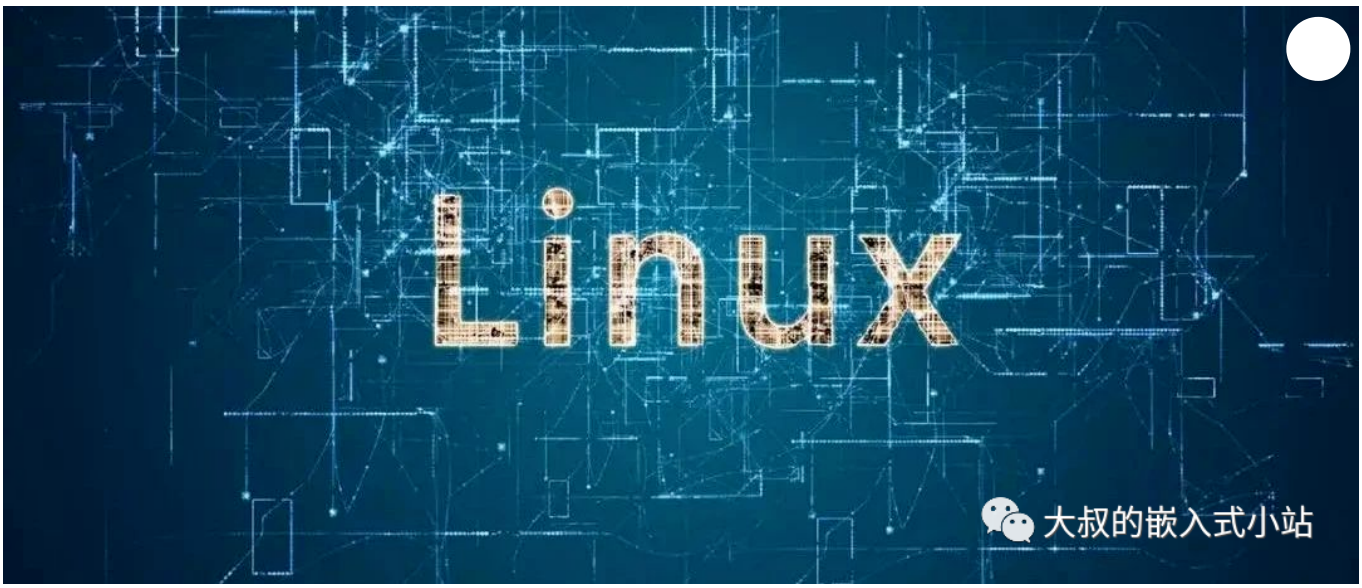
从Linux内核中学习高级C语言宏技巧

原创 UncleShine 大叔的嵌入式小站 2023-03-10 12:35 发表于浙江

收录于合集

#linux内核 2 #c语言 5 #linux 4

Linux内核可谓是集C语言大成者，从中我们可以学到非常多的技巧，本文来学习一下宏技巧，文章有点长，但耐心看完后C语言level直接飙升。



1.用do{}while(0)把宏包起来

```
#define init_hashtable_nodes(p, b)  do {      \
  int _i;                \
  hash_init((p)->htable##b);        \
  ...略去          \
} while (0)
```

Linux中常见如上定义宏的形式，我们都知道do{}while(0)只执行一次，**那么这个有什么意义呢?**

我们写一个更简单的宏，来看看

```
#define fun(x) fun1(x);fun2(x);
```

则在这样的语句中：

```
if(a)
  fun(a);
```

被展开为

```
if(a)
  fun1(x);fun2(x);;
```

fun2(x)将不会执行！有同学会想，加个花括号

```
#define fun(x) {fun1(x);fun2(x);}
```

则在这样的语句中

```
if (a)
  fun(a);
else
  fun3(a);
```

被展开为

```
if (a)
  {fun1(x);fun2(x);};
else
  fun3(a);
```

注意}后还有个;这将会**出现语法错误**。

但是假如我们写成

```
#define fun(x) do{fun1(x);fun2(x);}while(0)
```

则完美避免上述问题！

2.获取数组元素个数

写一个获取数组中元素个数的宏怎么写？显然用sizeof

```
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof(*arr))
```

可以用，但这样是存在问题的，先看个例子

```
#include<stdio.h>
int a[3] = {1,3,5};
int fun(int c[])
{
  printf("fun1 a= %d\n",sizeof(c));
}
int main(void)
{
  printf("a= %d\n",sizeof(a));
  fun(a);
  return 0;
}
```

输出：

```
a = 12;
b = 8; //32位电脑为4
```

为什么？**因为数组名和指针不是完全一样的，函数参数中的数组名在函数内部会降为指针！sizeof(a),在函数中实际上变成了sizeof(int *)。**

上面的宏存在的问题也就清楚了，**这是一个非常重大，且容易忽略的bug！**

让我们看看，内核中怎么写：

```
#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]) + __must_be_array(arr)
```

(arr)[0]是0长数组，不占用内存，GNU C支持0长数组，在某些编译器下可能会出错。（不过不是因为这个来避开上面的问题）

sizeof(arr) / sizeof((arr)[0]很好理解数组大小除去元素类型大小即是元素个数，真正的精髓在于后面__must_be_array(arr)宏

```
#define __must_be_array(a)  BUILD_BUG_ON_ZERO(__same_type((a), &(a)[0]))
```

先看内部的__same_type，它也是个宏

```
# define __same_type(a, b) __builtin_types_compatible_p(typeof(a), typeof(b))
```

__builtin_types_compatible_p 是gcc内联函数，在内核源码中找不到定义也无需包含头文件，在代码中也可以直接使用这个函数。（只要是用gcc编译器来编译即可使用，**不用管这个，**只需知道：

当 a 和 b 是同一种数据类型时，此函数返回 1。

当 a 和 b 是不同的数据类型时，此函数返回 0。

再看外部的（**精髓来了**）

```
#define BUILD_BUG_ON_ZERO(e) (sizeof(struct { int:-!!(e); }))
```

上来就是个小技巧：!!(e)是将e转换为0或1，加个-号即将e转换为0或-1。

再用到了位域：

有些信息在存储时，并不需要占用一个完整的字节， 而只需占几个或一个二进制位。例如在存放一个开关量时，只有0和1 两种状态，用一位二进位即可。这时候可以用位域

```
struct struct_a{
```

```
  char a:3;
  char b:3;
  char c;
};
```

a占用3位，b占用3位，如上结构体只占用2字节，位域可以为无位域名，这时它只用来作填充或调整位置，不能使用，如：

```
struct struct_a{
  char a:3;
  char  :3;
  char c;
};
```

当位数为负数时编译无法通过！

当a为数组时，__same_type((a), &(a)[0])，&(a)[0]是个指针，两者类型不同，返回0，即e为0，-!!(e)为0，sizeof(struct { int:0; })为0，编译通过且不影响最终值。

当a为指针时，__same_type((a), &(a)[0])，两者类型相同，返回1，即e为1，-!!(e)为-1，无法编译。

3.求两个数中最大值的宏MAX

思考这个问题，你会怎么写

3.1一般的同学：

```
#define MAX(a,b) a > b ? a : b
```

存在问题，例子如下：

```
#include<stdio.h>
#define MAX(x,y) x > y ? x: y
int main(void)
{
```

```
    int i = 14;
    int j = 3;
    printf ("i&0b101 = %d\n",i&0b101);
    printf ("j&0b101 = %d\n",j&0b101);
    printf("max=%d\n",MAX(i&0b101,j&0b101));
    return 0;
}
```

输出：

```
i&0b101 = 4
j&0b101 = 1
max=1
```

明显不对，因为>运算符优先级大于&，所以会先进行比较再进行按位与。

3.2稍好的同学：

```
#define MAX(a,b) (a) > (b) ? (a) : (b)
```

存在问题，例子如下：

```
#define MAX(x,y) (x) > (y) ? (x) : (y)
int main(void)
{
  printf("max=%d",3 + MAX(1,2));
  return 0;
}
```

输出：

```
max = 1
```

同样是优先级问题+优先级大于>。

附优先级表：同一优先级的运算符，运算次序由结合方向所决定。

优先级	运算符	名称或含义	使用形式	结合方向
1	[]	数组元素下标	数组名[常量表达式]	左到右
	()	圆括号、函数参数表	(表达式）/函数名(形参表)	
	.	成员选择（对象）	对象.成员名	
	->	成员选择（指针）	对象指针->成员名	
2	-	负号运算符	-表达式	右到左
	~	按位取反运算符	~表达式	
	++	自增运算符	++变量名/变量名++	
	--	自减运算符	--变量名/变量名--	
	*	取值运算符	*指针变量	
	&	取地址运算符	&变量名	
	!	逻辑非运算符	!表达式	
	(类型)	强制类型转换	(数据类型)表达式	
	sizeof	长度运算符	sizeof(表达式)	
3	/	除	表达式 / 表达式	左到右
	*	乘	表达式 * 表达式	
	%	余数（取模）	整型表达式 % 整型表达式	
4	+	加	表达式 + 表达式	左到右
	-	减	表达式 - 表达式	
5	<<	左移	变量 << 表达式	左到右
	>>	右移	变量 >> 表达式	
6	>	大于	表达式 > 表达式	左到右
	>=	大于等于	表达式 >= 表达式	
	<	小于	表达式 < 表达式	
	<=	小于等于	表达式 <= 表达式	
7	==	等于	表达式 == 表达式	左到右
	! =	不等于	表达式 != 表达式	

8	&	按位与	表达式 & 表达式	左到右
9	^	按位异或	表达式 ^ 表达式	左到右
10	|	按位或	表达式 | 表达式	左到右
11	&&	逻辑与	表达式 && 表达式	左到右
12	||	逻辑或	表达式 || 表达式	左到右
13	?:	条件运算符	表达式1? 表达式2: 表达式3	**右到左**
14	=	赋值运算符	变量 = 表达式	**右到左**
	/=	除后赋值	变量 /= 表达式	
	*=	乘后赋值	变量 *= 表达式	
	%=	取模后赋值	变量 %= 表达式	
	+=	加后赋值	变量 += 表达式	
	-=	减后赋值	变量 -= 表达式	
	<<=	左移后赋值	变量 <<= 表达式	
	>>=	右移后赋值	变量 >>= 表达式	
	&=	按位与后赋值	变量 &= 表达式	
	^=	按位异或后赋值	变量 ^= 表达式	
	|=	按位或后赋值	变量 |= 表达式	
15	,	逗号运算符	表达式, 表达式, ...	左到右

3.3良好的同学

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

避免了前两个出现的问题，但同样还有问题存在：

```
#include<stdio.h>

```

```
#define MAX(x,y) ((x) > (y) ? (x): (y))
int main(void)
{
  int i = 2;
  int j = 3;
  printf("max=%d\n",MAX(i++,j++));
  printf("i=%d\n",i);
  printf("j=%d\n",j);
  return 0;
}
```

期望结果：

```
max=3, i=3, j=4
```

实际结果

```
max=4, i=3, j=5
```

尽管用括号避免了优先级问题，但这个例子中的j++实际上运行了两次。

3.4Linux内核中的写法

```
#define MAX(x, y) ({        \
  typeof(x) _max1 = (x);      \
  typeof(y) _max2 = (y);      \
  (void) (&_max1 == &_max2);    \
  _max1 > _max2 ? _max1 : _max2; })
```

下面进行详解。

3.4.1.GNU C中的语句表达式

表达式就是由一系列操作符和操作数构成的式子。 例如三面三个表达式

```
a+b
```

```
i=a*2
a++
```

表达式加上一个分号就构成了**语句**，例如，下面三条语句：

```
a+b;
i=a*2;
a++;
```

A compound statement enclosed in parentheses may appear as an expression in GNU C.

——《Using the GNU Compiler Collection》6.1 Statements and Declarations in Expressions

GNU C允许在表达式中有复合语句,称为**语句表达式：**

```
({表达式1;表达式2;表达式3;...})
```

语句表达式内部可以有局部变量，语句表达式的值为内部最后一个表达式的值。

例子：

```
int main()
{
  int y;
  y = ({ int a =3; int b = 4;a+b;});
  printf("y = %d\n",y);
  return 0;
}
```

输出：y = 7。

这个扩展使得宏构造更加安全可靠，我们可以写出这样的程序：

```
#define max(x, y) ({        \
```

```
  int _max1 = (x);      \
  int _max2 = (y);      \
  _max1 > _max2 ? _max1 : _max2; })
int main(void)
{
  int i = 2;
  int j = 3;
  printf("max=%d\n",max(i++,j++));
  printf("i=%d\n",i);
  printf("j=%d\n",j);
  return 0;
}
```

但这个宏还有个缺点，只能比较int型变量，改进一下：

```
#define max(type, x, y) ({        \
  type _max1 = (x);      \
  type _max2 = (y);      \
  _max1 > _max2 ? _max1 : _max2; })
```

但这需要传入type，还不够好。

3.4.2 typeof关键字

GNU C 扩展了一个关键字 typeof，用来获取一个变量或表达式的类型。

例子：

```
int a;
typeof(a) b = 1;
typeof(int *) a;
int f();
typeof(f()) i;
```

于是就有了

```
#define max(x, y) ({        \
```

```
  typeof(x) _max1 = (x);      \
  typeof(y) _max2 = (y);      \
  _max1 > _max2 ? _max1 : _max2; })
```

3.4.3真正的精髓

对比一下，内核的写法：

```
#define max(x, y) ({        \
  typeof(x) _max1 = (x);      \
  typeof(y) _max2 = (y);      \
  (void) (&_max1 == &_max2);    \
  _max1 > _max2 ? _max1 : _max2; })
```

发现比我们的还多了一句

```
(void) (&_max1 == &_max2);
```

这才是真正的精髓，对于不同类型的指针比较，编译器会给一个警告：

```
warning: comparison of distinct pointer types lacks a cast
```

提示两种数据类型不同。

至于加void是因为当两个值比较，比较的结果没有用到，有些编译器可能会给出一个警告，加(void)后，就可以消除这个警告。

4.通过成员获取结构体地址的宏container_of

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
#define container_of(ptr, type, member) ({      \
  const typeof(((type *)0)->member) *__mptr = (ptr);  \
  (type *)((char *)__mptr - offsetof(type, member));  \
})
```

4.1作用

我们传给某个函数的参数是某个结构体的成员，但是在函数中要用到此结构体的其它成员变量，这时就需要使用这个宏：container_of(ptr, type, member)

ptr为已知结构体成员的指针，type为结构体名字，member为已知成员名字，例子：

```
struct struct_a{
  int a;
  int b;
};

int fun1 (int *pa)
{
  struct struct_a *ps_a;
  ps_a = container_of(pa,struct struct_a,a);
  ps_a->b = 8;
}

int main(void)
{
  float f = 10;
  struct struct_a s_a ={2,3};
  fun1(&s_a.a);
  printf("s_a.b = %d\n",s_a.b);
  return 0;
}
```

输出：s_a.b=8。

本例子中通过struct_a结构体中的a成员地址获取到了结构体地址，进而对结构体中的另一成员b进行了赋值。

4.2详解

首先来看：

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

这个是获取在结构体TYPE中，MEMBER成员的偏移位置。

定义一个结构体变量时，编译器会按照结构体中各个成员的顺序，在内存中分配一片连续的空间来存储。例子：

```
#include<stdio.h>
struct struct_a{
  int a;
  int b;
  int c;
};
int main(void)
{
  struct struct_a s_a ={2,3,6};
  printf("s_a   addr = %p\n",&s_a);
  printf("s_a.a addr = %p\n",&s_a.a);
  printf("s_a.b addr = %p\n",&s_a.b);
  printf("s_a.c addr = %p\n",&s_a.c);
  return 0;
}
```

输出

```
s_a   addr = 0x7fff2357896c
s_a.a addr = 0x7fff2357896c
s_a.b addr = 0x7fff23578970
s_a.c addr = 0x7fff23578974
```

结构体的地址也就是第一个成员的地址，每一个成员的地址可以看作是对首地址的**偏移**，上面例子中，a就是首地址偏移0，b就是首地址偏移4字节，c就是首地址偏移8字节。

我们知道C语言中指针的内容其实就是地址，我们也可以把某个地址强制转换为某种类型的指针，(TYPE *)0)即将地址0，通过强制类型转换，转换为一个指向结构体类型为 TYPE的常量指针。

&((TYPE *)0)->MEMBER自然就是MEMBER成员对首地址的偏移量了。

而(size_t)是内核定义的数据类型，在32位机上就是unsigned int，64位就是unsiged long int，就是强制转换为无符号整型数。

再来看：

```
#define container_of(ptr, type, member) ({      \
  const typeof(((type *)0)->member) *__mptr = (ptr);  \
  (type *)((char *)__mptr - offsetof(type, member));  \
})
```

第一句（其实这句才是精华）

```
const typeof(((type *)0)->member) *__mptr = (ptr);  \
```

typeof在前面讲过了，获取类型，这句作用是利用赋值来确保你传入的ptr指针和member成员是同一类型，不然就会出现警告。

第二句

```
  (type *)((char *)__mptr - offsetof(type, member));  \
```

有了前面的讲解，应该就很容易理解了，成员的地址减去偏移不就是首地址吗，为什么要加个(char *)强制类型转换？

因为offsetof(type, member)的结果是偏移的字节数，而指针运算，（char *）-1是减去一个字节，（int *）-1就是减去四个字节了。

最外面的 (type *)，即把这个值强制转换为结构体指针。

5.#与变参宏

5.1#和##

#运算符，可以把宏参数转换为字符串，例子

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

```
#define PSQR(x) printf("The square of " #x " is %d.\n",((x)*(x)))
int main(void)
{
    int y = 5;
    PSQR(y);
    PSQR(2 + 4);
    return 0;
}
```

输出：

```
The square of y is 25.
The square of 2 + 4 is 36.
```

##运算符,可以把两个参数组合成一个。例子：

```
#include <stdio.h>
#define PRINT_XN(n) printf("x" #n " = %d\n", x ## n);
int main(void)
{
    int x1 = 2;
    int x2 = 3;
    PRINT_XN(1);        // becomes printf("x1 = %d\n", x1);
    PRINT_XN(2);        // becomes printf("x2 = %d\n", x2);
    return 0;
}
```

该程序的输出如下：

```
x1 = 2
x2 = 3
```

5.2变参宏

我们都知道printf接受可变参数，C99后宏定义也可以使用可变参数。C99 标准新增加的一个 __VA_ARGS__ 预定义标识符来表示变参列表，例子：

```
#define DEBUG(...) printf(__VA_ARGS__)
int main(void)
{
  DEBUG("Hello %s\n","World! ");
  return 0;
}
```

但是这个在使用时，可能还有点问题比如这种写法：

```
#define DEBUG(fmt,...) printf(fmt,__VA_ARGS__)
int main(void)
{
  DEBUG("Hello World! ");
  return 0;
}
```

展开后

```
printf("Hello World! ",);
```

多了个逗号，编译无法通过，这时，只要在标识符 __VA_ARGS__ 前面加上宏连接符 ##，当变参列表非空时，## 的作用是连接 fmt，和变参列表宏正常使用；当变参列表为空时，## 会将固定参数 fmt 后面的逗号删除掉，这样宏也就可以正常使用了，即改成这样：

```
#define DEBUG(fmt,...) printf(fmt,##__VA_ARGS__)
```

除了这些，其实Linux内核中还有很多宏和函数写得非常精妙。Linux内核越看越有味道，看内核源码，很多时候都会不明所以，但看明白后又醍醐灌顶，又感慨人外有人！

往期精彩：

嵌入式Linux驱动学习-6.platform总线设备驱动模型

嵌入式Linux驱动学习-5.驱动的分层分离思想

嵌入式Linux学习经典书籍-学完你就是大佬

如何在Linux上优雅地写代码-Linux生存指南

Linux下C语言编程风格和规范

收录于合集 #linux内核 2

上一篇 · 浅谈面向对象设计思想，以及在Linux内核中的体现