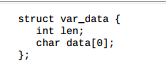
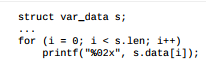
# Linux C内核编程

## GNU C 下的零长度下标数组和变量下标数组

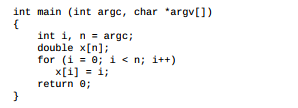
Linux上可用的C编译器是GNU C编译器，其对标准C进行了一系列的扩展。比如它允许使用零长度的数组，这个在定义变长对象的头结构应用时很有用：



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。



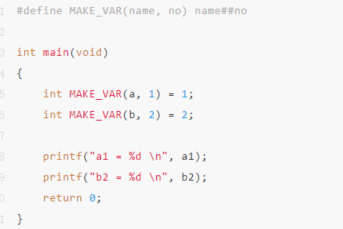
## GNU C支持的可变参数宏

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]

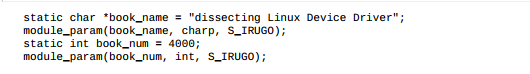
|  |  |  |
| --- | --- | --- |
| 分类 | 选项名称 | 功能 |
| 总体选项 | -c | 对源文件进行编译或汇编。 |
| -E | 对源文件进行预处理 |
| -S | 对源文件进行编译 |
| -o file | 指定输出目标文件名称 file |
| -v | 显示编译阶段的命令 |
| 语言选项 | -ansi | 支持符合ANSI标准的C语言程序 |
| 告警选项 | -W | 屏蔽所有的警告信息 |
| -Wall | 显示所有类型的警告信息 |
| -Werror | 出现任何警告信息就停止编译 |
| 调试选项 | -g | 产生调试信息 |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

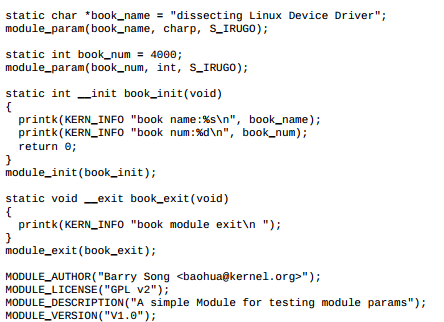
## 如何在模块加载的时候传递参数

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





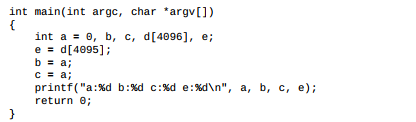
不传递参数值的时候就按照默认参数值。参数类型可以是byte、short、ushort、int、uint、long、ulong、charp（字符指针）、bool或invbool（布尔的反）

## 部分LINUX内核编程的标示含义

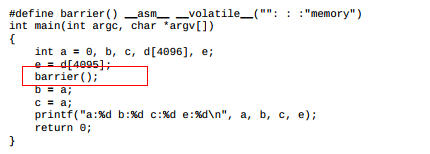
|  |  |  |
| --- | --- | --- |
| 符号 | 含义 | 备注 |
| \_\_init | 用于标示Linux内核模块加载函数。所有标识为\_\_init的函数如果直接编译进入内核，成为内核镜像的一部分，在连接的时候都会放在.init.text这个区段内，同时存放一份函数指针供系统的initcall调用 |  |
| \_\_exit | 标示Linux内核模块卸载函数 |  |
| \_\_initdata | 被定义为该类型的数据只是初始化阶段需要，内核在初始化完后可以释放它们占用的内存 |  |
| /proc/kallsyms | 对应内核符号表，记录了符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

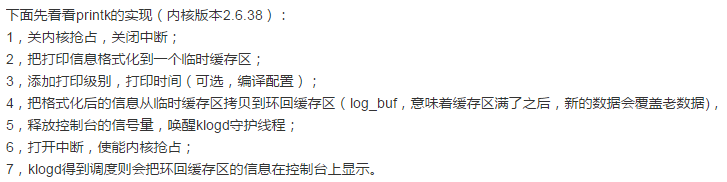
## 可重入函数的特点

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

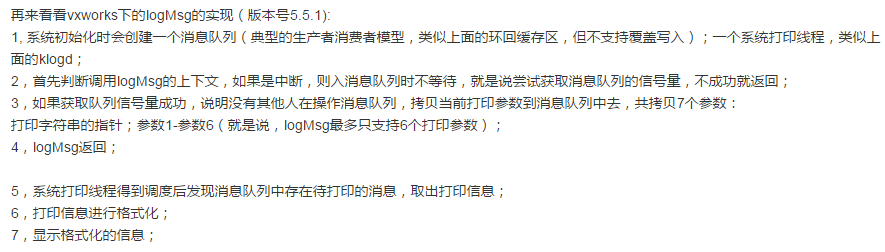
## 中断服务程序中实现打印

中断服务程序中不能调用不可重入函数。函数不可重入多数是因为在函数中引用了全局变量等，所以如printf，它会引用全局变量stdout，而这个是一个通用设备，很有可能因为抢占不到而被阻塞。另外比如malloc/free等操作会引用全局的内存分配表，并且还不好释放，也不能在中断程序中处理。还有浮点运算也是不允许的。

那么如果想在中断程序中进行打印该怎么办呢？其实如linux的printk和vxworks的logmsg都是可以选择的处理，但是前者的处理过程中大部分操作时在printk中实现的，如下所示，这样的话在大量打印情况下可能导致中断丢失，一些依赖于中断的处理可能会不及时，如高精度定时器



而对于logmsg其处理的步骤其实和printk类似但是操作的执行者和执行顺序上存在差异：



由于不需要类似格式化信息的操作，且限制了打印的参数数量，中断中需要处理的就只有参数入队而不需要其他操作，所以其实时性更强。但是它的问题比较明显就是对于记录的信息本身来说约束较强，如打印信息必须是const char\* 类型的，即打印信息不能动态生成的。

两者在大量打印的情况下都会有信息丢失，logmsg是消息队列满了之后就不再入队了，而printk是新数据会覆盖旧的数据。

# 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

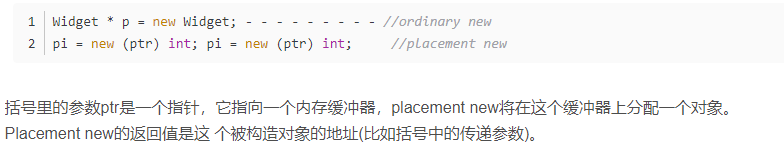
new (p)A(); //调用第三种operator new ，它在调用placement new之后，还会在p上调用A::A()，这里的p可以是堆中动态分配的内存，也可以是栈中缓冲。

**placement new**

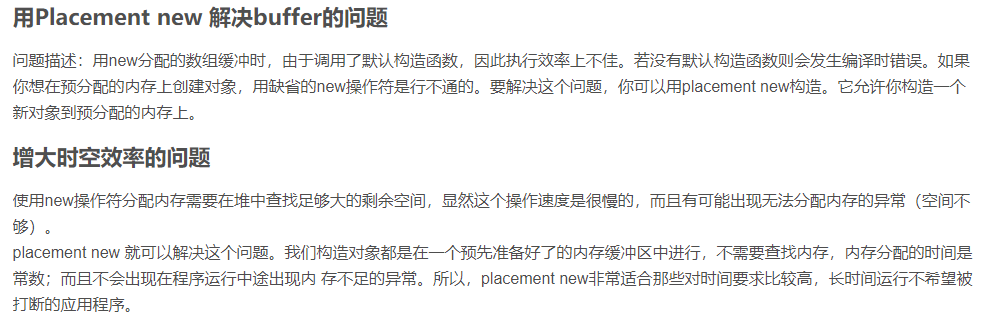
一般来说，使用new申请空间时，是从系统的“堆”（heap）中分配空间。申请所得的空间的位置是根据当时的内存的实际使用情况决定的。但是，在某些特殊情况下，可能需要在已分配的特定内存创建对象，这就是所谓的“定位放置new”（placement new）操作。 定位new操作的定义格式如下：



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



## 常量函数

<类型说明符><函数名>(<参数表>)const;

当一个函数被修饰为常量函数后，其函数中出现的对外部数据的任何写入或修改操作都将被系统检查为出错。其具备以下的一些特征：

1. 常量成员函数不能修改对象
2. 常量成员函数在定义和声明中都需要加上const
3. **非常量成员函数不能被常量成员函数调用，但构造函数和析构函数除外**
4. 常量对象只能调用常量成员函数。

## C++类型转换

### static\_cast

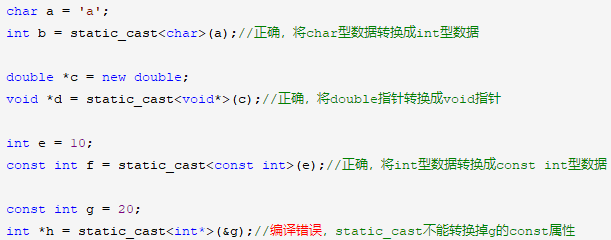
相当于传统的C语言里的强制转换，该运算符把expression转换为new\_type类型，用来强迫隐式转换，例如non-const对象转为const对象，编译时检查，用于非多态的转换，可以转换指针及其他，**但没有运行时类型检查来保证转换的安全性**。它主要有如下几种用法：

1. 用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换。

进行上行转换（把派生类的指针或引用转换成基类表示）是**安全的**；

进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是**不安全的**。

1. 用于**基本数据类型**之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。
2. 把空指针转换成目标类型的空指针。
3. 把任何类型的表达式转换成void类型。



### dynamic\_cast

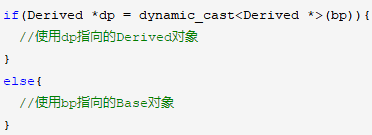
dynamic\_cast主要用于**类层次间的上行转换和下行转换**，还可以用于**类之间的交叉转换**（cross cast）。在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的；在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。**dynamic\_cast是唯一无法由旧式语法执行的动作**，也是唯一可能耗费重大运行成本的转型动作。Dynamic\_cast的基本格式如下：

dynamic\_cast<type\*>(e)

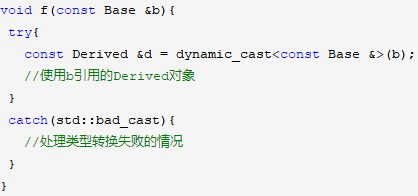
dynamic\_cast<type&>(e)

dynamic\_cast<type&&>(e)

type必须是一个类类型，在第一种形式中，type必须是一个有效的指针，在第二种形式中，type必须是一个左值，在第三种形式中，type必须是一个右值。在上面所有形式中，e的类型必须符合以下三个条件中的任何一个：e的类型是是目标类型type的公有派生类、e的类型是目标type的共有基类或者e的类型就是目标type的的类型。如果一条dynamic\_cast语句的转换目标是指针类型并且失败了，则结果为0。如果转换目标是引用类型并且失败了，则dynamic\_cast运算符将抛出一个std::bad\_cast异常(该异常定义在typeinfo标准库头文件中)。e也可以是一个空指针，结果是所需类型的空指针。



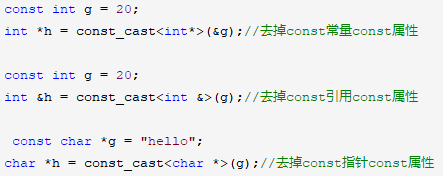
Base为包含至少一个虚函数的基类（基类必须是多态的），Derived是Base的共有派生类。在运行时将基类的指针转换为子类指针。



在引用转换失败时，会抛出std::bad\_cast异常

### const\_cast

用于修改类型的const或volatile属性，这也是C++种唯一可用于修订对应属性的操作符。主要应用有：常量指针被转化成非常量的指针，并且仍然指向原来的对象；常量引用被转换成非常量的引用，并且仍然指向原来的对象；const\_cast一般用于修改底指针。如const char \*p形式。

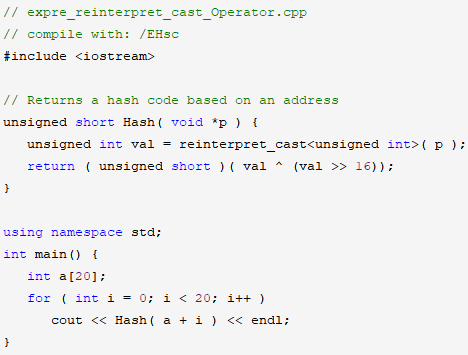


### reinterpret\_cast

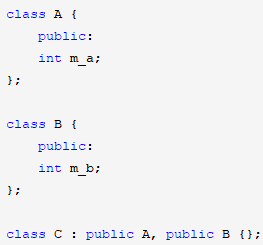
reinterpret\_cast意图执行低级转型，实际动作（及结果）可能取决于编辑器，这也就表示它不可移植。它所转换的类型

必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，再把该整数转换成原类型的指针，还可以得到原先的指针值）。**可以实现不同类型之间的转化**。

reinterpret\_cast意图执行低级转型，实际动作（及结果）可能取决于编辑器，这也就表示它不可移植。另外，使用reinterpret\_cast必须要谨慎，避免出现异常。



static\_cast和reinterpret\_cast的区别主要在于多重继承



前两个的输出值是相同的，最后一个则会在原基础上偏移4个字节，这是因为static\_cast计算了父子类指针转换的偏移量，并将之转换到正确的地址（c里面有m\_a,m\_b，转换为B\*指针后指到m\_b处），而reinterpret\_cast却不会做这一层转换。

## Lambda表达式

详细的说明和解释可以参考c++ primer或者相关的资料，这里只是针对一些基本概念和常用案例进行分析和总结。

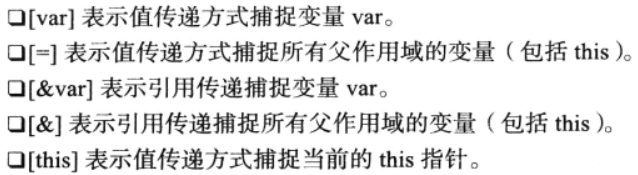
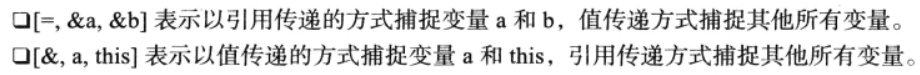
参考资料： https://www.cnblogs.com/Braveliu/p/4231818.html

### 基本格式说明

[capture] {parameters} mutable -> return-type {statement}

（1） （2） （3） （4） （5）

（1）——捕捉列表，编译器根据引出符[]判断接下来的代码是否是lambda函数，它捕捉上下文中的变量以供lambda函数使用

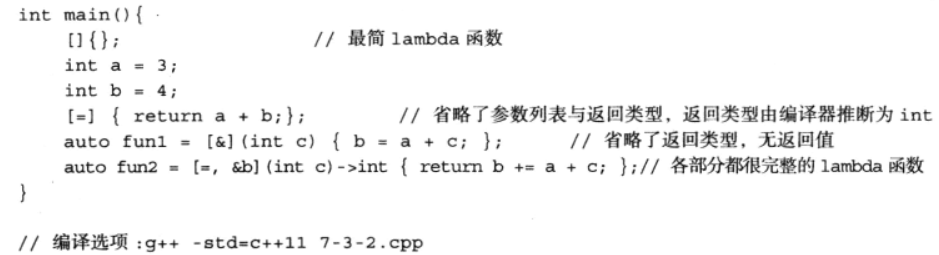
[] 是不捕获任何外部变量的意思。

（2）——参数列表，表征lambda函数的形参列表，如果不需要参数传递，则可以连同括号一起省略

（3）——修饰符，默认情况下，lambda函数总是一个const函数，mutable可以取消其常量性。在使用该修饰符时，参数列表不能省略（即使参数为空）

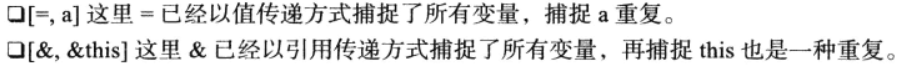
（4）——返回类型。如果不需要返回值的时候也可以连同符号->一起省略。此外，在返回类型明确的情况下，也可以省略该部分，让编译器对返回类型尽心推导。

（5）——函数体，除了可以使用形参外，还可以使用所有捕获的变量。

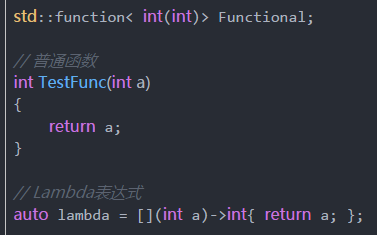
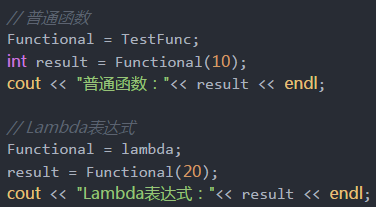


### 实例1——捕获列表的错误示范

捕获列表不允许变量的重复传递，以下方式将导致编译错误。



### 实例2——lambda表达式和function函数的耦合应用

通过std::function对C++中各种可调用实体（普通函数、Lambda表达式、函数指针、以及其它函数对象等）的封装，形成一个新的可调用的std::function对象；它其实类似于一种类型声明，其实我们更常用的是使用using来更为方便的声明它，比如上面例子中的Functional可以写为：

Using func\_ std::function<int(int)>;

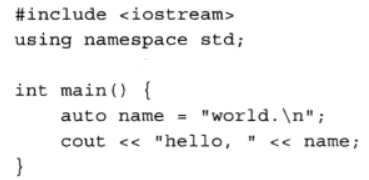
func\_ Functional;

这种类型使得普通函数和lambda表达式都可以直接给它赋值，只需要返回参数和形参一致即可。Lambda表达式不能被赋值，但它可以赋值给函数指针和std::function对象。

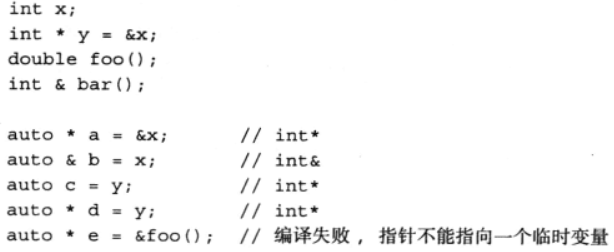
## Auto类型推导

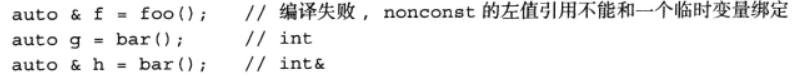
C/C++原本是静态类型的语言，即变量的类型是静态指定的，除非进行了强制转换否则一直保持。而像python这样的动态类型语言不需要静态指定变量的类型：

等价的C++语言代码如下：



### 实例1——auto在指针、引用的应用





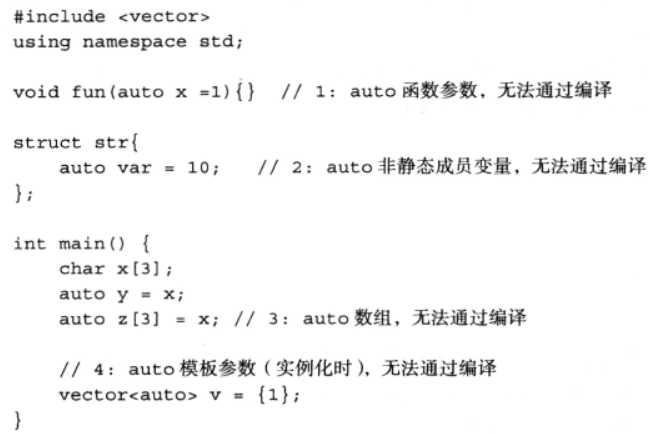
如果要使得auto声明的变量是另一个变量的引用，则必须使用auto &。另外，对于cv限制符（volatile和const），它们代表了变量的两种不同属性（易失的和常量的），c++11标准中规定auto可与它们一起使用，但是声明为auto的变量并不能从其初始化表达式中“带走”cv限制符，如：

Const int i= 10;

auto a = i; // 编译不会报错，但是a的类型是int而不是const int

### 实例2——auto使用上的一些约束

受制于语法的二义性或实现的困难性，auto也有一些使用约束。



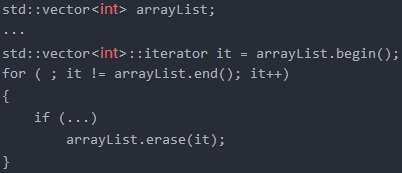
1. auto不能是函数形参类型，如果需要泛型的参数则要使用模板。
2. 对于结构体来说，非静态成员变量的类型不能是auto。虽然有初始值，但是编译器会阻止auto对结构体中非静态成员的推导
3. 声明auto数组是禁止的
4. 在实例化模板的时候使用auto作为模板参数是禁止的

## STL标准库

### 迭代器

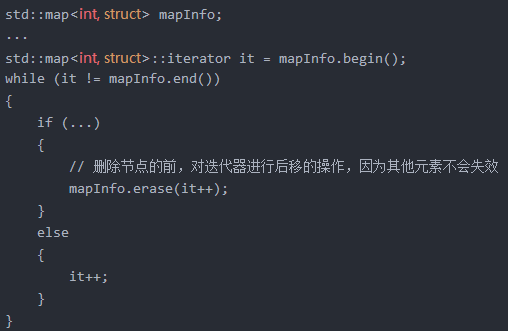
#### 实例1——STL容器在循环中删除迭代器的处理问题

主要问题在于执行迭代器的erase操作，它会使得容器中的元素被删除同时对应的迭代器也变为无效，如果再使用这个迭代器肯定会导致错误，如：



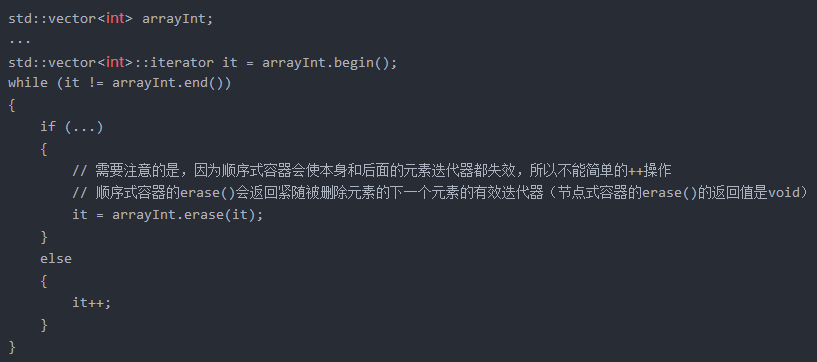
通常的规则是：对于节点式容器（map、list、set）元素的删除、插入操作会导致指向该元素的迭代器失效，其他元素迭代器不受影响；而对于顺序式容器(vector，string，deque)元素的删除、插入操作会导致指向该元素以及后面的元素的迭代器失效。

* 对于节点式/关联式容器（map、list、set）



Map这样的关联式结构使用的是红黑树来实现，插入、删除一个结点不会对其他结点造成影响。

* 对于顺序式容器(vector，string，deque)



顺序式容器使用了连续的内存删除一个元素导致后面所有的元素会向前移动一个位置。其执行后会返回下一个迭代器

这里有一点需要注意：对于list来说，**它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator（map不会），因此上面两种正确的方法都可以使用**。但是下面的使用是错误的：

auto iter = list\_tmp.begin();

while (iter != list\_tmp.end()) { list\_tmp.erase(iter); }

是返回下一个有效迭代器而不是在原有基础上++。

### Std::String类型应用说明

参考资料：<http://c.biancheng.net/view/400.html>

#### String类成员函数c\_str()的意义与用法

语法:

const char \*c\_str();

c\_str()函数返回一个指向正规C字符串的指针常量, 内容与本string串相同.

这是为了与c语言兼容，在c语言中没有string类型，故必须通过string类对象的成员函数c\_str()把string 对象转换成c中的字符串样式。**注意：**一定要使用strcpy()函数 等来操作方法c\_str()返回的指针，比如以下为错误的用法：

*char\* c;*

*string s="1234";*

*c = s.c\_str();* //c最后指向的内容是垃圾，因为s对象被析构，其内容被处理，同时，编译器也将报错——将一个const char \*赋与一个char \*。

合理的用法为：

char c[20];

string s="1234";

strcpy(c,s.c\_str());

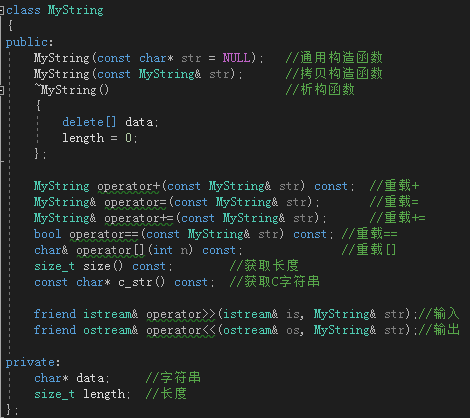
这样才不会出错，c\_str()返回的是一个临时指针，不能对其进行操作

### 独立实现一个string类型

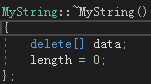
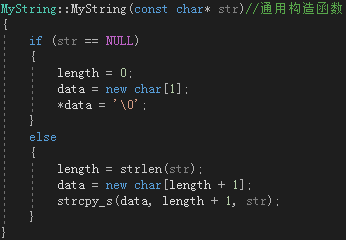
参考资料：https://www.cnblogs.com/zhizhan/p/4876093.html

要实现一个string类，至少需要实现：构造函数、析构函数、拷贝构造函数、重载赋值操作符（如果更深入的话需要实现取址运算符和取址运算符const）。

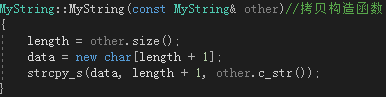
* 基础声明和头文件



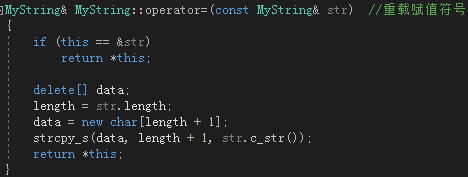
* 默认构造函数和析构函数



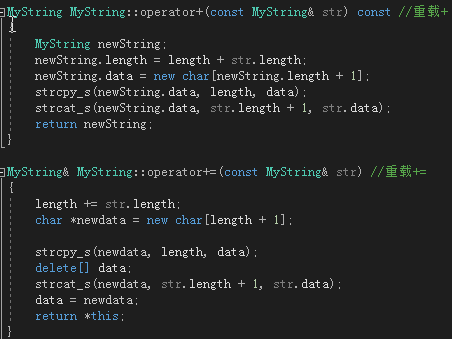
* 拷贝构造函数

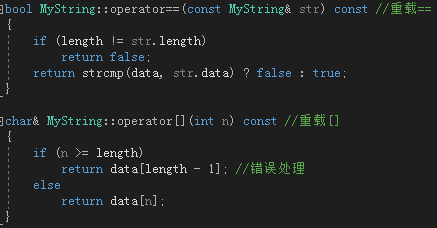


* 重载赋值操作符

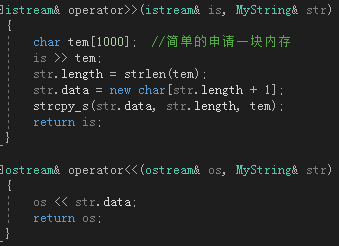


* 重载+ += == []





* 重载输入输出符



### Map的基本用法和注意事项

参考资料：<https://blog.csdn.net/sevenjoin/article/details/81943864>

概念要点：

* STL的一种关联容器，提供一对一的hash
* Map的一个元素有两个成员，关键字key和对应的值value，前者在map中唯一
* Map内部的实现为红黑树，其具有对数据自动排序的功能

基本用法：

* 使用map需要包含 map头文件
* 基本定义：std:map<int, string> personnel; 常用定义：typedef map<int,CString> UDT\_MAP\_INT\_CSTRING;

UDT\_MAP\_INT\_CSTRING enumMap;

* 插入元素方法

mapStudent.insert(pair<int, string>(000, "student\_zero")); //用insert函數插入pair

mapStudent.insert(map<int, string>::value\_type(001, "student\_one")); // 用insert函数插入value\_type数据

**mapStudent[123] = "student\_first"; // 用"array"方式插入**

前两者在插入时如果发现key不唯一会返回失败，返回值时一个pair类型的值，其第二个参数会是false。

最后一种会直接覆盖key对应的value

* 查找元素方法

std::map<int, string>::iterator iter = mapStudent.find("123");

迭代器会指向找到的元素，否则返回mapStudent.end()

* 删除与清空元素

iter = mapStudent.find("123"); mapStudent.erase(iter); //迭代器删除

mapStudent.erase("123"); // 关键字删除，成功返回1反之为0

mapStudent.erase(mapStudent.begin(), mapStudent.end()); //使用迭代器范围删除，等同于 mapStudent.clear()

* Map::size() 返回一个map中元素的个数
* 其它用法可以参考 <http://www.cplusplus.com/reference/map/map/?kw=map> 的member functions

Map注意事项：

* 使用下标进行map元素value获取之前必须进行key是否存在的判定，如果不存在该key，那么下标操作将会向map中插入一个新元素

### 智能指针的基本概念和使用

为了解决野指针、内存泄漏，重复释放的问题，C++11中引入和智能指针的概念来方便管理堆内存。智能指针是利用了一种叫做RAII（资源获取即初始化）的技术对普通的指针进行封装，这使得智能指针实质是一个对象，行为表现的却像一个指针。智能指针在C++11版本之后提供，包含在头文件<memory>中，shared\_ptr、unique\_ptr、weak\_ptr。

**智能指针的原理**

智能指针是一个类，这个类的构造函数中传入一个普通指针，析构函数中释放传入的指针。智能指针的类都是栈上的对象，所以当函数（或程序）结束时会自动被释放

#### Shared\_ptr的基本使用

shared\_ptr多个指针指向相同的对象。shared\_ptr使用引用计数，每一个shared\_ptr的拷贝都指向相同的内存。每使用他一次，内部的引用计数加1，每析构一次，内部的引用计数减1，减为0时，自动删除所指向的堆内存。shared\_ptr内部的引用计数是线程安全的，但是对象的读取需要加锁。

#include <iostream>

#include <memory>

int main() {

{

std::shared\_ptr<int> ptrx = std::make\_shared <int>(10);

std::cout << \*ptrx << std::endl; // 打印10

int a = 10;

std::shared\_ptr<int> ptra = **std::make\_shared<int>(a)**;//使用此方式初始化更好

std::shared\_ptr<int> ptra2(ptra); //copy shared指针引用数量增加

std::cout << ptra.use\_count() << std::endl; // 打印共享指针的引用数量为2

int b = 20;

int \*pb = &a;

//std::shared\_ptr<int> ptrb = pb; //**error**

std::shared\_ptr<int> ptrb = std::make\_shared<int>(b);

ptra2 = ptrb; //assign ptra2 指向ptrb指向的对象使得原有shared指针引用计数减一而当前计数值加一

pb = ptrb.get(); //**获取原始指针**

std::cout << ptra.use\_count() << std::endl;

std::cout << ptrb.use\_count() << std::endl;

}

}

* 不能直接将指针赋值给一个智能指针。例如std::shared\_ptr<int> p4 = new int(1);的写法是错误的，

Shared\_ptr<int> sp1(new int(22)); //这种写法是可以的

* 可以像上面代码中那样使用拷贝或赋值（函数）的方式初始化智能指针。**拷贝使得对象的引用计数增加1，赋值使得原对象引用计数减1（被赋值的那个引用计数减一），当计数为0时，自动释放内存**。后来指向的对象引用计数加1，指向后来的对象。
* Get函数获取原始指针
* **注意不要用一个原始指针初始化多个shared\_ptr，否则会造成二次释放同一内存，像上述例子中ptra ptrb都是原始指针。**
* 避免针对share类指针的循环引用，它会导致堆内存无法正确释放引发内存泄漏。

#### 何时需要使用Shared\_ptr

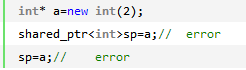
(1) 程序不知道自己需要使用多少对象. 如使用窗口类, 使用 shared\_ptr 为了让多个对象能共享相同的底层数据.

(2) 程序不知道所需对象的准确类型.

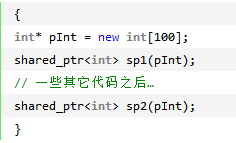
(3) 程序需要在多个对象间共享数据.

#### Shared\_ptr使用的注意事项

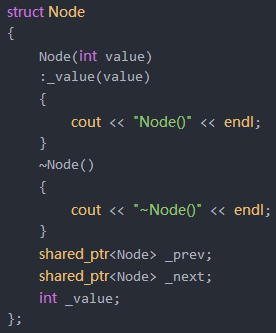
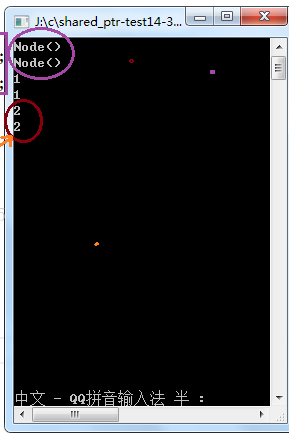
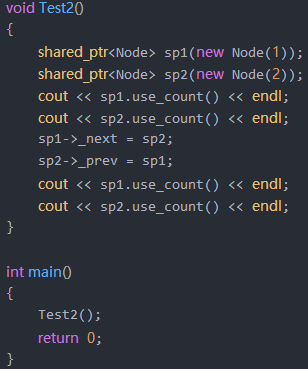
1. 禁止纯指针给智能指针赋值或者拷贝构造



1. Shared\_ptr多次引用统一数据，会导致两次释放同一内存



1. 使用share\_ptr造成循环引用

在上面的示例中，构造的sp1和sp2在出它们的作用域时（即Test2（））都没有被析构，从而造成了内存泄漏。这是因为在进行next赋值时，sp2的计数加一，而pre赋值的时候sp1的计数加一。这样实际上sp1和sp2的引用计数都是2。由于先构造的后释放，后构造的先释放可知，先释放的是sp2，那么因为它的引用计数为2，减去1之后就成为了1，不能释放空间，因为还有其他的对象在管理这块空间（就是sp1的next还指向这个空间）。但是sp2这个变量已经被销毁，因为它是栈上的变量，但是sp2管理的堆上的空间并没有释放。接下来释放sp1，同样，先检查引用计数，由于sp1的引用计数也是2，所以减1后成为1，也不会释放sp1管理的动态空间。

总结起来就是：sp2等待sp1 的next释放，而sp1也在等待sp2的pre释放。如果要解决这个问题，可以将类中的pre和next指针替换为weak\_ptr即可，原理可参考：<https://blog.csdn.net/qq_34992845/article/details/69218843> 因为weak\_ptr是使得

#### unique\_ptr的使用

unique\_pt——“唯一”拥有其所指对象，同一时刻只能有一个unique\_ptr指向给定对象（通过禁止拷贝语义、只有移动语义来实现）。unique\_ptr指针本身的生命周期：从unique\_ptr指针创建时开始，直到离开作用域。离开作用域时，若其指向对象，则将其所指对象销毁(默认使用delete操作符，用户可指定其他操作)。

unique\_ptr指针与其所指对象的关系：在智能指针生命周期内，可以改变智能指针所指对象，如创建智能指针时通过构造函数指定、通过reset方法重新指定、通过release方法释放所有权、通过移动语义转移所有权。

int main() {

{

std::unique\_ptr<int> uptr(new int(10)); //绑定动态对象

//std::unique\_ptr<int> uptr2 = uptr;  **//不能賦值**

//std::unique\_ptr<int> uptr2(uptr); **//不能拷貝**

std::unique\_ptr<int> uptr2 = std::move(uptr); //轉換所有權

uptr2.release(); //释放所有权

}

//超過uptr的作用域，內存釋放

return 0;

}

#### Weak\_ptr的使用

weak\_ptr是为了配合shared\_ptr而引入的一种智能指针，因为它不具有普通指针的行为，没有重载operator\*和->,它的最大作用在于协助shared\_ptr工作，像旁观者那样观测资源的使用情况。weak\_ptr可以从一个shared\_ptr或者另一个weak\_ptr对象构造，获得资源的观测权。但weak\_ptr没有共享资源，它的构造不会引起指针引用计数的增加。使用weak\_ptr的成员函数use\_count()可以观测资源的引用计数，另一个成员函数expired()的功能等价于use\_count()==0,但更快，表示被观测的资源(也就是shared\_ptr的管理的资源)已经不复存在。weak\_ptr可以使用一个非常重要的成员函数lock()从被观测的shared\_ptr获得一个可用的shared\_ptr对象， 从而操作资源。但当expired()==true的时候，lock()函数将返回一个存储空指针的shared\_ptr。

{

std::shared\_ptr<int> sh\_ptr = std::make\_shared<int>(10);

std::cout <<"1--"<< sh\_ptr.use\_count() << std::endl; //打印1

std::weak\_ptr<int> wp(sh\_ptr); // weak\_ptr引发引用计数增加，\_weaks值加一

std::cout <<"2--"<< wp.use\_count() << std::endl; //打印1

std::cout <<"3--"<< sh\_ptr.use\_count() << std::endl; //打印1

if(!wp.expired()){

std::shared\_ptr<int> sh\_ptr2 = wp.lock(); //get another shared\_ptr

\*sh\_ptr = 100;

std::cout <<"4--"<< wp.use\_count() << std::endl; //打印2

std::cout <<"5--"<< sh\_ptr.use\_count() << std::endl; //打印2 触发引用的是shared\_ptr类型所以\_uses值加1

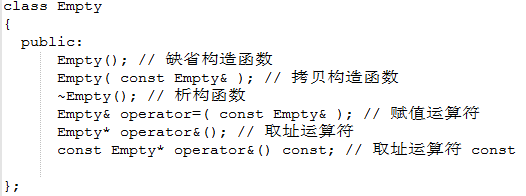
}

}

Weak\_ptr的引用计数增加的是类的 \_weaks值，而share\_ptr 引用计数增加是\_uses。wp.use\_count()返回的是\_weaks值而sh\_ptr.use\_count()返回的是\_uses 值。

## 类、对象、继承、多态

### C++空类的六个函数



类如果只是定义而不是创建类的对象，那么这些默认的函数是不会产生的。

* 拷贝构造函数特征

对于一个类X，如果一个构造函数的第一个参数是下列之一：

X& const X& volatile X& const volatile X&

且没有其他参数或者其他参数都有默认值，那么这个函数是拷贝构造函数，比如：

X：：X（const X&）； //是拷贝构造函数

X：：X（X&， int=1）； //是拷贝构造函数

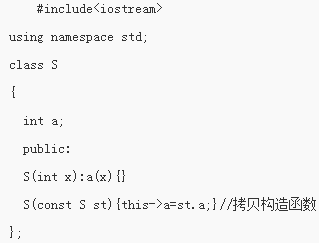
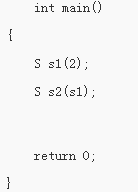
另外，类中可以存在超过一个拷贝构造函数，这个类似于对拷贝构造函数的重载，比如上面的参数中如果只有X&的拷贝构造函数存在，那么不能使用const X或volatile X的对象实行拷贝构造。如果不定义拷贝构造函数，那么编译器会根据上下文决定选择哪一个拷贝构造函数。

### 拷贝构造函数的参数为什么不能是值传递

当我们尝试把拷贝构造函数写成值传递的时候，会发现编译都不通过。比如下面的例子：

F:\codeblock-project\mycpp1\myfirst.cpp|11|error: invalid constructor; you probably meant 'String (const String&)'

这个意思是拷贝构造函数是无效的函数，其实这个是C++编译器的一种门禁，主要为的是防止出现可能的死循环。

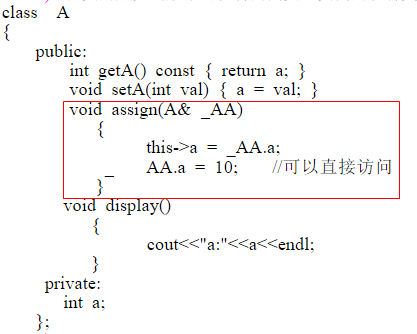
 

比如在上面的例子中，当给s2初始化的时候调用了s2的拷贝构造函数，由于是值传递，系统会给形参st重新申请一段空间，然后调用自身（也就是s1）的拷贝构造函数把s1的数据成员的值传递给st。当调用自身的拷贝构造函数的时候形参又是值传递，那么就会出现不断调用拷贝构造函数的死循环。所以，拷贝构造函数一定不能是值传递，但是是不是const不一定，但是从安全角度来看最好是加上。

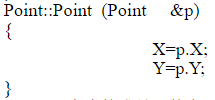
### 成员函数中访问对象的私有成员问题

参考资料：<https://wenku.baidu.com/view/4e13d5343968011ca300912c.html>

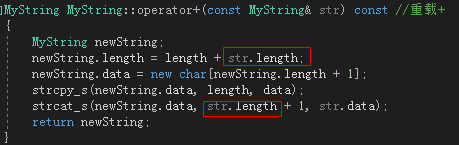
1. 在C++的类成员函数中，允许直接访问该类对象的私有成员变量
2. 在类的成员函数中可以访问同类型实例的私有变量



1. 拷贝构造函数中，可以直接访问另外一个同类对象（引用）的私有成员



1. 类的成员函数可以直接访问作为其参数的同类型对象的私有成员，注意下面的示例中如果是函数中的局部对象变量也可以这样访问



需要注意的是，上面的说法都只限定于同一个类型，即便是子类也不行。

### C++中类访问权限控制

1. private 成员

只能由：该类中的函数及其友元函数访问

不能被任何其他访问，该类的对象也不能访问

1. protected成员

只能由：该类中的函数、子类的函数以及其友元函数访问

但不能被该类的对象访问。

1. public成员

可以被该类中的函数；子类的函数；及其友元函数访问，也可以由该类的对象访问。

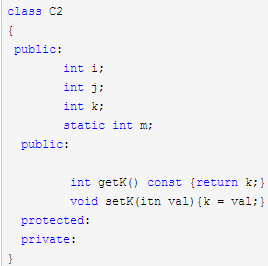
类继承后方法属性的变化如下：

|  |  |  |  |
| --- | --- | --- | --- |
|  | Public | Protected | Private |
| Public继承 | Public | Protected | 不可用 |
| Protected继承 | Protected | Protected | 不可用 |
| Private继承 | Private | Private | 不可用 |

## C++类的实例化对象的sizeof值

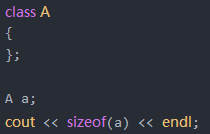
参考：https://blog.csdn.net/houqd2012/article/details/40264943

### （0）几个基本成员的计算

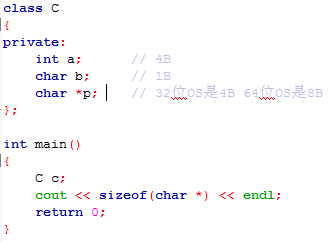


Sizeof(C2) =12 普通成员变量放在栈中，静态属性的成员放在全局数据区，成员函数放在代码区，所以sizeof结果是12.

### （1）空类对象的sizeof值

返回值是1，空类是可以实例化的，它说明编译器需要给他分配内存空间来来指示类实例的地址。这里编译器默认分配了一个字节以便标记可能初始化的类实例，同时使得空类占用的空间最少。

### （2）sizeof对于成员占用空间的计算



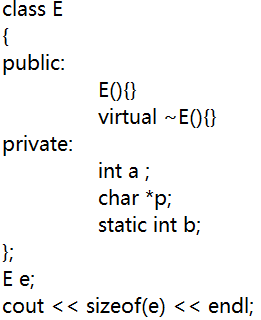
考虑到对齐的问题如果是32位OS那么上面的结果是12，但是如果是64位OS，那么结果是16。

### （3）sizeof中虚函数的空间占用情况

 该代码的执行结果是 12

当类含有虚函数时，（不论是自己的虚函数，还是继承来的），那么类中就有一个成员变量信息：虚函数指针（32位系统下4个字节）。这个指针指向一个虚函数表，虚函数表的第一项是类的typeinfo信息，之后的项为此类的所有虚函数的地址。当类中有虚函数的时候，编译器会为类插入一个不可见的数据并建立一个表，即虚函数表，而这个数据也就是虚表指针。**虚函数表保存了类中的虚函数地址，类似一个数组，其中的每个元素存放的就是类中虚函数的地址。当调用虚函数的时候，程序先取出虚表指针获取虚函数表，然后从这个表里取出对应函数的指针来调用该函数**。我理解编译器在虚函数实现的地方将实现类在虚函数表中进行了绑定和地址填写，推测是利用索引进行的绑定。

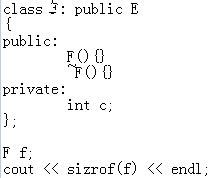
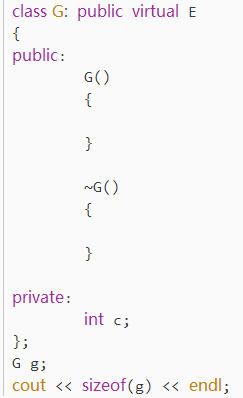
### （4）sizeof下静态成员变量占用的空间

 显示结果为12

静态成员变量是在静态存储区分配空间，它不属于实例的一部分，因此类中的static成员变量占据空间

### （5）派生类对象的sizeof结果

派生类对象的存储空间 = 基类存储空间 + 派生类特有的非static数据成员的空间。**如果是虚继承的话，类对象的存储空间大小 = 基类的存储空间 + 派生类特有的非static数据成员的存储空间 + 每一个类的虚函数存储空间**。

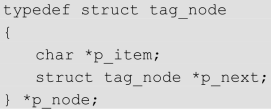
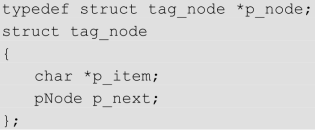
16  20

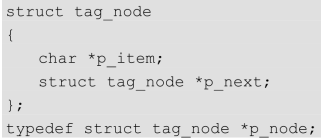
需要注意的是：sizeof(g) = 12(E基类的存储空间) + 4(G特有的非static数据成员的存储空间) + **4（E类的虚函数的存储空间，如果E类中有多个虚函数，只算一次）**）。

# 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";*

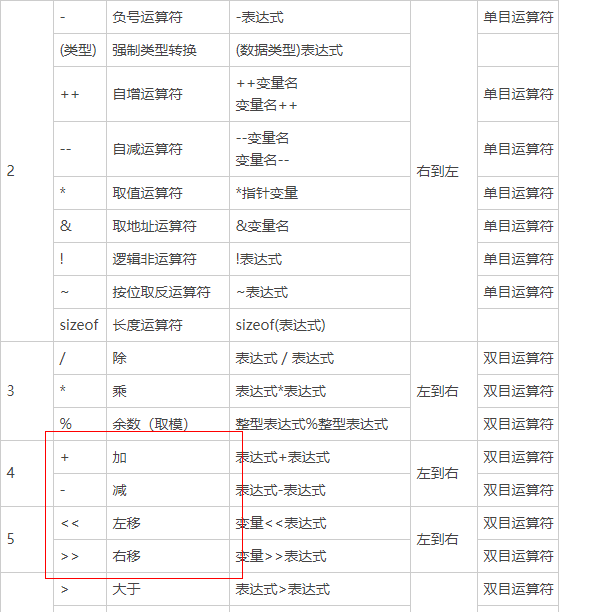
*++ptr1; /// error*

*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

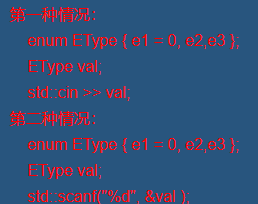


注意，左右移位的优先级是比加减乘除低的，所以下面的表达式可能得不到预期结果：

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

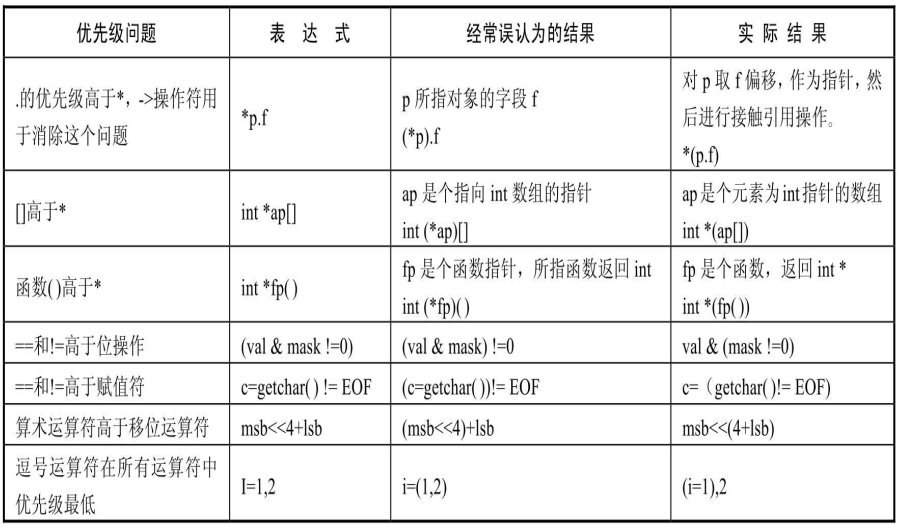
## 枚举变量的两种错误用法

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。

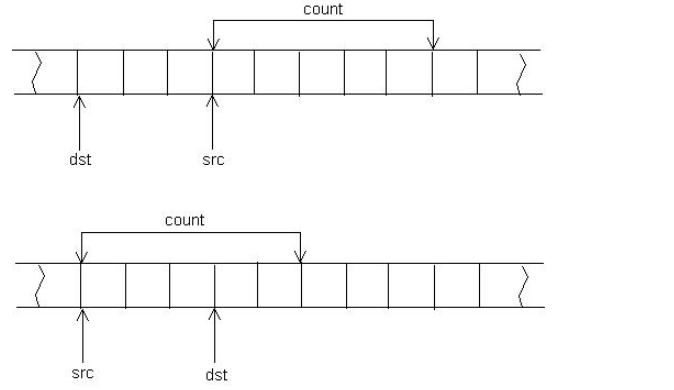
## 运算符优先级易错示例



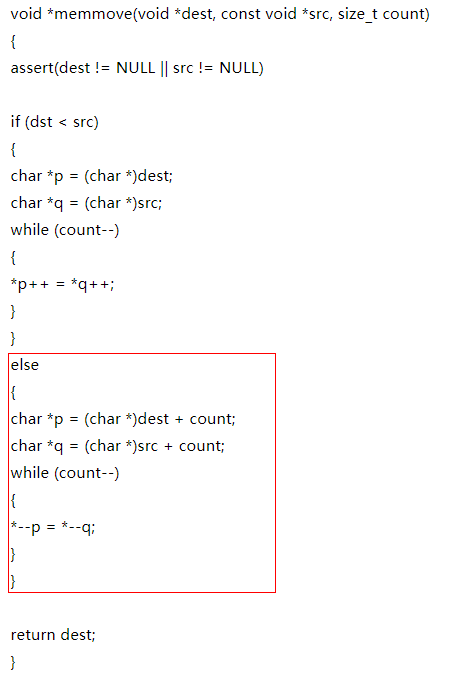
## Memcpy 与 memmove的区别

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

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

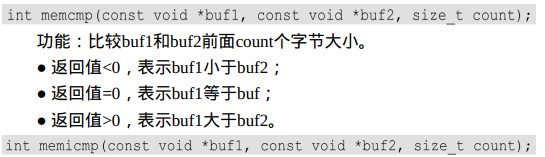


两种都是进行内存拷贝的C库接口，但是move函数在针对上述第二种情况的时候做了地址重叠的保护，



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。

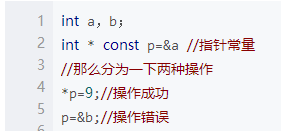
在第一次调用时，strtok()必需给予参数strToken字符串值，往后的调用则将其设置成NULL即可，当返回值变为NULL时说明已经到了末尾，我推测是初始的字符串被存储在了strtok中



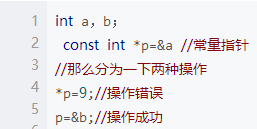
<http://c.biancheng.net/cpp/html/175.html>

## 指针常量、常量指针、指向常量的指针常量

指针常量：本质是一个常量而使用const进行修饰，代表其指针值是确定的，指向的值可变。



常量指针：指向常量的指针变量，其指针值可变（地址可变），指向的内容不可变。



指向常量的指针常量：相当于是上面两者的结合，

const int \* const b = &a;//指向常量的指针常量

## const常量赋值应用

const int para； //c编译器会报告警，c++编译器可能会报错（尤其是支持c++11的编译器）

char \*const para = “hello”

// 这个需要看编译器，vs下面会报错，但是codeblock下的c++11编译器会报warnning warning: ISO C++ forbids converting a string constant to 'char\*' [-Wwrite-strings]|

// C编译器不会报错

char \* p = “aa55”;

// C标准不会报错，但是C++可能会报错，提示常量 error C2440: “初始化”: 无法从“const char [5]”转换为“char \*”或者报告警：warning: ISO C++ forbids converting a string constant to 'char\*' [-Wwrite-strings]|

// 下面的定义方式在C++中也可能会出现上面的现象

char \*p = NULL;

p = “abcd”;



const char\* para;

cosnt char\* const para = “hello”;

// 以上表述是正确的

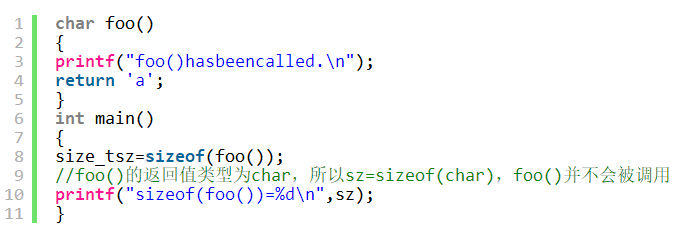
## C语言Sizeof

参考资料：<https://baike.baidu.com/item/sizeof/6349467?fr=aladdin>

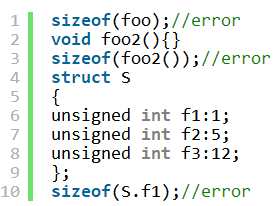
* sizeof计算对象的大小是转换为对象类型来计算的，比如：



* Sizeof对于一个函数调用可以求值，但是结果是函数返回值类型的大小，函数本身本不会被调用



* 函数、不确定类型的表达式以及位域成员不能被计算siezof值

 但是sizeof(s)是可以的

### 结构体的sizeof

* 结构体的sizeof一个重要点就是字节对齐，对齐的准侧有三个：

1)结构体变量的首地址能够被其最宽基本类型成员的大小所整除；

2)结构体每个成员相对于结构体首地址的偏移量（offset）都是成员大小的整数倍，如有需要编译器会在成员之间加上填充字节（internal padding）；

3)结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要编译器会在最末一个成员之后加上填充字节（trailing padding）。

* 结构体某个成员相对于结构体首地址的偏移量可通过宏offsetof()来获取，其定义实现为：

#define offsetof(s,m) (size\_t)&(((s\*)0)->m) //s是结构体类型名，m是结构体成员名

* 结构体的大小等于最后一个成员的偏移量加上其大小再加上末尾的填充字节数目。
* 结构体中存在位域类型的sizeof

struct BF1 {

char f1:3;

char f2:4;

char f3:5;

};

位域类型为char，第1个字节仅能容纳下f1和f2，所以f2被压缩到第1个字节中，而f3只能从下一个字节开始。因此sizeof(BF1)的结果为2。

struct BF2{char f1:3;short f2:4;char f3:5;};

相邻的数据类型不一致，不同的编译器有不一样的返回值，比如VC和GCC就是3，而dev-c++是2

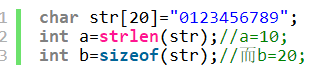
struct BF3{char f1:3;char f2;char f3:5;};

非位域字段穿插在其中，不会产生压缩，在VC6和Dev-C++中得到的大小均为3。

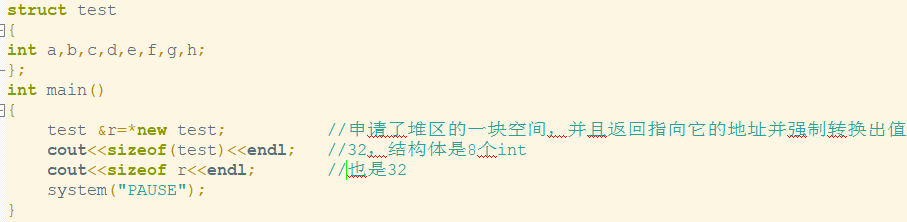
* Union 联合体的sizeof，它是每个成员sizeof的最大值的对齐结果。

### 与strlen的区别

1. sizeof是运算符而strlen是函数，前者可以用类型做参数，后者只能用char \*做参数，且必须是以‘\0’结尾的。并且sizeof可以使用函数做参数（注意是函数整体而不是函数名），相当于是用函数返回值计算（返回值不能是void）
2. 数组做sizeof的参数不退化，传递给strlen就退化为指针了。



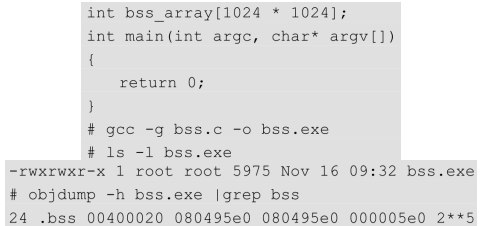
1. strlen的结果要在运行的时候才能计算出来，是用来计算字符串的长度，不是类型占内存的大小。
2. sizeof 操作符，计算的是对象在栈上的投影体积
3. 对引用的处理：sizeof 一个引用得到的结果是 sizeof 一个被引用的对象的大小



## 内存管理

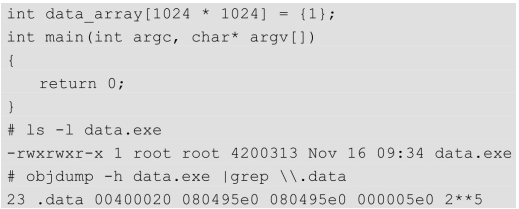
### 内存数据段含义

**.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”

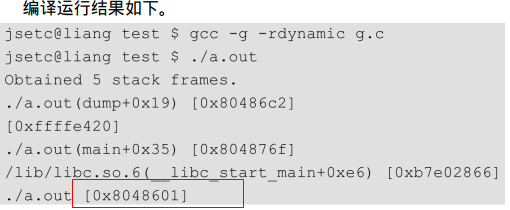
### 段错误调试的基本方法

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）。

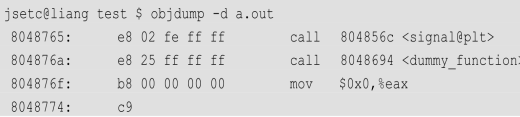
1. 使用GDB进行调试

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



## Memset的用法与使用注意

### 基本介绍

void \*memset(void \*s, uchar ch, size\_t n);

该函数将s中当前位置后面的n个字节 （typedef unsigned int size\_t ）用 ch 替换并返回 s。作用是在一段内存块中填充某个给定的值，它是对较大的结构体或数组进行清零操作的一种最快方法

### 常见问题和错误

* 按字节设置的问题

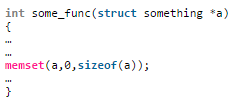
memset函数按字节对内存块进行初始化，所以不能用它将int数组初始化为0和-1之外的其他值（除非该值高字节和低字节相同），其实也就是说初始化的时候指针类型必须指向的是字节，否则可能出现异常，如：

int a[5] = {1};

memset(a,5,sizeof(a)); // 此时 a[0] 中的值并不是5而是 0x050505=84215045

也就是说设置的值只能是0~255之间的值，如果ch的值超过了8bit范围那么函数也只能取低8bit进行设置。

* 针对函数形参为指针时的设置

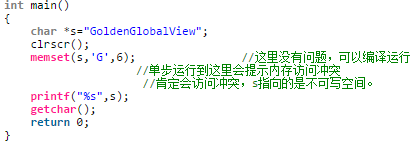


这里的问题是sizeof返回的是指针的大小而不是结构体的大小，这样导致设置的范围不能覆盖整个结构体

* 对于结构体的设置

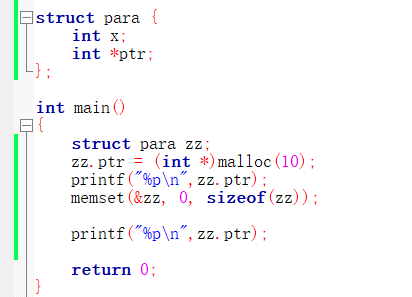
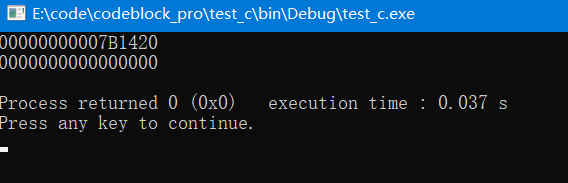
对于结构体 some x; 可以通过 memset(&x, 0 , sizeof(some)) 进行设置

对于结构体数组 some x[10]; 可以通过 memset(&x, 0 ,sizeof(some) \* 10)进行设置。但是以下代码是存在问题的：



由于s指向的是常量空间，所以它被当作常量放入程序存储空间了，如果修改为 char s[] = “…”就不会有问题了

另外，当结构体中包含指针的时候，也需要注意：

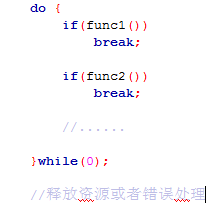
当memset初始化时，并不会初始化p\_x指向的int数组单元的值，而会把已经分配过内存的p\_x指针本身设置为0，造成内存泄漏。

## do{}while(0)的用法

参考链接：http://www.cnblogs.com/flying\_bat/archive/2008/01/18/1044693.html

### （1）消除goto语句

链接中说到的一种场景是在一个函数中要分配多个资源且需要分配一旦出错需要立即退出函数的场景。其实还有一种场景也是比较多的，就是一个函数中调用了多个API，每个API都需要检测返回值的场景。



### （2）宏定义中的使用

我们在很多代码的宏定义中通常看到使用do{…}while(0)结构，这个看似多余的处理实际上是为了使得代码更加简洁和通用而进行的处理。比如下面的例子：

#define SAFE\_DELETE(p) do{ delete p; p = NULL} while(0);

if(NULL != p) SAFE\_DELETE(p)  
else   ...do sth...

如果去掉do while会出现错误，如果给宏加上{}，那么else语句无法通过编译。当然还可以使用if(…){}else{…}的方式避免出现类似的情况。但是这样做需要所有代码都维持这样的规范，并且对于一些库函数或代码来说这样写并不通用，即不能够兼容所有的场景，所以使用do…while(0)是最好的选择

# Makefile

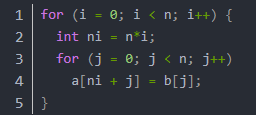
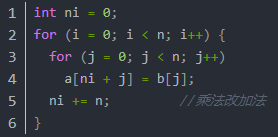
参考文档：<https://wiki.ubuntu.org.cn/%E8%B7%9F%E6%88%91%E4%B8%80%E8%B5%B7%E5%86%99Makefile>

# Clean code

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

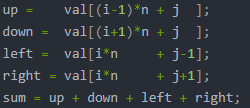
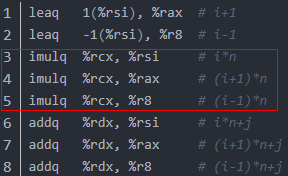
## 减小程序计算量

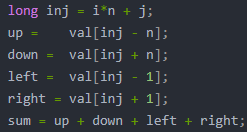
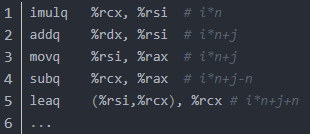
以加法替代乘法、以位移替代乘法

## 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

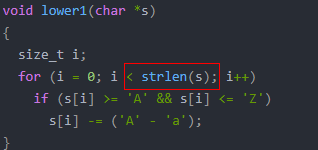
 汇编结果

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

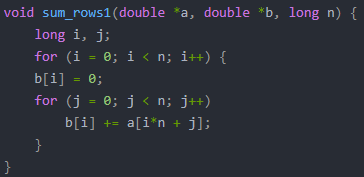
1. 有些计算或者表达式不需要有自变量或者不需要随循环索引变化的最好就在循环体外执行，避免影响

循环执行的效率

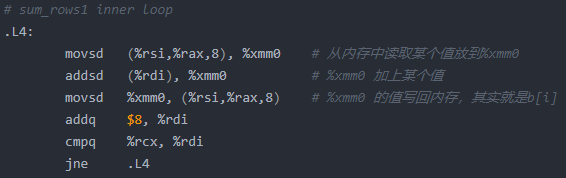


在字符串长度大于100000时，程序运行时间大涨，这个和strlen计算本身有关，其时间复杂度为O(N)，每次都调用相当于是O(N2)的复杂度

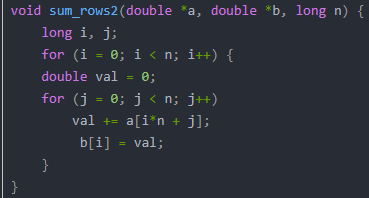
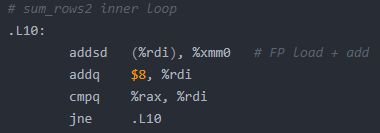
1. 有些循环中存在不必要的内存引用，比如以下示例



汇编结果

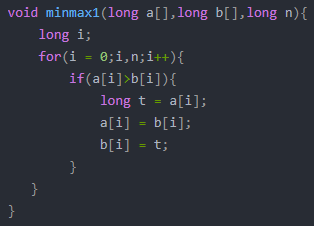
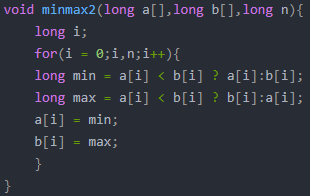


内循环每次都会去读取b[i]的值，然后再把b[i]写回内存 ，改进建议就是设置临时变量进行计算然后再赋值

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。

# 驱动代码技术与技巧

## 实现特性代码间的耦合

如果特性A和特性B需要配置同一表项的同一个字段，那么可以使用以下两种方法进行代码间的解耦。

* 采用状态机耦合处理

特性A和特性B各自独立向状态机注入状态，最终获取运算后的状态设置表项字段，解除特性间编码实现上的接口。例如这个字段是一个valid字段，状态机的初始状态是0，A、B进行设置的时候并不是直接设置而是将它们的设置值记录进行运算之后再将运算结果设置到这个字段。如何进行运算需要看这个字段的含义和特性A、B的需求。

* Notify机制

采用类似notify机制通知耦合特性需要进行对共同资源的操作，这种方式，代码不存在直接调用，但是实现逻辑上仍然关联

## 驱动软件系统与上层控制系统的接口注意事项

驱动软件和上层控制系统的接口通常被称为接口适配层（IAL），对于其定义有以下事项需要关注：

* IAL函数的接口参数不可以使用map、vector等STL数据类型，
* 不能直接在原有接口类中已有虚接口的中间（比如FUNC\_A和FUNC\_B之间）新增虚接口FUNC\_C，因为这些虚接口都是以地址的形式存放在虚表中的，如果单板驱动未和主控同步那么可能上层调用的接口是FUNC\_C实际调用到的却是FUNC\_B。这一情况主要存在于解耦板和主机之间，因为解耦单板或者解耦模块上的软件是独立的应该能够无缝兼容升级或者降级后的主机软件。其实在原有类中新增接口都是不对的，应该新建一个类来实现。