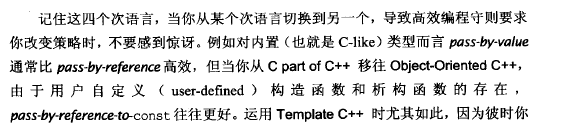
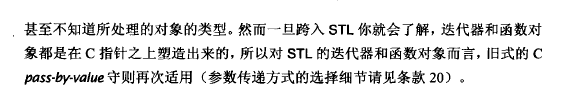


所以如果是引用的话~ Widget& w,则是调用Copy Assignement？就会相当于引用了。

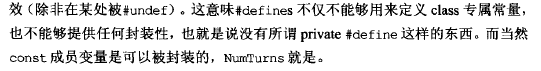


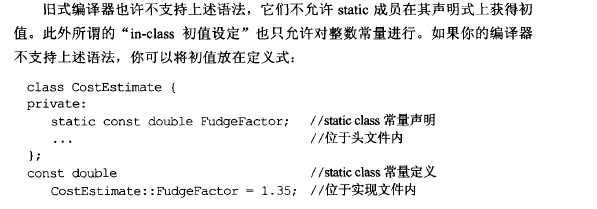


Const Widget& widget

## 用const等替换#define



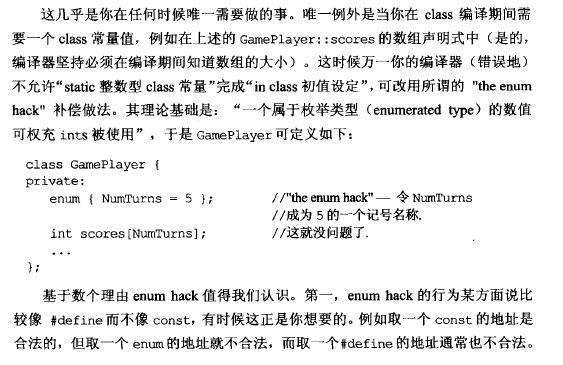




上述语法说的是，直接在类中声明的时候，就把值定了。

Static const double FudgetFactor = 1.35; 在CPP中，声明一下，不设初始值。

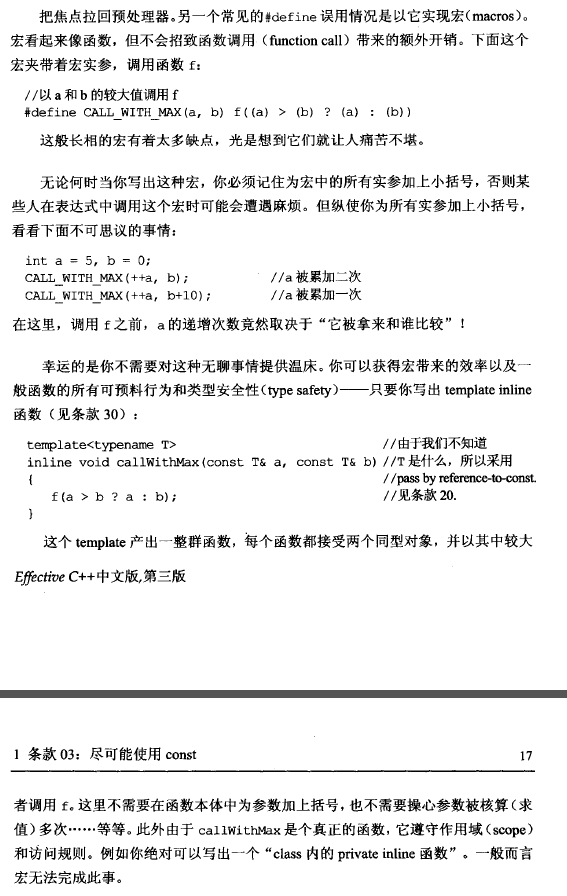
Const double CostEstimate::Fu…..

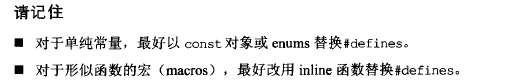


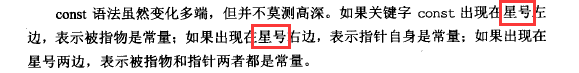
需要常量值，但是老编译器又不能通过static const的直接赋值时，

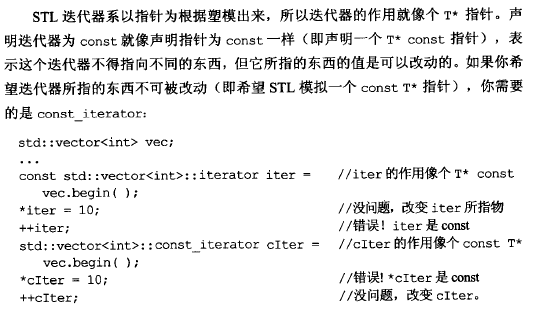


这个很有意思，用宏的话，可能会造成重复调用累加多次，当然有些函数也有可能，



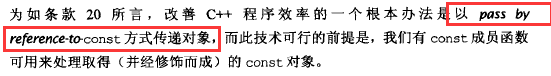




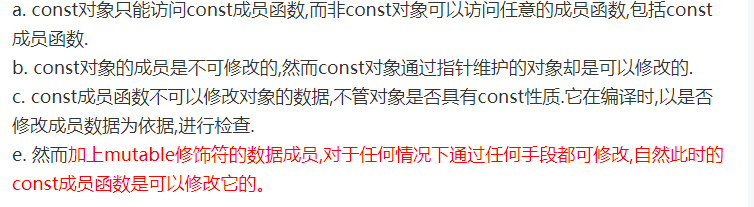


所以const 的迭代器，不能++,const\_iterator的迭代器，不能修改指向的值.

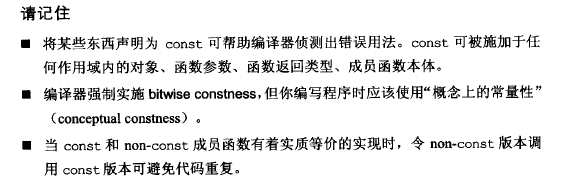
const 放在函数后面，成员函数，



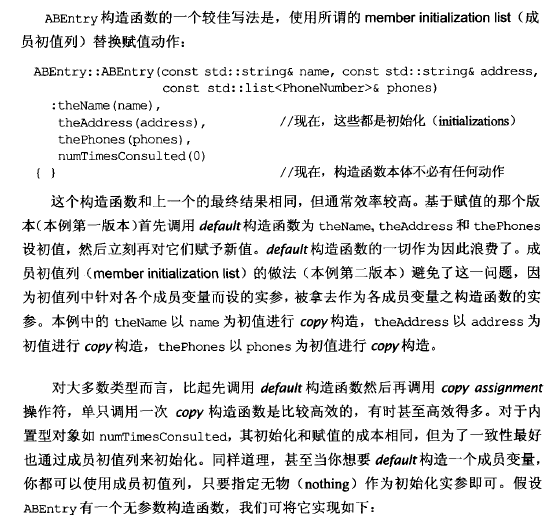


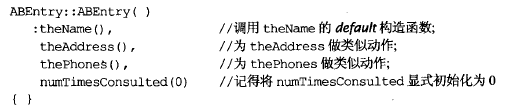


对const成员函数，可以通过指针去修改！！！(有空的话，去测试一下EventSystem里面的GetDesignSize还是啥的函数需要Size，size可以从Draw还是Arrange，是一个const的函数中获得，我想从中输出值却一直没办法，现在看来，可以用指针或是mutable！)



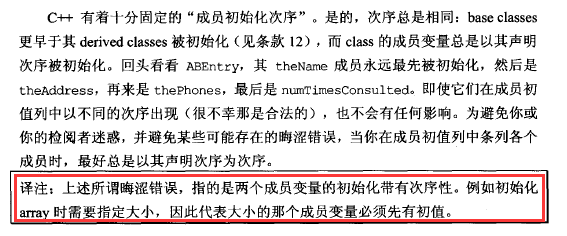
## 确定对象使用前已初始化





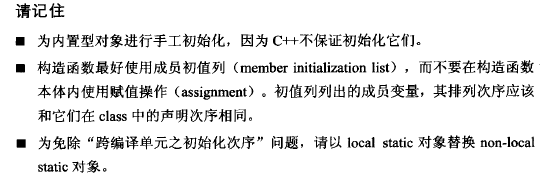
避免在构造函数中，使用theName = name这种，这样的是赋值，而不是初始化。

const 或是 references ，一定要初值，不能被赋值，所以最好总是使用成员初值列。



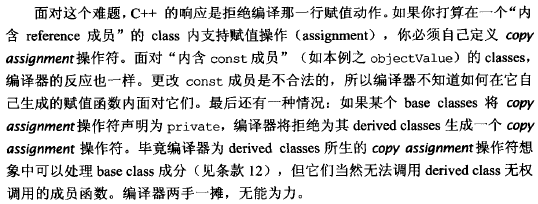
!!!这个很重要呀~用上一个声明来初始化下一个的数组这样的。





最后一条的意思，就很像是单例了~

# 构造，析构和赋值运算（64页）



总结一下其实是这个情况，就是如果你没有手动写，而是调用类中默认的copy assignment操作符的话，他会拒绝为string& nameValue这样的reference变量以及const T objectValue

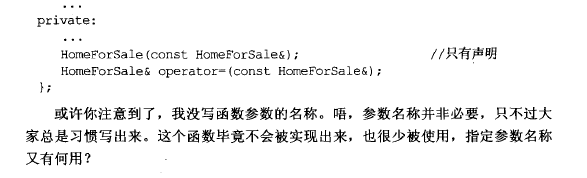
这样的const对象赋值。



## 若不想使用编译器自动生成的函数



你不声明的话，如果调用到，编译器也会帮你加上,方案1，自己进行显示声明为private，不够安全。

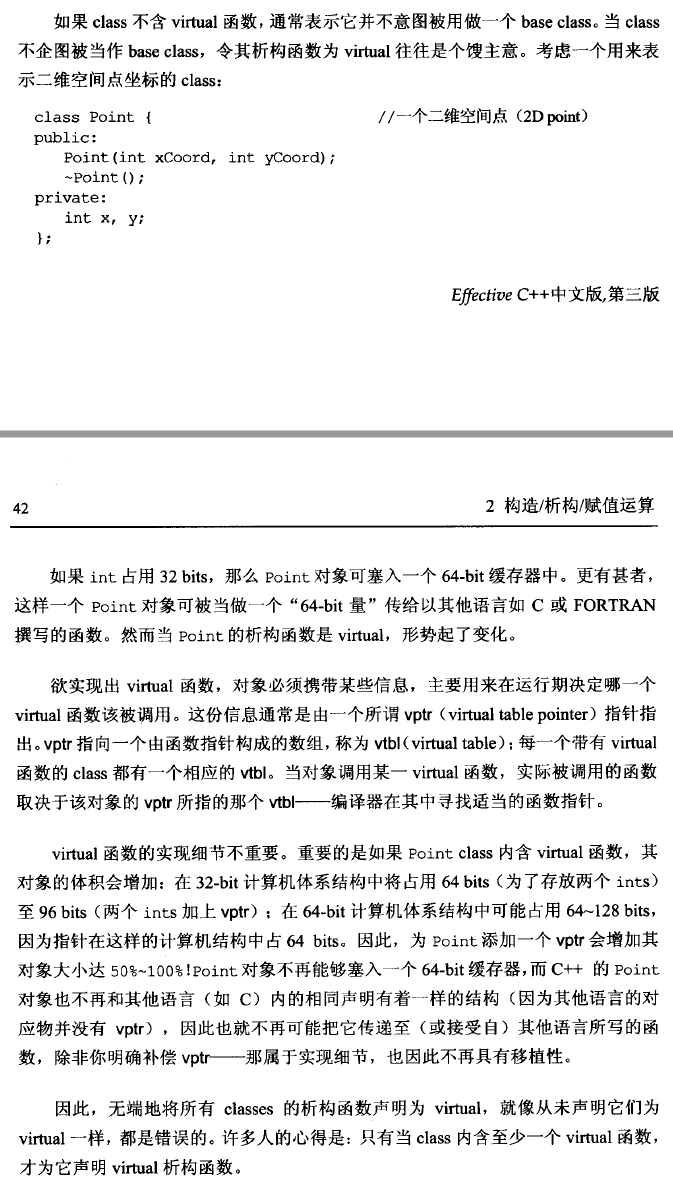


参数名称不是必要的！

方案2：用一个类，然后再去继承他,可能会多重继承就是了

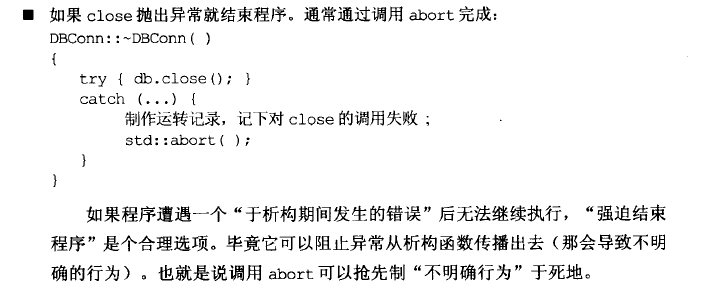
## （可记录）为多态基类声明virtual析构函数70页

这个真的非常有用！！积累的析构函数，声明virtual ~XXX()，这样，哪怕取的是基类的指针指向子类的对象，也能够在清除的时候正常调用子类的析构进而清除所有！



!!!！原来是这样，声明virtual 会导致该对象多携带一个VPTR指针！！原来没有virtual声明的话，本来可以占用很小的内存，但是声明之后呵呵！而且还会影响移植性！！！别的语言没有VPTR指针

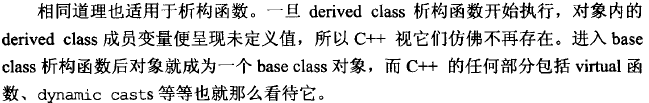
## 别让异常逃离析构函数



## （可记录）绝不要在构造函数和析构函数中调用virtual 函数！！！！



会被调用的是基类的！！构造函数先调用基类的构造函数，那么理所当然的，继承类都还没有初始化，运行起来肯定会有问题，所以会被编译器视而不见



最关键的在于，我们常常会使用Init函数来在构造函数中初始化，但是如果此时在Init中包含了virtual的函数，那么。。。。。这个还真的很容易用错，关键是不会引发编译器报错！在UE4中试一下呢~UC++ 此到第82页

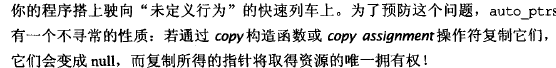
# 资源管理(91)

## （可记录）资源管理的智能指针相关部分可以整理下记录RAII

可以在某段函数内使用智能指针， auto\_ptr，则在离开该函数时，会自动销毁并删除它所指向的物体，所以不要让多个智能指针指向同一个变量，同时智能指针的复制，比如auto\_ptr<int> ptr1(aa)

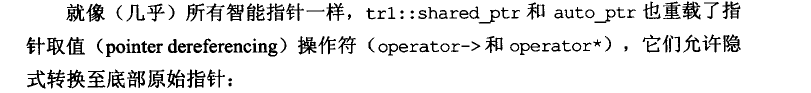
Ptr2 = ptr1;

则ptr2指向对象，ptr1被置位空.

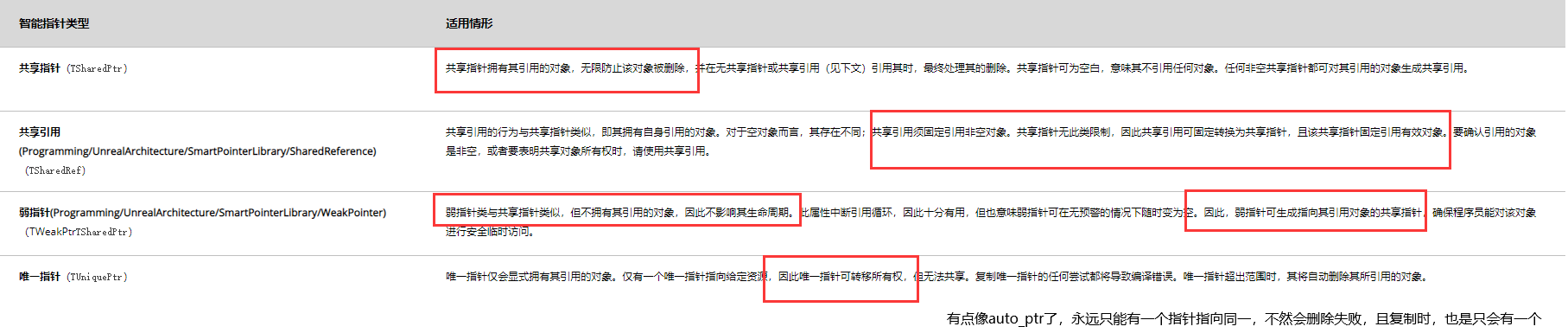


使用shared\_prt<XXX> AAA 时，可以在初始化的时候指定删除器！比如在构造函数中初始化的话： Actor( AA) : AAA(AA , function){} 则在shared\_prt引用计数器为0的时候，就会调用function而不是直接调用析构函数删除。

看到shared\_ptr和auto\_ptr，想到了UC++的智能指针，TSharedPtr，也是同样拥有Get方法获取其中的指针，也重载了operator->和operator\*,但是UC++还有智能引用，TSharedRef



UC++中的指针对比：



为RAII提供一个现实转换函数比如get()，就是很多的一般做法，也可以提供隐式转换函数，

Class Font {

……..

operator FontHandle() const

{

return handle;

}

//显示

FontHandle get() const

{

return handle;

}

FontHandle handle;

}

但是隐式转换会可能出问题！因为比如 FontHandle A1 = A2;

RAII资源管理类可以获取内部的变量，虽然感觉和封装有一定的违背，但是他真正的作用是为了，让资源被释放时，可以正确的删除（用智能指针去指向对象）。

智能指针还会有一个隐藏的问题，就是如果有这样一个函数，

void Dothing( std::tr1::shared\_ptr<Actor> InActor , string InString)

当我们调用时，传入的参数是std::tr1::shared\_ptr<Actor>，则如果调用方式写成这样

Dothing( std::tr1::shared\_ptr<Actor>(new Actor), ”哈哈哈哈哈哈”) ，那么OK，不会有问题，但是如果，第二个参数传递的函数，函数为

String GetString()

{

String[] aa = { “1”,”2” };

return aa[2];

}

Dothing( std::tr1::shared\_ptr<Actor>(new Actor), GetString())

这个时候！GetString是会报错的，那么此时，因为在执行Dothing函数前，会调用三个函数且他们的执行顺序不是一定的：

new Actor();

GetString();

std::tr1::shared\_ptr<Actor>()

如果GetString在new Actor之后执行，那么因为有异常，创建的new Actor返回的指针就丢失了，就造成了资源泄露。

所以，推荐把声明shared\_ptr的函数另起一行。

## typedef的灾难~

如果使用typedef 将数组定义了别名，则delete的时候很可能出现问题，比如

typedef int MyName[100];

int\* MyArray = new MyName;

由于MyName是该数组int的别名，100是数组的长度

则int\* 在释放的时候，需要delete[] int; 所以最好不要对数组形式typedef，

可以使用vector<int> MyName，相当于别名了不是。

# 设计与声明：108页

## （可整理记录）

使用外覆类型

Funciton( int a, int b, int c)

如果a b c有特殊的要求，并且容易因为填错造成问题的话，将三个参数使用struct包装成类型，就可以有效防止填错

Struct a {

Explicit a(int Ina) : val(Ina){};

int Ina;

}

Function( a(1) , b(1) c(1)); 还可以保证传入的顺序。以前在设计的时候，确实很少使用这点，都是在函数内去做判断，保证处理异常或是弹出警告等，这样单独使用外覆类型确实在放置策划填写错误上有奇效，但是如果对每一个都声明结构体，未免太过于复杂且繁琐。

声明为Struct还有个好处，在使用RPC的时候，可以方便的对某些自定义类型进行复制。

还有一点就是如果某个参数例如攻击类型，如果使用enum的话，可以被转化为int使用，若不想让其被转换，可以使用例如

class AttackType{

public:

static AttackType Comba1{ return AttackType(1); }

……

private:

AttackType (int InType);

};

这样可以直接把AttackType::Comba1当做参数，类似的用法在但丁中有：

UpdateInputKey(EInputKey::Move, IE\_Axis);

其中的EInputKey即为此类似用法，更加方便快捷，把FInputKey做了包装以简单方式调用。

struct EInput

{

static FInputKey None;

static FInputKey Move;

static FInputKey MoveForward;

}

## 可记录！这点其实真的很有用！

我们定义一个函数的时候，参数如果是一个对象，已值传递的方式的话，如

void Function(Actor InActor);

Actor AA;

Function(AA);

则此时，调用函数是是以原对象的复制值进行传递，因此会调用一次Actor的构造函数，以及Actor内的成员变量的构造函数！当Function结束时，会调用Actor的析构函数以及内部成员变量的析构函数，这么一来一回，开销就上去了！还有一点需要注意，如果函数形如

void Function(BaseActor InActor) 也是可以传递AA作为参数的，但是此时AA的Actor的继承部分，就会被全部抛弃。

当然我们一般写代码都是使用传递指针的方式，就规避了上述的问题，还有一种方式为使用const Actor& ,类似于struct不使用指针时，就可以用这样的方式，之所以要加const，是为了防止在函数内被修改，这个依据需求可以去掉。 其实const Actor& 在实现方式上，其实也是通过指针的方式。

类似这样的问题，如果返回值，使用了引用的话，有可能会出现返回的值在堆上生成，已经被释放掉了，返回了一个不存在的对象。

隐式转换，要不要将构造函数声明为explicit？依据情况来定，在很多情况下，我们需要例如

class Num {

Num( int A) {};

const Num operator @ (const Num& InB) const

……..

}

此时如果将构造函数声明为explicit, 则以下代码就出错了，

const Num A(1);

result = A @ 33;

这是因为允许了隐式转换，所以编译器发现33可以隐式转换为Num(33) ，并且调用

operator @() 。

模板全特化：

Template<>

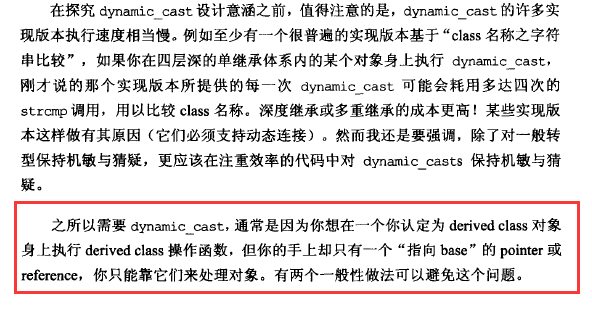
# 5 实现 p143

P148 将各个转换符号再写一遍：

const\_cast 通常用来进行常量移除，把const转化为非const ，也是唯一有此能力的转换符。

dynamic\_cast 主要用来进行，安全向下转型，这是唯一无法通过旧式语法执行的动作。

static\_cast



Base\* A;我知道他是指向一个DerivedD 类型的对象，用dynamic\_cast,可能会经过

DerivedA ,DerivedB,DerivedC,这几个过程！会很漫长。

Actor& GetActor() const { return m\_A; }

这样的写法，哪怕m\_A是private，哪怕函数声明的是const ，也会导致m\_A可以被外部修改，同样的，返回指针和迭代器这样的handles也能造成这个问题。但是可以通过返回

Const Actor来避免被修改，但是还是返回了





157 异常我还没有认真看！！！下次冲洗那看你！

## Inlining详细说明，与文件间的编译依存关系降低（可整理记录）

Inline函数只是一个申请而不是强制命令，而且该申请是可以隐喻提出的，将函数定义与class定义式内，例如：

class Actor {

public:

int GetOuter() {return Owner;}

}

但是内联函数如果定义多于一行，编译器就会忽略inline限定符。

内联函数也不是用的越多越好，任何对内联函数的修改， 都需要重新编译插入该函数的所有客户端，而如果是非内联函数，客户端只需要重新连接就好

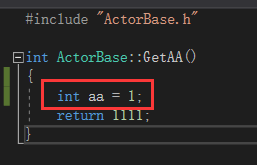
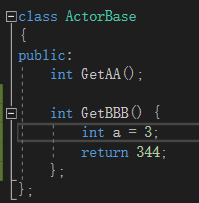
，而且过度的替换会造成代码膨胀，最关键的是，内联函数无法调试。

而virtual函数，由于内联函数大部分是在编译期进行的函数替换，此时无法确定要替换的virtual函数是哪一个函数，所以没法使用内联。

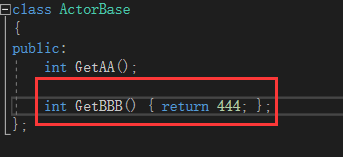
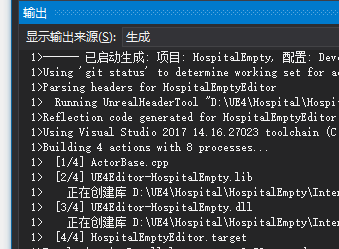
还有，如果使用了函数指针指向一个内联函数，那么编译器还是会为该函数生成一个outline版本。因此，特别是对于工程量庞大，引用部分多的工程来说，要非常小心的使用内联函数，也要避免将函数定义写在头文件中，更是千万不能将构造函数和析构函数声明为内联，因为他们并不像看上去的那样空，编译器会为其自动填充很多函数，例如对象的初始化与删除。

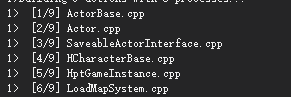
## P171，将文件间的编译依存关系降至最低：

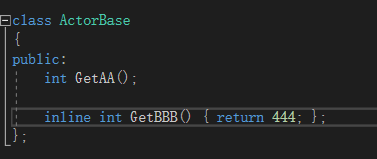
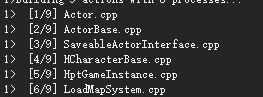
如果将函数实现写在.cpp中，可以有效减少编译数的。对于下面的类



如果修改了.CPP文件，可以看到，只需要重新编译一个.cpp文件，而如果修改.h中的一个返回值，需要编译所有引用了该头文件的函数（编译所有客户端）



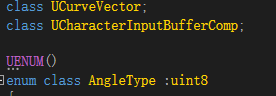
，如果使用显式inline，修改一个值更是要重新编译所有客户端

当然，就算不用内联函数，修改在头文件中实现的函数，也需要重新编译所有客户端。

可见，我们平时在编写类时，会尽量不在头文件中包含其余头文件，防止产生编译依存关系。

如果需要使用其他类的时候，我们一般会写出声明，在.cpp文件中再对具体的类头文件进行包含。



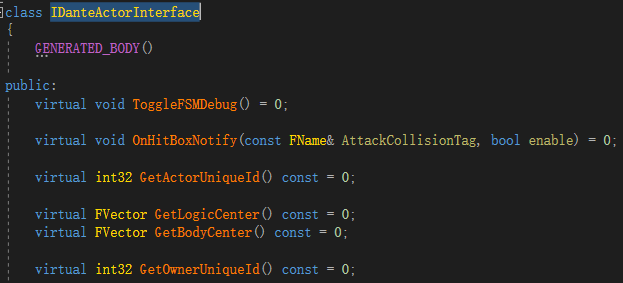
但是因为我们只使用一个类型声明式，所以可以声明类似 UCurveVector\* Vec;

TSharedPtr<UCurveVector> Vec; 这样的变量甚至使用类似

UCurveVector GetVector() 这样的函数声明，但不能声明 UCuirveVector Vec ， 因为声明一个指针变量，编译器只需要分配一个指针空间即可，不需要关心起具体占用多大控件，而涉及到定义某种类型，就必须要使用它的定义式了，而我们并没有包含其头文件。

为了减少我们的编译依存关系，

1、最常用的当然是我们的接口Interface方式了，interface中声明了要使用的方法，而具体的方法实现，则放到继承了该接口的类中，那么在该类的客户端中，就可以不包含其具体的实现，而只包含该接口的声明了，这样无论实现部分怎么修改，只要该接口的函数不变，客户端都不需要重新编译



2、还有一种方式，被称为Handle class

将一个类分成两个类，其中一个类只提供接口，而另一个实现类负责实现这些接口，对于客户端来说，只需要包含提供接口的类头文件即可，而真正的实现类无论怎么修改，只要接口不变，都不需要客户端重新进行编译。类似的例子如下,客户端只需要包含Actor.h即可，实现在ActorImpl中。

class ActorImpl;

Class Actor{

……..

public:

void SetActor();

public :

ActorImpl\* RealActor; //因为声明指针变量是不需要知道具体实现的

};

.cpp 中再正式包含ActorImpl的实现即可

#include “ActorImpl.h”

void Actor::SetActor()

{

RealActor-> SetActor();

}

两种方式的声明部分，都其实使用了隐式的Inline内联，虽然两种方式都会带来额外的开销（比如Interface由于都是virtual函数的关系，会需要一个间接调用的成本，而Handle class的方式，除了每一个访问都有一个间接性之外，还会因为需要多声明指针增加一些额外的开销。）但是在合理运用范围时，对工程的好处是巨大的。

# 6 继承与面向对象设计

## 重载继承与名称域的关系等问题：（可记录）

继承中有一些经常被忽略的问题（我就没记住过：）），如下所示的类中，ActorBase重载了A1：

class ActorBase {

…

public:

virtual void A1() = 0;

virtual void A1(int a);

void B1();

void B2();

};

class Actor : public ActorBase

{

….

public:

virtual void A1();

void B1();

};

如果如Actor这样只继承了A1的其中一个版本的话，那么ActorBase的A1(int a)就不会被继承了，此时如果按下面方法调用，就会报错，因为ActorBase的A1(int a)方法并没有被继承。

Actor AA

AA.A1(33)

那么如果我们只想要重写其中的一个函数，另一个函数使用父类的实现的话，在Actor中使用作用域标识即可:

class Actor : public ActorBase

{

….

public:

using ActorBase::A1;

virtual void A1();

void B1();

}

using会使ActorBase内的叫做A1的所有函数（变量）都在Actor作用域中可见，此时如果调用 Actor的A1，则无参数版本依旧调用Actor的重写版本，而带参数版本，就会调用ActorBase中的版本了。但是有一点要注意，using不能在private继承中使用，因为相当于将父类的函数(变量)都暴露出来了。

对于纯虚函数的继承，还有一种非常有用的用法，比如我希望基类的A方法一定被继承，但是我又希望该函数能够提供一些默认的方法，那么，可以选择将纯虚函数的实现也加上，没错，纯虚函数也是可以加上实现的，该实现需要使用者手动调用：

class ActorBase {

…..

public:

virtual void A() = 0;

};

void A() {

DoSomething();

}

class Actor : public ActorBase

{

virtual void A( ){ActorBase::A();}

};

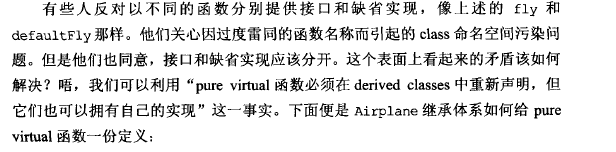
class Character:public ActorBase

{

virtual void A( ){DoNothing();}

}

这样的用法，可以让接口拥有默认的行为实现，又可以保证纯虚函数的写法。



声明为纯虚函数，还是可以为其写上实现函数，在子类中，通过父类::函数名来进行纯虚函数声明的调用。

199页

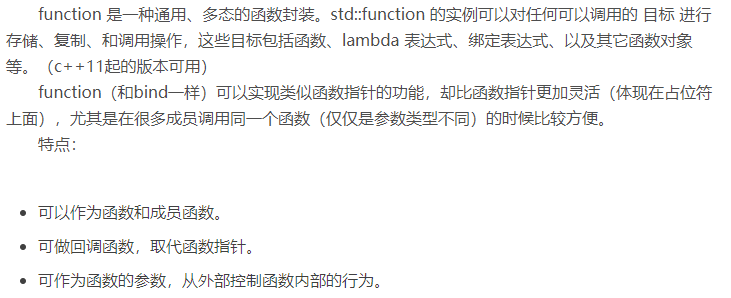
## （可选择性整理记录）

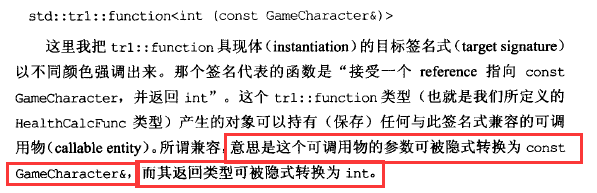
Non-Virtual Interface方法，有意思！！

C++的继承类可以重新定义继承来的private virtual函数

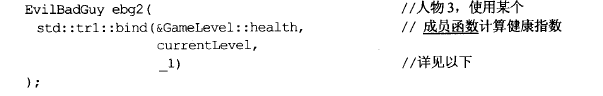
函数指针，如果引用函数名去油没有调用该函数，函数名将被自动解释为函数指针。P = AA等价于P = &AA

C++ std::tr1::function使用





## tr1::bind的用法！！！这特么是个啥！



## 绝不重新定义继承而来的non-virtual函数和缺省参数值（可整理记录）

class ActorBase {

void Dothing();

}

class Actor : public ActorBase {

void Dothing();

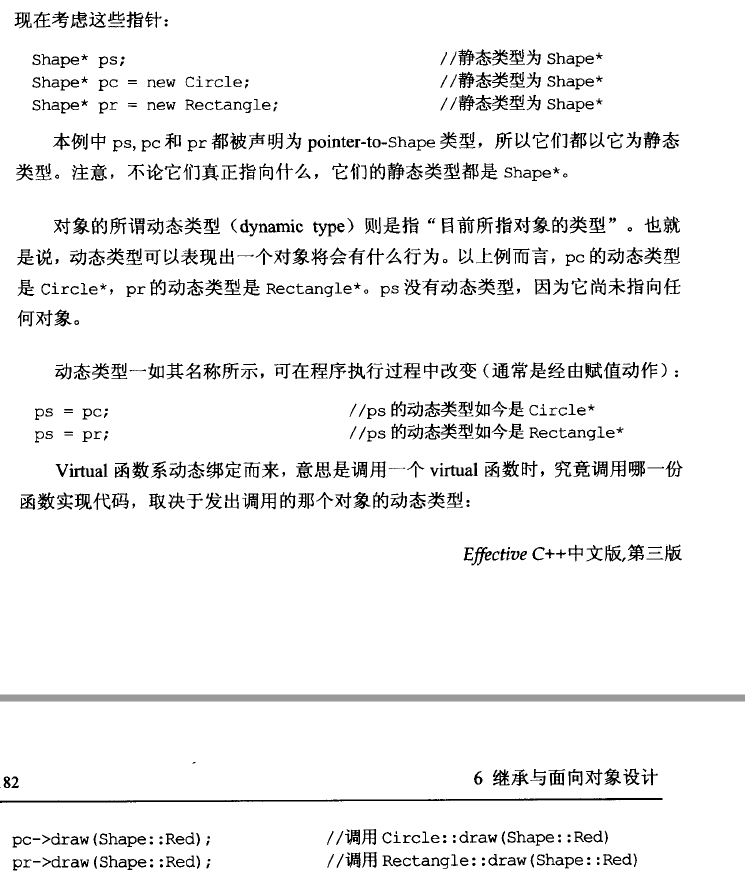
}

如果重写了非virtual函数，那么当一个Actor的对象A，为其声明一个指针

ActorBase\* ABP = &A; A->Dothing();

Actor\* AP = &A; A->Dothing(); 这两个调用，与实际对象无关，分别会调用ActorBase的Dothing和Actor的Dothing！如果是virtual函数，是动态绑定的，则无论以哪种方式调用，调用的都是继承类（如果有重写）的Dothing，因为对象是一个Actor对象；而如果是一个非virtual的函数，那么则会根据声明的指针类型，来调用对应的函数！

静态类型、静态绑定与动态类型动态绑定



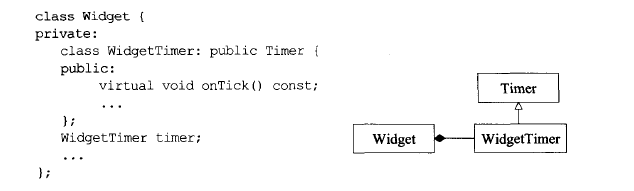


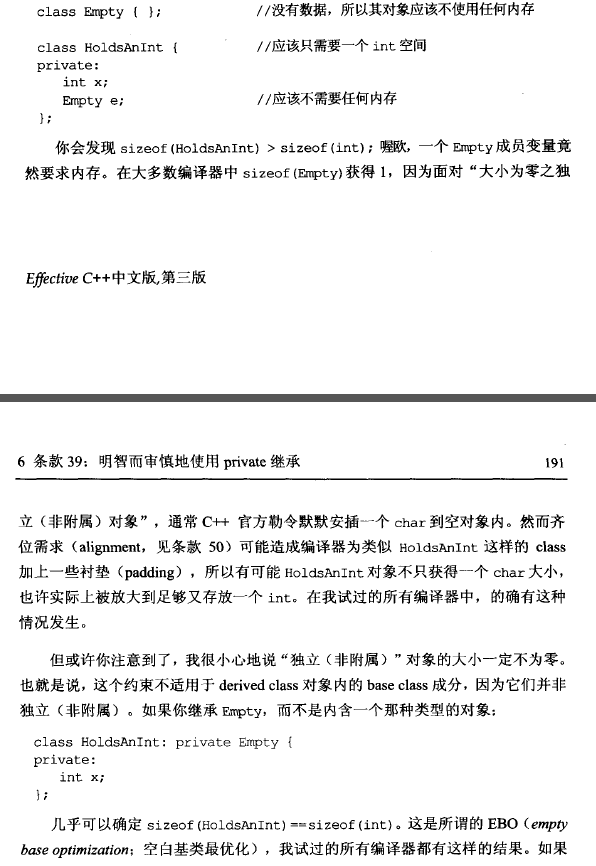
如果virtual 函数有缺省参数，那么就会特别麻烦,可以测试一下

谨慎的使用Private继承：

如果class之间的继承关系是private，那么编译器不会自动讲一个derived class对象转换为一个base class对象！

嵌套Class 骚啊,可以看下内部具体的函数实现是怎么声明的，Widget::WidgetTimer::OnTick()?

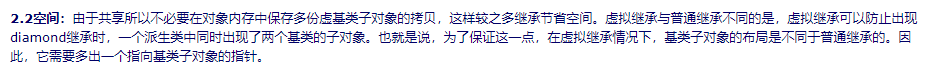
一个大小为0的独立对象，编译器会为其添加一个char（一般情况下）到空对象内，如果有特殊需求的话，



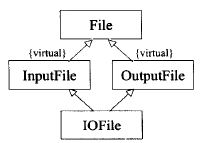
所以private继承还有一个功能，就是使那些空对象真的变成空的~但是现实中，应该根本不太可能这么用吧。

Virtual继承，

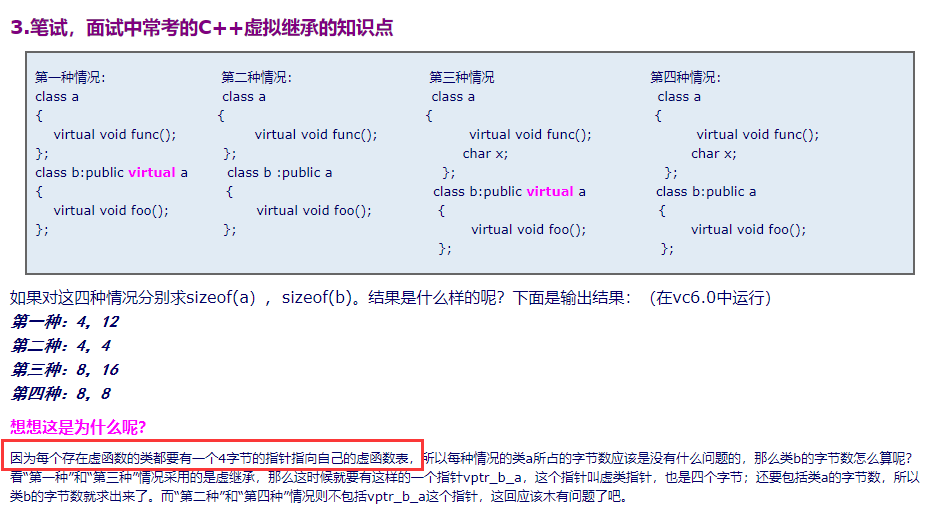
虚拟继承在一般的应用中很少用到，所以也往往被忽视，这也主要是因为在C++中，多重继承是不推荐的，也并不常用，而一旦离开了多重继承，虚拟继承就完全失去了存在的必要因为这样只会降低效率和占用更多的空间。

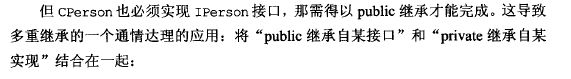


虚拟继承可以防止出现diamond继承时，一个派生类中同时出现了两个基类的子对象

这就是钻石型继承，IOFile继承自两个，这个可以测试一下，如果InputFile和OutputFile都有一个FileName这个字段。

非必要不使用virtual继承





## 关于继承这一块，可以再看一遍，然后写点代码测试

## （记录）再看了一遍继承的梳理：

不要重新定义继承而来的缺省参数值！因为缺省参数是在编译期就静态绑定了的，而virtual函数，则是在运行期动态绑定的，如果我们在继承的virtual函数中重新定义了缺省参数值，在调用时，是根据静态类型来判断缺省参数的调用的

例如：

class ActorBase {

virtual void Dothing( int32 A = 1 ){ print( A ) ;};

}

class Actor : public ActorBase {

virtual void Dothing( int32 A = 2 ) { print( A ) ;};

}

ActorBase\* A;

ActorBase\* B = new Actor();

A->Dothing();

B->Dothing();

那么此时，B->Doting()虽然是调用的Actor的动态类型指定的vritual函数Actor的Dothing，但是其缺省参数，却会因为B的静态类型是ActorBase而调用ActorBase的Dothing缺省参数A = 1！

那么如果为了保证缺省参数的执行结果一致，最好将virtual的缺省参数写成与父类一致，才能避免因为动态类型造成的问题。

Class Actor : public ActorBase {

virtual void Dothing( int32 A = 1 ) { print( A ) ;};

}

那么一旦ActorBase的缺省值改变，子类也必须跟着一起变，这太蠢了。

这里有一个之前没怎么关注过的设计，non-virtual interface,

即是将缺省参数值的函数设置为非virtual函数，并在内部去调用virtual函数，例如

class ActorBase : {

virtual void RealDothing(int32 A);

void Dothing( int32 A = 1 ){

//还可以把，各种不需要重写的代码，写在这里，例如生成和释放一些必要数据等，

//这样还可以防止，在继承类中，忘记写那些重要东西。

RealDothing(A);

};

}

## （记录，加上自己猜想）Private继承与多重继承的问题：

class ActorBase {};

class Actor :private ActorBase{};

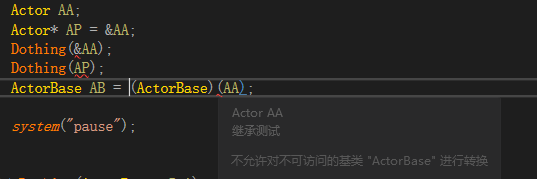
void DoThing( ActorBase& A );

ActorBase AB;

Actor AA;

DoThing(AA); //这样用，是会报错的！

因为private继承，编译器不会自动将一个继承类对象转化为一个基类对象，是指针，也不行转换，就算显示调用转换，也是不能通过编译的。



尽量不要使用private继承！

多重继承：

因为在UC++中，Uobject的类不允许多重继承，且平时很少使用多重继承，因此有很多多重继承的点确实没有留意。

class ActorBase{

void DoThing();

virtual void DothingTwo();

int m\_a = 1;

}

class ObjectBase {

int DoThing() {return 1;};

virtual int DothingTwo(){return 1;};

int m\_a = 2;

}

class Actor: public ActorBase ,public ObjectBase{

}

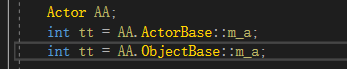
Actor AA;

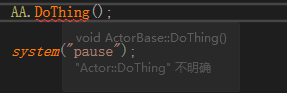
AA.DoThing();

AA. DothingTwo ();

此时编译器是不知道调用的是哪一个DoThing的，只能手动进行调用：

AA.ActorBase::DoThing();

还有，ActorBase与ObjectBase都有一个对象m\_a，那么多重继承后，Actor有几个m\_a？两个

Virtual的函数，也会不明确。

为了解决这些问题，总是应该将多重继承声明为virtual 继承，但是声明为virtual public却会占用更多的内存，访问非virtual函数时，访问速度也会变慢。

但是多重继承有一个很合理的用法在于，因为private继承其实并不是要表明继承类是一个基类，而是想要使用基类的一些方法，所以可以使用多重继承实现例如：

class Person {  
 virtual void Eat(){};

}

class Monster {

virtual void StrongEat(){}

}

class Man : public Person , private Monster{

virtual void Eat(){

StrongEat();

};

}

因为人其实不是个怪兽，但是想要使用怪兽的技能的话，用这样的方式是比较合适的，当然,如果不用继承而在Man类中使用组合的方式调用Monster类的StrongEat会更好。

# 模板与泛型编程 P228

## 隐式接口和编译期多态（记录，加上测试代码）

编译期多态，比如

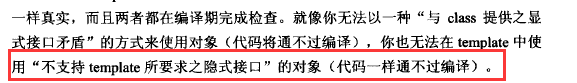
Template<typename T>

void DoThing(T &actor)

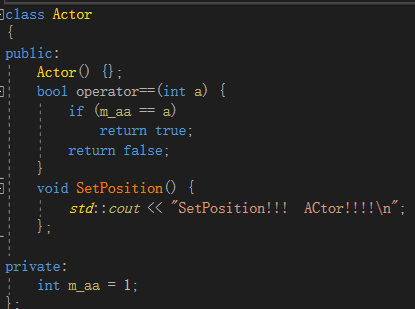
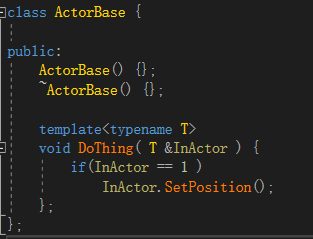
涉及actor的任意函数调用，包括operator，都可能造成templaye T的具现化，在编译期发生，具现化后，调用的就相当于不同的函数了。

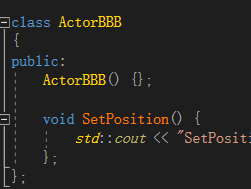
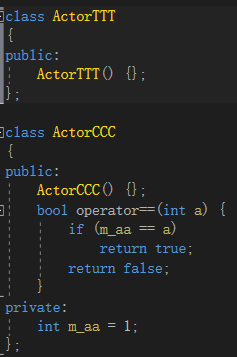
运行期多态，就是virtual函数，运行期哪一个virtual函数应该被绑定。

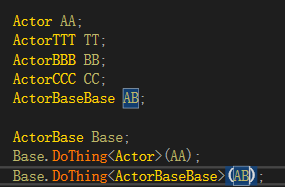
template的隐式接口：



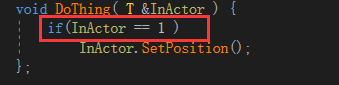
一般的函数声明，都是声明显式接口，而声明template其实是需要支持他的隐式接口的，并不是适用于任意类型，关于这点，之前真的从来没有留意过，所以测试如下，







其中，只有DoThing<Actor>(AA)与Base.DoThing<ActorBaseBase>(AB);能够通过编译，其他的模板由于没有满足对应的隐式接口，并不能通过编译。所以经常在人家的项目中，发现虽然用了模板但是并不支持很多自己写的类，就是这个原因了。需要注意的是，



运算符也算是需要隐式支持的接口，如果没有重载对应的运算符，也不能通过编译。

在template中，多态是通过template具现化和函数重载解析，在编译期就发生了。

## P234 了解typename的双重意义

有一个typename我从来没有注意过的点地方，当我们声明template函数时，

template<class A>与template