C++编程学习CQIU

2017年4月17日星期一

目 录

[1. 模板标题1 3](#_Toc1004646303)

[1.1 模板标题2 3](#_Toc1787875667)

[1.1.1 模板标题3 3](#_Toc9098727)

[1.1.1.1 模板标题4 3](#_Toc970603373)

[2. C++学习 3](#_Toc1753344490)

[2.1 OO训练营题解 3](#_Toc888260433)

[2.1.1 停车场问题 3](#_Toc1985968270)

[2.1.1.1 需求描述 3](#_Toc585955829)

[2.1.1.2 答题 4](#_Toc2122058697)

[2.2 C++学习整理 4](#_Toc296927350)

[2.2.1 C++学习思想 4](#_Toc358320894)

[2.2.1.1 C++代码编写原则 4](#_Toc1906015161)

[2.2.1.2 从编译器角度理解C++-李宝亮-清华 4](#_Toc1507191496)

[2.2.2 C++开发中的那些坑 5](#_Toc2125734551)

[2.2.2.1 适度使用宏，尽量杜绝使用字符串拼接宏## 5](#_Toc1185907813)

[2.2.3 C++中零散知识点总结学习 7](#_Toc637284828)

[2.2.3.1 const学习 7](#_Toc1973899259)

[2.2.3.2 函数的返回值。 8](#_Toc1341612606)

[2.2.3.3 构造函数私有化，如何派生对象 9](#_Toc493338468)

[2.2.3.4 虚函数表 9](#_Toc1561168792)

[2.2.3.5 &符号的使用,类型右边为引用，对象左边取地址。 10](#_Toc2001981495)

[2.2.3.6 函数指针学习。 10](#_Toc1353281579)

[2.2.3.7 <<操作符和友元函数 11](#_Toc1392015317)

[2.2.3.8 构造函数只有一个参数时，可以隐式转换 11](#_Toc1361901674)

[2.2.3.9 文件输入输出流学习 11](#_Toc1506182099)

[2.2.3.10 Int转换成string的方法 12](#_Toc779572290)

[2.2.3.11 string转换成int的方法 12](#_Toc175184846)

[2.2.3.12 引入空指针类型的关键字nullptr 12](#_Toc1631187685)

[2.2.3.13 所有类型的指针占据空间大小相同 13](#_Toc930093055)

[2.2.3.14 友元函数声明问题 13](#_Toc476791920)

## 模板标题1

见下文

### 模板标题2

见下文

#### 模板标题3

见下文

##### 模板标题4

见下文

###### 模板标题5

见下文

模板标题6

见下文

模板标题6

见下文

模板标题7

见下文



对于WPS来说，可以从”文件”-->”页面设置”-->”文档网络”，然后选择“无网络”

## C++学习

具体内容见下：

### OO训练营题解

主要整理学习过程中零散的知识笔记，由点穿线，以线带面，进而系统化的学习方法。

#### 停车场问题

##### 需求描述

1.停车管理员可以存放客人车辆，取回客人车辆

2.建立一个停车场，可以停放多辆车 在停车场

停车场是空还是满，能停多少量车，统计停多少量车

多辆车，只需写2，3辆即可，停车场数量不能写死，弄一个变量，停车场停车的数量

3.停车场从1个变成多个，停车管理员按顺序停放车辆，即一个停车场停满了放入下一个停车场

4.停车场管理员可以选择那个停车场空位多，将车停入那个停车场

5.扩建为多个停车楼，每个停车楼有多个停车场，每个停车楼有一个停车管理员，

原管理员升级为经理,停车经理可以随机将车辆分配给停车管理员，管理员可以按照任意策略停放车辆。

（包括2中方式，随机和顺序存储）

6.增加打印功能，可以打印出当前经理下的所有管理员管理的停车场停放情况（车辆、空位数、空置率）

按照空格缩进划分层次

打印格式：

M（经理）：车辆个数 空位数 空置率

A1（停车管理员）： 车辆个数 空位数 空置率

P1（停车场）：车辆个数 空位数 空置率

P2（停车场）：车辆个数 空位数 空置率

A2（停车管理员）： 车辆个数 空位数 空置率

P3（停车场）：车辆个数 空位数 空置率

P4（停车场）：车辆个数 空位数 空置率

7.增加新的打印功能，打印成xml格式

##### 答题

#### OO训练营visitor的技巧

感觉像是一种偷梁换柱的做法，visitor的模式，表面上看是保护了封装性，但是深深的存在欺诈行为。

##### 需求描述

略

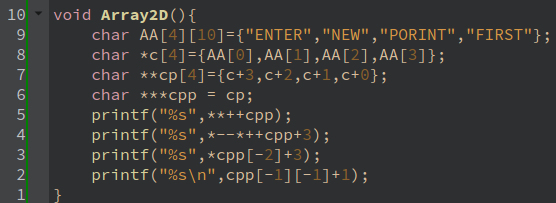
### C/C++专项小例题

具体内容见下：

#### 专项例题

##### 多级指针问题

###### 问题描述



求取这段代码的打印信息。

###### 基本公式

c[n] = \*(c+n)，指针c+n再取值后，就是c[n].

###### 基本规律

不论是多少级指针，形像这种



它表示的含义都是类似这种，

cp[0] = c+3,

cp[1] = c+2

cp[2] = c+1

cp[3] = c+0

也就是说，你线别管 char 后面带多少个\*了。

###### 进行推导求解

Step1：求这行打印：



表示cpp=cp，然后cpp是一个三级指针。

++cpp--> cpp = cp+1,

\*cpp = \*(cp+1) = c+2,why? \*(cp+1)=cp[1],查表，根据第7行，为c+2

\*\*cpp = \*(c+2)=c[2],接着查表，就是打印“PORINT”

注意状态，此时cpp=cp+1了。

Step2：求这行打印：

分析++cpp, 则 cpp=cp+2

分析\*(++cpp) 则 \*cpp=\*(cp+2)=cp[2]，查表=c+1

因此\*(++cpp)=c+1

分析--\*(++cpp)=--(c+1)=c

分析\*--\*(++cpp)=\*c=c[0],注意它是地址

分析\*--\*(++cpp)+3 = c[0]+3,表明它是地址c[0]开始再加3个字节，就是--“ER”

注意状态，此时cpp=cp+2了。

Step3: 求这行打印：

分析cpp[-2] = \*(cpp-2)=\*(cp+2-2)=\*cp=c+3

分析\*cpp[-2]=\*(c+3)=c[3]

分析\*cpp[-2]+3=c[3]+3 🡪”ST”

注意，cpp指针没动，还是cp+2的位置。

Step4:求这行打印：

分析cpp[-1]=\*(cpp-1)=\*(cp+2-1)=cp[1]=c+2，查表

分析cpp[-1][-1]=(c+2)[-1]= \*(c+2-1)=\*(c+1)=c[1]

分析cpp[-1][-1]+1= c[1]+1，表明是字符串“NEW”第1个位置，就是从E开始，就是

“EW”

因此整理后：得到 “PORINTERSTEW”

###### 总结

可以知道，根据基本公式和基本规律原则，对于这种类似的题，基本上一网打尽。

### 设计模式UML

在UML类图中，常见以下几种关系：泛化(Generalization),实现(Realization),关联(Association),聚合 (Aggregation),组合(Composition),依赖(Dependency)。

注意，这些关系，其实都是依赖哦。

关系强弱顺序 泛化>实现>组合>聚合>关联>依赖

这些关系是由强到若的过程。

泛化是一种is A的关系

像聚合，组合之类的，是has A的关系，能用若的依赖，就用若的，影响比较小

总体原则：个人理解。

实线，耦合性强

虚线，耦合性若

实心，耦合性强

空心，耦合性若

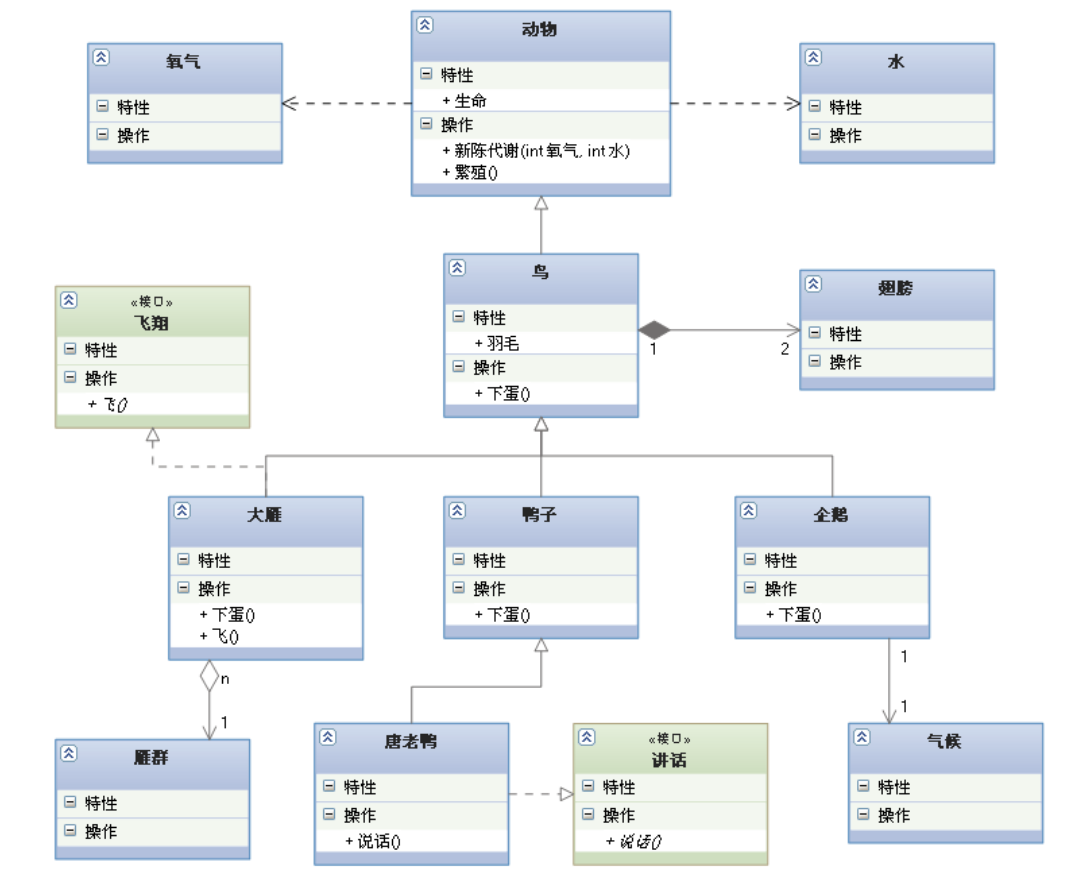
三角形箭头，表示继承的意思。

普通箭头，非继承了

离箭头近的一方，是被拥有者，被动的，带有部件之意，后者说是低级的，原始的，弱势的一方，性格比较低调，从事被动。

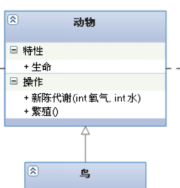
#### 总体图

总体图如下：



#### 泛化关系

箭头指谁，谁就是父亲，注意是三角形箭头，代表寻根的意思，指向根。

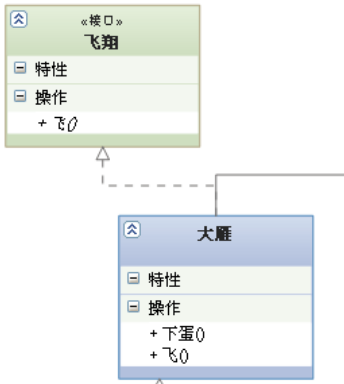


动物派生出了鸟，因此鸟的根是动物。

#### 实现关系

是一种类与接口的关系，它表示类是接口所有特征和行为的实现。

它是这种，带三角形箭头的虚线。



注意和泛化的区别，在于用虚线代替了实线，why？想想，接口都是纯虚函数啊，当然是抽象了，因此是虚线，箭头方向，都指向接口，因为他们的根都是这个接口了，注意这个是飞翔的接口。

#### 关联关系

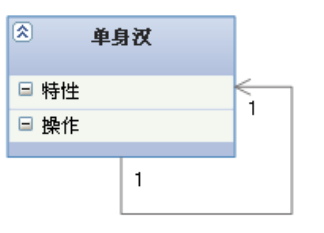
是一种拥有关系，它使得一个类知道另一个类的属性和方法，

代码体现，就是成员变量。

带箭头，指向被拥有者，比如课程被学生拥有，因此如下图所示，1个学生拥有n门课程



还有一种自关联：

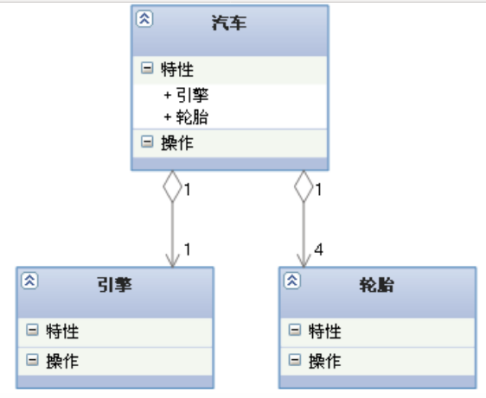


#### 聚合关系

是一种整体和部分的关系，部分可以离开整体单独存在，比如轮胎和引擎都属于汽车一部分。

特别注意：聚合其实也是一种关联关系，从语法上无法区分，必须考察具体逻辑关系。

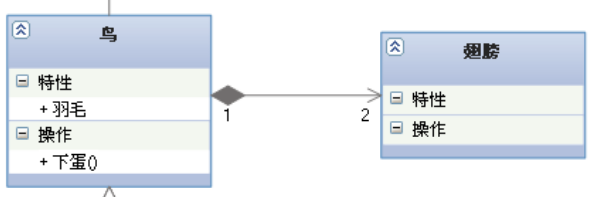
这样理解，箭头指向部件，我拥有部件。反向想就是菱形了，注意是空心的。



#### 组合关系

是一种整体和部分的关系，部分不能离开整体单独存在，比如鸟儿和翅膀的关系，翅膀被鸟儿拥有，箭头指向翅膀，好别扭，没办法，这是老美的思维方式，先局部，后整体。

后面菱形是实心的。

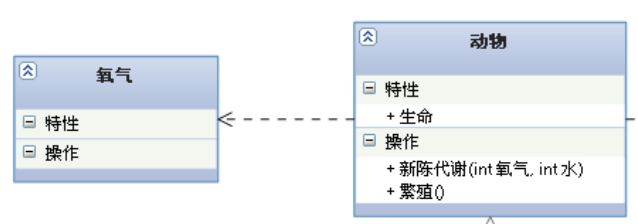


#### 依赖关系

是一种使用的关系，即一个类的实现需要另一个类的协助，尽量不要使用双向的互相依赖。

代码体现：局部变量/方法的参数或者对静态方法的调用，比如动物呼吸需要氧气，这个就是一种依赖，氧气被动物需要，但是氧气不属于动物。

注意是虚线，箭头指向被动方，这个算是最小的依赖了，因此用虚线吧。



### C++学习整理

主要整理学习过程中零散的知识笔记，由点穿线，以线带面，进而系统化的学习方法。

#### C++学习思想

##### C++个人总结随想

OOP中类的封装，继承和多态是非常伟大的想法。

做一件事情必须有依赖，解除依赖的办法是，提取稳定的公共的依赖，这个过程叫做抽象，也叫建模，最稳定的依赖就是抽象。

最稳定的抽象，演化成了接口，变成接口以后，大家都根据这个接口派生新的子类。

那么这些子类就有了共同的父类，它爹妈。

##### C++代码编写原则

1.代码一定要简洁

2.头文件要职责单一，尽量少的头文件，using namespace std，类似这种名空间，要谨慎使用。

3.关于重复的理解：描述相同知识的代码，叫重复，描述不同知识的，不叫，即便它功能类似。

4.使用gtest库，测试驱动开发的方法

5.==这种运算符，可以声明成友元函数。

6.一般Class 声明的类，首字母大写，比如class Mile

7.测试用例，开发用例，以及库，最好分开，比如gtest,src,testcase等等

8.对于私有成员的访问，友元函数是一种方法，不过也要慎用。

##### 从编译器角度理解C++-李宝亮-清华

C++代码学习过程中，如果从编译器的角度去看待问题，那么一定会写出高质量的C++代码。

###### 经典语录-过早的优化时万恶之源

过早的优化是万恶之源，why，

原因1：开始写代码的时候，每前进一小步，都要进行优化的化，会使得代码通用性变差，功能可拓展性变差，最后导致的结果是进度超慢，拖着优化一起做，有可能最后完不成项目。

原因2：每前进一小步都要优化，即使最后实现功能，也没有占到全局的观点进行优化，有可能最后发现算法，结构本身有很大的优化空间，最后导致的结果是，顶层做了优化，之前底层的小优化可能直接删掉，白白浪费时间做了小优化。

###### 内存泄漏问题简谈

内存泄漏问题：本质上是上一次程序运行开始时，申请了一段内存，一般由一个指针来指向一段内存空间，比如int \*a=new int[100]，然后程序运行结束后，这个指针没有释放空间，（如何释放空间 delete []a即可），那么程序运行结束后，这段空间依然占用着呢。下一次程序再执行时，又从操作系统上要了一段新的内存空间，同样的执行后，这段内存空间也没释放，久而久之，系统上一样没有足够的内存空间供程序使用了，因此程序执行崩溃。

问题1：为啥需要释放内存呢，编译器不能自动检查代码，在程序执行结束后释放空间么？

答：像这种方式

Int num = 1234;

int \*a=new int[num]，一般而言，num是一个变量，它执行时才决定具体数值是多少，因此在编译阶段，编译器无法知道到底申请了多少内存空间，也就没有办法释放该空间了。

问题2：比如我们程序中定义了

Int a,b,c

Int array[100]，为啥程序执行结束后，不需要释放空间呢？

答：很显然，a,b,c以及array，申请了多少内存空间都是确定的，因此在程序执行结束后，编译器会很聪明的帮助大家把这些变量占用的空间自动释放，因此就不需要代码开发者去释放空间了。

问题3：我们释放了指针指向的内存空间，此时这个指针指向了不确定的位置，叫野指针，那么这个野指针不需要释放么？

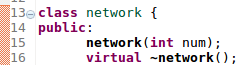
答：请注意，指针这个东西，它存的内容是地址，它本真也是一个变量，当我们销毁它的地址后，它的变量属性没变，既然是变量，就也像int a;这样，编译器会自动释放哦，不需要我们来管。

###### 如何避免内存泄漏

一般程序中new了多少次，就要delete多少次，他们是成对出现的，大家不要忘记这一点，代码的安全编程很重要，代码不但要写得干净，而且更要运行时更加安全。

###### C++中析构函数问题简谈

析构函数一般强制定义成虚函数



一般C++中的析构函数都是虚函数，虚函数的好处是，它的派生类时，派生类的析构函数执行时，先执行它本身的析构函数，如果不定义成虚函数，就没有这个特性。

###### 多继承与单继承问题简谈

多继承问题会出现各种诡异的现象，强烈建议使用单继承策略，这个问题，对于袁英杰的OO思想来说，他的团队使用过很多次多继承，需要两面看待。

#### C++开发中的那些坑

见下文

##### 适度使用宏，尽量杜绝使用字符串拼接宏##

接下来以OO训练营用金老师代码为例，说明使用宏的危害。

第一点，它使得代码的可读性变差，

第二，它会让宏参数调用过程中容易出问题。

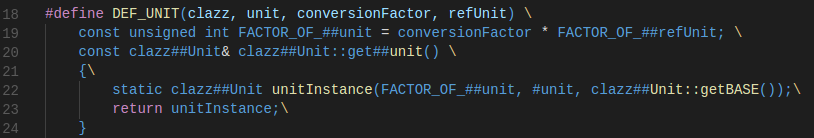
宏的根本问题在于，它没有进行数据类型检查的能力，导致了很多问题。

###### 定义宏参数

比如定义一段如下代码，希望传递的参数如下：

clazz,表示一个类，unit,表示一个单位，conversionFactor表示一个因子

refUnit表示相对单位，他们在编译器初始看来，都是字符串的。



Why是字符串呢？

因为编译器认为，当这个参数在宏参数体内部有##之类的拼接符号的时候，就认为它是一个字符串了。

还有一种情况是，这个宏参数首次出现，也通常被认为是字符串。这是C++编译器的规范。

###### 将宏参数进行接口调用



我们定义一个新的宏USER\_DEF，其实就是把宏参数DEF\_UNIT进行了二次包装。

如果我们这样直接使用DEF\_UNIT，是没有问题的。因为第1和第2个参数在宏定义体内部都用到了##连接符号，见19,20行代码。因此都被认为是字符串，因此能直接使用。



而事实上，TSP这个参数是一个宏

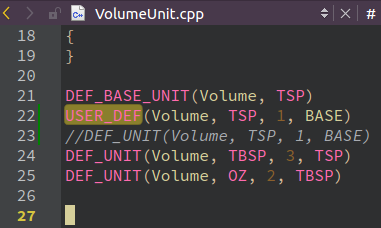


经过翻译后，等价于这个参数

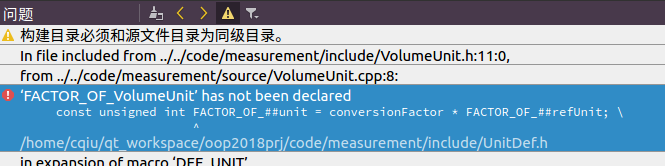


###### 宏参数接口调用后的问题

如果我们按照这种方式调用，将USER\_DEF打开，



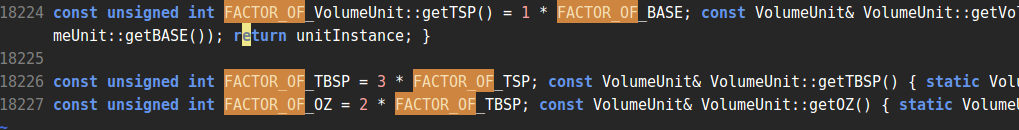
编译器会报告如下错误：



###### 问题分析，使用gcc -E命令



使用这条命令后，打开VolumeUnitUnroll.cpp文件，发现如下问题：



本来希望的路线是TSP是一个字符串，可是它被翻译成了VolumeUnit::getTSP（）函数。

这是因为，在宏调用宏的时候，参数传递发生了问题导致的结果。

###### 小结论

对于有参数的宏，宏内部尽量不要用字符串拼接的方式来做。非常容易出问题，因为宏参数传递的过程中，不能进行入参类型检查，所以编译器有时候也不知道你这个参数是变量，还是字符串，还是数据类型等等，很容易编译出错，很容易误导。

#### C++中零散知识点总结学习

见下文

##### 构造函数的理解

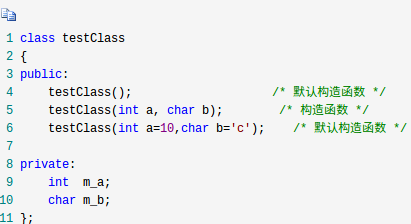
在C++中构造函数很复杂，也是重点，要好好研究，整理

###### 特性1--构造函数无返回类型。

定义一个构造函数的时候，不允许有返回类型，因为他本身就是一个类型，用它来修饰一个元素后，一般这个元素会作为其他函数的返回值，那么如果构造函数再有返回值，它返回啥呢？

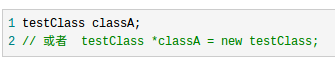
###### 特性2--默认构造函数。

A：默认构造函数是这种



比如第4行。

B:默认构造函数功能，注意这个功能和System Verilog要区分一下，这就已经表示了创建实例了，System Verilog时候，认为它只是一个声明而已。



C：默认构造函数有2种定义方式：

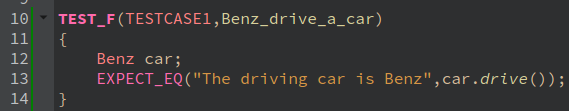
方式1：无参构造函数，比如第4行;

方式2：所有参数都有默认值，比如第 6行。

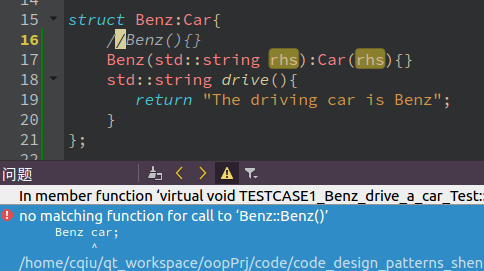
特别注意，一个类只能有一种方式的默认构造函数，第4,6行不能同时出现。

D：只要你自己定义了构造函数，那么编译器就不会再定义了。

例如：第12行，我们调用了构造函数Benz car;



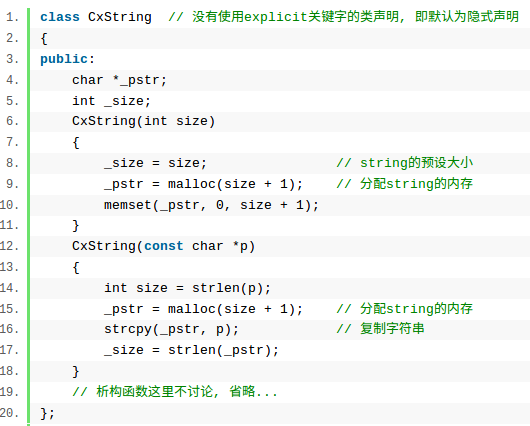
如果注销掉第16行，那么编译报错，提示没有Benz()的函数提供调用。



###### 特性3--构造函数只有一个参数时，可以隐式转换

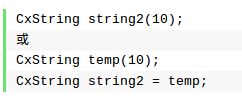
在C++中, 如果的构造函数只有一个参数时, 那么在编译的时候就会有一个缺省的转换操作:将该构造函数对应数据类型的数据转换为该类对象。

举例说明：





为啥呢？

等价于：

但是如果构造函数上，添加

，explicit，意思是必须让编译器进行显示转换，这样就能限制隐式转换了。

换句话说： CxString string2 = 10 ，等价与CxString string2(10)。

特别注意，这里面的“=”用来完成缺省的转换操作了，而不是对象的复制操作哦。

这个还有等号赋值的嫌疑。

##### 构造函数私有化，如何派生对象

一般而言，如果要使用类的方法或者属性，需要先实例化一个对象，如下这个流程图。

正常类的使用流程。

使用其方法aa.get()

类 A

实例化对象

类在实例化对象的时候，必然调用它本身的构造函数，如果它的构造函数是私有的，那么就不能显示调用哦。

比如 class A{xxx,xxx,A()}; A aa();这种声明方式就是显示调用哦，因为实在类外调用的，如果它的构造函数是私有的，那么就不能调用了，也就不能实例化了，更谈不上使用它的方法了，aa.get()，因此这掉路被堵死了。

感觉是，先有对象，再有方法，对象创建又是显示声明的，因此路就堵死了。

有没有对象在类内创建，然后再用的思路呢？

含静态成员函数的类的使用流程

使用静态方法

A A::get(){}

使用对象其他方法

Public：

a.set()之类

实例化对象

A A::get()

{Return A();}

我们巧妙的利用了静态成员函数（也叫静态方法）实在类外初始化的特性，（也就是说，它的初始化时，是有系统自动调用，不用先通过类实例化一个对象，再使用对象的方法，这种思路），实现在静态成员函数内部构造对象，然后利用函数返回值，返回一个对象，这样再用这个对象的其他正常方法，用这种思路来创建新的对象。

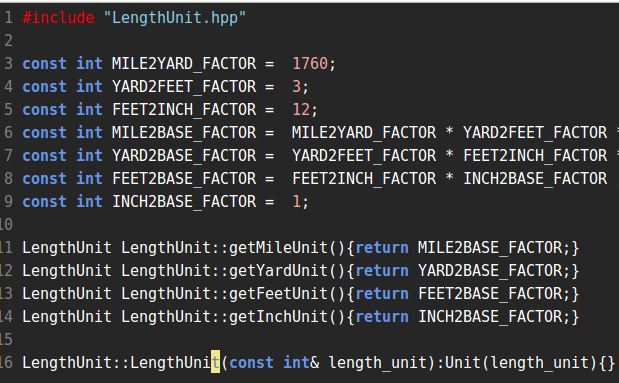
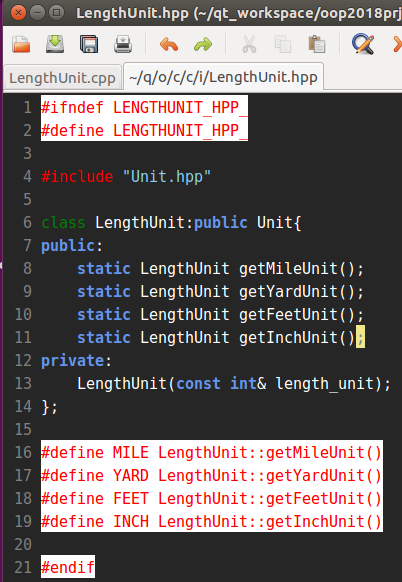
这是一种脱壳构造的方式。有点黑客编程，黑客帝国的意思。

##### 函数的返回值。

函数的返回值语句，所做的工作就是去初始化一个具有返回类型的匿名变量。这时候，将对照函数的返回类型检查返回表达式的类型，并执行所有标准的，或者用户定义的变换。

取自《C++程序设计语言，特别版，贝尔实验室，Bjarne Stroustrup》,第132页。

举例：



特别注意：右图中第11行

分析：LengthUnit LengthUnit::getMileUnit(){return MILE2BASE\_FACTOR;}

注意，本身MILE2BASE\_FACTOR，是一个整数哦，第6行有定义，但是我们不难看出，返回的类型，却是LengthUnit的，多奇怪啊。

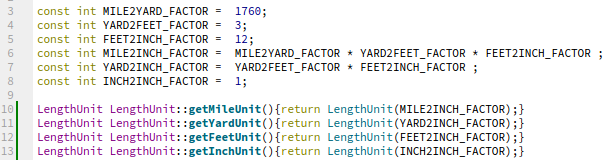
此时来个技巧，return 语句的返回值技巧，可以隐示的转换成函数的返回类型。

可以看看，返回的是1个参数，那么正好和这行语句，对象的创建一致。



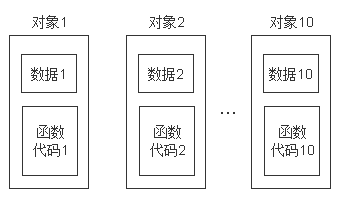
再特别注意，LegnthUnit本身可是private的，按照道理讲，它是不会采用正常的办法创建对象的，当然更不能派生子类了。可是C++用到了这种类似黑客的方法，通过函数返回值的方式，创建了这个对象，你说牛逼不牛逼啊。

当然，使用显示转换也没关系，代码都能成功运行，本实例最牛的地方体现在，第10行，getMileUnit()本身是静态成员函数，程序执行前，系统自动给它初始化，在初始化内部建立的构造函数 LengthUnit(MILE2INCH\_FACTOR)，注意，虽然LengthUnit（）这个构造函数本身虽然是私有的，但是在函数内部可以调用哦，所以不影响规则。

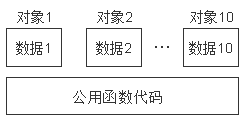


##### C++中类和对象的存储方式

用类去定义对象时，系统会为每一个对象分配存储空间。如果一个类包括了数据和函数，要分别为数据和函数的代码分配存储空间。按理说，如果用同一个类定义了10个对象，那么就需要分别为10个对象的数据和函数代码分配存储单元，如下图所示



这种方式好笨，函数浪费存储空间，能不能把函数弄成一个呢？



变成这种形式，C++编译器就是这么干的。

因此每个对象所占用的存储空间只是该对象的数据部分（虚函数指针和虚基类指针也属于数据部分）所占用的存储空间，而不包括函数代码所占用的存储空间。

那么是否可以理解，C++的对象深层次拷贝的时候，应该也是只需要拷贝数据和函数地址就可以了吧。

C++程序的内存格局通常分为四个区：全局数据区(data area)，代码区(code area)，栈区(stack area)，堆区(heap area)(即自由存储区)。

全局数据区存放全局变量，静态数据和常量；

所有类成员函数和非成员函数代码存放在代码区；

为运行函数而分配的局部变量、函数参数、返回数据、返回地址等存放在栈区；

余下的空间都被称为堆区。根据这个解释，我们可以得知在类的定义时，类成员函数是被放在代码区，而类的静态成员变量在类定义时就已经在全局数据区分配了内存，因而它是属于类的。对于非静态成员变量，我们是在类的实例化过程中(构造对象)才在栈区或者堆区为其分配内存，是为每个对象生成一个拷贝，所以它是属于对象的。

应当说明，常说的“某某对象的成员函数”，是从逻辑的角度而言的，而成员函数的存储方式，是从物理的角度而言的，二者是不矛盾的，成员函数的存储方式，事实上是在代码区的。

###### 虚拟内存空间的分配方式

介绍一下虚拟内存的排布方式，

Section1：0-4k，保护段，比如#define NULL (void\*) 0，这个就是放在了地址0.

Section2：.text，代码段，类成员函数和非成员函数，放在代码段，特别注意，也放类的静态成员函数哦。

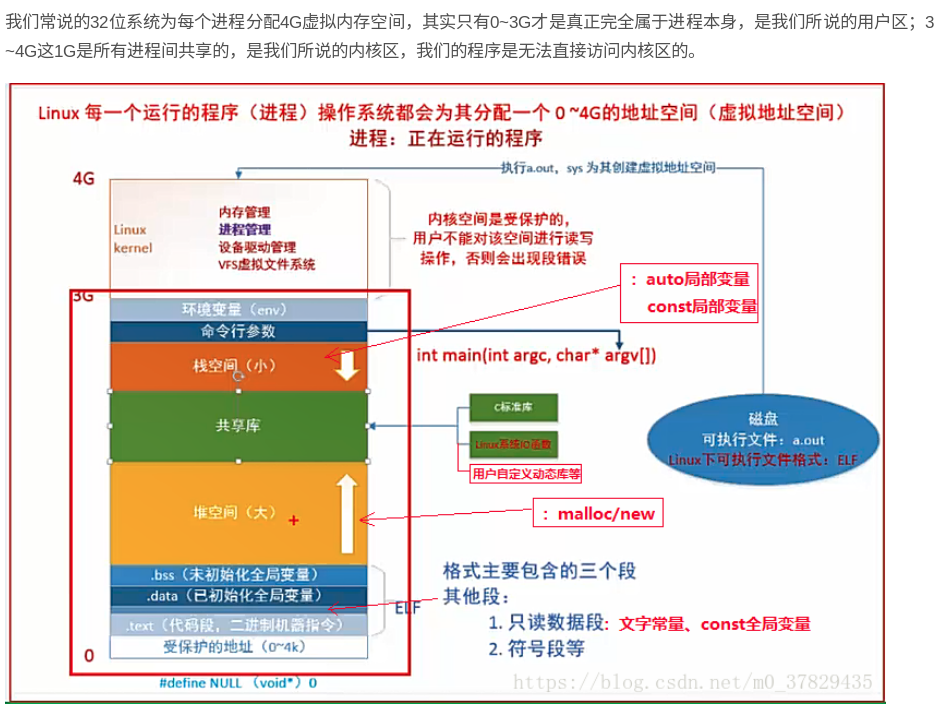
Section3：.data，全局数据段，存放全局变量，静态数据和常量;

Section4：.bss，未初始化的全局数据段

Section5：堆空间（heap area），自由存储区，malloc，free，new，delete等等

Section6：共享库，比如C标准库等等

Section7：栈空间，行函数而分配的局部变量、函数参数、返回数据、返回地址等存放在栈区，特别包括const的局部变量



Section8：命令行参数

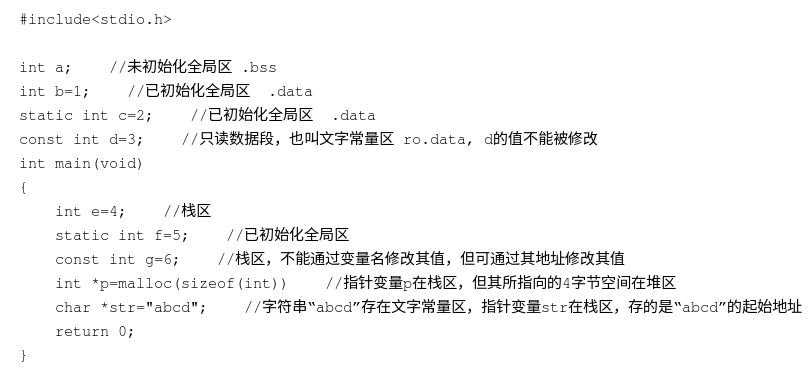
Section9：环境变量

Section：3G-4G之间，放内核代码的，是受保护的，用户不能对这段空间进行读写操作，常见的 段错误，就是伤到了内核区，地址访问越界等等错误哦。

Segment Fault。

举例说明：

特别说明：局部const变量，本身和局部变量一样，放在栈区的，但是虚拟地址，在编译阶段都确定好了。



###### 类的静态成员函数和非静态成员函数

###### 下面我们再来讨论下类的静态成员函数和非静态成员函数的区别：静态成员函数和非静态成员函数都是在类的定义时放在内存的代码区的，因而可以说它们都是属于类的，但是类为什么只能直接调用静态类成员函数，而非静态类成员函数(即使函数没有参数)只有类对象才能调用呢？原因是类的非静态类成员函数其实都内含了一个指向类对象的指针型参数(即this指针)，因而只有类对象才能调用(此时this指针有实值)。

##### 虚函数表

C++中虚函数的作用主要是为了实现多态的机制。

关于多态，简而言之就是用父类型级别的指针指向其子类型的实例，然后用父类型的指针调用实际子类的成员函数。这种技术可以让指针有“多种形态”，这是一种泛型技术。

所谓的泛型技术，说白了就是试图使用不变的代码来实现可变的算法。

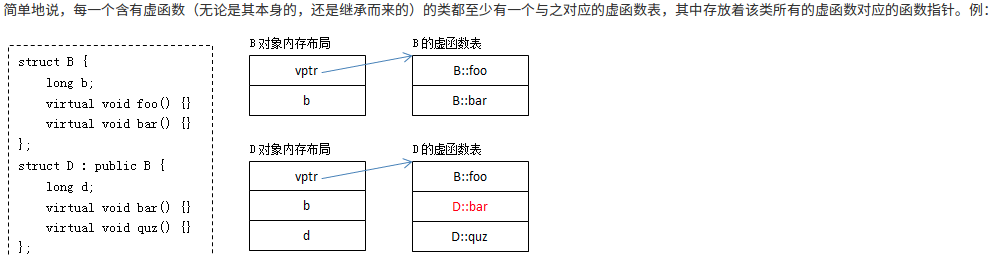
比如模板技术/RTTI技术/虚函数技术。要么试图在编译时决议，要么试图做到运行时决议。

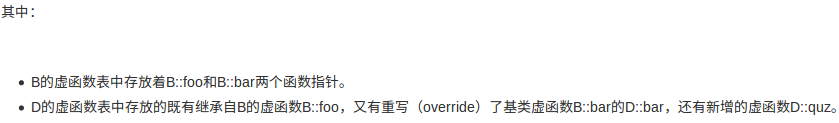
###### 虚函数实现机制

虚函数是通过一张虚函数表（virtual Table）来实现的。建成V-Table。在这个表中，主要是一个类的虚函数地址表，这张表决定了继承,覆盖的问题。保证了它真面貌是反映实际的函数。

有虚函数的类的实例中，这个表被分配在了实例的内存中，当我们调用父类指针操作一个子类的时候，这张虚函数表显得非常重要了，它像地图一样，指明了实际要调用的函数。

虚函数表存在于实际对象实例中最前面的位置（这是为了保证取到虚函数表的有效最高的性能--如果有多层继承或者是多重继承的情况下），因此我们可以通过对象实例的地址来得到这张虚函数表，然后就能遍历其中的函数指针，并调用相应的函数了。





##### C++赋值运算符重载函数(operator=)

小结论：

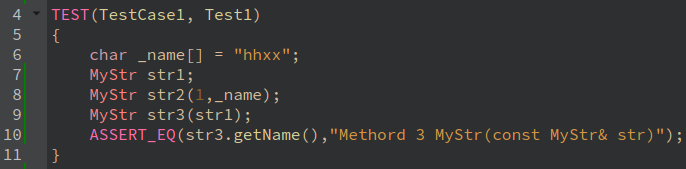
赋值运算符重载函数(operator=),这种函数本身不属于构造函数，他应该算是一种成员函数而已，因此它不能作为初始化赋值的方法。

在定义的时候，不能用它，过程赋值的时候才用。

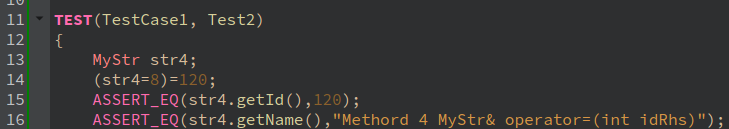
考虑如下类，它的构造函数有如下6种方式

###### 方式3调用方式

简单，好理解。

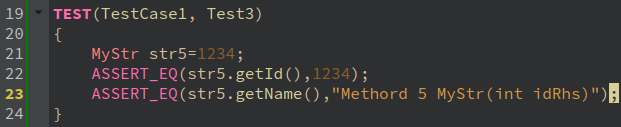


###### 方式4调用方式



特别注意，方式4的=重载时，是在定义之后才进行的重载。

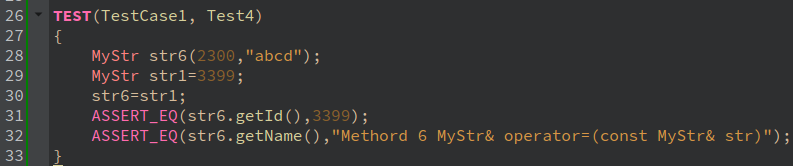
###### 方式5调用方式



注意方式5和方式4的区别，当我们初始化赋值的时候，根据构造函数的特性，“构造函数只有一个入参的时候，可以隐式转换”根据这个原则，隐式转换的时候，等号就相当于入参赋值了，因此这个初始化的过程走的是方式5的构造函数。

###### 方式6调用方式

这种方式和上面好区分，非初始化赋值，在过程赋值时，使用了=重载运算符。



##### const学习

const 能修饰常量，是被编译器放在内存中的只读区域，无论如何都不能被修改。

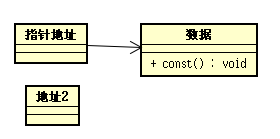
当有函数接口传参的时候，被定义成一个只读变量，

比如 const int nValue；nValue是一个不能修改的常量。

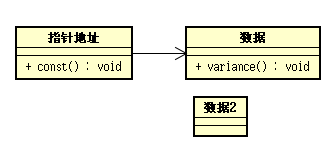
###### 型如const int nValue不能修改的常量

它nValue是不能修改的常量。

###### 何为常量指针和指针常量



常量指针，等价于 常量的指针，表示重点在指针，能移动，但是指向的值是常量的。



指针常量，等价于 指针的常量，表示重点在常量，指针不能移动，但是里面存放的值可以变动的。地址不改，数据改。

###### 组合情况分析

以char类型为例，一般就是const 和 char和 \* 的各种组合，那么就就是3x2x1总共6种组合形式。

比如 1) const char \* a;

2) const \* char a;

3) char const \* a;

4) char \* const a;

5) \* const char a;

6) \* char const a;

总共6种组合，对于5)和6)，没有\*排在前面的情况，删掉。

根据编译器定的规律，当const 与char 连续排放的时候，功能一致。

因此1）和3）属于同一种情况，他们是常量指针。

形2）和4）属于一种情况，他们是指针常量。

###### 具体记忆方法，const先左结合性。

首先定位一下const的位置，先让const向左看齐，如果左边是char，那么表示值不动，那么是常量指针

如果左边是\*，就表示地址是常量，它是指针常量。

第二，如果左边没有东西，再看右边即可。

第三，如果有括号存在，那么括号结合优先。

###### 型如const char \*pContent表示\*pContent不能修改（常量指针）

const char \*pContent;特别注意，\*pContent值，不能修改哦，但是pContent这个指针可以递增，递减，caffe里面有代码。

这种指针叫常量指针，表明指针的值可以动，它指向的值必须都是常量。

###### 型如 char const \*pContent表示\*pContent不能修改（常量指针）

这种写法和 const char \*pContent;一样的，都是\*pContent不能修改。

###### 型如 const (char\*) pContent表示pContent不能修改（指针常量）

const (char \*) pContent;表示 pConstent这个指针本身不能修改，但是\*pContent可以修改，需要和上述写法区分一下。

###### 型如 const int \*const p3=&me，都不能修改

组合情况 const int \*const p3=&me。此时p3,和\*p3都是不可改变的。

const 本身是左结合的修饰符。

###### 型如 void function(const int Var)

表示递过来的参数Var在函数内不可以改变. 好理解

###### 型如 const mytype function()

表示函数返回自定义类型mytype， 限定函数返回值的类型，一般这个不咋用。

###### 型如function() const，叫常量成员函数

我理解这个常成员函数，只会在类里面出现。

void function()const; //[常量成员函数](https://www.baidu.com/s?wd=%E5%B8%B8%E6%88%90%E5%91%98%E5%87%BD%E6%95%B0&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "/home/cqiu/Documents\x/_blank)， const成员函数是不能改变对象的成员函数，在不改变对象的成员函数原型中，加上const，比较适当。

理解啥叫不改变对象？

答：就是class类内部的变量，不能被修改，而且调用class内部的函数的时候，要求这个函数也必须是这种 “fucntion() const”类型的函数，比较变态，一般再格式化输出字符串的时候会用到些。

特别重点：

[构造函数](https://www.baidu.com/s?wd=%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "/home/cqiu/Documents\x/_blank)和析构函数对这个规则例外，它们从不定义为常量成员，但可被常量对象调用（被自动调用）。它们也能给常量的数据成员赋值，除非数据成员本身是常量。

##### &符号的使用,类型右边为引用，对象左边取地址。

一句话：&出现在类型的右边时为“引用”，

&出现在对象的左边时为取地址。

比如：

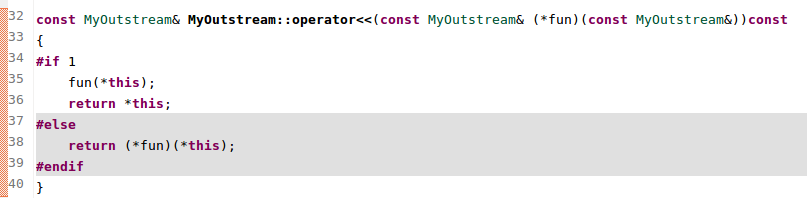
Int \*c;

Int &b=a;//类型int的右边，表示引用

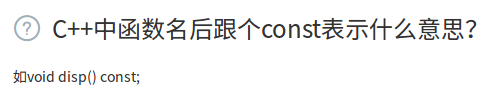
c=&b;//出现在对象b的左边，表示取b的地址。

特殊注意：第1,引用不能为空，定义引用时，必须要初始化，第2,引用不可改变指向，对于一个对象至死不渝。

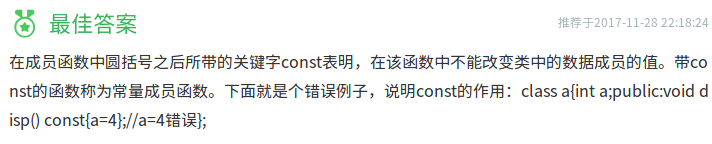
##### 函数指针学习。



知识点1：C++函数名后跟一个const



本代码就这样，



知识点2：类型右边跟着&，表示引用



知识点3：函数指针类型，型入下所示，



就是有一个函数比如 int (\*fun)(int )等等的，那么就是这种函数的类型

Int find(int a)等等。其中fun=find，fun表示它是一个函数指针，指向这一类函数。

这两行好理解了，传递一个\*this，注意它是引用，所以是\*this，是对象，不是指针。

Fun就是一个函数指针。36行返回的也是对象。

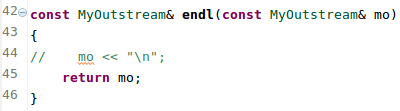


因为fun指针的函数型，就是这样的



入参是一个对象 const MyOutstream&，出参也是对象。Const MyOutstream&。

代表函数：endl



##### <<操作符和友元函数

C++中虚函数的作用主要是为了实现多态的机制。

##### 文件输入输出流学习

文件输入输出流学习，见下文。

###### ANSI C标准预定义宏5

\_FILE\_：当前源文件名，char字符，使用/FC选项产生全路径

\_LINE\_：当前源文件的行号，正数

\_DATE\_：当前编译日期，char字符串，格式：Aug  28  2011

\_TIME\_：当前编译时间，char字符串，格式：10：32：12

\_FUNC\_：当前函数

\_FUNCTION\_：当前函数

\_TIMESTAMP\_：最后一次修改当前文件的时间戳，char字符串，格式：

Sun  Aug  28  13：05：34   2014

##### Int转换成string的方法

我们希望string大行其道，摒弃printf的方法，让string传递到Log中，这种方法开发将会特别方便。C++11中的to\_string()函数将int转换成string。

具体用法：

Int a = 124;

std::to\_string(a);

如果版本不是C++ 11的编译器，那么也可以自己开发to\_string()函数哦，代码如下。

#define max 100

string to\_string\_user(int n)

{

int m=n;

int i=0,j=0;

char s[max];

char ss[max];

while(m>0)

{

s[i++]= m%10 + '0';

m/=10;

}

s[i]='\0';

i=i-1;

while(i>=0)

{

ss[j++]=s[i--];

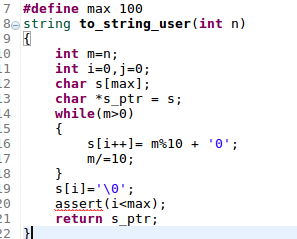
}

ss[j]='\0';

return ss;

}

改进后的优化版本：



##### string转换成int的方法

采用标准库中atoi函数。

string s = "12";   
 int a = atoi(s.c\_str());   
对于其他类型也都有相应的标准库函数，比如浮点型atof(),long型atol()等等。

##### C++中<string>和<string.h>区别

string.h是C语言里面关于字符数组的函数定义的头文件，常用函数有strlen、strcmp、strcpy等等，更详细的信息可以自己查看（面向过程）

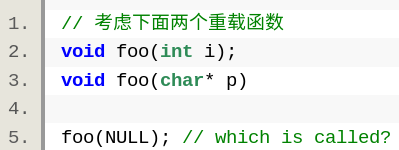
<string>是C++标准库(STL)中的类型，它是定义的一个类，定义在string头文件中。里面包含了对字符串的各种常用操作，它较char\*的优势是内容可以动态拓展，以及对字符串操作的方便快捷，用+号进行字符串的连接是最常用的操作。

推荐做法：C++中就用<string>,准没错。

##### 引入空指针类型的关键字nullptr

在C++11标准中，引入了nullptr，它表示空指针类型的关键字。名空间是std::nullptr,为什么不用NULL呢，因为NULL本身就是0的意思，在一些特殊场景下，容易有歧义。

根本原因是与函数重载有关系。



比如这段代码，我们知道NULL就是0的意思，很明显第2行和第3行都满足要求啊，编译器傻眼了，如果我们使用了nullptr来代替NULL，那么编译器就匹配上了第3行代码，第2行类型不符合，木有问题。

##### 所有类型的指针占据空间大小相同

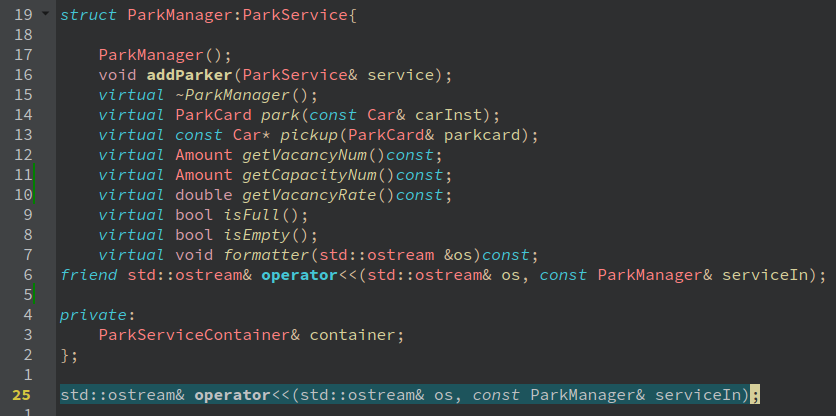
在C++中所有类型的指针变量占据的内存空间都是相同的，但是其指向的变量占用不同大小的存储空间。

这里要表达的意思是，比如

Car \* car; ; 其中 if(car == nullptr)这个是可以比较的，因为他们都是地址的意思哦。

##### 友元函数声明问题

特别注意，看看这段class的声明，发现友元函数在第25和第6行都进行定义了，定义了2遍，一定要注意，如果第25行不定义的话，其他程序可能会看不见这个友元函数。



友元函数一定需要声明2遍，一遍给class内部看，一遍对外彻底开放。

##### TU(Translation Unit)定义

解释什么是TU(Translation Unit)，就是传输单元，也叫编译单元。

通俗理解：就是一个.cpp文件，编译生成一个.o文件，这就叫一个编译单元TU。

一个.cpp文件（在将其包含的h文件展开后）就形成一个编译单元，因为编译器是将.cpp文件逐个编译，得到一个个.[obj文件](https://www.baidu.com/s?wd=obj%E6%96%87%E4%BB%B6&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "/home/cqiu/x/_blank)（或.o等，具体随编译器而异），然后将obj链接，生成可执行文件。

##### C语言中的类似匿名namespace作用

经常遇到的问题是，在C语言中，如果多个TU中（多个.cpp文件）中使用了同一个名字作为函数名或者全局变量，那么在链接的时候，会发生重定义错误。为了解决这个问题，我们可以在定义这些标识符的时候，前面加上static关键字，这样这个函数就只能在一个cpp中可见了。

比如 static void computeFunc(sth. ...);

##### C++中的匿名namespace作用

什么是匿名namespace

类似这种：

namespace{

struct AA{

};

Int bb;

}这类的代码

C++中也可以用static像C那样使用，

注意：全局表示符被static修饰后，它的linkage（关联），变成了internal linkage(内部关联)。

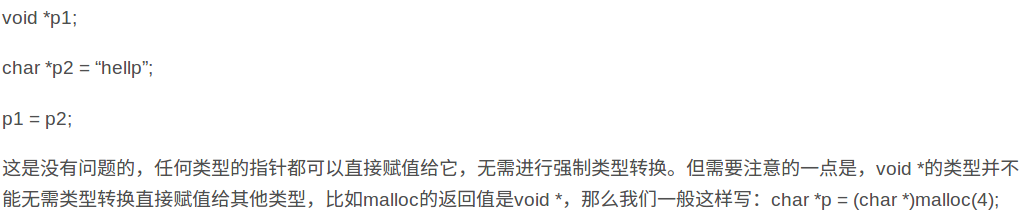
但是C++中的匿名namespace的做法是，它其实是使用了名字改编的方法，每个TU(cpp)中的匿名namespace实际上会有一个独一无二的名字，因此不同的cpp上的namespace相同标识符其实属于不同的namespace。

它没有使用internal linkage内部关联的手法，就是改名，这种方法好，在模板非类型参数中可以使用了。

##### void \*类型说明

本身它可以接收任何类型的指针。

比如：



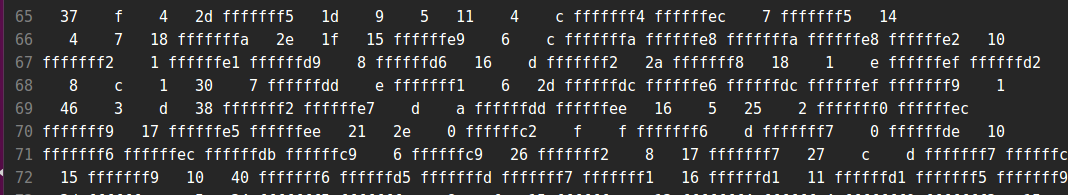
但是void \*类型不能做运算，它并不是一个真实的变量，它只是一个抽象的概念。

与void 之间的比较，

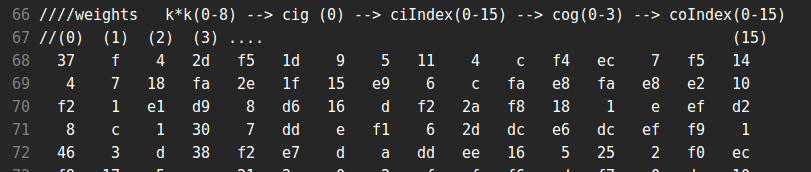
Void本身指的是空类型，比如返回值是空等等，void \*表示的是任意类型指针。

##### 按照字节打印数据

不好的打印方式：



希望的打印方式：可以看到当出现负数的时候，比如f5,只打印了f5,没有那些长长的ff ff ff f5



解决方法：







思路分析：

S1： 开始dataTmp数据是ap\_int<8>类型数据，

S2： 将dataTmp 进行取地址 &dataTmp[m]

S3： 进行强制转换，转换成uint8\_t方式 (uint8\_t \*) (&dataTmp[m])

注意，转换后，它还是指针的值，并不是变量哦

S4：对该指针取值 前面加 \* ,因此就有 \*(uint8\_t \*)(&dataTmp[m])这种方式。

小贴士：uint8\_t 类型：它本质上就是unsigned char类型



##### 新增知识点

新增知识点～～