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

Item 30 代理类

Item 31 让函数根据一个以上的对象类型来决定如何虚化