printf("This is function %s\n", \_\_FUNCTION\_\_);

main();

return ;

}

int main()

{

printf("This is function %s\n", \_\_FUNCTION\_\_);

return 0;

}

通过设置函数属性为constuctor, 可使其在main()之前运行.