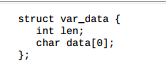
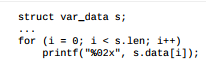
# Linux C内核编程

## GNU C 下的零长度下标数组和变量下标数组

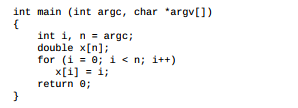
Linux上可用的C编译器是GNU C编译器，其对标准C进行了一系列的扩展。比如它允许使用零长度的数组，这个在定义变长对象的头结构应用时很有用：



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。



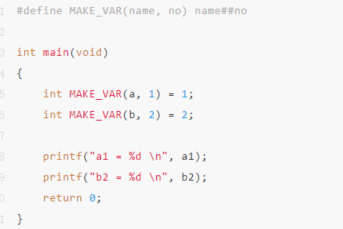
## GNU C支持的可变参数宏

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，如果不加##，那么会导致编译错误，宏解析出来会在fmt后多一个逗号。

“##”原本是用于把宏中的参数按照字符进行拼接，从而得到一个新的标识符，比如：



其实关于宏的可变参数，最简洁的是使用“...”代表可变参数，对应宏定义中使用“\_\_VA\_ARGS\_\_”来表示。



而像上面在“...”之前加上参数名时，就需要在定义中使用这个名字。同时最好加上“##”防止出现0参数的情况。

## 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]

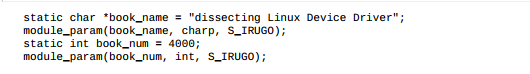
|  |  |  |
| --- | --- | --- |
| 分类 | 选项名称 | 功能 |
| 总体选项 | -c | 对源文件进行编译或汇编。 |
| -E | 对源文件进行预处理 |
| -S | 对源文件进行编译 |
| -o file | 指定输出目标文件名称 file |
| -v | 显示编译阶段的命令 |
| 语言选项 | -ansi | 支持符合ANSI标准的C语言程序 |
| 告警选项 | -W | 屏蔽所有的警告信息 |
| -Wall | 显示所有类型的警告信息 |
| -Werror | 出现任何警告信息就停止编译 |
| 调试选项 | -g | 产生调试信息 |
| 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 |

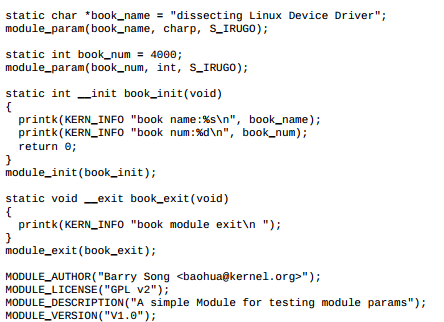
## 如何在模块加载的时候传递参数

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





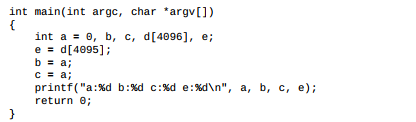
不传递参数值的时候就按照默认参数值。参数类型可以是byte、short、ushort、int、uint、long、ulong、charp（字符指针）、bool或invbool（布尔的反）

## 部分LINUX内核编程的标示含义

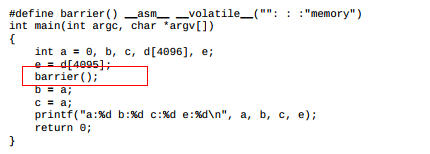
|  |  |  |
| --- | --- | --- |
| 符号 | 含义 | 备注 |
| \_\_init | 用于标示Linux内核模块加载函数。所有标识为\_\_init的函数如果直接编译进入内核，成为内核镜像的一部分，在连接的时候都会放在.init.text这个区段内，同时存放一份函数指针供系统的initcall调用 |  |
| \_\_exit | 标示Linux内核模块卸载函数 |  |
| \_\_initdata | 被定义为该类型的数据只是初始化阶段需要，内核在初始化完后可以释放它们占用的内存 |  |
| /proc/kallsyms | 对应内核符号表，记录了符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

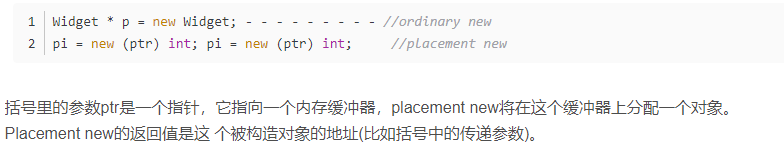
new (p)A(); //调用第三种operator new ，它在调用placement new之后，还会在p上调用A::A()，这里的p可以是堆中动态分配的内存，也可以是栈中缓冲。

**placement new**

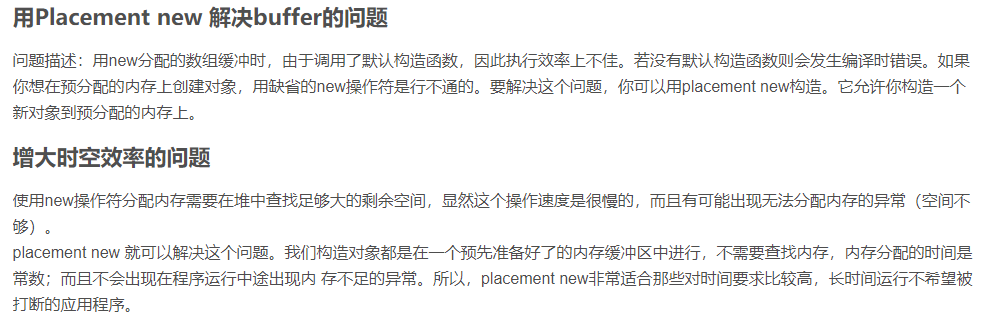
一般来说，使用new申请空间时，是从系统的“堆”（heap）中分配空间。申请所得的空间的位置是根据当时的内存的实际使用情况决定的。但是，在某些特殊情况下，可能需要在已分配的特定内存创建对象，这就是所谓的“定位放置new”（placement new）操作。 定位new操作的定义格式如下：



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



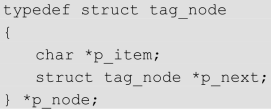
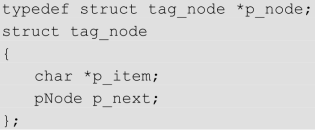
该函数的主要应用

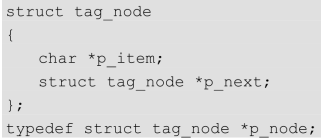


# C语言编程知识点

## 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";*

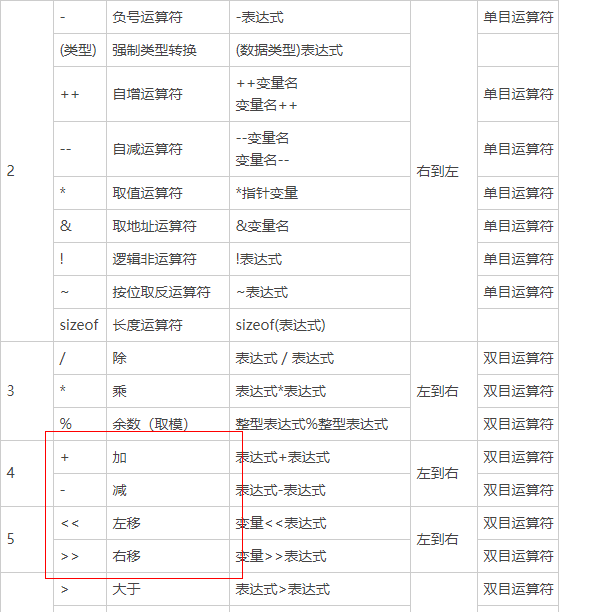
*++ptr1; /// error*

*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

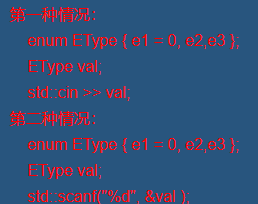


注意，左右移位的优先级是比加减乘除低的，所以下面的表达式可能得不到预期结果：

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

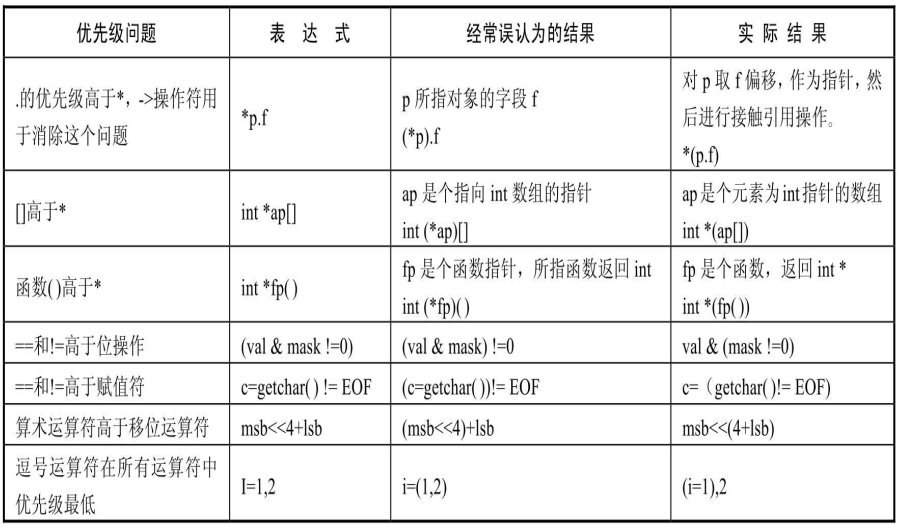
## 枚举变量的两种错误用法

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。

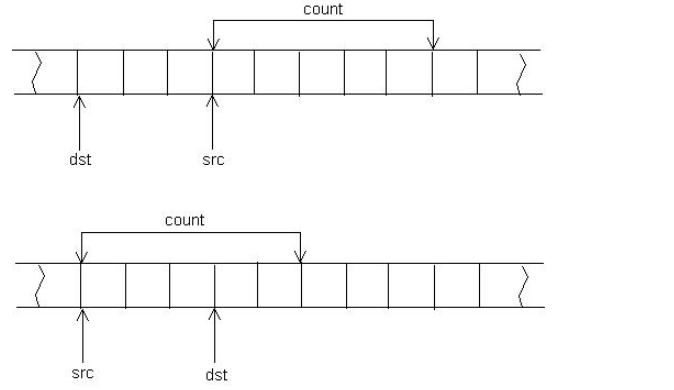
## 运算符优先级易错示例



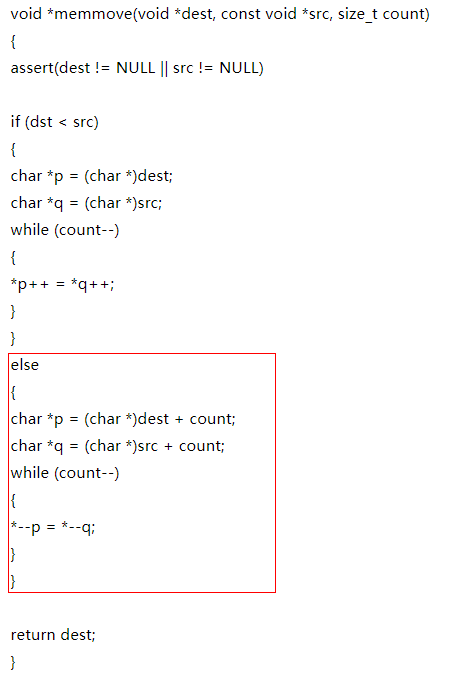
## Memcpy 与 memmove的区别

void \*memcpy(void \*dst, const void \*src, size\_t count);

void \*memmove(void \*dst, const void \*src, size\_t count);

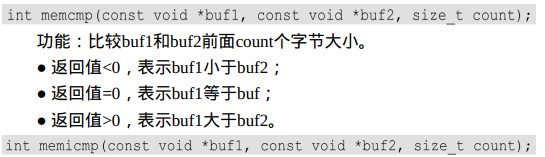


两种都是进行内存拷贝的C库接口，但是move函数在针对上述第二种情况的时候做了地址重叠的保护，



针对这种情况采用了最后一个字节开始COPY的方法解决，所以实际使用的时候memmove应该是比memcopy更安全的。其实在实际编码中本来不应该出现这种源和目的COPY内容地址重叠的情况，这样增加了运行时代码的复杂性，也使得这部分代码不具备安全性。

## Memcmp 和 memicmp 的区别



返回结果都一样，只不过后者不区分大小写

## C库中一些有用的冷门string处理函数



strrev将字符串string中的字符顺序颠倒过来，NULL结束符位置不变。返回调整后的字符串的指针。

\_strupr、strupr将string中所有小写字母替换成相应的大写字母，其他字符保持不变。返回调整后的字符串的指针。

\_strlwr、strlwrr将string中所有大写字母替换成相应的小写字母，其他字符保持不变。返回调整后的字符串的指针



Strchr 查找字符c在字符串string中首次出现的位置，NULL结束符也包含在查找中。返回一个指针，指向字符c在字符串string中首次出现的位置，如果没有找到，则返回NULL。



Srerchr 查找字符c在字符串string中最后一次出现的位置，也就是对string进行反序搜索，包含NULL结束符。返回一个指针，指向字符c在字符串string中最后一次出现的位置，如果没有找到，则返回NULL。



Strstr 在字符串string中查找strSearch子串。返回子串strSearch在string中首次出现位置的指针，如果没有找到子串strSearch，则返回NULL；如果子串strSearch为空串，函数返回string值。



Strncat 将源串strSource开始的count个字符添加到目标串strDest后，源串strSource的字符会覆盖目标串strDestination后面的结束符NULL。如果count大于源串长度，则会用源串的长度值替换count值，得到的新串后面会自动加上NULL结束符。与strcat函数一样，本函数不能处理源串与目标串重叠的情况，函数返回strDestination值。**也没有溢出检查，所以存在一定的不安全性**。



Strncpy 将源串 strSource 开始的 count 个字符复制到目标串strDestination所指定的位置，**如果count值小于或等于strSource串的长度**，**不会自动添加NULL结束符目标串中（所以使用该API需要注意末尾加‘\0’）**，而count大于strSource串的长度时，则将strSource用NULL结束符填充补齐count个字符，复制到目标串中，不能处理源串与目标串重叠的情况，函数返回strDestination值





Strset 将string串的所有字符设置为字符c，遇到NULL结束符停止。函数返回内容调整后的string指针。

Strnset 将string串开始count个字符设置为字符c，如果count值大于string串的长度，将用string的长度替换count值。函数返回内容调整后的string指针。



Strspn 查找任何一个不包含在strCharSet串中的字符（字符串结束符NULL除外）在string串中首次出现的位置序号。返回一个整数值，指定在string中全部由characters中的字符组成的子串的长度；如果string以一个不包含在strCharSet中的字符开头，函数将返回0值。

参考链接：https://www.runoob.com/cprogramming/c-function-strspn.html



Strcspn 查找strCharSet串中任何一个字符在string串中首次出现的位置序号，包含字符串结束符NULL。返回一个整数值，指定在string中全部由非characters中的字符组成的子串的长度，如果string以一个包含在strCharSet中的字符开头，函数将返回0值。



Strspnp查找任何一个不包含在strCharSet串中的字符（字符串结束符NULL除外）在string串中首次出现的位置指针。返回一个指针，指向非strCharSet中的字符在string中首次出现的位置。



Strpbrk查找strCharSet串中任何一个字符在string串中首次出现的位置，不包含字符串结束符NULL。返回一个指针，指向strCharSet中任一字符在string中首次出现的位置，如果两个字符串参数不含相同字符，则返回NULL值。



Strtok在strToken串中查找下一个标记，strDelimit字符集则指定了在当前查找调用中可能遇到的分界符。返回一个指针，指向在strToken中找到的下一个标记。如果找不到标记，就返回NULL值。每次调用都会修改strToken内容，用NULL字符替换遇到的每个分界符。

在第一次调用时，strtok()必需给予参数strToken字符串值，往后的调用则将其设置成NULL即可，当返回值变为NULL时说明已经到了末尾，我推测是初始的字符串被存储在了strtok中

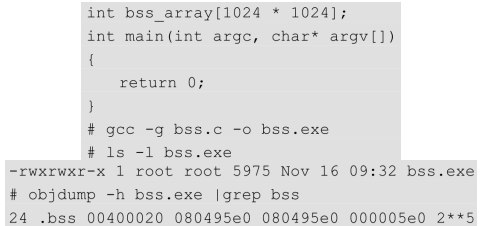


<http://c.biancheng.net/cpp/html/175.html>

## 内存管理

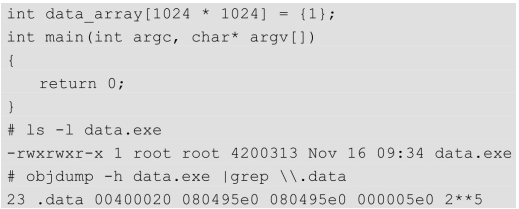
### 内存数据段含义

**.bss段**——用来存放那些没有初始化和初始化为0的全局变量



注意其中使用的gcc编译方法和objdump命令（objdump命令是Linux下的反汇编目标文件或者可执行文件的命令，它以一种可阅读的格式让你更多地了解二进制文件可能带有的附加信息。——http://blog.chinaunix.net/uid-15007890-id-3788056.html）使用方法

**.data段** ——用来存放那些初始化为非零的全局变量。如果数据全是零，为了优化考虑，编译器把它当作bss来处理。



注意：从可执行文件大小可以看出初始化之后的全局数据既要占用运行空间又要占用文件空间，且程序运行的整个期间data数据一直存在。**所以，我们在定义全局数据时最好定义时不要初始化而是等到使用时或者运行初始化时再给其赋值**。

**.rodata段** —— ro代表read only，rodata就是用来存放常量数据的。

* 常量不一定就放在 rodata 里，有的立即数直接和指令编码在一起，存放在代码段（.text）中。
* 对于字符串常量，编译器会自动去掉重复的字符串，保证一个字符串在一个可执行文件（EXE/SO）中只存在一份复制。
* rodata是在多个进程间是共享的，这样可以提高运行空间利用率。
* 在有的嵌入式系统中，rodata放在ROM（或者NOR Flash）里，运行时直接读取，无须加载到RAM内存中。
* 在嵌入式Linux系统中，也可以通过一种叫作XIP（就地执行）的技术，也可以直接读取，而无须加载到RAM内存中。
* 常量是不能修改的，修改常量在Linux下会出现段错误。

**.text段** —— 代码段，存放代码（如函数）和部分整数常量，其余rodata相似，主要不同在于这个段是可执行的

**栈（stack）**—— 用于存放临时变量和函数参数，另一个主要应用则是递归（ 几乎必用）。通常情况下，栈向下（低地址）增长，每向栈中PUSH一个元素，栈顶就向低地址扩展，每从栈中POP一个元素，栈顶就向高地址回退

**堆（heap）**—— 一种生命周期完全由使用者控制的内存。为防止内存泄漏和溢出：Linux下有valgrind，它的使用方法很简单，大家可以试用一下，以后每次写完程序都应该用valgrind跑一遍。

valgrind参考资料：<https://www.jianshu.com/p/5a31d9aa1be2> <https://www.cnblogs.com/AndyStudy/p/6409287.html>

### 野指针

指向“垃圾”内存的指针。产生原因：指针变量没有被初始化；指针被释放或delete之后没有被赋值为NULL。

### 涉及内存编码的一些注意事项

1. 内存分配时未判定返回值以确保正确地分配地址空间（malloc未判定返回值）
2. 内存分配之后未进行初始化处理就使用或者使用时出现引用越界的情况
3. 分配和释放内存未能配对导致内存泄漏或重复释放
4. 释放了内存但仍然使用“野指针”

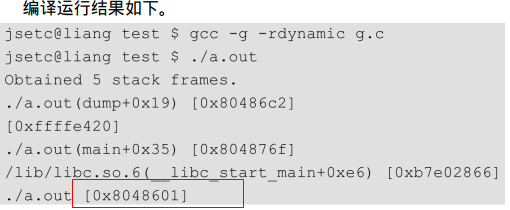
### 段错误调试的基本方法

产生段错误就是访问了错误的内存段，一般是你没有权限，或者根本就不存在对应的物理内存，尤其常见的是访问0地址（程序访问了访问的内存超出了系统所给这个程序的内存空间）。

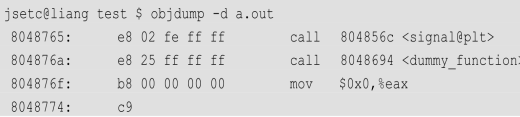
1. 使用GDB进行调试

程序编译的时候需要使用 -g –rdynamic 选项。然后再针对输出文件执行gdb命令或者在运行环境执行gdb（这个实施细节可能还需要探究），如果只是gcc进行构建的话，直接使用gdb + a.out 就可以进行调试了。

1. 利用backtrace 和objdump进行分析



如上图所示执行到这个地址挂掉了，那么使用objdump 进行查看时哪个函数引发的异常



## 弱符号\_\_attribute\_\_((weak))的含义和用法

若两个或两个以上全局符号（函数或变量名）名字一样，而其中之一声明为weak symbol（弱符号），则这些全局符号不会引发重定义错误。链接器会忽略弱符号，去使用普通的全局符号来解析所有对这些符号的引用，但当普通的全局符号不可用时，链接器会使用弱符号。当有函数或变量名可能被用户覆盖时，该函数或变量名可以声明为一个弱符号。弱符号也称为weak alias（弱别名）。

主要应用就是在SDK中定义一个默认实现，防止构建的中断也为外部定义该函数逻辑实现提供便利性，其实这个有点类似C++的父类和子类的多样性实现。

参考资料：

<https://blog.csdn.net/sea_snow/article/details/83650519>

<https://www.cnblogs.com/linhaostudy/p/9237357.html>

\_\_weak 和 \_\_attribute\_\_((weak)) 关键字的使用

<https://itexp.blog.csdn.net/article/details/106816700>

## 利用宏实现动态调用不同的函数



# Makefile

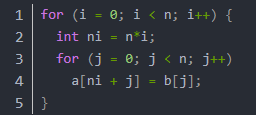
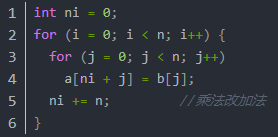
参考文档：<https://wiki.ubuntu.org.cn/%E8%B7%9F%E6%88%91%E4%B8%80%E8%B5%B7%E5%86%99Makefile>

# Clean code

参考： <https://blog.csdn.net/qq_16933601/article/details/111639650>

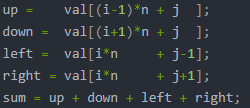
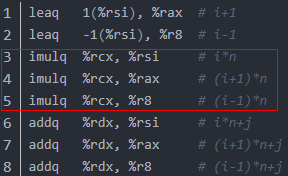
## 减小程序计算量

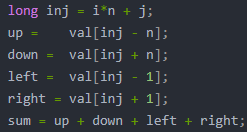
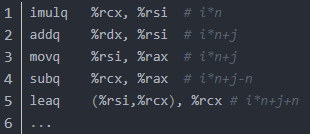
以加法替代乘法、以位移替代乘法

## 提取代码中的公共部分

最常见的是多个函数中有重复的逻辑可以单独提取出来进行抽象，还有一些是不容易发现的，比如下面的例子：

 汇编结果

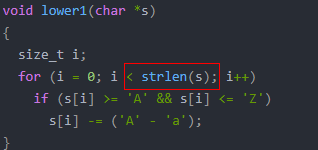
 汇编结果

提取一个公共的中间运算结果防止编译后产生不必要的指令，当然有的编译器根据优化等级是可以自动完成以上优化处理的

## 消除循环中的低效代码

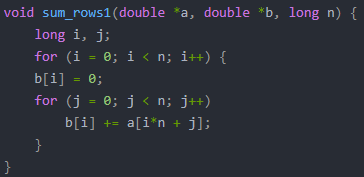
1. 有些计算或者表达式不需要有自变量或者不需要随循环索引变化的最好就在循环体外执行，避免影响

循环执行的效率

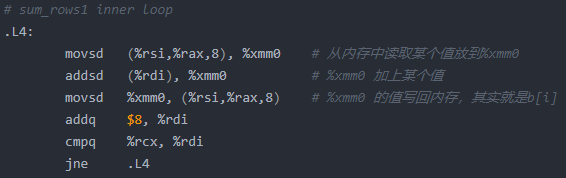


在字符串长度大于100000时，程序运行时间大涨，这个和strlen计算本身有关，其时间复杂度为O(N)，每次都调用相当于是O(N2)的复杂度

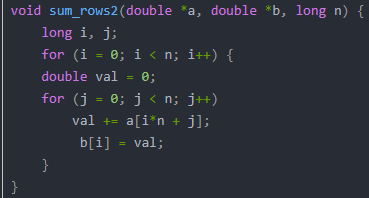
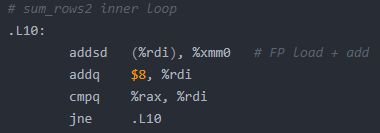
1. 有些循环中存在不必要的内存引用，比如以下示例



汇编结果

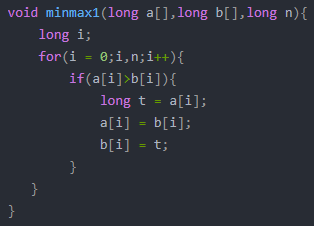
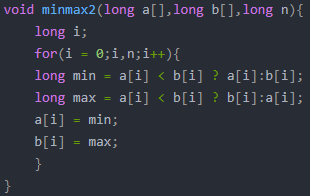


内循环每次都会去读取b[i]的值，然后再把b[i]写回内存 ，改进建议就是设置临时变量进行计算然后再赋值

1. 循环体中使用条件传送风格的代码以替换if else操作

现代处理器的流水线性能使得处理器的工作远远超前于当前正在执行的指令。处理器中的分支预测在遇到比较指令时会进行预测下一步跳转到哪里。如果预测错误，就要重新回到分支跳转的原地。分支预测错误会严重影响程序的执行效率。因此，我们应该编写让处理器预测准确率提高的代码，即使用条件传送指令。我们用条件操作来计算值，然后用这些值来更新程序状态

改进：

在原代码中，需要对a[i]和b[i]进行比较，再进行下一步操作，这样的后果是每次都要进行预测。改进后的代码实现这个函数是计算每个位置i的最大值和最小值，然后将这些值分别赋给a[i]和b[i]，而不是进行分支预测。