结构体最后的长度为0或者1的数组

在Linux系统里，/usr/include/linux/if\_pppox.h里面有这样一个结构：  
  
struct pppoe\_tag {  
  
\_\_u16 tag\_type;  
  
\_\_u16 tag\_len;  
  
char tag\_data[0];  
  
} \_\_attribute ((packed));  
  
  
  
最后一个成员为可变长的数组，对于TLV（Type-Length-Value）形式的结构，或者其他需要变长度的结构体，用这种方式定义最好。使用起来非常方便，创建时，malloc一段结构体大小加上可变长数据长度的空间给它，可变长部分可按数组的方式访问，释放时，直接把整个结构体free掉就可以了。例子如下：  
  
struct pppoe\_tag \*sample\_tag;  
  
\_\_u16 sample\_tag\_len = 10;  
  
sample\_tag = (struct pppoe\_tag \*)malloc(sizeof(struct pppoe\_tag)+sizeof(char)\*sample\_tag\_len);  
  
sample\_tag->tag\_type = 0xffff;  
  
sample\_tag->tag\_len = sample\_tag\_len;  
  
sample\_tag->tag\_data[0]=....  
  
  
  
...  
  
释放时，  
  
free(sample\_tag)  
  
是否可以用 char \*tag\_data 代替呢？其实它和 char \*tag\_data 是有很大的区别，为了说明这个问题，我写了以下的程序：  
  
例：test\_size.c  
  
10  struct tag1  
  
20  {  
  
30      int a;  
  
40      int b;  
  
50  }\_\_attribute ((packed));  
  
60  
  
70  struct tag2  
  
80  {  
  
90      int a;  
  
100     int b;  
  
110     char \*c;  
  
120  }\_\_attribute ((packed));  
  
130  
  
140  struct tag3  
  
150  {  
  
160      int a;  
  
170      int b;  
  
180      char c[0];  
  
190  }\_\_attribute ((packed));  
  
200  
  
210  struct tag4  
  
220  {  
  
230      int a;  
  
240      int b;  
  
250      char c[1];  
  
260  }\_\_attribute ((packed));  
  
270  
  
280  int main()  
  
290  {  
  
300      struct tag2 l\_tag2;  
  
310      struct tag3 l\_tag3;  
  
320      struct tag4 l\_tag4;  
  
330  
  
340      memset(&l\_tag2,0,sizeof(struct tag2));  
  
350      memset(&l\_tag3,0,sizeof(struct tag3));  
  
360      memset(&l\_tag4,0,sizeof(struct tag4));  
  
370  
  
380      printf("size of tag1 = %d\n",sizeof(struct tag1));  
  
390      printf("size of tag2 = %d\n",sizeof(struct tag2));  
  
400      printf("size of tag3 = %d\n",sizeof(struct tag3));  
  
410  
  
420      printf("l\_tag2 = %p,&l\_tag2.c = %p,l\_tag2.c = %p\n",&l\_tag2,&l\_tag2.c,l\_tag2.c);  
  
430      printf("l\_tag3 = %p,l\_tag3.c = %p\n",&l\_tag3,l\_tag3.c);  
  
440      printf("l\_tag4 = %p,l\_tag4.c = %p\n",&l\_tag4,l\_tag4.c);  
  
450      exit(0);  
  
460  }  
  
  
  
\_\_attribute ((packed)) 是为了强制不进行字节对齐，这样比较容易说明问题。  
  
程序的运行结果如下：  
  
size of tag1 = 8  
  
size of tag2 = 12  
  
size of tag3 = 8  
  
size of tag4 = 9  
  
l\_tag2 = 0xbffffad0,&l\_tag2.c = 0xbffffad8,l\_tag2.c = (nil)  
  
l\_tag3 = 0xbffffac8,l\_tag3.c = 0xbffffad0  
  
l\_tag4 = 0xbffffabc,l\_tag4.c = 0xbffffac4  
  
  
  
从上面程序和运行结果可以看出：tag1本身包括两个位整数，所以占了8个字节的空间。tag2包括了两个位的整数，外加一个char \*的指针，所以占了12个字节。tag3才是真正看出char c[0]和char \*c的区别，char c[0]中的c并不是指针，是一个偏移量，这个偏移量指向的是a、b后面紧接着的空间，所以它其实并不占用任何空间。tag4更加补充说明了这一点。所以，上面的struct pppoe\_tag的最后一个成员如果用char \*tag\_data定义，除了会占用多个字节的指针变量外，用起来会比较不方便：方法一，创建时，可以首先为struct pppoe\_tag分配一块内存，再为tag\_data分配内存，这样在释放时，要首先释放tag\_data占用的内存，再释放pppoe\_tag占用的内存；方法二，创建时，直接为struct pppoe\_tag分配一块struct pppoe\_tag大小加上tag\_data的内存，从例一的行可以看出，tag\_data的内容要进行初始化，要让tag\_data指向 strct pppoe\_tag后面的内存。  
  
struct pppoe\_tag {  
  
\_\_u16 tag\_type;  
  
\_\_u16 tag\_len;  
  
char \*tag\_data;  
  
} \_\_attribute ((packed));  
  
  
  
struct pppoe\_tag \*sample\_tag;  
  
\_\_u16 sample\_tag\_len = 10;  
  
方法一：  
  
sample\_tag = (struct pppoe\_tag \*)malloc(sizeof(struct pppoe\_tag));  
  
sample\_tag->tag\_len = sample\_tag\_len;  
  
sample\_tag->tag\_data = malloc(sizeof(char)\*sample\_tag\_len);  
  
sample\_tag->tag\_data[0]=...  
  
释放时：  
  
free(sample\_tag->tag\_data);  
  
free(sample\_tag);  
  
  
  
方法二：  
  
sample\_tag = (struct pppoe\_tag \*)malloc(sizeof(struct pppoe\_tag)+sizeof(char)\*sample\_tag\_len);  
  
sample\_tag->tag\_len = sample\_tag\_len;  
  
sample\_tag->tag\_data = ((char \*)sample\_tag)+sizeof(struct pppoe\_tag);  
  
sample\_tag->tag\_data[0]=...  
  
释放时：  
  
free(sample\_tag);  
  
所以无论使用那种方法，都没有char tag\_data[0]这样的定义来得方便。  
  
  
  
讲了这么多，其实本质上涉及到的是一个C语言里面的数组和指针的区别问题。char a[1]里面的a和char \*b的b相同吗？《Programming Abstractions in C》（Roberts, E. S.，机械工业出版社，.6）页里面说：“arr is defined to be identical to &arr[0]”。也就是说，char a[1]里面的a实际是一个常量，等于&a[0]。而char \*b是有一个实实在在的指针变量b存在。所以，a=b是不允许的，而b=a是允许的。两种变量都支持下标式的访问，那么对于a[0]和b[0]本质上是否有区别？我们可以通过一个例子来说明。  
  
  
  
例二：  
  
10  #include <stdio.h>  
  
20  #include <stdlib.h>  
  
30  
  
40  int main()  
  
50  {  
  
60      char a[10];  
  
70      char \*b;  
  
80  
  
90      a[2]=0xfe;  
  
100     b[2]=0xfe;  
  
110     exit(0);  
  
120  }  
  
  
  
编译后，用objdump可以看到它的汇编：  
  
080483f0 <main>:  
  
80483f0:       55                      push   %ebp  
  
80483f1:       89 e5                   mov    %esp,%ebp  
  
80483f3:       83 ec 18                sub    $0x18,%esp  
  
80483f6:       c6 45 f6 fe             movb   $0xfe,0xfffffff6(%ebp)  
  
80483fa:       8b 45 f0                mov    0xfffffff0(%ebp),%eax  
  
80483fd:       83 c0 02                add    $0x2,%eax  
  
8048400:       c6 00 fe                movb   $0xfe,(%eax)  
  
8048403:       83 c4 f4                add    $0xfffffff4,%esp  
  
8048406:       6a 00                   push   $0x0  
  
8048408:       e8 f3 fe ff ff          call   8048300 <\_init+0x68>  
  
804840d:       83 c4 10                add    $0x10,%esp  
  
8048410:       c9                      leave  
  
8048411:       c3                      ret  
  
8048412:       8d b4 26 00 00 00 00    lea    0x0(%esi,1),%esi  
  
8048419:       8d bc 27 00 00 00 00    lea    0x0(%edi,1),%edi  
  
  
  
可以看出，a[2]＝xfe是直接寻址，直接将xfe写入&a[0]+2的地址，而b[2]=0xfe是间接寻址，先将b的内容（地址）拿出来，加，再xfe写入计算出来的地址。所以a[0]和b[0]本质上是不同的。但当数组作为参数时，和指针就没有区别了。  
  
int do1(char a[],int len);  
  
int do2(char \*a,int len);  
  
这两个函数中的a并无任何区别。都是实实在在存在的指针变量。  
  
  
顺便再说一下，对于struct pppoe\_tag的最后一个成员的定义是char tag\_data[0]，某些编译器不支持长度为0的数组的定义，在这种情况下，只能将它定义成char tag\_data[1]，使用方法相同