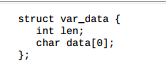
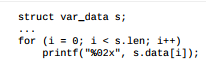
# Linux C内核编程

## GNU C 下的零长度下标数组和变量下标数组

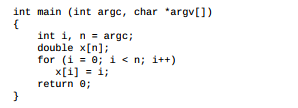
Linux上可用的C编译器是GNU C编译器，其对标准C进行了一系列的扩展。比如它允许使用零长度的数组，这个在定义变长对象的头结构应用时很有用：



char data[0]仅仅意味着程序中通过var\_data结构体实例的data[index]成员可以访问len之后的第index个地址，它并没有为data[]数组分配内存，因此sizeof（struct var\_data）=sizeof（int）。如果var\_data的数据就保存在该数据结构紧接着的内存区域中，则可以通过如下代码进行数据访问：



针对以下代码GNU C中是合法的，即支持变量作为数据的长度标示



## GNU C语句表达式解决标准C的宏产生的副作用

#define min(x,y) ((x) < (y) ? (x) : (y))

标准C库下定义的min宏，存在副作用，比如：代码min（++ia，++ib）会展开为（（++ia）<（++ib）?（++ia）：（++ib）），传入宏的“参数”增加两次。

改用如下定义方式可以避免出现两次增加的副作用。

**#define min\_t(type,x,y) \**

**(｛type \_ \_x =(x); type \_ \_y = (y); \_ \_x<\_ \_y ? \_ \_x : \_ \_y; } )**

int ia, ib, mini;

float fa, fb, minf;

mini = min\_t(int, ia, ib);

minf = min\_t(float, fa, fb);

根本原因是： GNU C把包含在括号中的复合语句看成是一个表达式，称为语句表达式，它可以出现在任何允许表达式的地方。我们可以在语句表达式中使用原本只能在复合语句中使用的循环、局部变量等

针对上面的宏定义还有一种GNU C编译器下的处理可以优化以实现min和原有的格式一样。即typeof ，它可以获取对象的类型

#define min(x,y) ({ \

const typeof(x) \_x = (x); \

const typeof(y) \_y = (y); \

(void) (&\_x == &\_y); \ // 注（1）

\_x < \_y ? \_x : \_y; })

注（1）：检查两者的数据类型是否一致。如果不一致GNU C编译器会产生如下告警。



## GNU C支持的可变参数宏

和标准C的可变参数函数类似，GNU C下可的宏可以接受可变数目的参数：



这里arg表示其余的参数，可以有零个或多个参数，这些参数以及参数之间的逗号构成arg的值，在宏扩展时替换arg，比如： pr\_debug("%s:%d",filename,line) ===》 printk("%s:%d", filename, line) 。使用“##”是为了处理arg不代表任何参数的情况，即 pr\_debug("success!\n") 需要使用 “##”将fmt后的逗号干掉

## GNU C下的特殊属性含义及声明

GNU C允许声明函数、变量和类型的特殊属性，以便手动优化代码和定制代码检查的方法。属性添加格式是在声明后面添加 \_\_attribute\_\_（（ATTRIBUTE））。其中ATTRIBUTE为属性说明，如果存在多个属性，则以逗号分隔。

|  |  |  |
| --- | --- | --- |
| 属性 | 含义 | 示例 |
| noreturn | 作用于函数，表示该函数从不返回。促使编译器优化代码，并消除不必要的警告信息 |  |
| format | 用于函数，表示该函数使用printf、scanf或strftime风格的参数，指定format属性可以让编译器根据格式串检查参数类型 | asmlinkage int printk(const char \* fmt, ...) \_\_attribute\_\_ ((format (printf, 1, 2)));  上述代码中的第1个参数是格式串，从第2个参数开始都会根据printf（）函数的格式串规则检查参数。 |
| unused | 作用于函数和变量，表示该函数或变量可能不会用到，这个属性可以避免编译器产生警告信息。 |  |
| aligned | 用于变量、结构体或联合体，指定变量、结构体或联合体的对齐方式，以字节为单位 | 表示该结构类型的变量以4字节对齐 |
| packed | 作用于变量和类型，用于变量或结构体成员时表示使用最小可能的对齐，用于枚举、结构体或联合体类型时表示该类型使用最小的内存 |  |
|  |  |  |
|  |  |  |

Asmlinkage 含义：<https://blog.csdn.net/qq84395064/article/details/86593469>

针对不同硬件微架构的形参存储位置对编译器进行提示

## GCC编译器编译常用选项含义

Gcc命令的常用格式为：

gcc [option1] [filename1] [option2] [filename2]

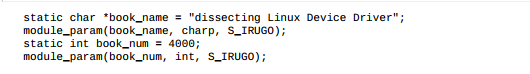
|  |  |  |
| --- | --- | --- |
| 分类 | 选项名称 | 功能 |
| 总体选项 | -c | 对源文件进行编译或汇编。 |
| -E | 对源文件进行预处理 |
| -S | 对源文件进行编译 |
| -o file | 指定输出目标文件名称 file |
| -v | 显示编译阶段的命令 |
| 语言选项 | -ansi | 支持符合ANSI标准的C语言程序 |
| 告警选项 | -W | 屏蔽所有的警告信息 |
| -Wall | 显示所有类型的警告信息 |
| -Werror | 出现任何警告信息就停止编译 |
| 调试选项 | -g | 产生调试信息 |
| 优化选项 | -O1 | 对目标文件的性能进行优化 |
| -O2 | 在-O1 的基础上进一步优化，提高目标文件的运行性能 |
| -O3 | 在-O2 的基础上进一步优化，支持函数集成优化 |
| -O0 | 不进行优化 |
| 连接器选项 | -static： | 使用静态链接。 |
| -l library | 链接 library 函数库文件 |
| -L dir | 指定连接器的搜索目录 dir |
| -shared | 生成共享文件 |
| 目录选项 | -I dir | 指定头文件的搜索目录 dir 。 |
|  | -Ldir | 指定搜索目录 dir |

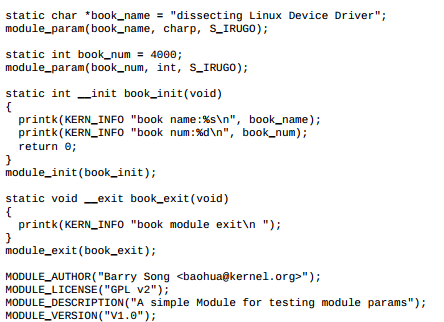
## 如何在模块加载的时候传递参数

模块参数是模块被加载的时候可以传递给它的值，它本身对应模块内部的全局变量。

参考：<https://www.cnblogs.com/mylinux/p/5670279.html>

可以传递单个的参数，也可以传递数组





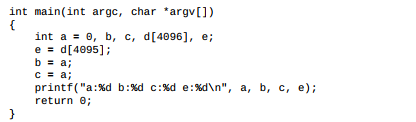
不传递参数值的时候就按照默认参数值。参数类型可以是byte、short、ushort、int、uint、long、ulong、charp（字符指针）、bool或invbool（布尔的反）

## 部分LINUX内核编程的标示含义

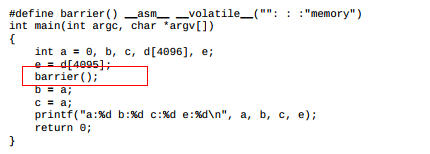
|  |  |  |
| --- | --- | --- |
| 符号 | 含义 | 备注 |
| \_\_init | 用于标示Linux内核模块加载函数。所有标识为\_\_init的函数如果直接编译进入内核，成为内核镜像的一部分，在连接的时候都会放在.init.text这个区段内，同时存放一份函数指针供系统的initcall调用 |  |
| \_\_exit | 标示Linux内核模块卸载函数 |  |
| \_\_initdata | 被定义为该类型的数据只是初始化阶段需要，内核在初始化完后可以释放它们占用的内存 |  |
| /proc/kallsyms | 对应内核符号表，记录了符号以及符号所在的内存地址 | 后者只适用于包含GPL许可权的模块 |
|  |  |  |
|  |  |  |
|  |  |  |

## 使用编译屏障处理编译乱序问题

编译乱序实际上并不是一种错误，而是编译器为了优化代码性能，减少逻辑上不必要的访存，以及尽量提高Cache命中率和CPU的Load/Store单元的工作效率，针对代码访存指令进行乱序。比如下面所示的代码：



在使用-O2的编译优化之后，可能 b=a的赋值语句会在 e = d[4095] 之前执行，而如果我们不想这样，一是可以取消编译优化选项；二是加入barrier()进行访存隔离



这样barrier()之前的代码就会保持原来的顺序。另外，volatile的含义是变量为易变的，它主要避免编译器进行优化时将内存访问行为进行合并。比如：线程A访问了var变量2次而没有修改它，那么优化时编译器认为第二次没有必要。而加了volatile限定修饰后说明，可能会有线程B/C针对var进行访问和修订，那么A针对var的第二次访问就不会被优化掉。

经典拓展案例：

volatile T\* pInstripe= 0;

T\* get\_instance() {

if (pInst == NULL) {

lock();

if (pInst == NULL) {

pInst = new T; //（1）

}

unlock();

}

return pInst;

}

（1）处包含了分配内存、在内存位置调用构造函数、将内存地址复制给pInst，如果后两个被乱序执行优化而调换了顺序，那么另外一个线程在执行到第一个if的时候就会发现指针不为空，但是构造函数初始化可能并没有完成，导致未初始化的数据泄露出去了

修订：

volatile T\* pInstripe= 0;

T\* get\_instance() {

if (pInst == NULL) {

lock();

if (pInst == NULL) {

T\* temp = new T;

***barrier();***

pInst = temp;

}

unlock();

}

return pInst;

}

## Linux内核编码中的关中断操作

参考资料：

<https://blog.csdn.net/u012247418/article/details/104180225>

注意它们均不能禁止内核所有的中断，只能禁止本处理器的中断；在确认中断没有被其它地方diable的时候可以使用下面的中断关闭方式

void local\_irq\_save(unsigned long flags);

void local\_irq\_restore(unsigned long flags);

void local\_irq\_enable(void);

void local\_irq\_disable(void);

// 禁止中断的底半部

local\_bh\_disable()

local\_bh\_enable()

## 可重入函数的特点

1. 不使用任何（局部）静态或全局的非const变量
2. 不返回任何（局部）静态或全局的非const变量的指针
3. 仅依赖于调用方提供的参数
4. 不依赖任何单个资源的锁（mutex等）
5. 不调用任何不可重入的函数

# C++编程

## 如何让 new 操作符只构造不申请内存——placement new

参考链接：<https://blog.csdn.net/weixin_44363885/article/details/93746137>

<https://blog.csdn.net/linuxheik/article/details/80449059>

new操作符：new operator，就是我们常用于申请内存、调用构造函数生成类对象并返回相应指针的过程

operator new 是重载函数（前2种不调用构造函数，这点区别于new operator）

void\* operator new (std::size\_t size) throw (std::bad\_alloc);

void\* operator new (std::size\_t size, const std::nothrow\_t& nothrow\_constant) throw();

void\* operator new (std::size\_t size, void\* ptr) throw();

第一种分配size个字节的存储空间，并将对象类型进行内存对齐。如果成功，返回一个非空的指针指向首地址。失败抛出bad\_alloc异常。

第二种在分配失败时不抛出异常，它返回一个NULL指针。

第三种是placement new版本，它本质上是对operator new的重载，定义于#include <new>中。它不分配内存，调用合适的构造函数在ptr所指的地方构造一个对象，之后返回实参指针ptr。

**第一、第二个版本可以被用户重载，定义自己的版本，第三种placement new不可重载**。

示例：

A\* a = new A; //调用第一种

A\* a = new(std::nothrow) A; //调用第二种

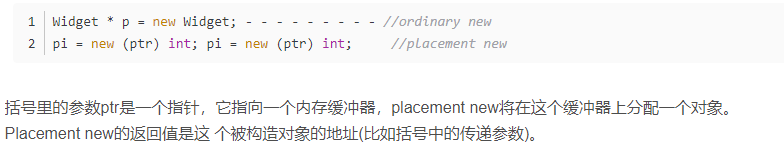
new (p)A(); //调用第三种operator new ，它在调用placement new之后，还会在p上调用A::A()，这里的p可以是堆中动态分配的内存，也可以是栈中缓冲。

**placement new**

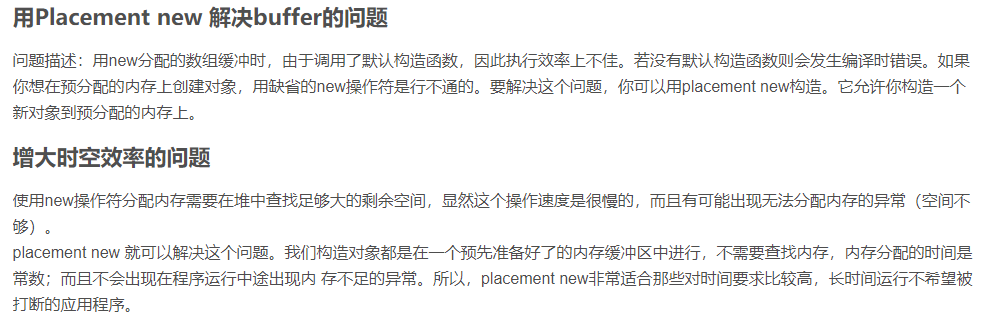
一般来说，使用new申请空间时，是从系统的“堆”（heap）中分配空间。申请所得的空间的位置是根据当时的内存的实际使用情况决定的。但是，在某些特殊情况下，可能需要在已分配的特定内存创建对象，这就是所谓的“定位放置new”（placement new）操作。 定位new操作的定义格式如下：



一般都用如下语句A\* p=new A;申请空间，而定位放置new操作则使用如下语句A\* p=new (ptr)A;申请空间，其中ptr就是程序员指定的内存首地址。第一个参数可以不传递由编译器根据



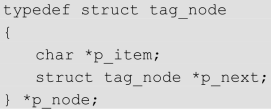
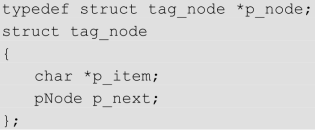
该函数的主要应用

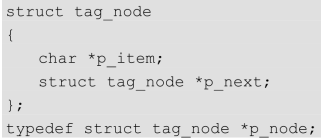


# C语言编程细节

## typedef定义结构体的问题

使用typedef定义结构体时，可能会遇到需要定义指向自己的指针的成员，以下三种都算是正确的做法，但是一般规范性或者推荐性的做法是第三种。



## typedef与const共同使用的问题

*typedef char\* cs;*

*const cs ptr1 = "pointer";*

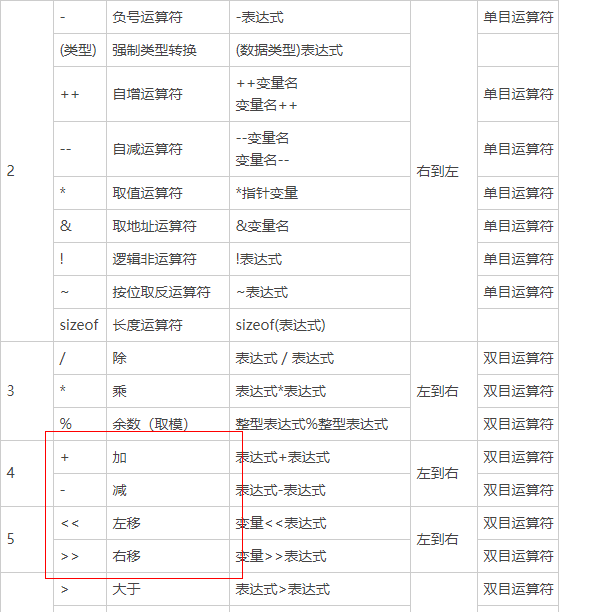
*++ptr1; /// error*

*const char\* ptr2 = "pointer";*

*++ptr2; /// runs fine*

定义中 cs已经成为了一个数据类型（char指针类型），使用const进行修饰之后，常量应该是ptr1，即这个指针类型而不是cs中的char。所以对ptr1进行自增是试图修改指针常量的值，是错误的。在这里typedef并不像是#define那样进行简单的宏展开。参考解释：https://qa.1r1g.com/sf/ask/830547511/

## 易错点：左右移位操作符的优先级

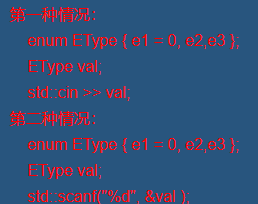


注意，左右移位的优先级是比加减乘除低的，所以下面的表达式可能得不到预期结果：

IGR\_PORT\_PAUSE\_EN\_BASE + port << IGRS\_PORT\_FIELD\_OFFSET; // 偏移会将前面加法结果给左移了

## 枚举变量的两种错误用法

枚举类型在C编译器下和整型变量有很多的一致性和可互换性，但是在某些场景下也不能使用枚举类型来代替整型类型



第一种情况会因为std::cin没有定义对应的枚举类型的重载>>运算符而出错，这就说明枚举类型是一种独立和鉴别的类型；第二种情况不会在编译时出现问题，枚举类型变量的尺寸不一定和int类型相同，这样一来我们采用%d就是说将枚举类型变量val当作4字节的int变量来看待并进行参数压栈。而在某些编译器下**sizeof( val )等于1字节**，这样scanf函数就会将val变量地址中的后续的三字节地址也压入栈中，并对其进行赋值，也许val变量后续的三个字节的地址没有特殊含义可以被改写（比如是字节对齐的空地址空间），可能会认为他不会出现错误，其实不然，在scanf函数调用结束后会进行栈清理，这样一来会导致scanf函数清理了过多的地址空间，从而破坏了外围函数的栈指针的指向，从而必然会导致程序运行时错误。

# Makefile

参考文档：<https://wiki.ubuntu.org.cn/%E8%B7%9F%E6%88%91%E4%B8%80%E8%B5%B7%E5%86%99Makefile>