技术

Item 25 将constructor和non-member functions虚化

这实际上是一层包装的技术。如果考虑$A \to B, A \to C$的继承体系，那么不可避免地可以推荐这个东西：

```
list<A*> container;
```

用于存放ABC这三个类的对象指针，反正利用RTTI可以做到很多事情。这样存放还可以达到构造函数虚化的效果（注意：构造函数是没办法虚化的，所以是达到效果）。

除此以外，还有复制操作，这相对于构造函数而言可能更加简单，因为可以直接`clone()`并将其虚拟化来完成，至于原来的复制操作？`private`即可。这里有个新特性：在继承体系中，可以返回不同的指针类型，这提升了虚函数的灵活性。

第三个例子是`operator<<`，如果这玩意也是`virtual`的话，那么你的使用方式可能是比较滑稽的，比如`t<<cout`。这照样也可以通过包装来解决问题，外面就用`const A&`来作为变量类型就行。

Item 26 限制某个classes所能产生的对象数量

假如你在写一个打印机程序，并且要求只能产生一个打印机，那么比较建议的方法是

```
Printer& thePrinter()
{
    static Printer p;
    return p;
}
```

之后调用`thePrinter()`就可以获得那个`Printer`。类里面放一个`friend`就行，以方便外面的构造函数调用，或者利用

这里有两点需要注意的：

- 如果可以用函数内`static`，就不要用`class`内的`static`。因为`class`内的`static`会要求不论变量是否产生都会构造析构一遍，反正少一点东西就是少一点麻烦。
- `static`和`inline`的互动需要考虑。这可能现在已经消除问题，但是还是建议不要让两者搭配，因为`inline`可能会导致有多个`static`副本，当然编译器可能会拒绝`inline`，因为这只是一个建议。

有的人可能会这么处理，用一个类内`static`变量标记有多少个这样的变量出现，但是要注意一个小问题，就是通过继承、包含的方式弄出来的变量也会导致这玩意+1，有可能会引发意料之外的`exception`。但是这玩意也有优点，就是可以设置为任意数字，而不像之前的方法只允许一个东西。

当然，你还可以把整个计数类作为一个基类，拿来private继承而不是放一个变量在里面。

```
template <class T>
class Counted
{
public:
    class TooManyObj{};
```

```
    static int objectCount() { return numObjects; }
protected:
    Counted() { init(); }
    Counted(const Counted& rhs) { init(); }
    ~Counted() { --numObjects; }
private:
    static int numObjects;
    static const size_t maxObjects;
    void init()
    {
        if(numObjects >= maxObjects) throw TooManyObj();
        ++numObjects;
    }
};
```

之后再用到的类里面放两行可以让外面也调用里面的东西出来。对于`static const`就不一定要求在哪里一定写上什么值，反正没写编译就不过是正常的。

```
using Counted<Printer>::objectCount;
using Counted<Printer>::TooManyObj;
```

Item 27 要求对象产生于heap之中

如果你要求让某种类型的对象有"自杀"能力，即`delete this`不报错的能力，那么你必须要让东西分配在heap中，而不是stack。

要求对象产生于heap中

为了让对象一定产生于heap中，我们必须要求客户不得以`new`之外的方法产生对象。通常来说，stack中的东西再定义点自动构造，而在寿命结束时自动析构，所以只要让隐式调用的构造、析构函数不合法就可以了。

通常来说最简单的办法是让其称为`private`方法，并且使用其他东西代偿。其中相对而言析构函数的屏蔽可能会比较方便，因为不一定能抓住所有的构造函数。而此时想要把东西析构掉只需要：

```
void destroy() { delete this; }
```

由于把构造析构函数设置成`private`会影响继承和内含，所以通常来说只需要把东西设置成`protected`就可以了。

判断某个对象是否在heap中

检测对象是否在heap中不是一件简单的事。而把`operator new`上打一个flag，并在之后的构造函数中检测的方法一般不会奏效，特别是应对数组初始化和用某个元素初始化的时候。就像

```
A* number = new A[100];
A* pn = new A(*new A);
```

这里需要强调的一点是，第二个的顺序不一定是预期中的先分配内存再构造，有可能是分配两段之后分别构造，这一样会引发问题。

在通常的程序空间中，stack向低地址走，而heap向高地址走。可以通过取地址的方法来判断是否在堆里面：

```
bool onHeap(const void* address)
{
    char onStack;
    return address < &onStack;
}
```

在原理上行得通，但是会有其他东西也可以通过检测，比如`static`对象、名称空间和全局变量。如果你在哪里遇到了困难，记得躺下。这时需要反过来看，你需要`delete this`的东西和heap里面的对象是否等价？其实不一定，万一有人想用内含的方法来构建自己的类，那么这个时候就`delete`不了，因为根本没有`new`出来。

此时回头来想，你仅仅是想要解决删除指针是否安全，可能不必这么大动干戈。有可能重新设计你自己的`new`和`delete`是一个好办法，用一个`set`维护一下就可以解决很多问题。但是这个时候就会有三个问题作为代价，你必须要了解。

- 你真的需要一个全局对象吗？你动了全局`new`和`delete`的话影响相当深远，而且堆的空间其实还是很宝贵的。
- 效率方面，你希望维护的是安全删除的地址，那么这一定要比原版的`new`慢上不少。
- 你不能保证安全删除这个动作总是稳定完成，特备是涉及到多重继承、虚拟继承的时候。

于是新的方法又出现了：还是包装，这次把需要的类都给包装起来，在里面重新定义`operator new`和`operator delete`。如果我们不希望有自行定义问题的话，可以考虑把析构函数设成纯虚函数。记得不同的指针虽然指向了同一片对象（就是一个继承体系、不同类指针表述的同一对象），但是可能会因为偏移而其值有所不同，所以需要使用`dynamic_cast<const void *>`来把东西打回原形。

禁止对象产生于heap中

说实话刚才的例子已经非常不错了，如果要求禁止对象产生于heap中，那么直接把分配机制掐断不就行了么？编译器的找内存独立于我们写的`operator new`，所以不存在那种情况。

Item 28 智能指针

通常来说，你使用智能指针，要的就是统一的构造析构、复制和赋值、解引用的方案，考虑到智能指针的普适性，`template`的时必须要有的。

这里会更多参考`auto_ptr`的实现，而不是更广为传播的`shared_ptr`的实现。

- 如果你的智能指针不应该出现复制构造函数和复制运算符，那么上`private`把东西屏蔽掉。
- 智能指针应当像内建指针一样使用。

智能指针的构造、赋值、析构

相对而言，构造和析构比较简单，因为只需要要求指针的自动化管理，而不需要去管理指针所指的内容。

```
template <class T>
class auto_ptr
{
public:
    auto_ptr(T* ptr = 0) : pointee(ptr) { }
    ~auto_ptr() { delete pointee;}
private:
    T* pointee;
};
```

此时如果发生赋值，通常来说有这些选择：

- 共享这个值。这么做需要做好标记，哪些或者有几个东西拿到了这个值，如果不标记好，何时析构是不明确的，不析构是要RTE的。
- 移交控制权。`auto_ptr`使用了这个方案，因为一是时间复杂度比较低，二是不容易出问题。注意一个问题：复制构造函数个`operator=`在这里的表现是不一样的。
- 复制一份。如果刚性需求、性能不在乎的话其实没啥。

哦对了，如果你打算通过移交控制权达到赋值目的的话，不要用by-value的方式传递函数参数，再怎么样也得用`const &`的方式来传递，不然东西都没了。

解引用

通常来说直接用`operator*`就行，`operator->`其实也差不多，仅仅是调用细节稍微有点差距。

```
template <class T>
T& SmartPtr<T>::operator*() const
{
    // 检验该指针是否有效
    return *pointee;
}
```

测试是否为nullptr

如果我们在原有定义上直接拿这玩意和0来比较，或者按照习惯的方式判断是否为空指针，那么你会拿到一个编译错误。为什么？因为你没有定义这个东西。所以这个时候我们需要新增一些隐式转换的操作来填补习惯。

- 隐式的类型转换。比如`operator void*()`，这会有无法避免不同类指针可以互相比较的问题。
- `operator!()`，这会比较影响代码书写，因为只允许!操作符，而`ptr==0`和`ptr`操作都会过不了编译。

智能指针转换为裸指针

这在之前要求会更进一步。这个时候就不要犹豫了，直接用`operator T*()`就可以。这样做会有一个优势和一个劣势：

- 之前的`!ptr`，`ptr==0`，`ptr`全部都能过测试。
- 会产生裸指针泄露的问题。这个问题其实相当严重，也是没法直接这么搞的原因，比如

```
auto_ptr<int> a = new int;
// ...
delete a;
```

理论上这是不能运行的，但是在隐式转换发生之后，就可以直接这么搞了。之后调用析构函数的时候就直接爆炸了。

所以说嘛，不要直接隐式转换成裸指针，提供一个`normalize`多香。

智能指针和继承

在继承体系中，我们写的这个智能指针可能效果不是很好。在继承体系中，这个类还是这个类，那个类就是那个类，智能指针哪里会管这么多？所以就需要一个强制转换的方法，让其能在继承体系中跳跃。

- 可以考虑单一跳转，只允许从一个智能指针类跳转到另一个，这种方法写的比较多，但是胜在稳定，完全可以自己把控。

- 可以考虑在其中再进行一次跳转。
- 不过不管怎么样，有句话不得不提：编译器禁止一次执行一个以上的“用户定制的类型转换函数”。

编译器在套用转换的时候，会寻找一些可以转换的方式：（比如把A类型转换为B类型）

- 在`auto_ptr<B>`找到`auto_ptr<A>`的单一自变量构造函数。
- 在`auto_ptr<A>`中找到`auto_ptr<B>`隐式转换操作符。
- 在`auto_ptr<A>`中找到可以实例化的`template`转换函数。

在使用这种方法时，会有三个缺点：

- 再有重载定义时，可能会造成二义性错误。在智能指针眼中，继承体系里面的类是互相独立的，并不存在“真正的“继承关系。
- 这种方法使用的不多，但是移植性不高，没有人知道一个没有流行的技术能走多远。
- 涉及到的知识不简单。需要了解的有：函数调用的自变量匹配规则，隐式类型转换函数，template functions的暗自实例化、成员函数模板。

智能指针与`const`

`const`自始自终是一个好玩的东西。在类型上面则不能修改指向的东西，但是可以修改指向哪里；在`*`附近则是不能修改指向位置，可以修改指向的东西。

但是在智能指针眼里，`const`带不带似乎就是两个类了。类型转换涉及`const`的，就是一条单行道：non-`const`转到`const`是安全的，反过来则是不安全的。对于`const`的操作可以对non-`const`操作，但是反过来就不行。

那么就可以考虑使用继承的方法来处理这个问题：先设置一个常量类，然后用另一个类来继承这玩意。不过要小心，这玩意要用到`union`。

Item 29 引用计数

引用计数这个技术在C++中可以参考一下`shared_ptr`，这主要是为了解决多个对象含有一样的值的问题，毕竟少点构造就可以少点空间和时间损耗。在这个条件下，我们现在需要做的事情其实不多：存储、记录重复次数。但是这个重复次数可能会引来很多麻烦。

引用计数的实现

通常来说，我们可以为每一个对应的对象设置一个引用次数，这样可以不用为每一个对象准备空间。这样可以以一个代理类的方式放在类里面，用`private`保护起来，外面一个指针，就可以完成了。此时如果调用多次相同的构造函数，那么引用计数同样不会增多。

```
String s1("Hello world!");
String s2("Hello world!");
// 这才是在当前条件下，能达到目的的调用
String s1("Hello world!");
String s2 = s1;
```

同时在引入引用次数这个概念之后，我们要注意只有在次数为0的时候才可以删除该对象，相对而言在`operator=`中更容易忽视这个问题。

对于刚才的“错误”调用，建议上个`map`做hash来处理。

写时才复制

在多个对象共有一块内存的时候，如果遇到了需要修改的情况，一个简单的思路是把东西拿出来，单独开一块内存，那边引用数目-1。但是在这类问题下，会遇到一个老生常谈的问题：如何判断`operator[]`的读写问题。虽然你可以保证`const`对象的`operator[]`是绝对人畜无害的，但是绝大部分情况下，你需要用没有`const`的东西，所以这种情况下，除了代理类以外，基本就是一刀切了。

指针、引用和写时才复制

刚才的问题看上去确实解决了，但是在某些新的场景下这个问题又会跑出来，比如说有个指针知道了里面去，现在要修改指针指向的内容，那么岂不是把一堆统统改掉了？这不是我们希望的效果。这个时候我们需要给原来的类中添加一个新的变量，用来标记该对象是否可以被共享，同时设定如下规则：

- 默认情况下这个东西是可以被共享的，如果有个新东西进来了就+1，反之-1。
- 如果因为`operator[]`导致可能被修改，那么将其分离的同时共享性反转。此时再有东西共享数值将会重新找一篇内存。

引用计数基类

在之前的基础上，可以考虑用一个基类来完成需要的结构，之后直接拿出来引用就行了。

```
class RCObject {
public:
    RCObject(): refCount(0), shareable(1) {}
    RCObject(const RCObject& rhs): refCount(0), shareable(1) {}
    RCObject& operator=(const RCObject& rhs) { return *this; }
    virtual ~RCObject() = 0;
    void addReference() { ++refCount; }
    void removeReference() { if(--refCount == 0) delete this; }
    void markUnshareable() { shareable = false; }
    bool isShareable() const { return shareable; }
    bool isShared() const { return refCount > 1; }
private:
    int refCount;
    bool shareable;
};
RCObject::~RCObject() {}
```

针对这段代码有问题是很正常的一件事情，特别是针对前面三个构造方法，来看看解释：

- 在大多数情况下，决定引用数应当交给程序员自己。随意预设会带来不可预料的后果，特别是同类之间和不同类之间的赋值是不一样的。
- 仔细思考一下，把一个这玩意复制到另一个身上的时候，我们应该做什么？这不应该导致修改值，如果真的修改了，应当体现在`addReferece`和`removeReference`里面。

后面的内容不想写了= =

不过要注意一下203页开始的代码，之前的努力都放在那里了。

如果想看下如何引用已有程序库的代码，建议看下209页。

Item 30 代理类

首先要注意一个问题：就是变量不可以作为数组大小，即数组的尺寸必须在编译期已知。C++甚至不支持一个与二维数组相关的堆内存分配行为。

实现二维数组

对于动态定义二维数组，除了使用`vector`套`vector`以外，还可以用一个`template`来实现。至于在这个类的构造函数里面实现什么，就不是程序本身控制的了，我们还可以进行迂回操作（比如`new`一堆出来）

```
template <class T>
class Array2D
{
public:
    Array2D(int dim1, int dim2);
    // ...
};
```

但是这会带来一个新的问题：就是我们如何找到其中的内容？比如说data[3][6]，显然是不存在这样的`operator`的，同样的你可以通过迂回操作，用`operator()`来定义一下，但是这样会带来使用手感上的麻烦：毕竟和`int`数组是不一样的。这两个迂回操作的本质区别是：一个是在实现方法上迂回，一个是在输入方式上迂回。

参考二维数组的某种理解方式，一个二维数组由一系列一维数组组成，而每一个一维数组都有固定数目的元素。那么如果我们以代理类的方式，实现类似的"一维数组"来顶替`operator[]`的结果，再在里面实现一个`operator[]`返回真正的int。

区分`operator[]`的读写动作

`operator[]`可以用于读写两个用途，在我们的期望下，读取不应该修改内容，而写入应当修改内容。在引用计数的情况下，修改内容应当引发引用计数的修改和新对象的产生。但是实际上不论是否用`const`识别，都会默认使用后一种，这是因为`const`是对于对象是否是`const`，而不是其行为是否是`const`。

所以新增一个代理类来顶替结果，在代理类内通过`operator`的类型转换和复制函数来区分左右值的运用，这个问题就解决了。

```
class String {
public:
    class CharProxy {
    public:
        CharProxy(String& str, int index);
        CharProxy& operator=(const CharProxy& rhs) {
            if(theString.value->isShared())
                theString.value = new StringValue(theString.value->data);
            theString.value->data[charIndex] = rhs.theString.value-
>data[rhs.charIndex];
            return *this;
        }
        CharProxy& operator=(char c) {
            if(theString.value->isShared())
                theString.value = new StringValue(theString.value->data);
            theString.value->data[charIndex] = c;
            return *this;
        }
        operator char() const { return theString.value->data[charIndex]; }
    private:
        String& theString;
        int CharIndex;
    };
    const CharProxy operator[](int index) const {
```

```
        return CharProxy(const_cast<String&>(*this), index);
    }
    CharProxy operator[](int index) {
        return CharProxy(*this, index);
    }
    friend class CharProxy;
private:
    RCPtr<String> value;
};
```

限制

代理类可以做到两件事情：一个是剥离我们的对象和进行的操作，将之具象化；另一个是关闭其他的操作途径，只允许已经定义过的内容。在第二点的要求下，直接对原内容取指就会编译不通过。主要有两个东西：没有`operator&`和转换无法完成。

所以直接定义一下不就可以了？在`const`版本中，我们直接拿出来，因为这不会影响是否共享；在非`const`版本中，应当和修改一样操作，因为你不能保证现在或者未来是否会被修改，而这类修改将会导致结构本身不自洽。

在对象处理上，这类方法同样有所限制。如果我们直接对数组进行套用，那么会导致你拿到的是一个代理类，本身是没有对应函数可供调用的，于是编译失败。除此以外，因为本身返回不是一个引用，这玩意在要求对应reference参数的函数中无法运作。这一切在剥离之后，你都要重新进行实现，这基本是不可承受的代价，因为每一个东西都要重新定义；也会有相当多的好处：至少能看到更多的编译错误。

评估

代理类能帮助我们做到一些事情。

- 多维数组。
- 左值右值的区分。
- 压抑隐式转换。

也同时你需要付出一些代价。

- 代理类是一种临时对象，需要被产生和被销毁。
- 代理类增加了系统的复杂度，使其产品更难被维护。
- `class`语义本身的改变，因为代理类把和真实对象合作改成了与替身对象合作。

Item 31 让函数根据一个以上的对象类型来决定如何虚化

这可能是一个头疼的问题，你要是说想还是想不到，但是居然还能经常遇到。

```
class GameObject {};
class SpaceShip: public GameObject {};
class SpaceStation: public GameObject {};
class Asteroid: public GameObject {};
```

于是挑战出来了，你如何实现这几个东西之间的碰撞？众所周知的是我们可以写九个函数分别处理，但是这傻不傻？好像有点，这简直就是浪费动态类型这个特点。于是你很容易就能写出一个入口函数，里面有个调用，然后就很尴尬地发现，这个调用并没有剖开整个问题，只是作为一个外界入口比较好。

虚函数+RTTI

我们可以让父类带一个纯虚函数，交给子类去实现，通过动态类型来决定调用者的类型；在里面进行不同类型的识别，于是乎就会产生这样的场面：

```
class GameObject {
    // ...
    virtual void collide(GameObject& otherObject) = 0;
};
class SpaceShip: public GameObject {
    // ...
    virtual void collide(GameObject& otherObject) {
        const type_info& objtype = typeid(otherObject);
        if(objtype == typeid(SpaceShip)) { /*...*/ }
        else if(objtype == typeid(SpaceStation)) { /*...*/ }
        else if(objtype == typeid(Asteroid)) { /*...*/ }
        else {}// Exception
    }
};
```

这样子实现看起来挺不错的，就是有一个不大不小的问题：你不知道什么时候会加一个新的类，那个时候就超级尴尬了。在那种条件下，可能别人对于这个需求漠不关心，但是你需要对你的代码作出适当的修改，但是所有人都不得不对于新的需求要修改和重新编译一遍，这都是时间。

只使用虚函数

如果我们只是用虚函数的话，那么接下来的应该就是一个滑稽场面：每一个类都要有对于四个的调用方法，当然里面有些简单的可以直接反过来调用就可以。

这个方法的问题和之前其实差不多，就是在不停的修改时意味着所有人都要不停编译，而且看着不难受么。

自行仿真虚函数表格

我们考虑这样做：

```
class GameObject {
public:
    virtual void collide(GameObject& rhs) = 0;
    // ...
};
class SpaceShip: public GameObject {
private:
    typedef void (SpaceShip::*HitFunction)(GameObject&);
    static HitFunction lookup(const GameObject& x);
    typedef map<string, HitFunction> HitMap;
public:
    virtual void collide(GameObject& rhs);
    virtual void hitSpaceShip(SpaceShip& rhs);
    virtual void hitSpaceStation(SpaceStation& rhs);
    virtual void hitAsteroid(Asteroid& rhs);
};
```

然后再`collide`里面调用另外三个即可。

```
void SpaceShip::collide(GameObject& rhs) {
    HitFunction hfp = lookup(rhs);
    if(hfp) (this->*hfp)(rhs);
    else throw();
}
```

其中`lookup`函数放了一个`map`，在这里面建立一个`class`和函数之间的关系。

```
SpaceShip::HitFunction SpaceShip::initializeCollisionMap() {
    // ...
}
SpaceShip::HitFunction SpaceShip::lookup(const GameObject& x) {
    static HitMap G = initializeCollisionMap();
    HitMap::iterator it = G.find(typeid(x).name());
    if(it == G.end()) return 0;
    return it->second; // (*it).second
}
```

在初始化的时候就会发现一个问题：传入参数都不一样，这会直接导致`map`的键值对写不进去。

将自行仿真的虚函数表格初始化

刚才的讨论是基于已经有这张表的基础上进行的操作，现在讨论的是如何把这个表建立成我们想要的样子，同时针对这个过程可能会对使用方法有所修改。在这个表的初始化过程中，我们需要插入三个键值对，未来可能会插入更多。从函数的返回值来看，我们需要针对这个`map`进行复制以便转移，可以直接返回指针来避免这个问题：

```
SpaceShip::HitFunction SpaceShip::initializeCollisionMap() {
    // ...
}
SpaceShip::HitFunction SpaceShip::lookup(const GameObject& x) {
    static auto_ptr<HitMap> G(initializeCollisionMap());
    HitMap::iterator it = G.find(typeid(x).name());
    if(it == G.end()) return 0;
    return it->second; // (*it).second
}
```

在指针的条件下就会有刚才的问题出现：函数指针会因为传入参数不对而写不进去。这样的解决办法是修改成对应的函数参数，并且在对应的处理中加入`dynamic_cast`。至少一个动态转型要比`reinterpret_cast`要好得多，后者可是有点欺骗编译器的行为，反而有可能会害了自己，前者还会在转型不成功的时候扔异常出来。

使用“非成员函数”的碰撞处理函数

这个方法看上去如此花里胡哨的，下一个方法把所有方法拿了出来，直接放在一个`namespace`里面。然后`map`要修改成`map<pair<string, string>, HitFunction>`，其他的操作方式做出适当修改就可以完成。

“继承”+“自行仿真的虚函数表格”

如果我们在其中丰富继承树的树枝，那么会得到一个更奇葩的情况：就是原来的函数不可以使用了。在原来的实现中，我们会使用动态类型的字符串作为输入，但是在从`map`中拿到准确函数的时候会要求函数名必须准确，于是编译出错。在这种情况下就真的没什么办法了，只能重新编译一下。

将自行仿真的虚函数表格初始化（再度讨论）

这个体系还是不很灵活：除了撞击这个动作之外，我们似乎无法添加更多的内容。这个时候就需要对`map`重新包装，利用`static`和`private`化的构造函数来确定唯一的方法管理数据库进行实现。