C语言高级用法

**作者：向仔州**

数据类型的特殊定义…………………………………………………………………………………..2

Void的使用问题………………………………………………………………………………………..9

extern 可以让C++编译器编译C代码的时候按照标准C语言来编译…………….11

const到底是不是常量……………………………………………………………………………….13

Volatile就是告诉编译器老老实实去内存取变量的值，不要搞什么优化…………..15

Struct结构体的一些bug…………………………………………………………………………...15

计算结构体里面成员的偏移量……………………………………………………………………..20

union问题……………………………………………………………………………………………….21

enum的使用方法……………………………………………………………………………………..22

引号和单引号的问题………………………………………………………………………………….23

#define的一些问题…………………………………………………………………………………..24

#if #else和if else的区别………………………………………………………………………….29

编译器编译C代码的时候打印出的warning，error，是怎么实现的？……………31

强制类型转换，转换的是数据还是地址？……………………………………………………..33.

指针使用注意事项………………………………………………………………………………………34

sizeof使用注意………………………………………………………………………………………….35

struct结构体高级用法………………………………………………………………………………..36

数组指针，用来操作二维数组………………………………………………………………………40

指针数组，数组里面全部放的是指针变量………………………………………………………41

函数指针…………………………………………………………………………………………………….43

typedef修饰结构体的好处…………………………………………………………………………..48

定义结构体指针…………………………………………………………………………………………..49

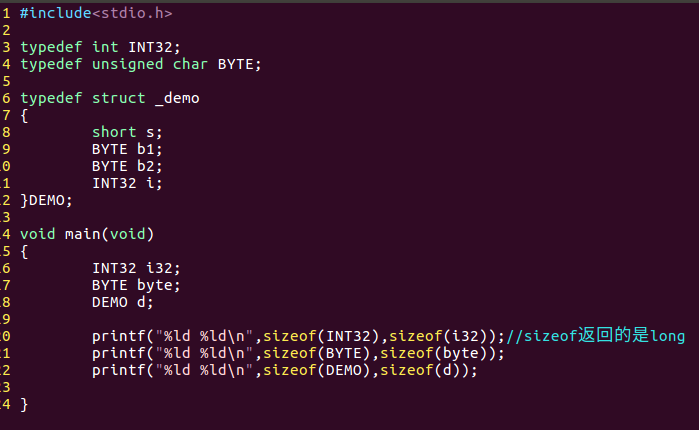
二级指针一般用来指向指针数组……………………………………………………………………50

C语言各种程序在内存分布的情况分析…………………………………………………………..52

函数递归调用....57 C语言字符串处理函数汇总............59

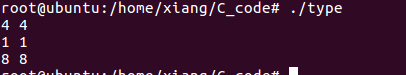
Linux系统下C文件制作静态库.........66 Linux系统下C文件制作动态库........67

**数据类型的特殊定义**



**这个INT32 就代表了int**

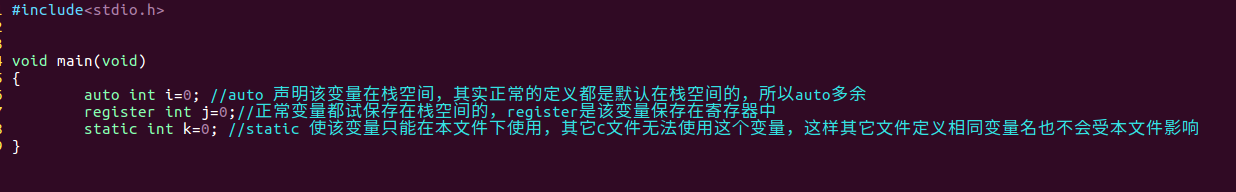
**typedef就是把int这种数据类型换一个好记的方式定义了一道**



**INT32也就是int 占用4个字节32位**

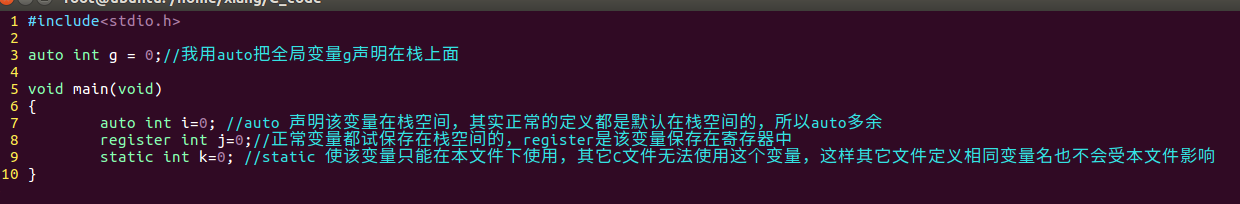
**BYTE 就是char 占用1个字节8位**

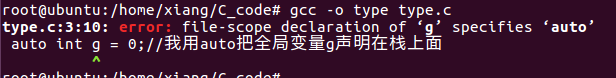
**结构体shart 是2字节 BYTE是1字节，INT32是4字节，也就是shart+BYTE+BYTE+INT32 = 8字节**



Register 定义变量要特别注意，就是你的系统是32位的，那么register最多定义int，不能定义long

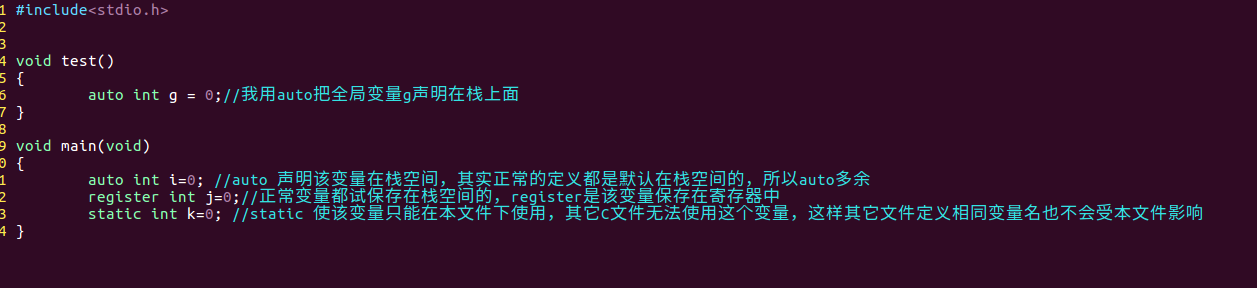
如果你的系统是64位的，那么register可以定义long。Register定义的变量必须符合系统位宽





**我发现用auto定义全局变量报错**

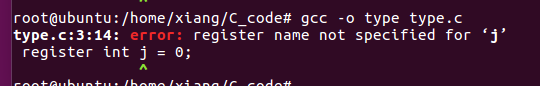
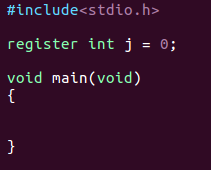
这是因为全局变量是放在全局区的，auto声明的变量是放在栈上面的，全局区没有栈这个东西，你让g怎么放在栈上面？



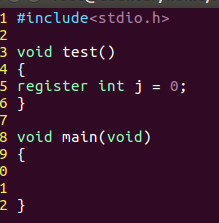
**因为子函数是申请在栈上面的，我在栈上用auto申请个变量是很正常的**

你看成功编译

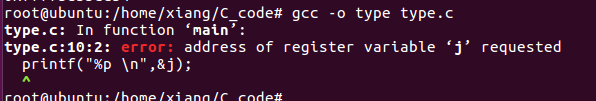
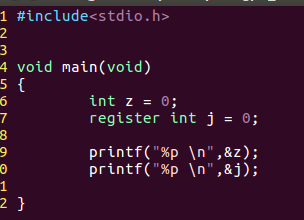
其实auto是不需要用的，系统会根据你的变量是在子函数还是在全局变量区域，来自动为你的变量进行分配，auto只是锦上添花多余。



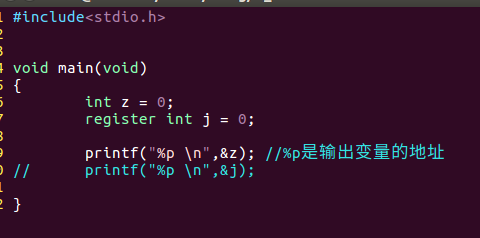
register变量不能放在全局区，寄存器变量只能放在栈里面



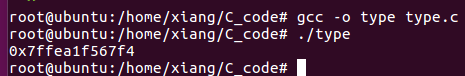
**register变量要比在栈上申请的变量读写速度快，如果我们的一个子函数要求运行很快，就可以在子函数里面多申请几个resigter变量，register申请变量一般用上实时系统上**



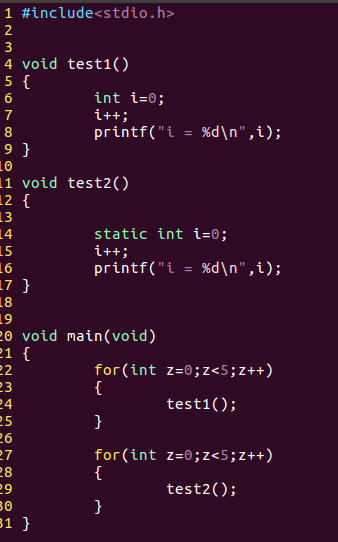
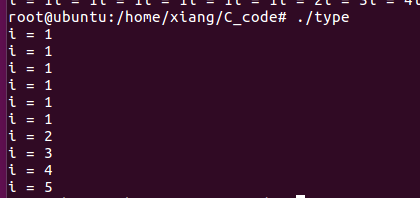
**register 寄存器地址是在CPU上面，不是在内存栈上面，在X86平台是不能获取地址的，在单片机里面可以，在linux平台下不清楚**



**Int直接定义的变量在内存栈上**



只有普通变量是可以访问地址的。因为普通变量在内存栈上。

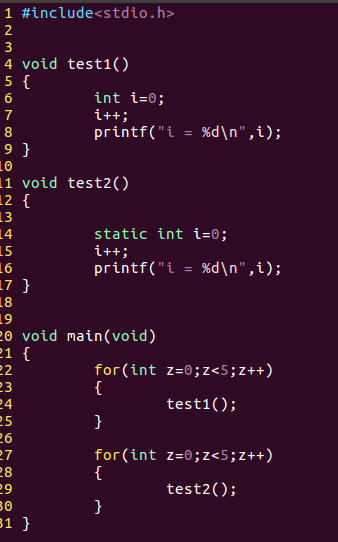
 

**而test2能累加到5**

**为什么test1每次输出都是1**

**Test2的输出**

**test1的输出**



**每次进入函数发现i是静态变量，所以没有对i进行再次清0，而是让i接着用**

**而static int i 用static将i放在了全局区，(注意static静态变量和全局变量都放在全局区只是功能不一样而已。一个是局部使用，一个是全局使用)**

**那是因为int i是放在栈空间里面的，每次函数执行完了都会给i清0**

虽然static定义的变量作用范围在子函数内部，但是子函数运行结束后，变量是没有被释放的，虽然你在主函数可以定义同一个i而且不会被子函数的i影响，但是子函数i占着内存这个事实是一直存在的，只是i的作用位置不同，但是i占着内存是没有释放的。

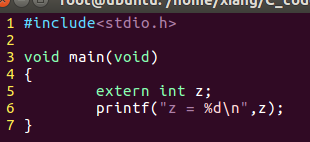
static还有一个非常重要关键的作用

我有两个C文件，type.c和type2.c

type2.c里面有个int 类型的变量

因为没有定义在函数里面，所以该变量是全局的

type.c文件去调用这个全局变量

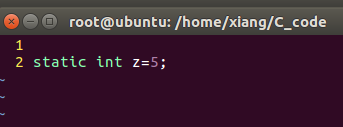
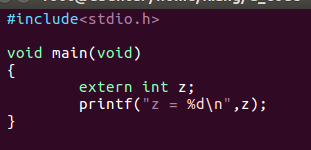


**这里用extern 才能声明其它C文件里面的全局变量**

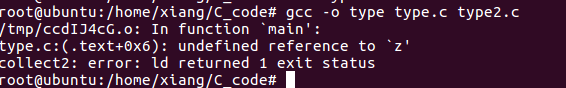
我们type.c里面输出的z值是type2.c的值。所以这就是全局变量的作用，它不仅在一个C文件里面全局，它在整个系统里面都是全局的，我根本不需要知道这个全局变量在哪个C文件，我只要知道有这个变量，然后用extern导入进我现在用的C文件，那么这个变量就可以给我用。所以为什么很多人说少用全局变量，就是因为这种全局变量任何文件都可以去篡改，很危险。

但是有个办法可以解决这个问题

type2.c type.c

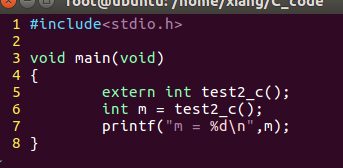
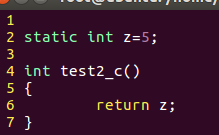
**在type2.c文件里面给全局变量加static**



你看编译就直接报错，说你的type.c文件用extern声明了一个不存在的全局变量。

但是我还是想使用这个全局变量怎么办？你可以用全局函数来承载这个文件内的全局变量

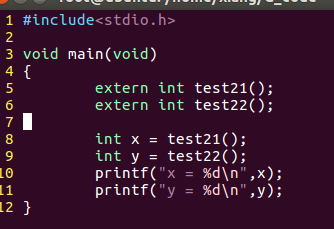
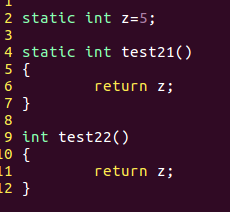
type.c type2.c

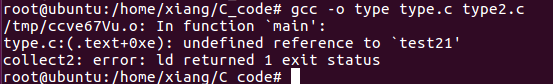
type.c文件成功调用了type2

文件里面的代码，从这里可以看出来函数和变量是一样的可以被局部使用，也可全局使用。

所以以后写程序如果函数不是被其它文件需要调用，就尽量在函数前加static

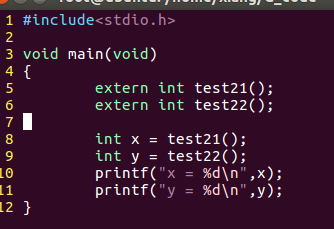
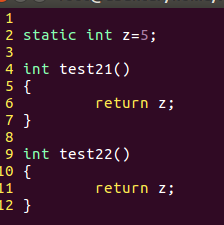
 

**在子函数前面加static，这样该函数的作用范围就是本文件**



你看其他文件调用test21()函数就会编译报错，和前面我们讲变量作用范围一个意思

type.c type2.c

**加了static的变量虽然不会成为整个系统的全局变量，但是可以成为本文件里面所有函数的全局变量**

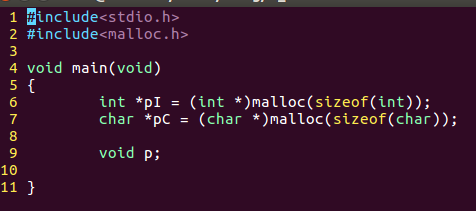


至于extern和include有什么区别呢？

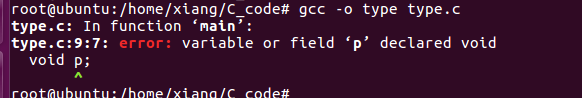
extern是提供本文件调用其它C文件函数的通道，比如我要调用test21() 我就在本文件写extern test21() ，这样本文件才能调用其它文件的test21()函数

include 里面的函数全部是声明，不具备extern的功能，你发现刚才文件之间没有include一样的可以相互调用，所以include只是为了让用代码的人知道用了哪些文件里面的函数。没有什么实际的意义。

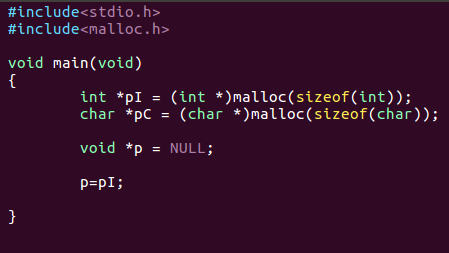
**Void的使用问题**



**C语言不允许有void类型变量**



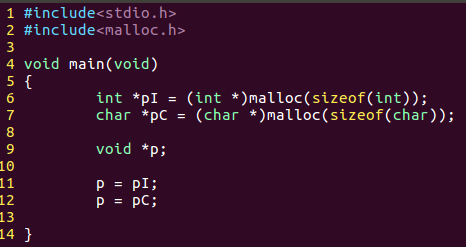
C语言中不允许有void类型的变量



**但是C语言运行有void类型的指针**

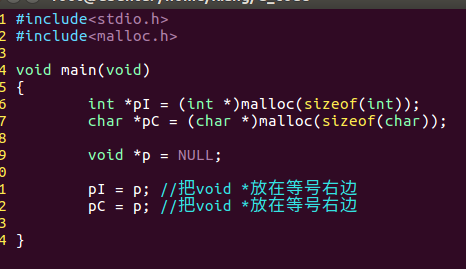
我们知道C语言规定只有相同类型的指针才可以相互赋值，比如int \*value = int \*temp 两边都是int指针，如果char \*value = int \*temp 这样就不行

但是void 指针可以接受任何类型的数据地址，比如char，int，long

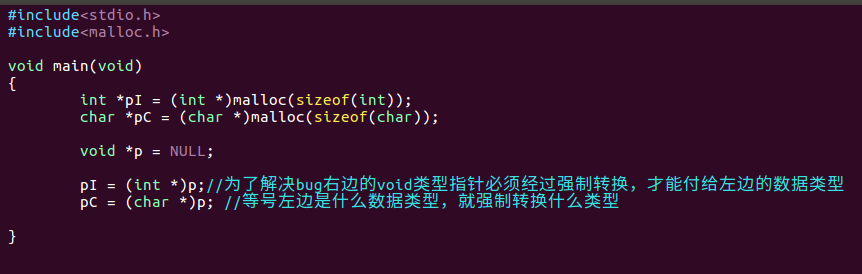
这样去操作p给\*p赋值是可以的。

**也可以char赋地址给p**

**你看 int赋地址给void p**



如果是这样void放在等号右边，虽然gcc编译器让你编译通过了，但是运行会出现bug

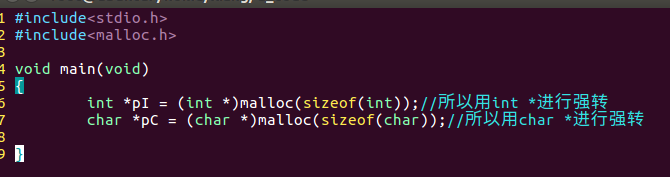


**(类型 \*)地址 = 强转**

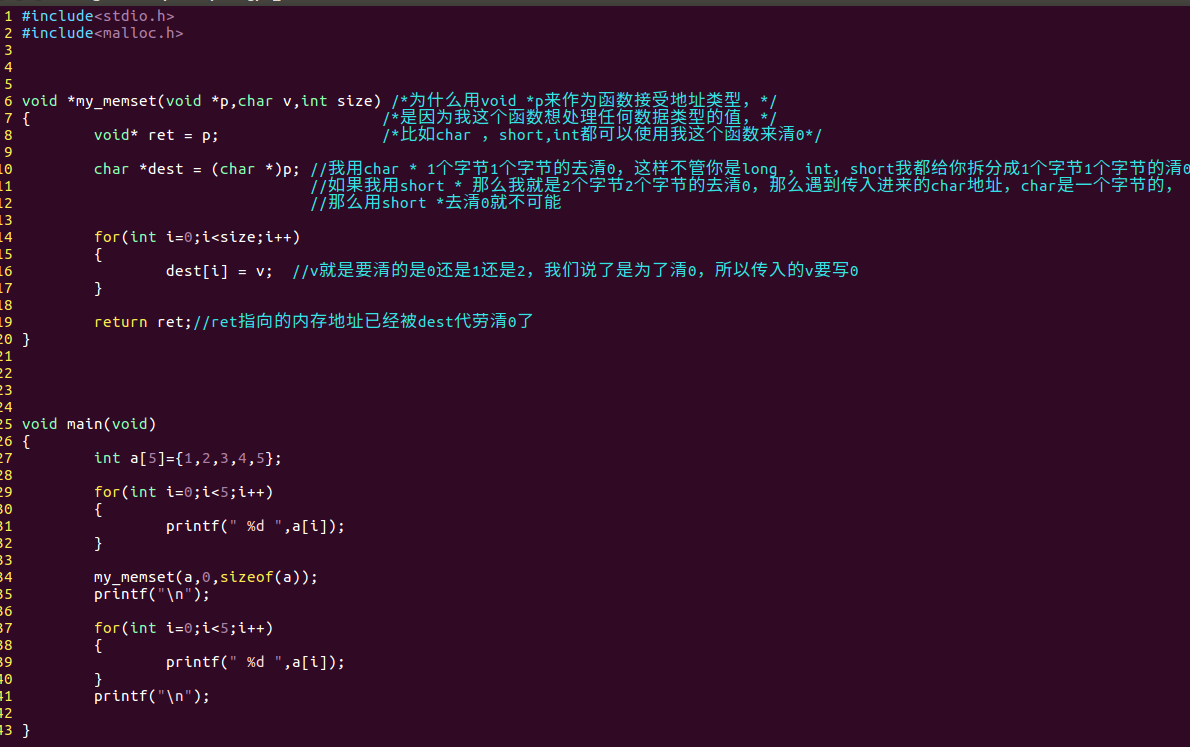


因为malloc原形是extern void \*malloc(unsigned int num\_bytes);

malloc返回值是void \* 也就和上面一样，void\*在等号右边



我们下面用void来做一个memset 内存清0的函数功能



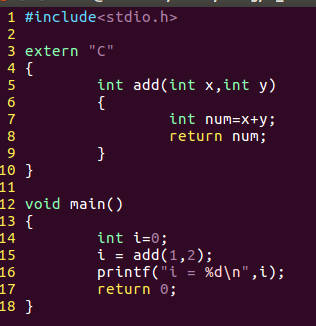
在linux驱动中void\* 就是两个C文件，A.c随便定义个结构体，传给B.c，B.c用void\* 来接收这个A.c传过来的数据，然后B.c再定义一个和A.c一样的结构体去装载void\*接收的数据，这样b.c就可以操作A.c传过来的数据了，这样的好处是A.c可以定义不同的变量和数据结构来传给B.c，比较灵活



这就是void的应用实例。

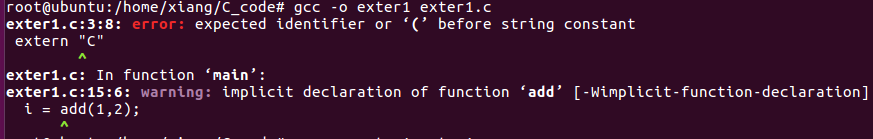
**extern 可以让C++编译器编译C代码的时候按照标准C语言来编译**

我们前面知道extern可以用来声明全局变量和全局函数，但是extern还有个用处就是在C++代码里面添加C语言代码时，它可以让C++编译器再编译C代码这部分按照标准C代码来编译，避免出现bug

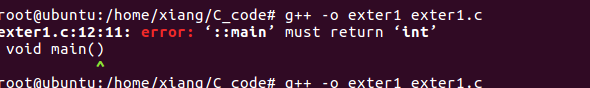


**这样用gcc编译是不行的，因为extern “C”{}**

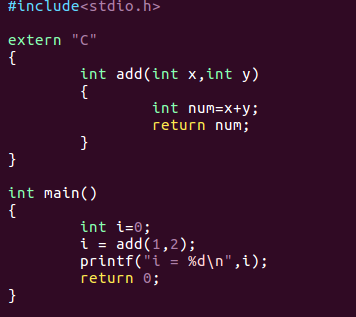
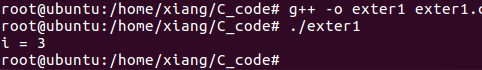
**这种结构是给C++编译器用的，在C++代码里面嵌入C代码用这种方式包裹**



你看gcc编译报错了，那么我们用g++编译器来编译



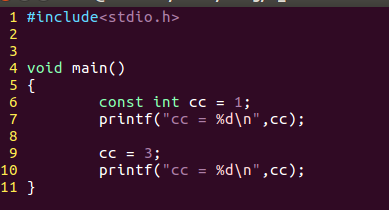
这个错误是因为C++代码不允许用void，void是一个不确定的数据类型，C++对这些东西做了严格要求。就是不准你用这种不确定的东西。

**将void改成int**

这样就编译过了，可以正常使用了。这就是extern的另一种用法，在C++里面用

**const到底是不是常量**

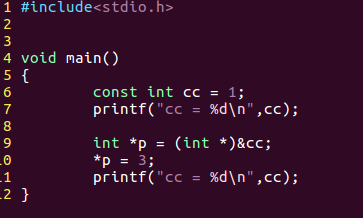


**发现报错，因为cc是常量所以是不能修改的**

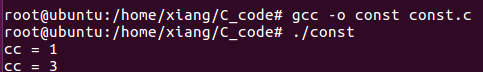
**我们来修改下常亮的值**



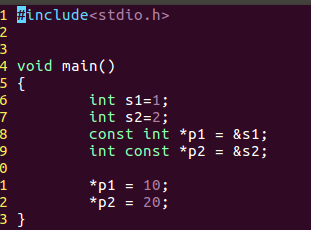
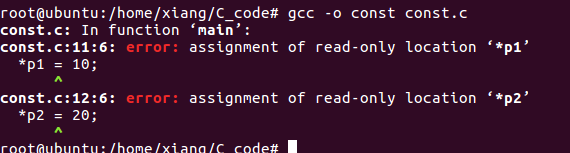
给变量加const就是告诉编译器变量是只读的



**我们发现直接用指针去操作cc的地址，用指针赋值给cc是可行的**

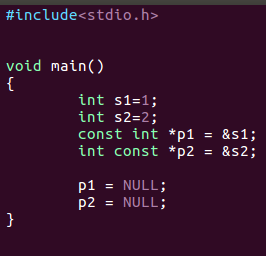


从以上我们看出来const对编译器有用对运行程序是没有用的，所以只要绕开编译器这个关卡，其实const变量的值还是可以被修改的，所以我们之前去const变量的地址上修改数值

**我们给const在\*左边的变量赋值**

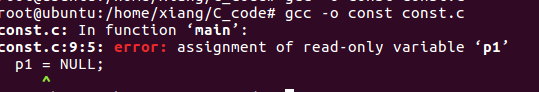
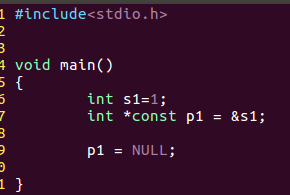
编译出现了错误，const指针口诀，左数右值，上面就是const在\*左边，就是左数

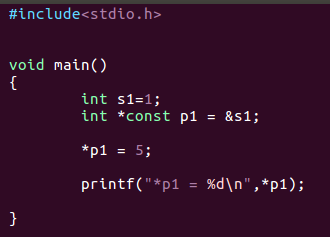
**左数就是地址上存放的数值不能改，但是地址本身是可以修改的**

你看我把p1，p2指向s1，s2的地址修改成NULL是没有问题的。

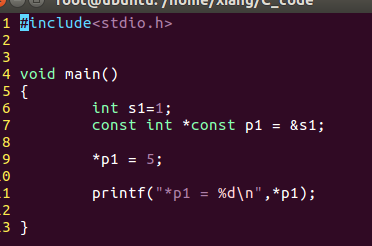
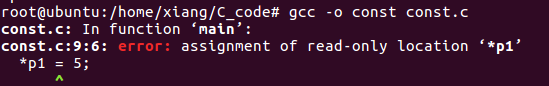
下面我们来看看右指是什么情况



**const在\*号右边，就是不能修改变量地址，但是可以修改变量值**



**const在\*号右边，你看我不能修改地址，但是可以修改变量值**

**如果一个变量\*左右都有const，那么这个变量地址和值都不能被改变**

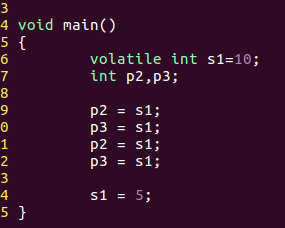
**Volatile就是告诉编译器老老实实去内存取变量的值，不要搞什么优化**



**但是哪晓得一句代码改变了S1的值，导致编译器自作聪明让S1的值始终为10，所以S1还是10**

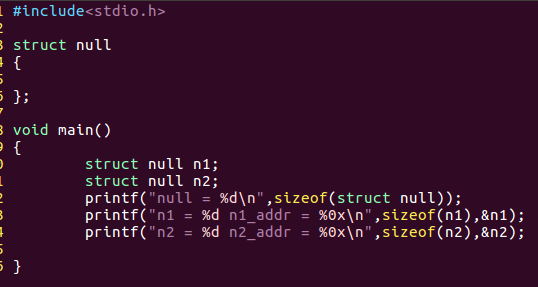
**这么一段代码，如果S1一直是在给其它变量赋值的话，编译器优化的时候会认为s1是个永远不变的值10**

这种情况有些编译器会优化成问题，有些编译器不会，看自己运气。

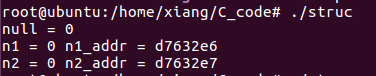
所以加volatile是为了让变量保险一点。

**加了volatile之后，编译器就不会去优化自作聪明把S1认为一直是10，编译器按照传统方法编译**

**Struct结构体的一些bug**

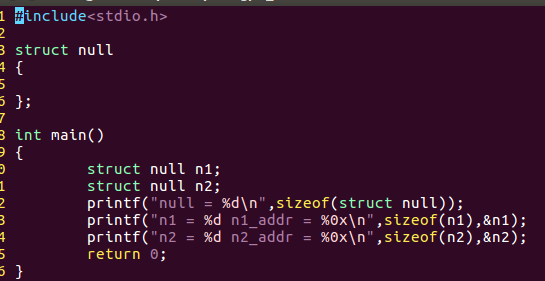


**两个空结构体初始地址不一样，这是因为我们这个版本的gcc编译器做了改进，以前的gcc编译器两个空结构体地址是一样的**



**我们发现空结构体是不占用内存空间的**

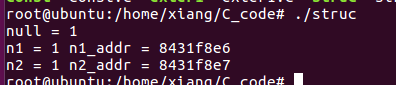
所以以上定义空结构体问题在C语言里面是有bug的



我们将main返回值改为int，这样就可以用g++来编译了



**我们发现g++编译器对C语言进行了严格的控制，给空结构体也分配了1个字节内存**



这样就让空结构体代码安全性更高

柔性数组



**这里分配的长度是多少呢？其实我们在分配之前先看看结构体长度到底是多大**

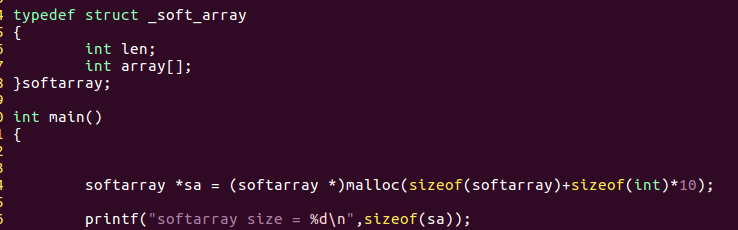
**这就是柔性数组，一般数组在初始化的时候要写数组长度，这里不写数组长度就是柔性数组**



softarray结构体长度是4字节

4字节

意思就是结构体初始化只有变量len的长度占了内存，array并没有占用内存



malloc已经分配内存了为什么只打印8

这是因为sa是地址，相当于打印了softarray结构体本身长度+指针的长度

但是实际上是多分配了40个字节长度的

40字节

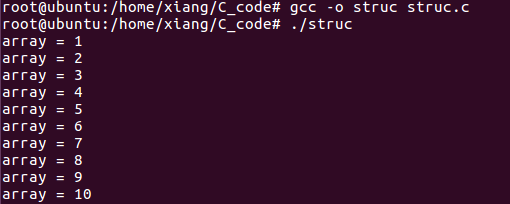
4字节

你看这40个字节是接在len长度后面的。

我们打印的地址是sa指针自己地址的长度。

这种柔性数组有个好处就是，你可以重复调用该函数接口，每次调用都可以分配不同长度的内存空间





柔性数组实际应用，用柔性数组来做斐波拉茨数列

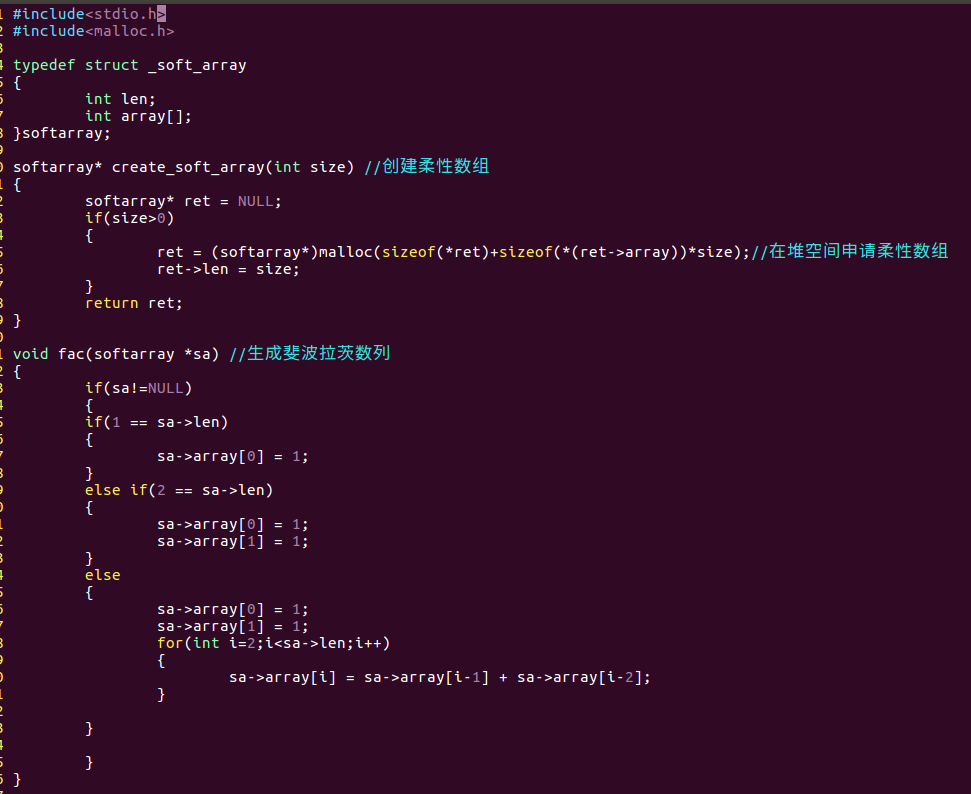
斐波拉茨数列数列原理 = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 //意思就是数组前两项为1，第三项开始，每一项都是前两项的和

定义柔性数组结构 typedef struct

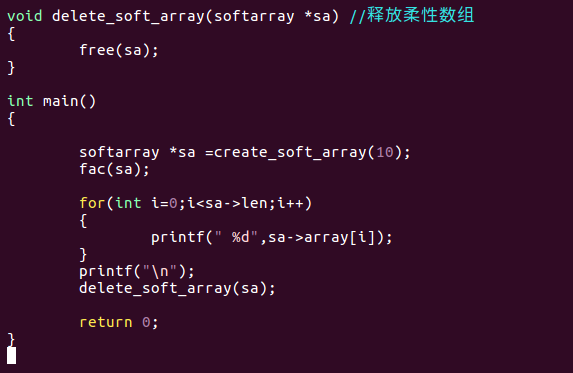
创建柔性数组creat\_sofr\_array

生成斐波拉茨数列

释放柔性数组delete\_soft\_array

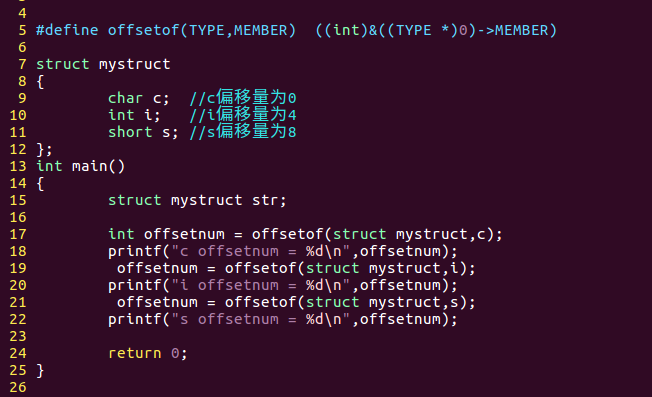


**这种\*方法就是直接分配结构体里面数据类型的大小，如果结构体数据类型变了，这个函数还是可以继续给结构体分配**

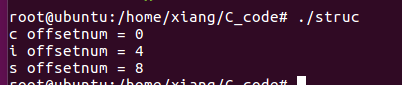


这就是斐波拉茨数列数列

**计算结构体里面成员的偏移量**



**offsetof内核函数，用来计算结构体每个成员的偏移量**

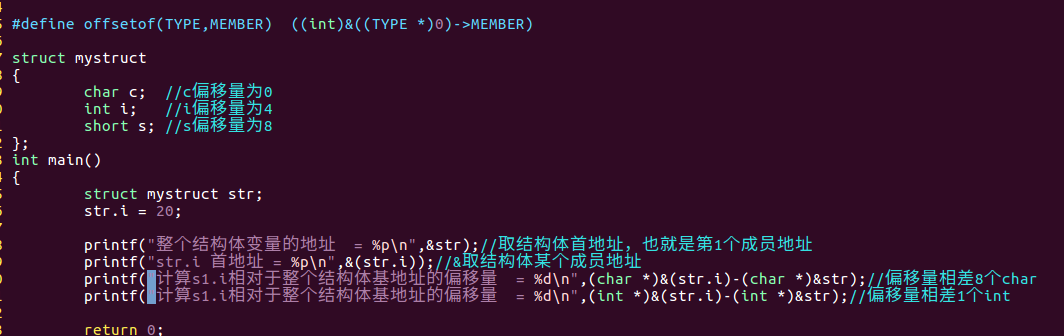


#define offsetof(TYPE,MEMBER) ((int)&((TYPE \*)0)->MEMBER) //这句代码如何分析

第一步 (TYPE \*)0 ： 类似(int \*) 0，我们这里是(struct mystruct \*)0，就是把这个0强制转换成存放struct mystruct 类型结构体的地址。

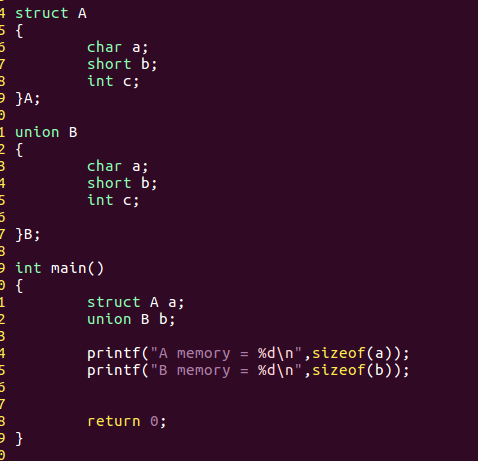
第2步 ((TYPE \*)0)->MEMBER：这句是取结构体里面的成员名，类似p->data；这个p的地址是0.因为我们用(TYPE \*)0先把结构体地址写成0了。

第3步 &((TYPE \*)0)->MEMBER : 类似&(((TYPE \*)0)->MEMBER) 把结构体某个成员名的地址取出来



这个和用offsetof宏一样，只是为了让大家理解这个原理

**union问题**



**union里面的成员只有最大的那个变量占用内存**

**Struct里面的成员都要占用内存**



union联合体一般用在内存很小或者节省内存的系统中，因为你不管使用union里面的哪一个变量，其余变量都会被清0。union要分大端小端。

**int i=1；**

小端

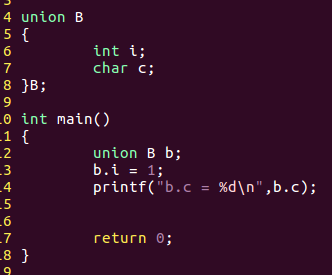
**int i=1；**

大端

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | |
| 0x04 | 0x03 | 0x02 | 0x01 | 地址 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 0 | 0 | 0 |
| 0x04 | 0x03 | 0x02 | 0x01 | |

这就是大端，数据从地址高位开始向低位排列 这是小端，数据从低位向高位排列

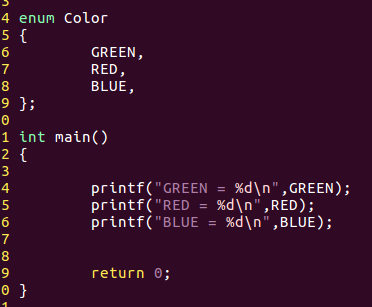
**证明我们ubuntu系统是小端模式**

**但是实际是1**

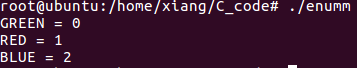
**我赋值的是i所以C应该是0**

**给联合体i赋值**

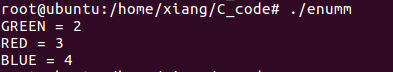
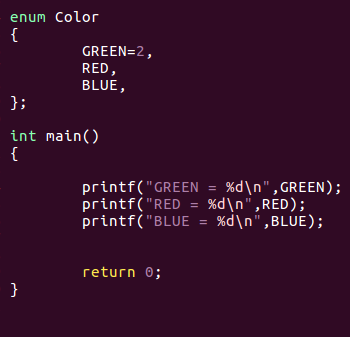
**enum的使用方法**



**enum类型里面的成员是从0开始，多一个成员自动+1**

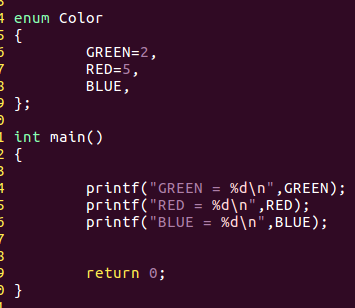


**enum类型里面的成员是从0开始，多一个成员自动+1**



**你看后面的成员从2开始+1**

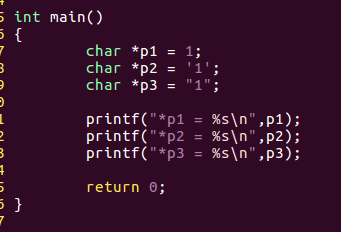
**如果我将没有定义数字的成员，前一个成员定义位2，那么后面的成员根据前面的成员挨着+1**



**你看第三个成员是根据第二个成员的数向后加1的**

这就是枚举enum类型的用法

**引号和单引号的问题:**





你看编译警告，运行报错。问题出在哪里呢？

**所以地址是有了但是地址里面没有值**

**这种定义char \*p等于某个数其实是给p赋地址，而不是赋值，要赋值必须在定义后使用\*P等于什么..….**

|  |  |
| --- | --- |
| 地址 | 地址上面的数 |
| 1 |  |
|  |  |
|  |  |
| 49 |  |
|  |  |
| 50000 | “1” |
|  | ‘\0’ |

内存

char \*p1 = 1

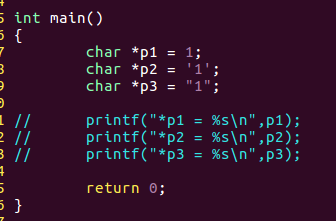
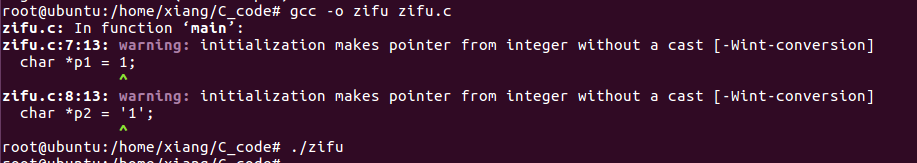
char \*p1 =‘1’

char \*p1 =”1“

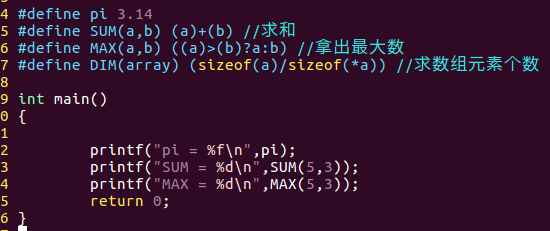
**char \*p 等于某个单引号的数其实也是赋地址，只是这个地址是ASCII码值，1的ASCII码值是49**

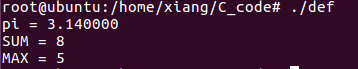
**char \*p 等于某个双引号，就是真正的给某个地址赋值了，只是地址是随机分配**

**我把输出屏蔽了，编译报警，程序可以运行通过，但是\*P1和\*p2的做法是不对的。\*P3做法正确**

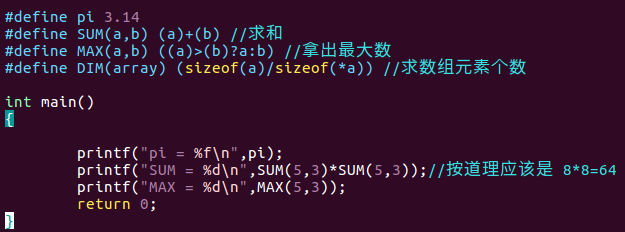
 

**#define的一些问题**





这样是没有问题的



这是编译器在编译过程造成的，

**怎么不是64而是23**

我们来看看编译器编译原理

**问题出在编译到中间文件.i文件时候造成的**

file.i

预处理器

file.c

file.s

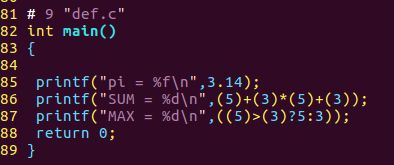
gcc编译器

file.h

我们以往都是gcc –o file file.c直接编译成执行文件，我们这次先编译-E中间文件看看

**编译出的中间文件**

所以宏最好是在定义的时候直接运算出结果。

**你看宏展开后是(5)+(3)\*(5)+(3)**

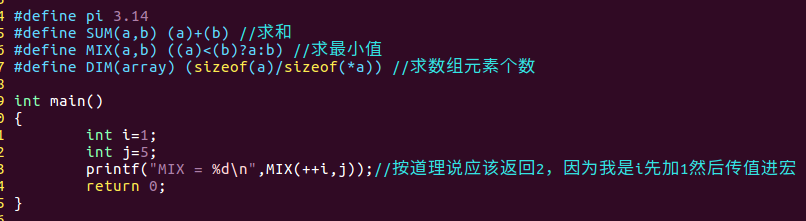
**很明显(3)\*(5)优先级高所以就是23**

宏的第二个运算问题

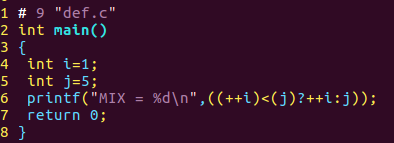


求最小值

没有问题，如果我下面修改一下呢？

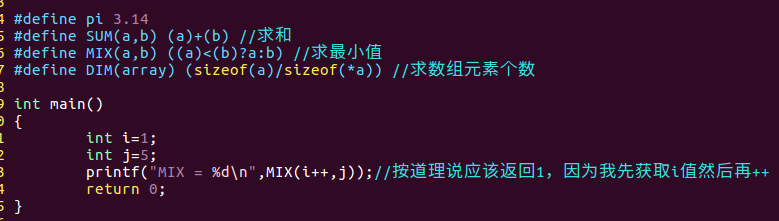


为什么结果是3，我们编译成.i文件看看



**i=2返回的时候又加了次i**

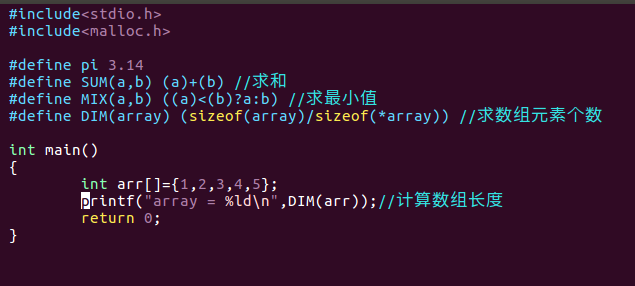
**第一次先加i然后取值i=2**



为什么结果是2

你编译成中间.i文件就知道了。所以当宏定义放进主函数执行的时候，最好直接得到宏运算后的值，不要在主函数里面运算两个带有运算功能的宏。

宏计算数组长度的问题

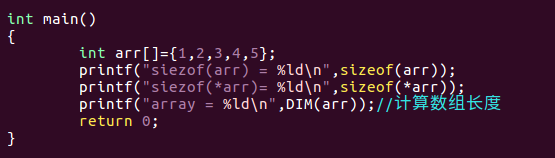


**为什么是ld，因为sizeof返回值是long型**

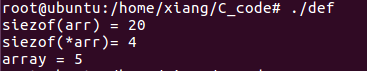
计算结果没有问题，



这个计算原理是什么？



**sizeof取数组地址，就是计算整个数组占用多少个字节**



内存

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 5 |

**1个格子代表一个字节**

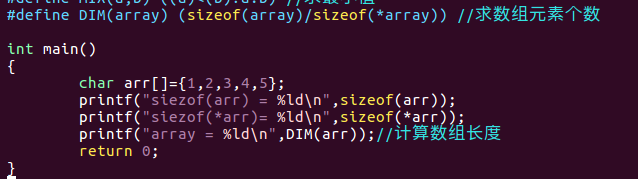
内存

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 5 |

**sizeof取数组指针就是计算数组里面一个元素占多少个字节**

#define DIM(array) (sizeof(array)/sizeof(\*array))

一个数组占用内存字节数/一个数组元素占用内存的字节数 = 数组长度





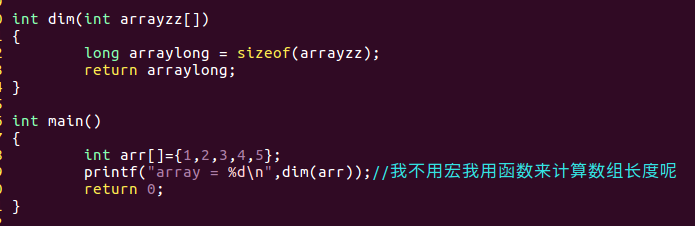
**我改成char，sizeof取数组指针一个元素就是一个字节**

内存

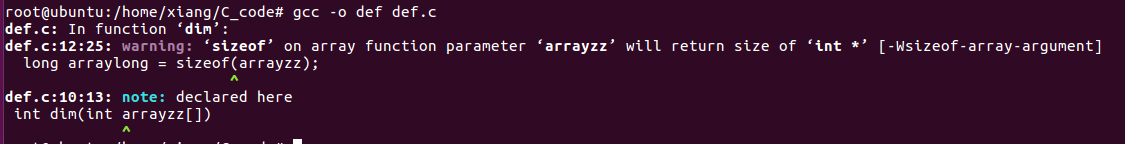
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

1格代表一个字节

所以明显计算1，2,3,4,5,这种数组用char比用int节省空间。但是在linux用char不一定节省空间，因为一个char就霸占一个内存地址，一个内存地址是32位，所以霸占了8位，后面24位只有空着，然后在申请一个char就从新的内存地址开始分配。所以我们分配数组最好是声明的时候就直接连续定义长度，或者用malloc来连续分配动态内存。

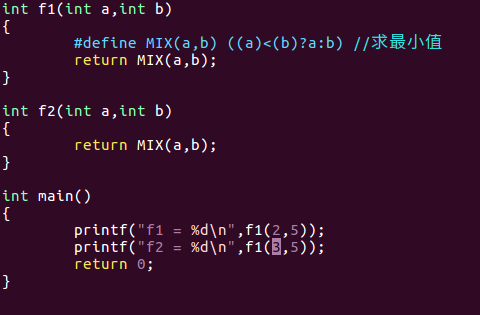


**这样写有问题**



为什么会有警告，在子函数里面计算数组长度有问题，因为arrayzz[]形参自动退化成了指针，所以你sizeof(arrayzz)计算出来的数组长度就是一个指针的4字节长度，和我们前面使用宏计算数组长度的结果不一样，所以这里用宏来计算数组长度比用子函数来计算数组长度优势大多了。

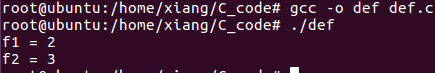
#undef用法



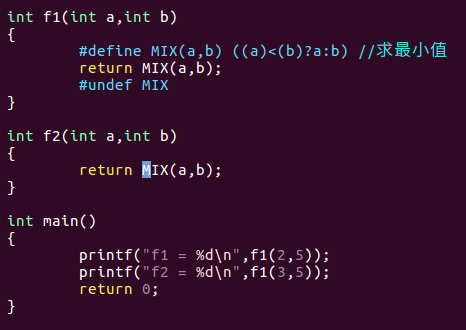
**这是因为宏在代码里面定义之后，文件里面任何一段代码都可以调用它**

**这就是宏的特性**

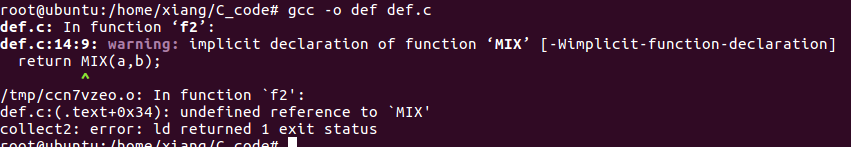
**我们发现在子函数f1定义的宏，子函数f2可以使用**

输出结果正常

如果我们只想让宏在某个函数里面执行，其它函数不能调用这个宏怎么办呢？



**在使用这个宏的函数结束位置加#undef就可以了，这个宏就只限制于本函数**



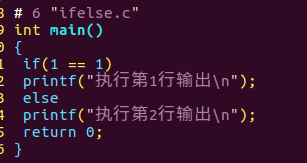
你看f2想去调用f1的宏就报错

**#if #else和if else的区别**



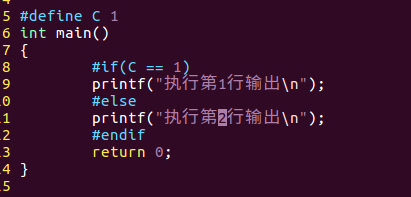
**这是我们常用的if else**

输出正常



**在i文件里面编译器把if和else的输出都编译进去了**

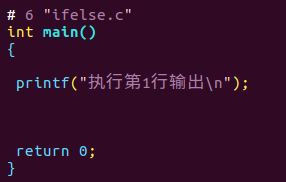
下面这是#if #else



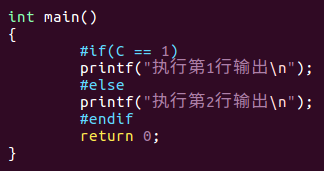


**输出结果一样**

但是你看看gcc -E的中间文件，



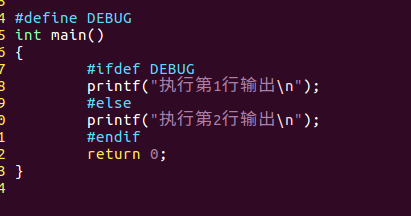
**只编译进#if成立的执行段代码。不成立的就不编译进二级制**



**我如果不定义宏，那么就输出#else代码段**

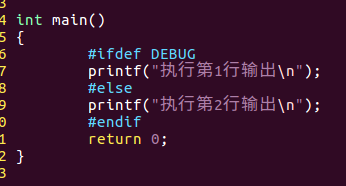


还有一种不用给C赋值的条件判断语句，#ifdef #else #endif



**用ifdef直接在宏这里定义名字就是了**

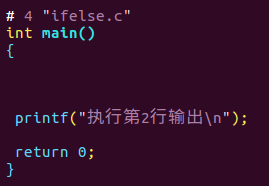




**这是没有在宏定义名字的**

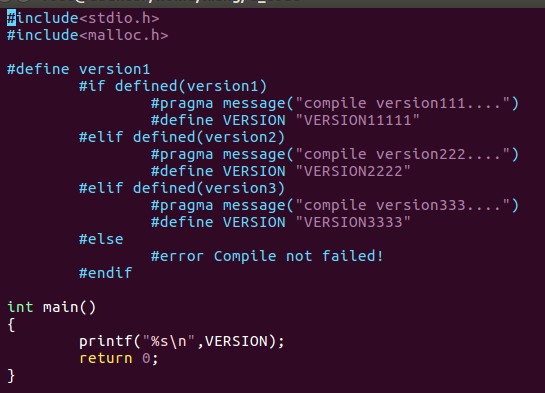


其实这种条件编译语句适合用来做代码调试，比如我写了很多printf和调试程序，正式发布程序的时候不需要这些，但是调试的时候需要，但是太多的调试代码一个一个去删除很麻烦，如果删除了那天又想拿回来调试又要重新写，所以用条件编译时最好的解决方法。

 #ifdef和#if一样的都是选择性编译。

**编译器编译C代码的时候打印出的warning，error，是怎么实现的？**

#error，#warning，#pragma这些都是预处理指令，就是靠这些实现的

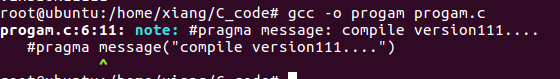


**#elif就是第一个#if不成立就执行我现在这elif的语句和 else if一样**

**#error 编译出错的时候输出错误信息**

**把编译版本号写入VERSION，然后编译完成执行程序时主函数可以打印出来**

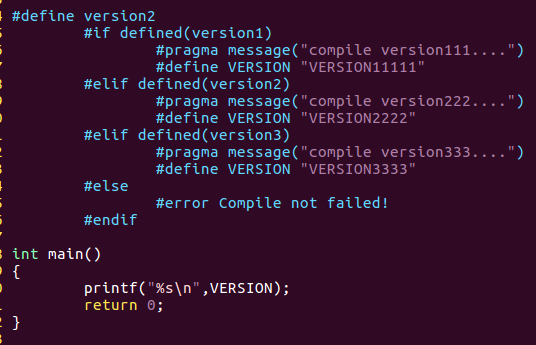
**#pragma message 就是编译的时候输出你编译的是哪一段代码**





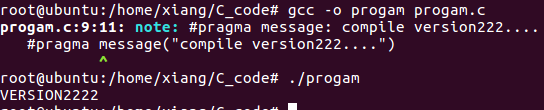
**这就是pragma写入字符输出**

比如我把宏#define version1改成version2



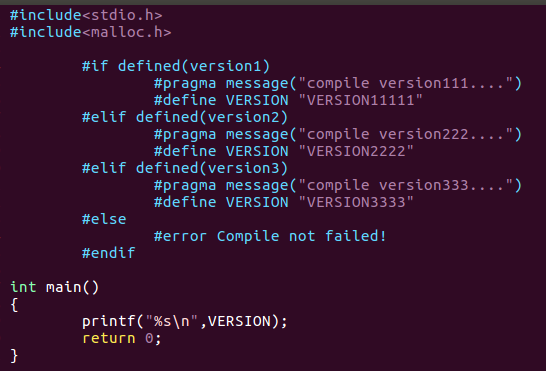
**编译的时候就应该执行这一段**

**我把宏改成了版本2**



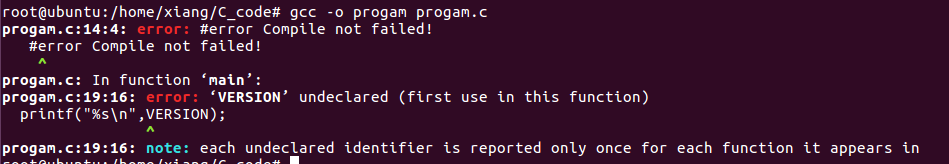
你看编译的时候执行elif 里面的输出，主程序也是执行VERSION222输出

如果我取消宏呢?



**输出直接打印报错error里面的字符串**

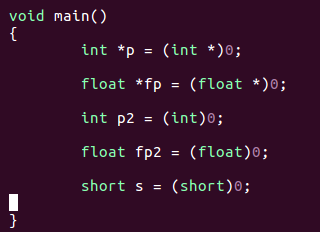
**我取消了#define version1或者#define version2这一段**



所以编译程序输出信息用#pragma，#error，#warning，执行程序输出信息用printf

#progma这个东西根据编译器的不同，是有很大区别的，所以在不同编译器使用#progma的时候，最好先实验下#progma的功能。

**强制类型转换，转换的是数据还是地址？**



内存

|  |  |
| --- | --- |
| 地址 | 数据 |
| 0x01 |  |
| 0x00 |  |

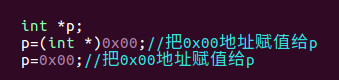
**这种方式就是我们把0当做一个地址赋值给p，p指向内存0这个地址**



内存

|  |  |  |
| --- | --- | --- |
|  | 地址 | 数据 |
|  | 0x01 |  |
| p= | 0x00 |  |

类似与下面这样



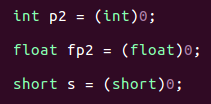
内存

|  |  |  |
| --- | --- | --- |
|  | 地址 | 数据 |
|  | 0x01 |  |
| \*p= | 0x00 | 5 |



所以括号里面带\*强制类型转换时转换地址，把某个

地址赋值给指针，然后向指针里面写数据就是向地址里面写数据。

这种括号里面不带\*的写法又是什么意思呢

我们知道内存的最小单元是char字节，所以内存一个地址只能存放一个字节8位的数，而不是16位，32位

内存(32位系统)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据 |  |  |  |  |
| 地址 | 0x00000007 | 0x00000006 | 0x00000005 | 0x00000004 |

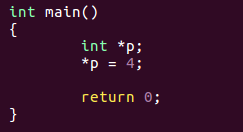
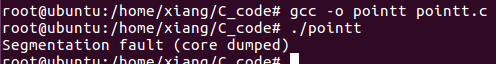
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据 |  |  |  |  |
| 地址 | 0x00000003 | 0x00000002 | 0x00000001 | 0x00000000 |

**指针使用注意事项**

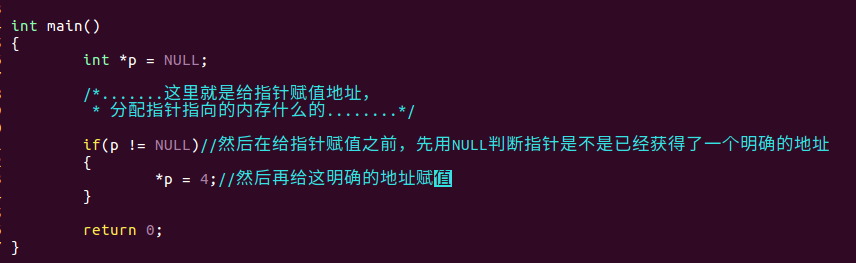
野指针处理方法

**因为指针的地址不确定，所以这个4也不知道放在内存哪个地址上，就会出现段错误**

**指针p被赋值前，没有给指针赋地址**

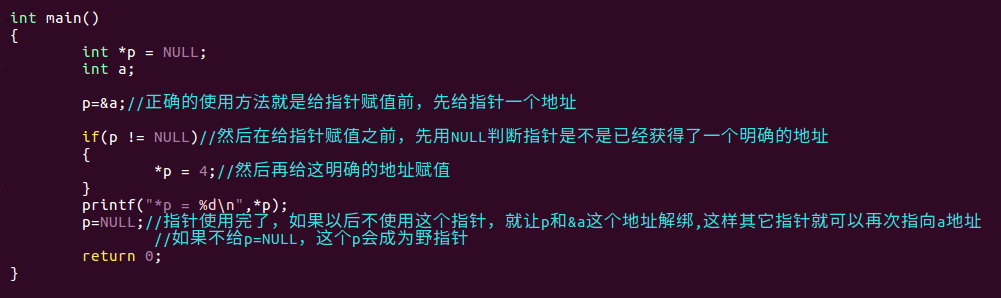
 

这种就是野指针的段错误。





用NULL判断指针是否有地址，这样可以防止段错误

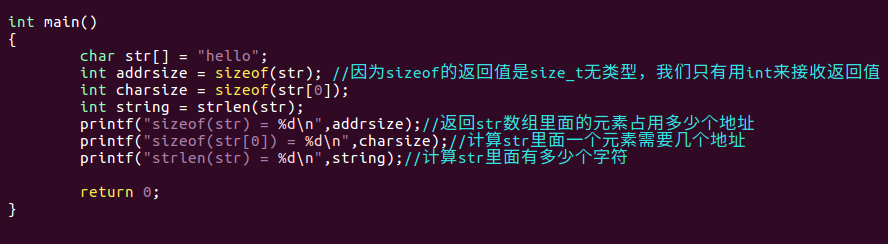


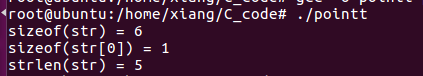


指针用完后，给指针地址改成NULL，这样可以防止野指针。

你如果水平可以也可以在给\*p赋值之前不用判断p是不是为NULL，自己把控指针使用方法。

**sizeof使用注意**

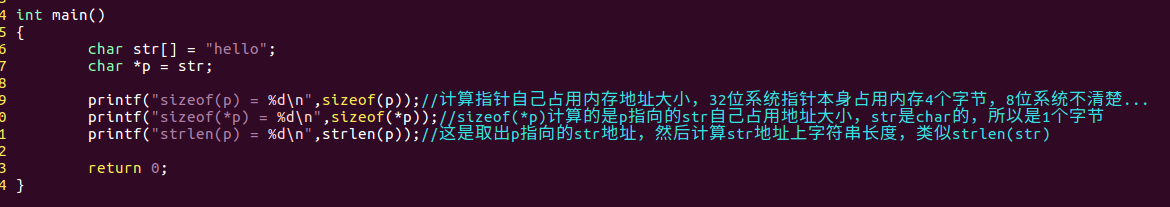


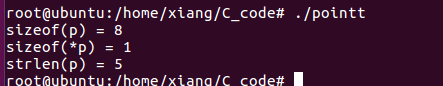


**为什么是6，因为计算的是str里面所有字符串加起来占多少个地址，我们知道1个字符占用1个字节，这个字符串除了hello之外还有个\0结束符，所以加起来是6**

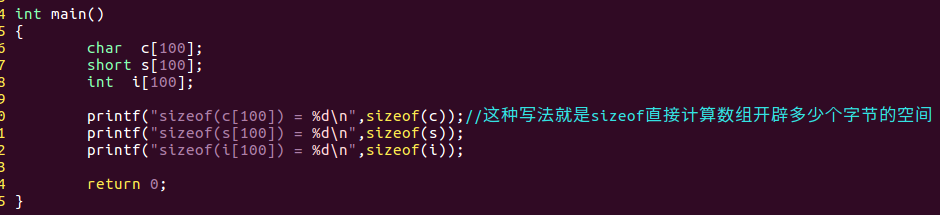
**这是计算数组里面我们看得见的字符个数，所以hello是5个字符，而不是前面使用sizeof计算数组占用地址大小**

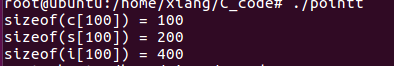
**这是计算1个字符的大小，1个字符8位1个字节**



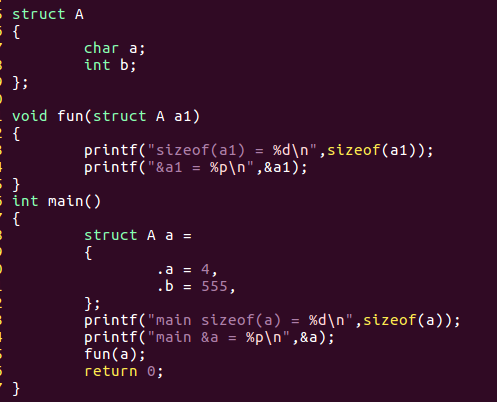


**这里指针怎么占用8个字节，因为我们这里用的是ubuntu 64位操作系统**





**struct结构体高级用法**



**主函数和子函数地址都不一样，所以主函数只是拷贝数据给子函数，子函数运行结束，子函数的数据都会被释放**

**函数用结构体做形参和函数用int形参是一样的，都是拷贝数据**

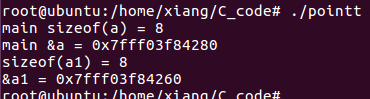
**在gcc编译器中结构体可以这样初始化，这样初始化的好处是如果我不想初始化a我可以直接**

**Struct A a =**

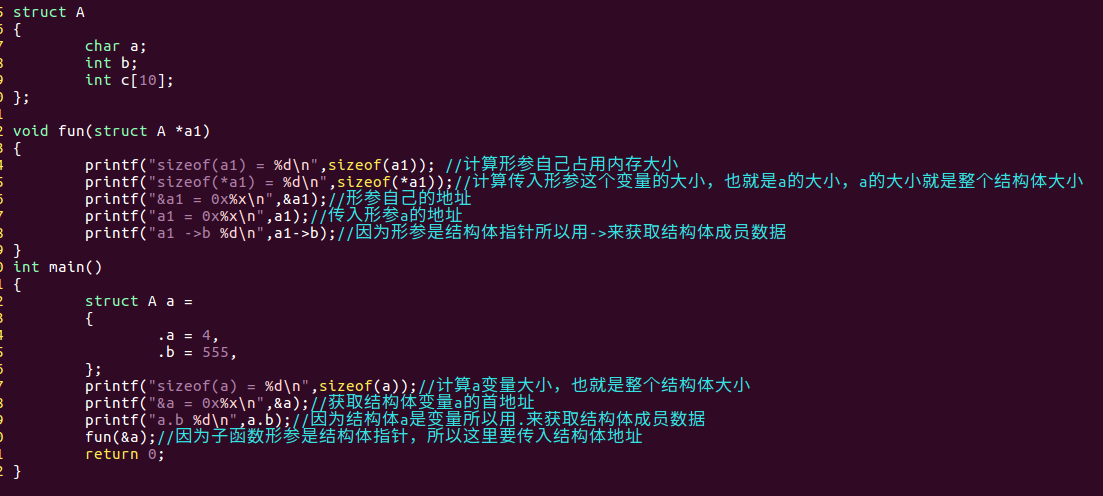
**{**

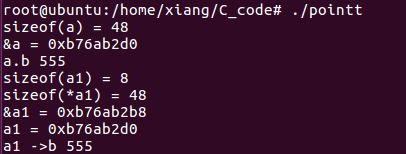
**.b = 555，**

**}；**

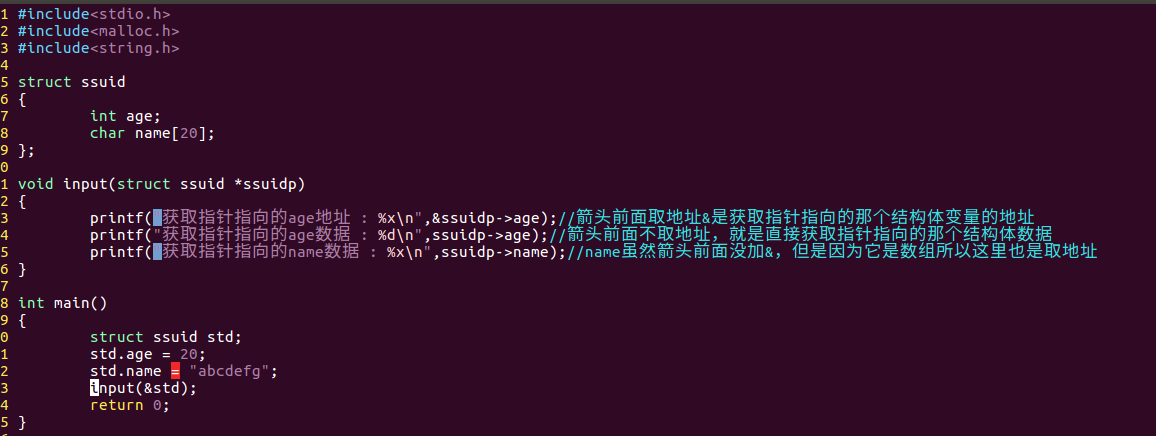


像这种结构体传参数给子函数的方法效率很低，因为结构体一般情况下会定义很多变量和数据，所以在传递形参过程中相当于拷贝数据，你说慢不慢。解决这种效率低下的方法就是用指针来接收结构体地址。





如何取结构体指针成员里面的地址



C语言只有在定义字符数组的时候才能用“=”来初始化变量，其它情况下是不能直接用“=”来为字符数组赋值的

char a[10] = "123"; /\*正确，在定义的时候初始化\*/

---------------------------------

char a[10];

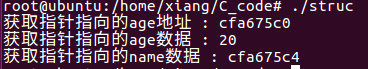
a = "123"; /\*错误，不能用“=”直接为字符数组赋值\*/

---------------------------

strcpy(a, "123"); /\*正确，使用strcpy函数复制字符串\*

**这里为什么给字符数组赋值会报错？**





多级结构体使用

创建结构体1

创建结构体2

结构体1

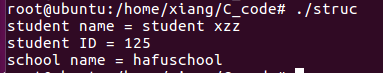
(结构体1包含结构体2)

结构体2



**用2个点访问school结构体里面的stu结构体**

**用1个点访问school结构体**



这种多级结构体都是用来管理变量和内存的，如果变量少用这种方法没有意义反而麻烦。那么用多级结构体有什么用？比如我要定义一个学校，学校这个变量就包含了树木，房子，老师，操场，学生，这些变量。在学生这个变量里面还包含学生，年龄，成绩，健康这些变量，所以我们就用了一级结构体来定义学校的成员，二级结构体来定义学生自己本身的内容。当然你也可以不这么做，你也可以直接写n多个变量来做。只是看起不是很直观。

其实这种直接访问多级结构体还是没有看出它有什么优势，但是我们用多级结构体指针就能看出它的优势了

结构体多级指针使用

结构体多级指针一般使用在多个结构体指针里面

**这种用结构体指针的话，我可以在程序运行的时候自动分配学校和学生这些变量，然后程序执行到某一段不需要学校和学生这些变量了，程序会自动释放掉这些变量，内存空间使用就相当智能，想用学校学生就自动分配，不想用就自动释放**

创建结构体2

创建结构体1

结构体2

结构体1

包含结构体2(而且是定义的结构体2是指针)



**访问本身结构体**

**然后再释放本身的结构体**

**释放学生和学校变量的时候一定要先释放结构体里面包含的结构体**

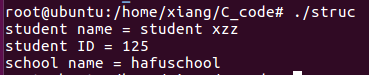
**访问结构体里面的结构体**

**我分配第二级，也就是school里面的结构体变量时也是用的指针，所以我要用2个箭头**

**我分配第一级的school结构体变量就是用指针，**

**所以访问school的时候用1个箭头**

**我在school结构体里面创建了一个结构体指针**



结构体指针虽然比上面直接访问结构体多了几行代码，但是这样就可以做到自动申请自动释放。

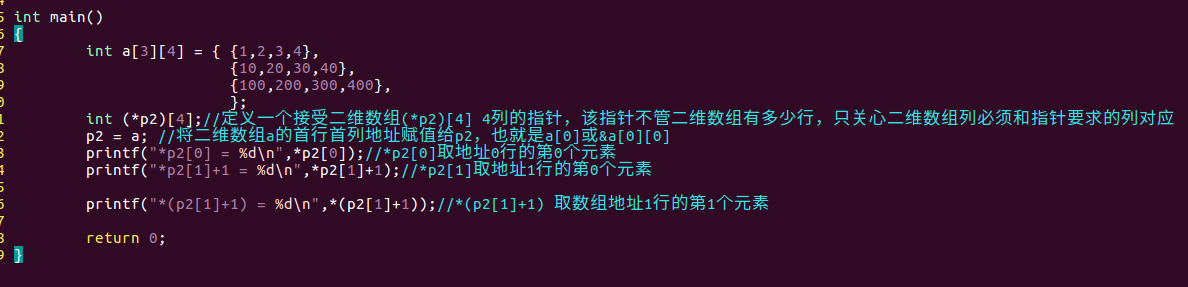
如果你使用3级结构体，就多写几个箭头，->->->

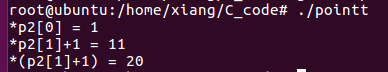
如果你使用4级结构体，就多写几个箭头，->->->->

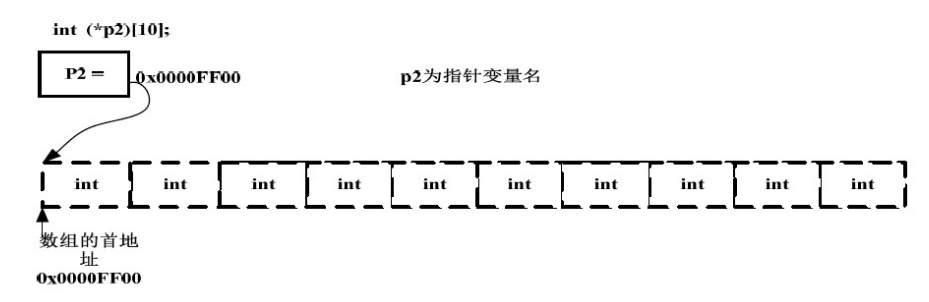
还有结构体指针访问比直接访问结构体运行效率高。

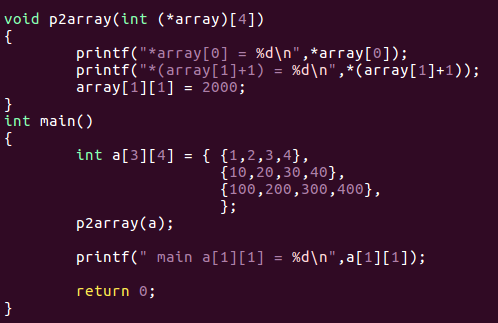
**数组指针，用来操作二维数组**

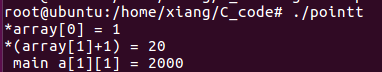
定义 int (\*p)[n];  
()优先级高，首先说明p是一个指针，指向一个整型的一维数组，这个一维数组的长度是n





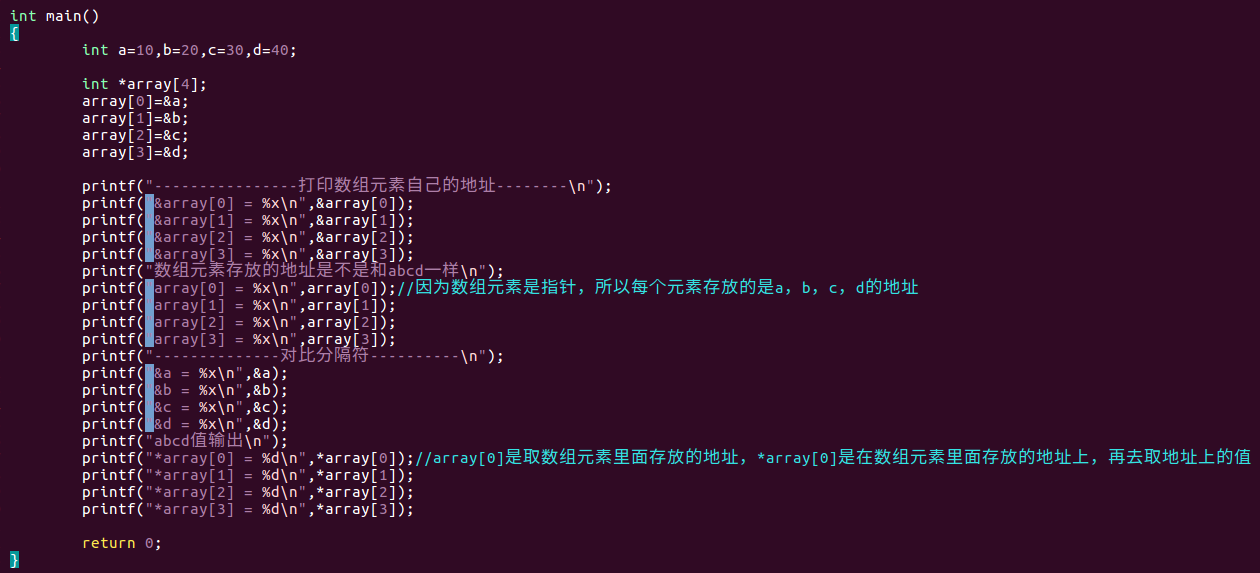




这就是数组指针用在子函数接口封装，修改原本二位数组值的好处。

**指针数组，数组里面全部放的是指针变量**

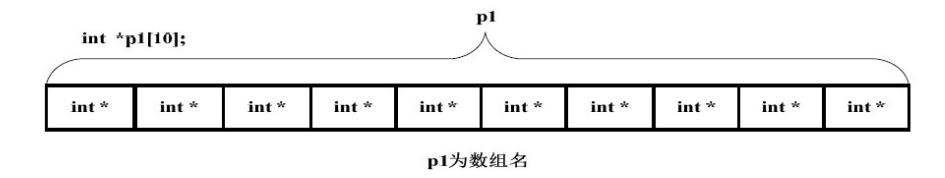
定义 int \*p[n];  
[]优先级高，先与p结合成为一个数组，再由int\*说明这是一个整型指针数组，它有n个指针类型的数组元素。



**指针数组里面每一个元素不是用来放值的，而是用来放地址**

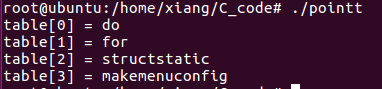


**因为array[0] = &a,所以指针数组不用\*号直接用下标取出来的是地址**





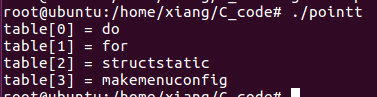
**如果指针数组不做函数形参，那么相互之间传递地址要用下标[]位置**



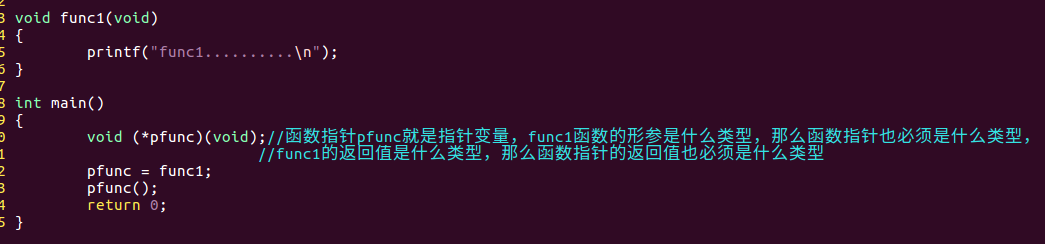


**指针数组比较适合用来做字符串传递，比如main函数的argv**

**如果指针数组做函数形参，那么相互之间传递地址就不用指定下标位置**



**函数指针**









这就是函数指针多参数用法。

**这是func1子函数占用的内存空间大小，每个子函数在内存都是连续的**

|  |  |
| --- | --- |
| Func1 | 0x0c |
| Func1 | 0x0b |
| Func1 | 0x0a |
| Func1 | 0x09 |
|  | 0x08 |
|  | 0x07 |
| pfunc | 0x06 |
|  | 0x05 |
| 数据 | 地址 |

内存

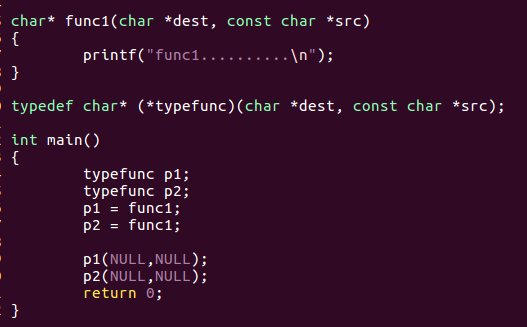
**函数指针变量存放在0x06地址这个位置**

**这个pfunc变量存放的是func1函数的首地址，所以执行pfunc就是执行func1**

typedef修饰函数指针的好处



如果我想多定义几个函数指针，指向func1函数，我就要写很多个，因为func1的形参太多，所以我要每个函数指针都要重复写很多形参，导致我太累。



**以后看到这种定义，先搞清楚是定义的结构体，还是函数指针**

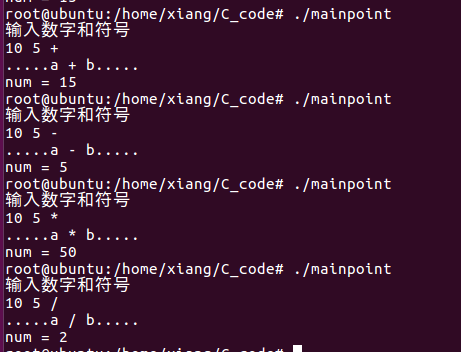
**然后我不管想创建多少个函数指针，都不用每个都去写形式参数**

**我用typedef定义一个函数指针的类型**



函数指针模拟函数重载使用



其实我觉得上面用函数指针做子函数调用，和直接调用子函数一样，我为什么还要这么麻烦用函数指针去调用子函数呢？

如果是用函数指针去做调用子函数确实没有什么意义，反而麻烦，但是我下面准备做分层编程，那么函数指针就有用了。

我们把上面计算器这个程序，进行分层，让多个人来完成这一个计算器功能。

我创建了两个文件，framework.c是架构文件。

Calcolator.c是具体执行的文件，一个人负责一个文件，所以就需要两个人来做这个事情。

**另一个人负责写这个c文件**

**一个人负责写这个c文件**

calcolator.c

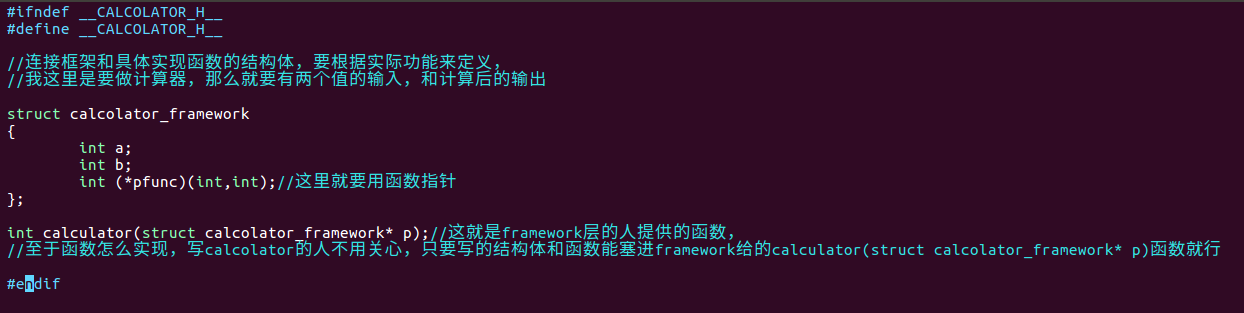
framework.c

calcolator.h

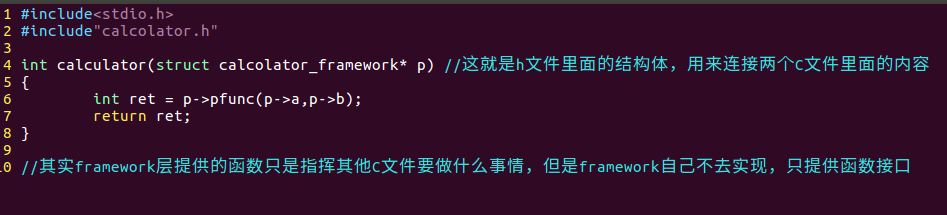
**calcolator.c里面的函数要调用framework.c里面的函数，那就需要一个桥梁，这个桥梁就是头文件，我们创建一个calcolator.h头文件**

我们创建了一个h文件当桥梁

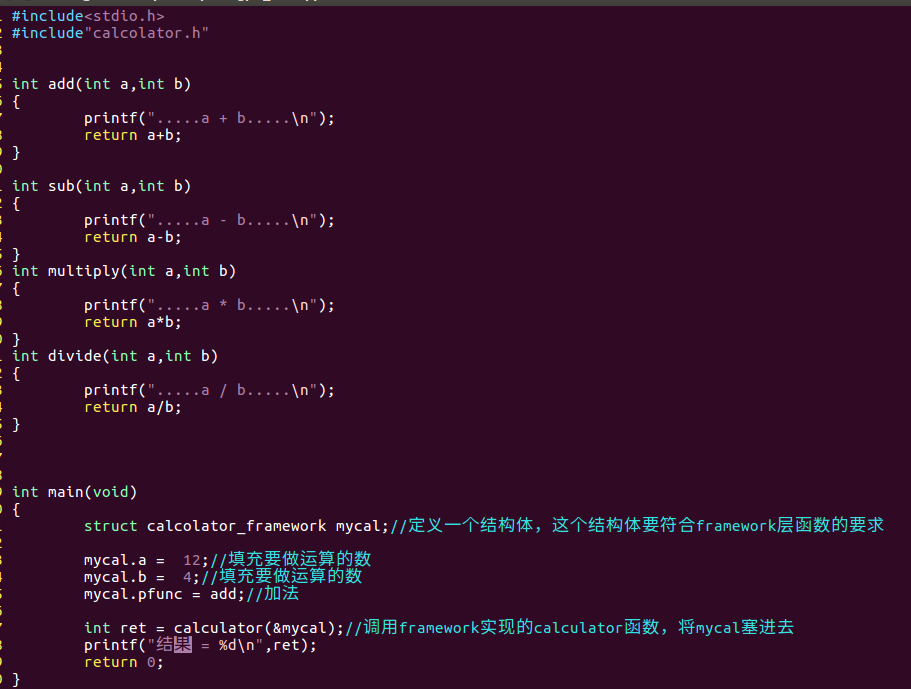
我们先创建桥梁calcolator.h



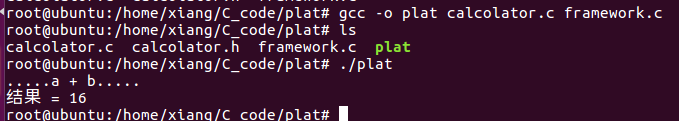
Framework层的人负责头文件的内容编写



在framework.c里面Framework层的人还要实现给下层人的函数接口



calcolator.c 由下层的人负责实现具体功能代码，然后填充framework层人设计的结构体和函数。



将两个C文件编译起来，就可以使用了，我们计算的是12 + 4 =16

这就是函数指针的用处，我们以前都是先写调用函数，在调用函数里面实现代码。然后主函数调用子函数

framework实现函数接口

子函数的C文件

实现子函数C文件的内容

子函数填充framework的函数接口

子函数调用framework函数执行程序

主函数调用子函数

这是以前的代码框架 这是分层后的代码框架

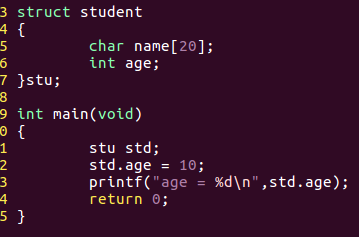
这样看分层貌似没有什么特别的用处，但是当你几十个人在完成一个项目的时候你就会发现很有用，只是我们现在这个例子太简单，所以反而感觉是杀鸡用牛刀，现在记住有这个思想就是了。

**typedef修饰结构体的好处**



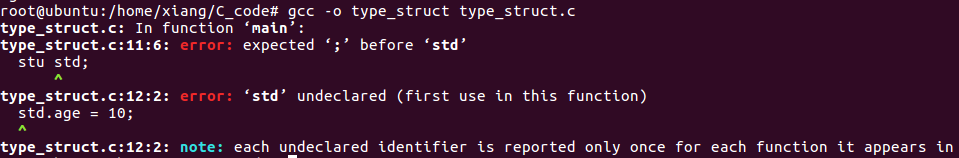
**我感觉每次定义结构体变量前面都要加struct太累了**



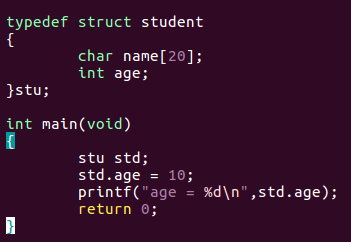


**然后用这个类型名曲定义结构体变量，这样就不用在前面加struct这么累了**

**我在结构体分号前定义一个类型名**



发现编译出错，那么我们再来修改下。

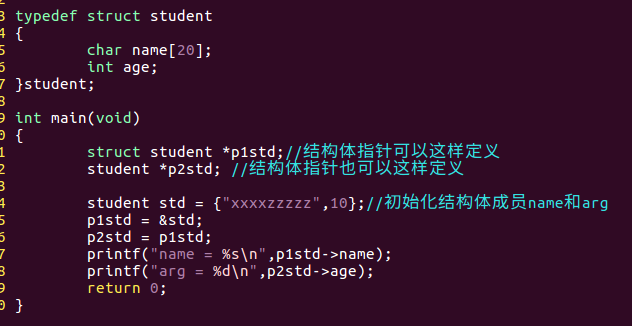


**这样写虽然能节省写struct的工作量，在你每次使用的时候，你要注意这是结构体类型，还是函数指针类型哦**

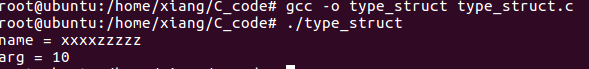
**这样就成功编译过了**

**我们在结构体前面加了typedef，来让结构体可以在分号前定义结构体类型**

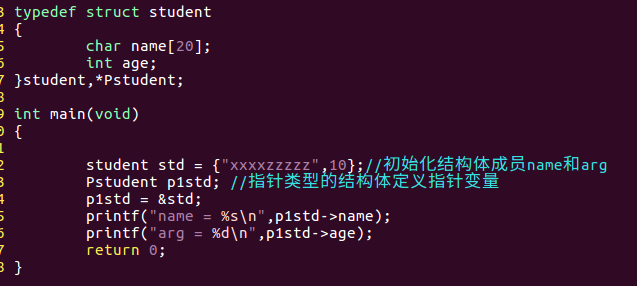
**定义结构体指针**



**这个名字可以和类型名字一样**

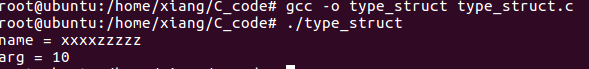


这就是结构体定义指针的基本方法



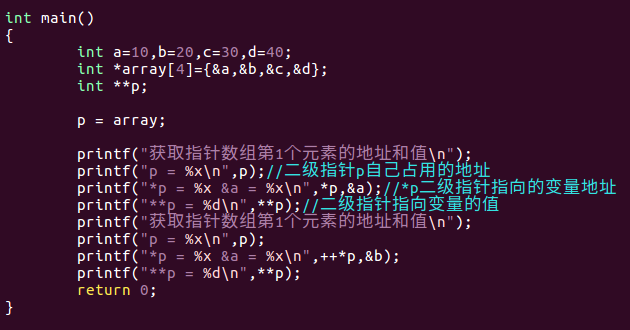
**类似struct student \* p1std**

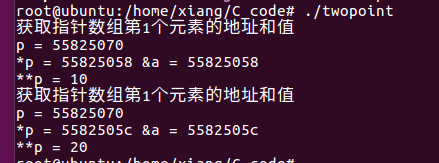
**结构体可以定义类型，还可以定义指针类型**



结果一样

**二级指针一般用来指向指针数组**

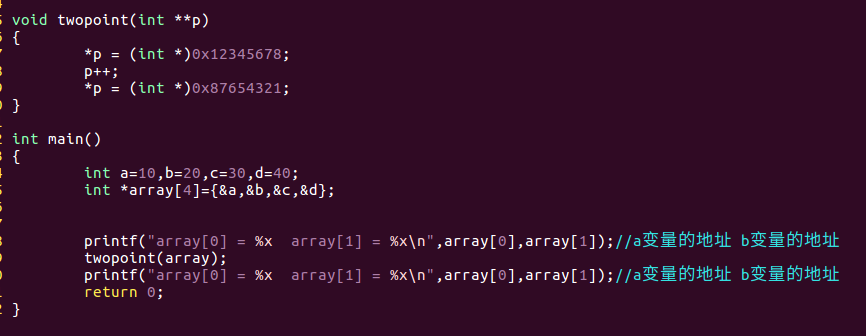


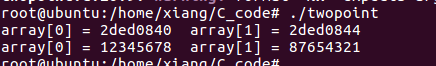


这就是二级指针大部分都是用来指向指针数组的，用来获取指针数组里面存放的各个元素指向变量的地址。

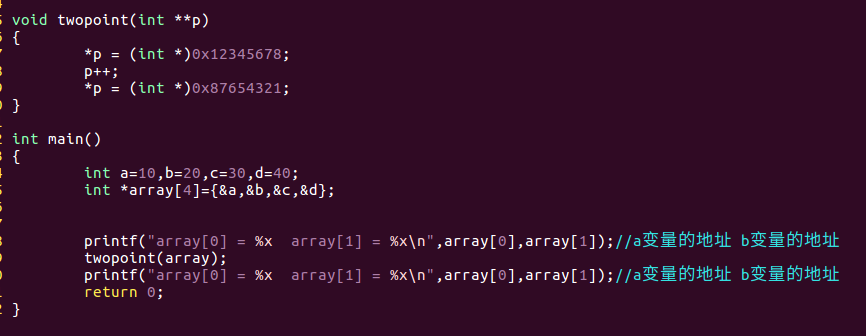
但是这样有意义吗？我不知道直接用指针数组吗？

我们用二级指针来做子函数形参就有意义了





这就是二级指针，用子函数去修改主函数里面变量的地址。



|  |  |
| --- | --- |
| a | 0x09 |
| b | 0x08 |
| c | 0x07 |
| d | 0x06 |
|  | 0x05 |
| 0x06 | 0x04 |
| 0x07 | 0x03 |
| 0x08 | 0x02 |
| 0x09 | 0x01 |
| 数据 | 地址 |

内存

**将abcd的变量地址分别给数组的每个元素，这样数组每个元素存放的就是变量的地址，而不是值，这就是指针数组**

|  |  |
| --- | --- |
| a | 0x09 |
| b | 0x08 |
| c | 0x07 |
| d | 0x06 |
|  | 0x05 |
| 0x06 | 0x04 |
| 0x07 | 0x03 |
| 0x08 | 0x02 |
| 0x09 | 0x01 |
| 数据 | 地址 |

内存

|  |  |
| --- | --- |
| a | 0x09 |
| b | 0x08 |
| c | 0x07 |
| d | 0x06 |
|  | 0x05 |
| 0x06 | 0x04 |
| 0x07 | 0x03 |
| 0x08 | 0x02 |
| 0x09 | 0x01 |
| 数据 | 地址 |

内存

**二级指针\*\*p里面 (p访问的是数组元素自己的地址)**

**二级指针\*\*p里面 (\*p访问的是数组元素上的值，这值也是地址值)**

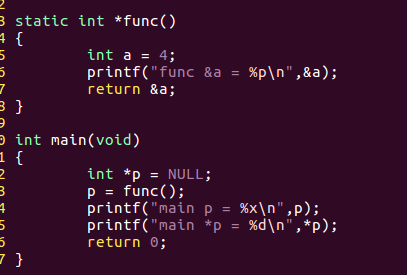




上面说我用p是取数组自己元素的地址，用\*p是取数组元素上面存放的地址，但是这个值是地址，因为我是指针数组。然后\*\*p就是取数组元素存放地址上存放的值。

操作系统把一整块内存划分位堆，栈，.data区域

**C语言各种程序在内存分布的情况分析**



|  |  |
| --- | --- |
| 4 | 0x12 |
|  | 0x11 |
|  | 0x10 |
|  | 0x0f |
|  | 0x0e |
|  | 0x0d |
|  | 0x0c |
| 44 | 0x0b |
|  | 0x0a |
|  | 0x09 |
|  | 0x08 |
|  | 0x07 |
|  | 0x06 |
|  | 0x05 |
|  | 0x04 |
|  | 0x03 |
|  | 0x02 |
|  | 0x01 |
|  | 0x00 |
| 数据 | 地址 |

内存

栈

按道理来说这个子程序结束0x12地址的数值4应该被清空，因为子函数在栈空间，子函数执行结束栈应该被释放，确实栈的代码是被释放了，但是变量的数字还在内存中

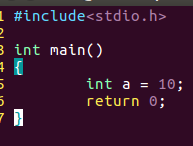
如果是这样，那么我们几个子函数相互调用栈怎么处理？



堆

func1定义的b变量放在内存里面是不会消失的，如果func2定义a变量也占用了这块内存地址，那么这个a变量本身是b变量的值44，只是我们习惯每次变量使用前都要赋值，所以就把0x0b地址的值赋值成了4

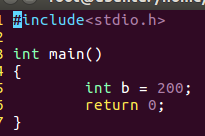
所以同一块栈空间是可以被多个子函数反复使用的。一般操作系统是对每个进程分配一个栈空间

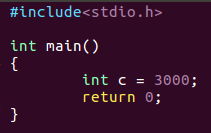


栈

|  |  |
| --- | --- |
| 10 | 0x12 |
|  | 0x11 |
|  | 0x10 |
|  | 0x0f |
|  | 0x0e |
| 200 | 0x0d |
|  | 0x0c |
|  | 0x0b |
|  | 0x0a |
|  | 0x09 |
| 3000 | 0x08 |
|  | 0x07 |
|  | 0x06 |
|  | 0x05 |
|  | 0x04 |
|  | 0x03 |
|  | 0x02 |
|  | 0x01 |
|  | 0x00 |
| 数据 | 地址 |

内存



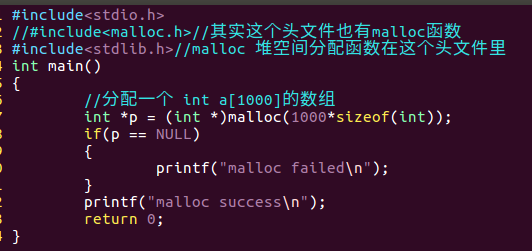
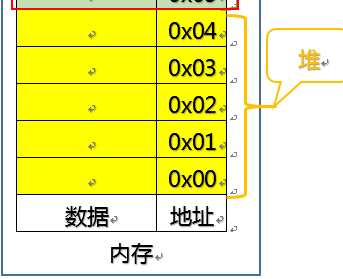


操作系统会对每个hello进程分配一个4k大小的栈空间，也有可能8k根据系统本身决定，这4k就是给一个进程使用的，这个进程里面的子函数，线程可以在这4k空间里面任意存放，但是如果你子函数的分配的数组大小或者结构体变量过多，大于4k，就会出现内存溢出，这个溢出就是超过了操作系统给你进程分配的4k内存大小，但是对整个内存条来讲你进程分配的空间是没有溢出的，你溢出的是操作系统给你规定的栈空间大小

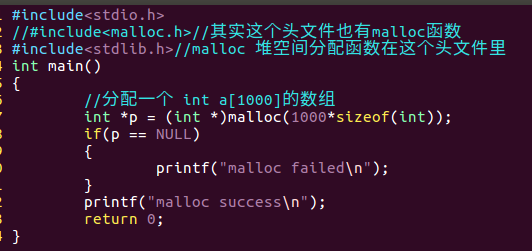
堆

所以在定义变量存放大批量数据的时候不要用局部变量数组，全局变量数组。比如int a[1000000]；或者结构体成员过多 struct{int a[1000000000000] , int b[1000000000] ,………………………}；这些都会超过栈空间大小。

定义变量存放大批量数据我们要用malloc堆来分配，这样可以解决栈空间不够内存溢出问题。

就是malloc把int a[1000]本来分配在栈的变量，放在了堆上。只不过把变量名改成了p



为什么是4096，因为一个int是4字节

堆

|  |  |
| --- | --- |
| 栈 | |
| int[1000] | 0x4096 |
| …….. | ………. |
| int[0] | 0x00 |
| 数据 | 地址 |

内存

malloc(2)

堆空间

|  |  |
| --- | --- |
| char 8位 | char 8位 |

malloc自己只返回分配了多少个8位的空间

malloc(10\*sizeof（int))

堆空间

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | ……… |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

malloc分配了40个8位的空间

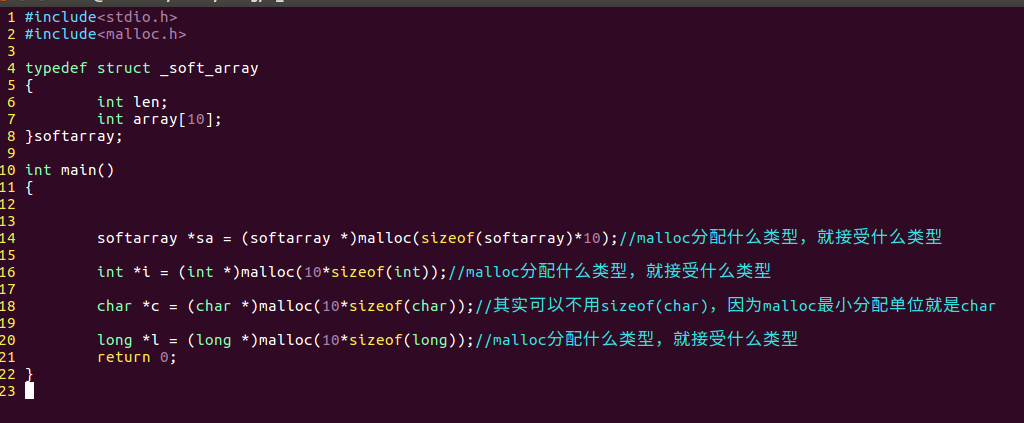
(int \*)malloc(10\*sizeof(int))

堆空间

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | ……… |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

void \*malloc(…….) //返回void\*的意思不是返回空NULL，而是malloc开辟了很多个字节的内存，但是它不知道怎么划分，返回个void\*，你可以将这块内存用(int \*) (char \*)来定义

用（int \*），将malloc分配的40个8位变成了10个32位的空间，那么每1个空间可以装1个int的值



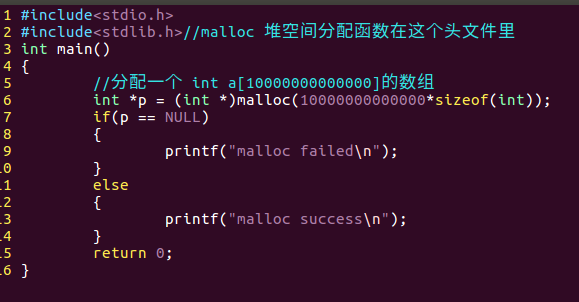
这就是malloc的好处，分配多少个空间就使用多少个空间，这些空间在堆里面，可以反复的使用。

所以当你在开发新的cpu平台时，或者新版本linux操作系统时，要先查阅内存分配的堆，栈大小。

堆空间分配错误问题



堆分配成功

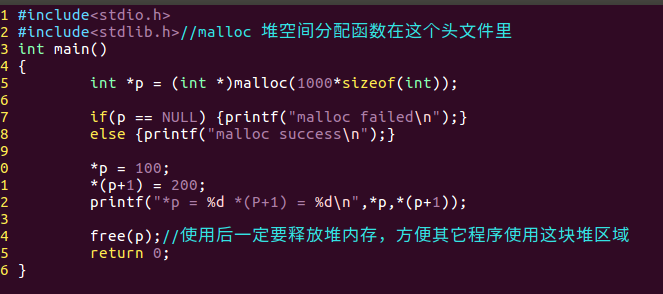


堆分配失败

这是因为你分配的堆空间大小，已经超过了内存堆区域的大小了。

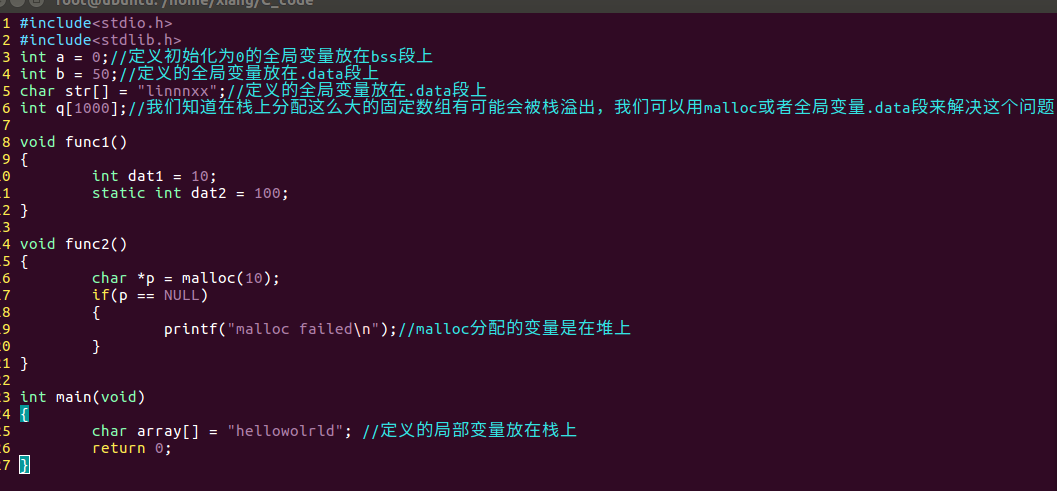
总结：不管是哪个进程，对变量进行堆分配，都在一个堆区域里面。不像栈区域，每个进程都有自己的4k栈，不管怎么定义变量，都在自己的栈里面，不会影响其它进程定义的变量。但是堆是公共厕所，所以某个进程使用完堆，一定要执行free，否则其它进程就无法再使用堆

堆使用后一定要释放

数据段(.data) ， 代码段，bss段

所以内存除了堆，栈，还有数据段，代码段，bss段



|  |
| --- |
| .data段 |
| 代码段 |
| bss段 |
| 堆 |
| 栈 |

内存

所以最好在使用的开发板上搞清楚各区域的大小

代码段：就是放函数里面的if …else…for之内的语句

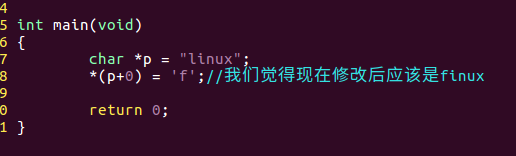
.data段：就是放定义的非0全局变量和加有static的局部变量，记住哦局部变量加了static那么就放在.data段上，而不是栈上。

bss段：就是放非0的全局变量，但是你给全局变量修改值后，就右放在.data段了所以bss也算.data段

局部变量放栈上

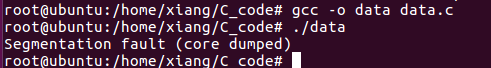
malloc的变量指针和区域放堆上

还有一种特殊的情况

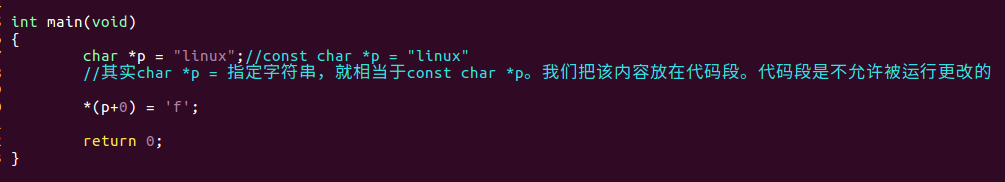


|  |
| --- |
| .data段 |
| 代码段 |
| bss段 |
| 堆 |
| 栈 |

内存



为什么是段错误？



**函数递归调用**



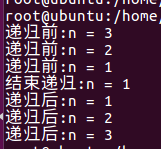


我们用函数递归调用计算5的阶乘，5 X 4 X 3 X 2 X 1 =120

阶乘就是一个数乘以比它小一个单位的数，一直乘到1，最后得出来一个数。比如上面这个就是。

函数递归绝对不是函数循环。

递归的原理：



函数递归还可以用来求斐波拉茨数列

理解递归要用栈的概念

栈是一种先进后出的结构

写入1

|  |  |  |
| --- | --- | --- |
|  |  | 1 |

写入2

|  |  |  |
| --- | --- | --- |
|  | 2 | 1 |

压栈

|  |  |  |
| --- | --- | --- |
| 3 | 2 | 1 |

写入3

|  |  |  |
| --- | --- | --- |
| 3 | 2 | 1 |

|  |  |  |
| --- | --- | --- |
|  | 2 | 1 |

|  |  |  |
| --- | --- | --- |
|  |  | 1 |

读出3

读出2

读出1

出栈

所以入栈就类似向箱子里面放盘子，放很多个。出栈就是拿箱子的盘子，但是拿的这个盘子不是放入箱子的第一个盘子，而是放入箱子的最后一个盘子先拿出来。

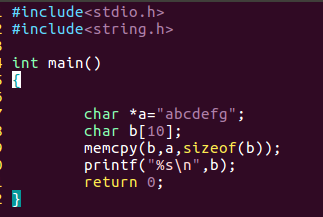
**C语言字符串处理函数汇总**

字符串处理函数在linux系统C库下面，其它操作系统就不清楚了

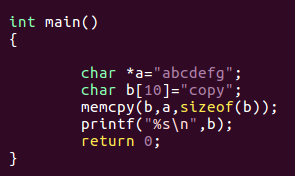
memcpy函数

void \*memcpy(void \*dest, const void \*src, size\_t n);

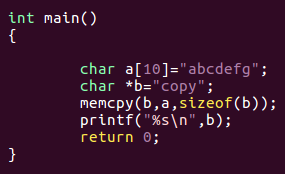
字符串复制函数，从内存区域src拷贝n个字节到dest，必须确保src和dest是内存不重复的区域



将a指针的字符串全部拷贝给b数组存放



就算b数组里面有字符串也会被memcpy覆盖



如果是将数组的字符串拷贝给指针是不行的，拷贝到的对象必须是确定好的内存比如数组

memmove函数

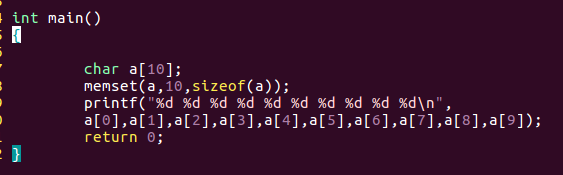
memset函数

extern void \*memset(void \*buffer, int c, int count)

buffer：为指针或是数组

c：是赋给buffer的值

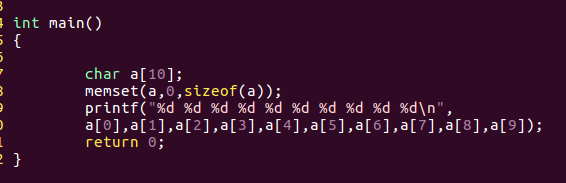
count：是buffer的长度.



就是给数组每个元素赋同样的值



我们一般把数组或者内存全部填0就叫清空内存



全部填0就是清空内存



memcmp函数

int memcmp(const void \*buf1, const void \*buf2, unsigned int count);

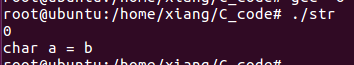
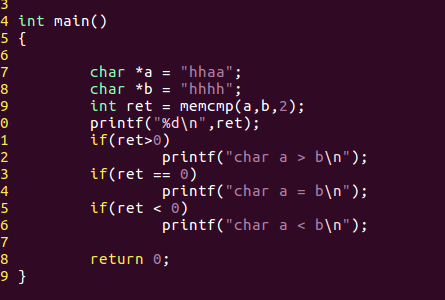
当buf1<buf2时，返回值小于0

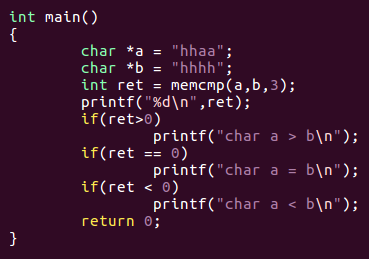
当buf1==buf2时，返回值=0

当buf1>buf2时，返回值大于0

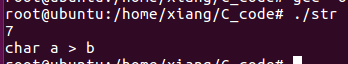
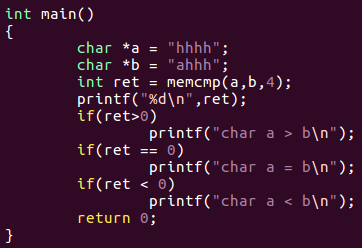
count 是比较字符次数，如果两个字符串变量第一个相等就继续向下比，根据count大小决定向下比好多个。但是一旦发现字符串比较结果为大于或者小于就立即返回，不再向下比较。不管你写多少个count都不会继续向下比较。

我写了count比较两个字符串变量的前两个字符，得到的结果是相等，返回0





我将count改成3，memcmp函数发现比较到第三个字符串的时候，根据ASCII码，h>a，所以根据buf1<buf2返回负数



我将count改成4，但是发现只比较了第一个字符就返回了，所以count成立条件必须是buf1和buf2的字符串相等，才按照count继续向下比较，在比较的过程中出现了大于或者小于就停止对比，直接返回

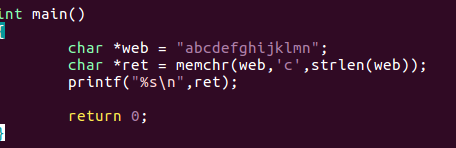
memchr函数

void \*memchr(const void \*str, int c, size\_t n)

str : 要处理的字符串数组，或者指针首地址

c ：我要寻找的字符，找到该字符memchar就返回该字符在数组中的地址，也就是元素下标

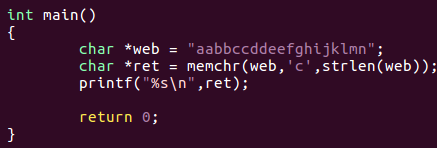
n : 这个字符串有多长



从ret地址开始继续输出后面的字符串

返回找到的字符串地址给ret







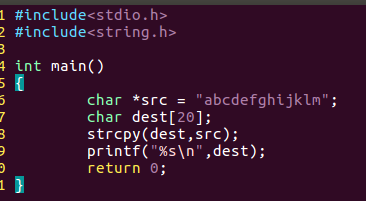
memchar函数就是不管你有多少个相同的字符串，都是以第一个找到的字符串为准，返回它的地址。

strcpy函数

char\* strcpy(char\* des,const char\* source)

//参数：des为目标字符串，source为原字符串。

把source空间的字符串以‘\0’结尾全部复制到des空间内



复制成功

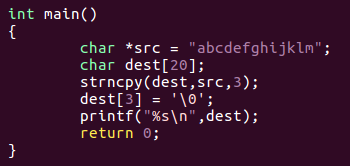
strncpy函数

char \*strncpy(char \*dest,char \*src,size\_t n);

src是源字符存放空间

dest是要将src里面的字符要赋值到的位置

n是要复制多少个



Strncpy复制后的字符不自带’\0’，所以要自己在后面加

strncpy和strcpy相比strncpy是指定要复制多少个字符，而strcpy是全部复制

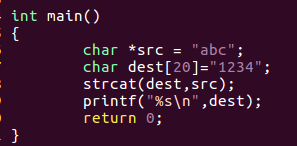


strcat函数

extern char \*strcat(char \*dest, const char \*src);

把src所指字符串添加到dest结尾处(覆盖dest结尾处的'\0')。

相当于就是合并两个字符串



dest开辟的空间要足够存放自己的字符串和合并过来的字符串



Strncat

strcmp函数

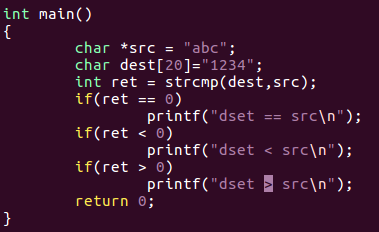
extern int strcmp(const char \*s1,const char \*s2);

当s1<s2时，返回为负数；

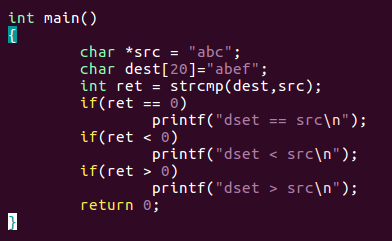
当s1==s2时，返回值= 0；

当s1>s2时，返回正数。

两个[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6)串自左向右逐个字符相比（按ASCII值大小相比较），直到出现不同的字符或遇'\0'为止





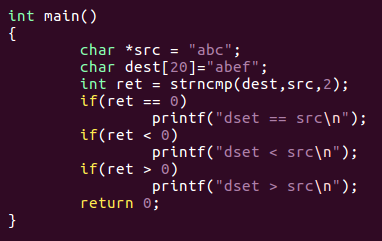


比较两个字符串，前面ab都是一样的就继续向下比较，发现e和c不一样，就对比大小

strncmp

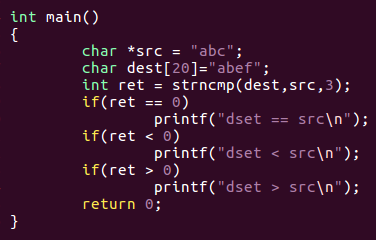
int strncmp ( const char \* str1, const char \* str2, size\_t n );

str1，srt2比较，和strcmp不一样的是，指定比较的个数，n为个数



这里写2就是只比较前面两个，两个字符串前面两个都是ab所以结果是相等





这里3就是只比较前面3个，两个字符串第3个正好不相等，就比较出大小了



strchr函数

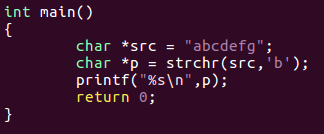
char \*strchr(const char\* \_Str,char \_Val)

str要查找的字符串

val 字符串里面的某个字符

返回查找到字符串某个字符的地址

我从左向右查找第1个b，然后把b的地址返回给指针，指针就从b这个地址开始向后操作



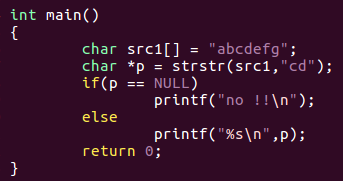


strstr函数

extern char \*strstr(char \*str1, const char \*str2);

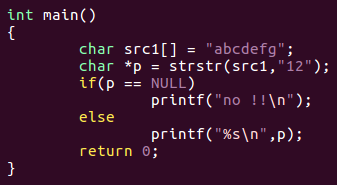
str1字符串地址

str2写一个字符串，然后查找str1字符串里面有没有包含str2写的字符，有的话就截取包含字符的地址给返回值，没有的话就返回NULL



该cd字符串src1包含了的





该12字符串src1不包含

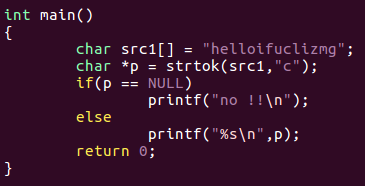


strtok函数

char\* strtok (char\* str,constchar\* delimiters );

str字符串地址

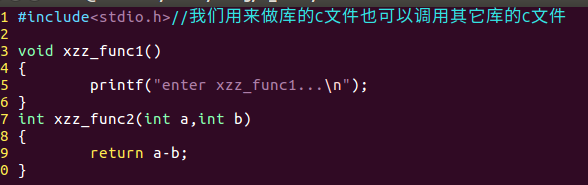
delimiters 填入用字符串里面的什么字符来当做分割符



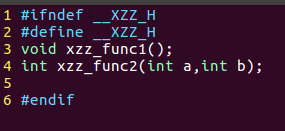
Src1字符串里面c开始后面的字符我就不要了



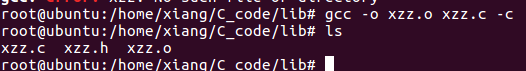
**Linux系统下C文件制作静态库**



准备一份C文件xzz.c

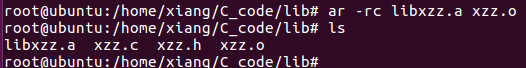
准备一份和C对应的头文件xzz.h

然后我有其它的C文件要调用xzz.c里面的函数，那么我先得把这个xzz.c文件作出静态库



把c文件制作成.o

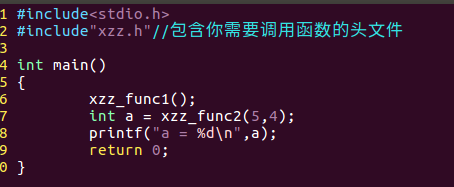
用gcc –o指定编译出来的文件名字，.o后缀是中间文件，因为制作静态库不能用.c文件直接编译成.a，必须经过中间文件.o 所以我们制作出了xzz.o文件



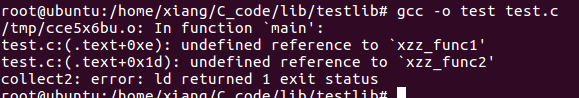
ar –rc libxzz.a xzz.o 就是把xzz.o制作成静态库，静态库都必须要求在你的文件名字前面加lib，所以叫libxzz.a，然后其他程序就可以调用libxzz.a里面的函数了。



我新建了一个test.c文件，在这个文件里面我调用了xzz.c里面的函数，那么我就需要把libxzz.a和xzz.h这两个文件放在和test.c同一个目录下



调用libxzz.a文件里面的函数，也就是xzz.c文件里面的函数



这种.text+….是属于链接报错，编译是通过了，链接出问题

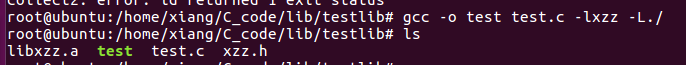
但是链接怎么出错了？ 这是因为你test.c调用了xzz文件里面的函数，所以你编译的时候你要把你调用的文件都指定进来。



因为我们的libxzz.a不在gcc编译器默认寻找的路径库文件里面，所以我们要用-l来指定链接库

Linux系统就是很扯淡，编译库的时候在文件前面加lib，-l链接库的时候要取消库文件前面的lib

但是为什么还是出错呢？，但是这次不是链接错误，是找不到库文件



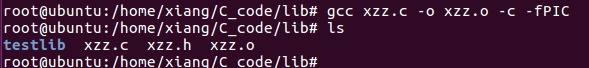
加-L就可以了，这个-L大写L是指定你的库文件在哪个路径里面，gcc编译器它不知道你库文件在哪个路径下，所以要用-L来指定。 -l(小写)指定库文件名，-L(大写)指定库文件存放路径



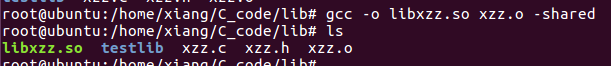
**Linux系统下C文件制作动态库**



我们还是用前面制作静态库的文件来制作动态库



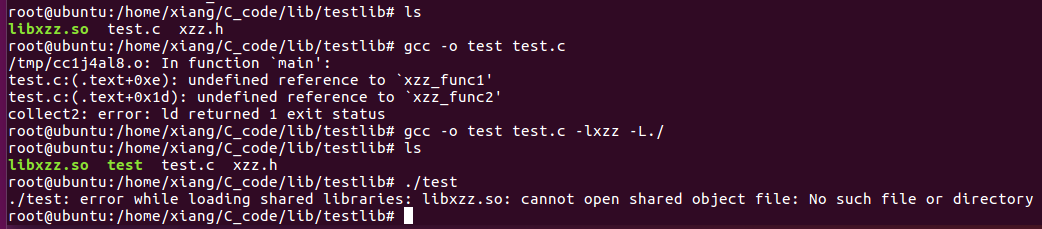
先用-fPIC把xzz.c编译成xzz.o，这个.o具备动态库制作的属性



再用-shared把.o文件编译成.so动态库，libxzz.so就是动态库



和静态库一样把动态库.so和h文件放进调用他们函数的目录里面



运行失败

生成可执行文件

指定库后编译成功

编译失败，和静态库一样，需要指定库

为什么会这样呢？，这就是静态库和动态库的区别了

静态库编译原理：

编译的时候直接把libxzz.a静态库和test.c文件直接编译在一个文件里面

libxzz.a

test.c

test

这样的话静态库编译出来的执行程序就比较大，因为里面包含了静态库整个代码，test.c整个代码，所以你直接-l –L指定就可以编译出来正常运行。

动态库编译原理:

test.c

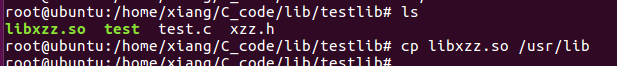
libxzz.so

test

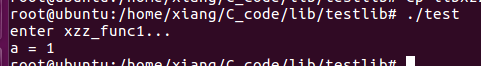
编译成test的时候只是标记了libxzz.so的位置，但是并没有把整个libxzz.so的代码编译进test文件

所以libxzz.so里面的程序没有链接进test，test程序就比静态库小很多

当执行test程序的时候，test会去默认标记的目录位置找libxzz.so文件，但是我们用-L把libxzz.so标记到当前目录下，所以test执行的时候用默认链接库的方法找不到。因为默认链接库不是你的当前目录，而是/usr/lib目录，这是linux系统规定的，你无法用-L去修改动态库的链接位置。



我们把libxzz.so放在/usr/lib目录下



你看这样就能执行了。所以编译的方法没有错，错是错在忘记把.so库文件放在/usr/lib目录下

所以在ARM平台上，你arm-linux-gcc在X86上面编译的库一定要拷贝一份到ARM平台的文件系统下的/usr/lib目录下，然后你在开发板上执行的程序，用到了动态库会自动去开发板的/usr/lib目录下找

在X86平台上你编译的库可能要依赖其他库，所以这个其他库编译出来后要放在交叉编译器的/usr/lib下