2

构造/析构/赋值运算

Constructors, Destructors, and Assignment Operators

几乎你写的每一个 class 都会有一或多个构造函数、一个析构函数、一个 *copy assignment* 操作符。这些很难让你特别兴奋，毕竟它们是你的基本谋生工具，控制着基础操作，像是产出新对象并确保它被初始化、摆脱旧对象并确保它被适当清理、以及赋予对象新值。如果这些函数犯错，会导致深远且令人不愉快的后果，遍及你的整个 classes。所以确保它们行为正确是生死攸关的大事。本章提供的引导可让你把这些函数良好地集结在一起，形成 classes 的脊柱。

条款 05：了解 C++ 默默编写并调用哪些函数

Know what functions C++ silently writes and calls.

什么时候 empty class（空类）不再是个 empty class 呢？当 C++ 处理过它之后。是的，如果你自己没声明，编译器就会为它声明（编译器版本的）一个 *copy* 构造函数、一个 *copy assignment* 操作符和一个析构函数。此外如果你没有声明任何构造函数，编译器也会为你声明一个 *default* 构造函数。所有这些函数都是 public 且 inline（见条款 30）。因此，如果你写下：

```
class Empty { };
```

这就好像你写下这样的代码：

```
class Empty {
public:
    Empty() { ... }                                  //default 构造函数
    Empty(const Empty& rhs) { ... }                  //copy 构造函数
    ~Empty( ) { ... }                                //析构函数, 是否该是
                                                     // virtual 见稍后说明.
    Empty& operator=(const Empty& rhs) { ... }       //copy assignment 操作符.
};
```

惟有当这些函数被需要（被调用），它们才会被编译器创建出来。程序中需要它们是很平常的事。下面代码造成上述每一个函数被编译器产出：

```
Empty e1;                    //default构造函数
                             //析构函数
Empty e2(e1);                //copy构造函数
e2 = e1;                     //copy assignment操作符
```

好，我们知道了，编译器为你写函数，但这些函数做了什么呢？唔，*default* 构造函数和析构函数主要是给编译器一个地方用来放置“藏身幕后”的代码，像是调用 base classes 和 non-static 成员变量的构造函数和析构函数。注意，编译器产出的析构函数是个 non-virtual（见条款 7），除非这个 class 的 base class 自身声明有 virtual 析构函数（这种情况下这个函数的虚属性; virtualness; 主要来自 base class）。

至于 *copy* 构造函数和 *copy assignment* 操作符，编译器创建的版本只是单纯地将来源对象的每一个 non-static 成员变量拷贝到目标对象。考虑一个 NamedObject template，它允许你将一个个名称和类型为 T 的对象产生关联：

```
template<typename T>
class NamedObject {
public:
   NamedObject(const char* name, const T& value);
   NamedObject(const std::string& name, const T& value);
   ...
private:
   std::string nameValue;
   T objectValue;
};
```

由于其中声明了一个构造函数，编译器于是不再为它创建 *default* 构造函数。这很重要，意味如果你用心设计一个 class，其构造函数要求实参，你就无须担心编译器会毫无挂虑地为你添加一个无实参构造函数（即 *default* 构造函数）而遮盖掉你的版本。

NamedObject 既没有声明 *copy* 构造函数，也没有声明 *copy assignment* 操作符，所以编译器会为它创建那些函数（如果它们被调用的话）。现在，看看 *copy* 构造函数的用法：

```
NamedObject<int> no1("Smallest Prime Number", 2);
NamedObject<int> no2(no1);                    //调用 copy 构造函数
```

编译器生成的 *copy* 构造函数必须以 no1.nameValue 和 no1.objectValue 为初值设定 no2.nameValue 和 no2.objectValue。两者之中，nameValue 的类型是 string，而标准 string 有个 *copy* 构造函数，所以 no2.nameValue 的初始化方式是调用 string 的 *copy* 构造函数并以 no1.nameValue 为实参。另一个成员 NamedObject<int>::objectValue 的类型是 int（因为对此 template 具现体而言 T 是 int），那是个内置类型，所以 no2.objectValue 会以“拷贝 no1.objectValue 内的每一个 bits”来完成初始化。

编译器为 NamedObject<int> 所生的 *copy assignment* 操作符，其行为基本上与 *copy* 构造函数如出一辙，但一般而言只有当生出的代码合法且有适当机会证明它有意义（见下页），其表现才会如我先前所说。万一两个条件有一个不符合，编译器会拒绝为 class 生出 operator= 。

举个例子，假设 NamedObject 定义如下，其中 nameValue 是个 *reference to* string，objectValue 是个 const T：

```
template<class T>
class NamedObject {
public:
   //以下构造函数如今不再接受一个 const 名称，因为 nameValue
   //如今是个 reference-to-non-const string。先前那个 char* 构造函数
   //已经过去了，因为必须有个 string 可供指涉。
   NamedObject(std::string& name, const T& value);
   ...                          //如前，假设并未声明 operator=
private:
   std::string& nameValue;  //这如今是个 reference
   const T objectValue;     //这如今是个 const
};
```

现在考虑下面会发生什么事：

```
std::string newDog("Persephone");
std::string oldDog("Satch");
NamedObject<int> p(newDog, 2);   //当初撰写至此，我们的狗 Persephone
                                 //即将度过其第二个生日。
NamedObject<int> s(oldDog, 36);  //我小时候养的狗 Satch 则是 36 岁，
                                 //-- 如果她还活着。
p = s;                           //现在 p 的成员变量该发生什么事?
```

赋值之前，不论 p.nameValue 和 s.nameValue 都指向 string 对象（当然不是同一个）。赋值动作该如何影响 p.nameValue 呢？赋值之后 p.nameValue 应该

指向 s.nameValue 所指的那个 string 吗？也就是说 reference 自身可被改动吗？如果是，那可就开辟了新天地，因为 C++ 并不允许“让 reference 改指向不同对象”。换一个想法，p.nameValue 所指的那个 string 对象该被修改，进而影响“持有 pointers 或 references 而且指向该 string”的其他对象吗？也就是对象不被直接牵扯到赋值操作内？编译器生成的 *copy assignment* 操作符究竟该怎么做呢？

面对这个难题，C++ 的响应是拒绝编译那一行赋值动作。如果你打算在一个“内含 reference 成员”的 class 内支持赋值操作（assignment），你必须自己定义 *copy assignment* 操作符。面对“内含 const 成员”（如本例之 objectValue）的 classes，编译器的反应也一样。更改 const 成员是不合法的，所以编译器不知道如何在它自己生成的赋值函数内面对它们。最后还有一种情况：如果某个 base classes 将 *copy assignment* 操作符声明为 private，编译器将拒绝为其 derived classes 生成一个 *copy assignment* 操作符。毕竟编译器为 derived classes 所生的 *copy assignment* 操作符想象中可以处理 base class 成分（见条款 12），但它们当然无法调用 derived class 无权调用的成员函数。编译器两手一摊，无能为力。

请记住

- 编译器可以暗自为 class 创建 *default* 构造函数、*copy* 构造函数、*copy assignment* 操作符，以及析构函数。

条款 06：若不想使用编译器自动生成的函数，就该明确拒绝

Explicitly disallow the use of compiler-generated functions you do not want.

地产中介商卖的是房子，一个中介软件系统自然而然想必有个 class 用来描述待售房屋：

```
class HomeForSale { ... };
```

每一位真正的地产中介商都会说，任何一笔资产都是天上地下独一无二，没有两笔完全相像。因此我们也认为，为 HomeForSale 对象做一份副本有点没道理。你怎么可以复制某些先天独一无二的东西呢？因此，你应该乐意看到 HomeForSale 的对象拷贝动作以失败收场：

```
HomeForSale h1;
HomeForSale h2;
HomeForSale h3(h1);          //企图拷贝 h1 — 不该通过编译
h1 = h2;                     //企图拷贝 h2 — 也不该通过编译
```

啊呀，阻止这一类代码的编译并不是很直观。通常如果你不希望 class 支持某一特定机能，只要不声明对应函数就是了。但这个策略对 *copy* 构造函数和 *copy assignment* 操作符却不起作用，因为条款 5 已经指出，如果你不声明它们，而某些人尝试调用它们，编译器会为你声明它们。

这把你逼到了一个困境。如果你不声明 *copy* 构造函数或 *copy assignment* 操作符，编译器可能为你产出一份，于是你的 class 支持 *copying*。如果你声明它们，你的 class 还是支持 *copying*。但这里的目标却是要阻止 *copying*！

答案的关键是，所有编译器产出的函数都是 public。为阻止这些函数被创建出来，你得自行声明它们，但这里并没有什么需求使你必须将它们声明为 public。因此你可以将 *copy* 构造函数或 *copy assignment* 操作符声明为 private。藉由明确声明一个成员函数，你阻止了编译器暗自创建其专属版本；而令这些函数为 `private`，使你得以成功阻止人们调用它。

一般而言这个做法并不绝对安全，因为 member 函数和 friend 函数还是可以调用你的 private 函数。除非你够聪明，不去定义它们，那么如果某些人不慎调用任何一个，会获得一个连接错误（linkage error）。“将成员函数声明为 `private` 而且故意不实现它们”这一伎俩是如此为大家接受，因而被用在 C++ iostream 程序库中阻止 *copying* 行为。是的，看看你手上的标准程序库实现码中的 `ios_base`, `basic_ios` 和 `sentry`。你会发现无论哪一个，其 *copy* 构造函数和 *copy assignment* 操作符都被声明为 `private` 而且没有定义。

将这个伎俩施行于 `HomeForSale` 也很简单：

```
class HomeForSale {
public:
   ...
private:
   ...
   HomeForSale(const HomeForSale&);                    //只有声明
   HomeForSale& operator=(const HomeForSale&);
};
```

或许你注意到了，我没写函数参数的名称。唔，参数名称并非必要，只不过大家总是习惯写出来。这个函数毕竟不会被实现出来，也很少被使用，指定参数名称又有何用？

有了上述 class 定义，当客户企图拷贝 `HomeForSale` 对象，编译器会阻挠他。如果你不慎在 member 函数或 friend 函数之内那么做，轮到连接器发出抱怨。

将连接期错误移至编译期是可能的（而且那是好事，毕竟愈早侦测出错误愈好），只要将 *copy* 构造函数和 *copy assignment* 操作符声明为 private 就可以办到，但不是在 HomeForSale 自身，而是在一个专门为了阻止 *copying* 动作而设计的 base class 内。这个 base class 非常简单：

```
class Uncopyable {
protected:                                          //允许 derived 对象构造和析构
    Uncopyable() {}
    ~Uncopyable() { }
private:
    Uncopyable(const Uncopyable&);                  //但阻止 copying
    Uncopyable& operator=(const Uncopyable&);
};
```

为求阻止 HomeForSale 对象被拷贝，我们唯一需要做的就是继承 Uncopyable：

```
class HomeForSale: private Uncopyable {             //class 不再声明
    ...                                             //copy 构造函数或
};                                                  //copy assign. 操作符
```

这行得通，因为只要任何人——甚至是 member 函数或 friend 函数——尝试拷贝 HomeForSale 对象，编译器便试着生成一个 *copy* 构造函数和一个 *copy assignment* 操作符，而正如条款 12 所说，这些函数的“编译器生成版”会尝试调用其 base class 的对应兄弟，那些调用会被编译器拒绝，因为其 base class 的拷贝函数是 private。

Uncopyable class 的实现和运用颇为微妙，包括不一定得以 public 继承它（见条款 32 和 39），以及 Uncopyable 的析构函数不一定得是 virtual（见条款 7）等等。Uncopyable 不含数据，因此符合条款 39 所描述的 empty base class optimization 资格。但由于它总是扮演 base class，因此使用这项技术可能导致多重继承（译注：因为你往往还可能需要继承其他 class）（多重继承见条款 40），而多重继承有时会阻止 empty base class optimization（再次见条款 39）。通常你可以忽略这些微妙点，只像上面那样使用 Uncopyable，因为它完全像“广告”所说的能够正确运作。也可以使用 Boost（见条款 55）提供的版本，那个 class 名为 noncopyable，是个还不错的家伙，我只是认为其名称有点……呃……不太自然。

请记住

- 为驳回编译器自动（暗自）提供的机能，可将相应的成员函数声明为 private 并且不予实现。使用像 Uncopyable 这样的 base class 也是一种做法。

条款 07：为多态基类声明 virtual 析构函数

Declare destructors virtual in polymorphic base classes.

有许多种做法可以记录时间，因此，设计一个 TimeKeeper base class 和一些 derived classes 作为不同的计时方法，相当合情合理：

```
class TimeKeeper {
public:
    TimeKeeper();
    ~TimeKeeper();
    ...
};
class AtomicClock: public TimeKeeper { ... };   //原子钟
class WaterClock: public TimeKeeper { ... };    //水钟
class WristWatch: public TimeKeeper { ... };    //腕表
```

许多客户只想在程序中使用时间，不想操心时间如何计算等细节，这时候我们可以设计 factory（工厂）函数，返回指针指向一个计时对象。Factory 函数会“返回一个 base class 指针，指向新生成之 derived class 对象”：

```
TimeKeeper* getTimeKeeper();            //返回一个指针，指向一个
                                        //TimeKeeper 派生类的动态分配对象
```

为遵守 factory 函数的规矩，被 getTimeKeeper()返回的对象必须位于 heap。因此为了避免泄漏内存和其他资源，将 factory 函数返回的每一个对象适当地 delete 掉很重要：

```
TimeKeeper* ptk = getTimeKeeper();          //从 TimeKeeper 继承体系
                                            //获得一个动态分配对象。
...                                         //运用它...
delete ptk;                                 //释放它，避免资源泄漏。
```

条款 13 说“倚赖客户执行 delete 动作，基本上便带有某种错误倾向”，条款 18 则谈到 factory 函数接口该如何修改以便预防常见之客户错误，但这些在此都是次要的，因为此条款内我们要对付的是上述代码的一个更根本弱点：纵使客户把每一件事都做对了，仍然没办法知道程序如何行动。

问题出在 getTimeKeeper 返回的指针指向一个 derived class 对象（例如 AtomicClock），而那个对象却经由一个 base class 指针（例如一个 TimeKeeper*指针）被删除，而目前的 base class（TimeKeeper）有个 non-virtual 析构函数。

这是一个引来灾难的秘诀，因为 C++ 明白指出，当 derived class 对象经由一个 base class 指针被删除，而该 base class 带着一个 non-virtual 析构函数，其结果未有定义——实际执行时通常发生的是对象的 derived 成分没被销毁。如果 getTimeKeeper 返回指针指向一个 AtomicClock 对象，其内的 AtomicClock 成分（也就是声明于 AtomicClock class 内的成员变量）很可能没被销毁，而 AtomicClock 的析构函数也未能执行起来。然而其 base class 成分（也就是 TimeKeeper 这一部分）通常会被销毁，于是造成一个诡异的“局部销毁”对象。这可是形成资源泄漏、败坏之数据结构、在调试器上浪费许多时间的绝佳途径喔。

消除这个问题的做法很简单：给 base class 一个 virtual 析构函数。此后删除 derived class 对象就会如你想要的那般。是的，它会销毁整个对象，包括所有 derived class 成分：

```
class TimeKeeper {
public:
   TimeKeeper( );
   virtual ~TimeKeeper();
   ...
};
TimeKeeper* ptk = getTimeKeeper();
...
delete ptk;                                    //现在，行为正确。
```

像 TimeKeeper 这样的 base classes 除了析构函数之外通常还有其他 virtual 函数，因为 virtual 函数的目的是允许 derived class 的实现得以客制化（见条款 34）。例如 TimeKeeper 就可能拥有一个 virtual getCurrentTime，它在不同的 derived classes 中有不同的实现码。任何 class 只要带有 virtual 函数都几乎确定应该也有一个 virtual 析构函数。

如果 class 不含 virtual 函数，通常表示它并不意图被用做一个 base class。当 class 不企图被当作 base class，令其析构函数为 virtual 往往是个馊主意。考虑一个用来表示二维空间点坐标的 class：

```
class Point {                                  //一个二维空间点（2D point）
public:
   Point(int xCoord, int yCoord);
   ~Point();
private:
   int x, y;
};
```

如果 int 占用 32 bits，那么 Point 对象可塞入一个 64-bit 缓存器中。更有甚者，这样一个 Point 对象可被当做一个“64-bit 量”传给以其他语言如 C 或 FORTRAN 撰写的函数。然而当 Point 的析构函数是 virtual，形势起了变化。

欲实现出 virtual 函数，对象必须携带某些信息，主要用来在运行期决定哪一个 virtual 函数该被调用。这份信息通常是由一个所谓 vptr（virtual table pointer）指针指出。vptr 指向一个由函数指针构成的数组，称为 vtbl（virtual table）；每一个带有 virtual 函数的 class 都有一个相应的 vtbl。当对象调用某一 virtual 函数，实际被调用的函数取决于该对象的 vptr 所指的那个 vtbl——编译器在其中寻找适当的函数指针。

virtual 函数的实现细节不重要。重要的是如果 Point class 内含 virtual 函数，其对象的体积会增加：在 32-bit 计算机体系结构中将占用 64 bits（为了存放两个 ints）至 96 bits（两个 ints 加上 vptr）；在 64-bit 计算机体系结构中可能占用 64~128 bits，因为指针在这样的计算机结构中占 64 bits。因此，为 Point 添加一个 vptr 会增加其对象大小达 50%~100%!Point 对象不再能够塞入一个 64-bit 缓存器，而 C++ 的 Point 对象也不再和其他语言（如 C）内的相同声明有着一样的结构（因为其他语言的对应物并没有 vptr），因此也就不再可能把它传递至（或接受自）其他语言所写的函数，除非你明确补偿 vptr——那属于实现细节，也因此不再具有移植性。

因此，无端地将所有 classes 的析构函数声明为 virtual，就像从未声明它们为 virtual 一样，都是错误的。许多人的心得是：只有当 class 内含至少一个 virtual 函数，才为它声明 virtual 析构函数。

即使 class 完全不带 virtual 函数，被“non-virtual 析构函数问题”给咬伤还是有可能的。举个例子，标准 string 不含任何 virtual 函数，但有时候程序员会错误地把它当做 base class：

```
class SpecialString: public std::string {  //馊主意！std::string有个
   ...                                      //non-virtual 析构函数
};
```

乍看似乎无害，但如果你在程序任意某处无意间将一个 pointer-to-SpecialString

转换为一个 pointer-to-string，然后将转换所得的那个 string 指针 delete 掉，你立刻被流放到“行为不明确”的恶地上：

```
SpecialString* pss = new SpecialString("Impending Doom");
std::string* ps;
...
ps = pss;                  //SpecialString* => std::string*
...
delete ps;                 //未有定义！现实中*ps 的 SpecialString 资源会泄漏，
                           //因为 SpecialString 析构函数没被调用。
```

相同的分析适用于任何不带 virtual 析构函数的 class，包括所有 STL 容器如 vector, list, set, tr1::unordered_map（见条款 54）等等。如果你曾经企图继承一个标准容器或任何其他“带有 non-virtual 析构函数”的 class，拒绝诱惑吧！（很不幸 C ++ 没有提供类似 Java 的 final classes 或 C# 的 sealed classes 那样的“禁止派生”机制。）

有时候令 class 带一个 pure virtual 析构函数，可能颇为便利。还记得吗，pure virtual 函数导致 abstract（抽象）classes —— 也就是不能被实体化（instantiated）的 class。也就是说，你不能为那种类型创建对象。然而有时候你希望拥有抽象 class，但手上没有任何 pure virtual 函数，怎么办？唔，由于抽象 class 总是企图被当作一个 base class 来用，而又由于 base class 应该有个 virtual 析构函数，并且由于 pure virtual 函数会导致抽象 class，因此解法很简单：为你希望它成为抽象的那个 class 声明一个 pure virtual 析构函数。下面是个例子：

```
class AWOV {                          //AWOV = "Abstract w/o Virtuals"
public:
   virtual ~AWOV( ) = 0;              //声明 pure virtual 析构函数
};
```

这个 class 有一个 pure virtual 函数，所以它是个抽象 class，又由于它有个 virtual 析构函数，所以你不需要担心析构函数的问题。然而这里有个窍门：你必须为这个 pure virtual 析构函数提供一份定义：

```
AWOV::~AWOV() { }          //pure virtual 析构函数的定义
```

析构函数的运作方式是，最深层派生（most derived）的那个 class 其析构函数最先被调用，然后是其每一个 base class 的析构函数被调用。编译器会在 AWOV 的 derived

classes 的析构函数中创建一个对~AWOV 的调用动作，所以你必须为这个函数提供一份定义。如果不这样做，连接器会发出抱怨。

“给 base classes 一个 virtual 析构函数”，这个规则只适用于 polymorphic（带多态性质的）base classes 身上。这种 base classes 的设计目的是为了用来“通过 base class 接口处理 derived class 对象”。TimeKeeper 就是一个 polymorphic base class，因为我们希望处理 AtomicClock 和 WaterClock 对象，纵使我们只有 TimeKeeper 指针指向它们。

并非所有 base classes 的设计目的都是为了多态用途。例如标准 string 和 STL 容器都不被设计作为 base classes 使用，更别提多态了。某些 classes 的设计目的是作为 base classes 使用，但不是为了多态用途。这样的 classes 如条款 6 的 Uncopyable 和标准程序库的 input_iterator_tag（条款 47），它们并非被设计用来“经由 base class 接口处置 derived class 对象”，因此它们不需要 virtual 析构函数。

请记住

- polymorphic（带多态性质的）base classes 应该声明一个 virtual 析构函数。如果 class 带有任何 virtual 函数，它就应该拥有一个 virtual 析构函数。
- Classes 的设计目的如果不是作为 base classes 使用，或不是为了具备多态性（polymorphically），就不该声明 virtual 析构函数。

条款 08：别让异常逃离析构函数

Prevent exceptions from leaving destructors.

C++ 并不禁止析构函数吐出异常，但它不鼓励你这样做。这是有理由的。考虑以下代码：

```
class Widget {
public:
   ...
   ~Widget( ) { ... }          //假设这个可能吐出一个异常
};
void doSomething()
{
   std::vector<Widget> v;
   ...
}                              //v 在这里被自动销毁
```

当 vector v 被销毁，它有责任销毁其内含的所有 Widgets。假设 v 内含十个 Widgets，而在析构第一个元素期间，有个异常被抛出。其他九个 Widgets 还是应该被销毁（否则它们保存的任何资源都会发生泄漏），因此 v 应该调用它们各个析构函数。但假设在那些调用期间，第二个 Widget 析构函数又抛出异常。现在有两个同时作用的异常，这对 C++ 而言太多了。在两个异常同时存在的情况下，程序若不是结束执行就是导致不明确行为。本例中它会导致不明确的行为。使用标准程序库的任何其他容器（如 list, set）或 TR1 的任何容器（见条款 54）或甚至 array，也会出现相同情况。容器或 array 并非遇上麻烦的必要条件，只要析构函数吐出异常，即使并非使用容器或 arrays，程序也可能过早结束或出现不明确行为。是的，C++ 不喜欢析构函数吐出异常！

这很容易理解，但如果你的析构函数必须执行一个动作，而该动作可能会在失败时抛出异常，该怎么办？举个例子，假设你使用一个 class 负责数据库连接：

```
class DBConnection {
public:
   ...
   static DBConnection create();        //这个函数返回
                                        //  DBConnection 对象；
                                        //为求简化暂略参数。
   void close();                        //关闭联机；失败则抛出异常。
};
```

为确保客户不忘记在 DBConnection 对象身上调用 close()，一个合理的想法是创建一个用来管理 DBConnection 资源的 class，并在其析构函数中调用 close。这一类用于资源管理的 classes 在第 3 章有详细探讨，这儿只要考虑它们的析构函数长相就够了：

```
class DBConn {          //这个 class 用来管理 DBConnection 对象
public:
   ...
   ~DBConn()            //确保数据库连接总是会被关闭
   {
        db.close( );
   }
private:
   DBConnection db;
};
```

这便允许客户写出这样的代码：

```
{                                          //开启一个区块（block）。
DBConn dbc(DBConnection::create());        //建立 DBConnection 对象并
                                           //  交给 DBConn 对象以便管理。
...                                        //通过 DBConn 的接口
                                           //  使用 DBConnection 对象。
}                                          //在区块结束点，DBConn 对象
                                           //  被销毁，因而自动
                                           //  为 DBConnection 对象调用 close
```

只要调用 close 成功，一切都美好。但如果该调用导致异常，DBConn 析构函数会传播该异常，也就是允许它离开这个析构函数。那会造成问题，因为那就是抛出了难以驾驭的麻烦。

两个办法可以避免这一问题。DBConn 的析构函数可以：

- 如果 close 抛出异常就结束程序。通常通过调用 abort 完成：

```
DBConn::~DBConn( )
{
   try { db.close(); }
   catch (...) {
        制作运转记录，记下对 close 的调用失败；
        std::abort( );
   }
}
```

如果程序遭遇一个“于析构期间发生的错误”后无法继续执行，“强迫结束程序”是个合理选项。毕竟它可以阻止异常从析构函数传播出去（那会导致不明确的行为）。也就是说调用 abort 可以抢先制“不明确行为”于死地。

- 吞下因调用 close 而发生的异常：

```
DBConn::~DBConn( )
{
   try { db.close(); }
   catch (...) {
        制作运转记录，记下对 close 的调用失败；
   }
}
```

一般而言，将异常吞掉是个坏主意，因为它压制了“某些动作失败”的重要信息！然而有时候吞下异常也比负担“草率结束程序”或“不明确行为带来的风险”好。为了让这成为一个可行方案，程序必须能够继续可靠地执行，即使在遭遇并忽略一个错误之后。

这些办法都没什么吸引力。问题在于两者都无法对“导致 close 抛出异常”的情况做出反应。

一个较佳策略是重新设计 DBConn 接口，使其客户有机会对可能出现的问题作出反应。例如 DBConn 自己可以提供一个 close 函数，因而赋予客户一个机会得以处理“因该操作而发生的异常”。DBConn 也可以追踪其所管理之 DBConnection 是否已被关闭，并在答案为否的情况下由其析构函数关闭之。这可防止遗失数据库连接。然而如果 DBConnection 析构函数调用 close 失败，我们又将退回“强迫结束程序”或“吞下异常”的老路：

```
class DBConn {
public:
  ...
  void close( )                          //供客户使用的新函数
  {
    db.close( );
    closed = true;
  }
  ~DBConn()
  {
     if (!closed) {
        try {                            //关闭连接（如果客户不那么做的话）
           db.close( );
        }
        catch (...) {                                  //如果关闭动作失败，
           制作运转记录，记下对 close 的调用失败；     //  记录下来并结束程序
           ...                                         //  或吞下异常。
        }
     }
  }
private:
  DBConnection db;
  bool closed;
};
```

把调用 close 的责任从 DBConn 析构函数手上移到 DBConn 客户手上（但 DBConn 析构函数仍内含一个“双保险”调用）可能会给你“肆无忌惮转移负担”的印象。你甚至可能认为它违反条款 18 所提忠告（让接口容易被正确使用）。实际上这两项污名都不成立。如果某个操作可能在失败时抛出异常，而又存在某种需要必须处理该异常，那么这个异常必须来自析构函数以外的某个函数。因为析构函数吐出异常

就是危险，总会带来“过早结束程序”或“发生不明确行为”的风险。本例要说的是，由客户自己调用 close 并不会对他们带来负担，而是给他们一个处理错误的机会，否则他们没机会响应。如果他们不认为这个机会有用（或许他们坚信不会有错误发生），可以忽略它，倚赖 DBConn 析构函数去调用 close。如果真有错误发生——如果 close 的确抛出异常——而且 DBConn 吞下该异常或结束程序，客户没有立场抱怨，毕竟他们曾有机会第一手处理问题，而他们选择了放弃。

请记住

- 析构函数绝对不要吐出异常。如果一个被析构函数调用的函数可能抛出异常，析构函数应该捕捉任何异常，然后吞下它们（不传播）或结束程序。
- 如果客户需要对某个操作函数运行期间抛出的异常做出反应，那么 class 应该提供一个普通函数（而非在析构函数中）执行该操作。

条款 09：绝不在构造和析构过程中调用 virtual 函数

Never call virtual functions during construction or destruction.

本条款开始前我要先阐述重点：你不该在构造函数和析构函数期间调用 virtual 函数，因为这样的调用不会带来你预想的结果，就算有你也不会高兴。如果你同时也是一位 Java 或 C# 程序员，请更加注意本条款，因为这是 C++ 与它们不相同的一个地方。

假设你有个 class 继承体系，用来塑模股市交易如买进、卖出的订单等等。这样的交易一定要经过审计，所以每当创建一个交易对象，在审计日志（audit log）中也需要创建一笔适当记录。下面是一个看起来颇为合理的做法：

```
class Transaction {                                    //所有交易的 base class
public:
   Transaction( );
   virtual void logTransaction() const = 0;            //做出一份因类型不同而不同
                                                       //的日志记录（log entry）
   ...
};
```

```
Transaction::Transaction()                          //base class 构造函数之实现
{
   ...
   logTransaction();                                //最后动作是志记这笔交易
}
class BuyTransaction: public Transaction {          //derived class
public:
   virtual void logTransaction() const;             //志记（log）此型交易
   ...
};
class SellTransaction: public Transaction {         //derived class
public:
   virtual void logTransaction() const;             //志记（log）此型交易
   ...
};
```

现在，当以下这行被执行，会发生什么事：

```
BuyTransaction b;
```

无疑地会有一个 BuyTransaction 构造函数被调用，但首先 Transaction 构造函数一定会更早被调用；是的，derived class 对象内的 base class 成分会在 derived class 自身成分被构造之前先构造妥当。Transaction 构造函数的最后一行调用 virtual 函数 logTransaction，这正是引发惊奇的起点。这时候被调用的 logTransaction 是 Transaction 内的版本，不是 BuyTransaction 内的版本——即使目前即将建立的对象类型是 BuyTransaction。是的，base class 构造期间 virtual 函数绝不会下降到 derived classes 阶层。取而代之的是，对象的作为就像隶属 base 类型一样。非正式的说法或许比较传神：在 base class 构造期间，virtual 函数不是 virtual 函数。

这一似乎反直觉的行为有个好理由。由于 base class 构造函数的执行更早于 derived class 构造函数，当 base class 构造函数执行时 derived class 的成员变量尚未初始化。如果此期间调用的 virtual 函数下降至 derived classes 阶层，要知道 derived class 的函数几乎必然取用 local 成员变量，而那些成员变量尚未初始化。这将是一张通往不明确行为和彻夜调试大会串的直达车票。“要求使用对象内部尚未初始化的成分”是危险的代名词，所以 C++ 不让你走这条路。

其实还有比上述理由更根本的原因：在 derived class 对象的 base class 构造期间，

对象的类型是 base class 而不是 derived class。不只 virtual 函数会被编译器解析至（*resolve to*）base class，若使用运行期类型信息（runtime type information，例如 dynamic_cast（见条款 27）和 typeid），也会把对象视为 base class 类型。本例之中，当 Transaction 构造函数正执行起来打算初始化“BuyTransaction 对象内的 base class 成分”时，该对象的类型是 Transaction。那是每一个 C++ 次成分（见条款 1）的态度，而这样的对待是合理的：这个对象内的“BuyTransaction 专属成分”尚未被初始化，所以面对它们，最安全的做法就是视它们不存在。对象在 derived class 构造函数开始执行前不会成为一个 derived class 对象。

相同道理也适用于析构函数。一旦 derived class 析构函数开始执行，对象内的 derived class 成员变量便呈现未定义值，所以 C++ 视它们仿佛不再存在。进入 base class 析构函数后对象就成为一个 base class 对象，而 C++ 的任何部分包括 virtual 函数、dynamic_casts 等等也就那么看待它。

在上述示例中，Transaction 构造函数直接调用一个 virtual 函数，这很明显而且容易看出违反本条款。由于它很容易被看出来，某些编译器会为此发出一个警告信息（某些则否，见条款 53 对警告信息的讨论）。即使没有这样的警告，这个问题在执行前也几乎肯定会变得显而易见，因为 logTransaction 函数在 Transaction 内是个 pure virtual。除非它被定义（不太有希望，但是有可能，见条款 34）否则程序无法连接，因为连接器找不到必要的 Transaction:: logTransaction 实现代码。

但是侦测“构造函数或析构函数运行期间是否调用 virtual 函数”并不总是这般轻松。如果 Transaction 有多个构造函数，每个都需执行某些相同工作，那么避免代码重复的一个优秀做法是把共同的初始化代码（其中包括对 logTransaction 的调用）放进一个初始化函数如 init 内：

```
class Transaction {
public:
    Transaction( )
    { init( ); }                                    //调用 non-virtual...
    virtual void logTransaction() const = 0;
    ...
private:
    void init()
    {
        ...
        logTransaction();                           //这里调用 virtual!
    }
};
```

这段代码概念上和稍早版本相同，但它比较潜藏并且暗中为害，因为它通常不会引发任何编译器和连接器的抱怨。此时由于 logTransaction 是 Transaction 内的一个 pure virtual 函数，当 pure virtual 函数被调用，大多执行系统会中止程序（通常会对此结果发出一个信息）。然而如果 logTransaction 是个正常的（也就是 impure）virtual 函数并在 Transaction 内带有一份实现代码，该版本就会被调用，而程序也就会兴高采烈地继续向前行，留下你百思不解为什么建立一个 derived class 对象时会调用错误版本的 logTransaction。唯一能够避免此问题的做法就是：确定你的构造函数和析构函数都没有（在对象被创建和被销毁期间）调用 virtual 函数，而它们调用的所有函数也都服从同一约束。

但你如何确保每次一有 Transaction 继承体系上的对象被创建，就会有适当版本的 logTransaction 被调用呢？很显然，在 Transaction 构造函数(s)内对着对象调用 virtual 函数是一种错误做法。

其他方案可以解决这个问题。一种做法是在 class Transaction 内将 logTransaction 函数改为 non-virtual，然后要求 derived class 构造函数传递必要信息给 Transaction 构造函数，而后那个构造函数便可安全地调用 non-virtual logTransaction。像这样：

```
class Transaction {
public:
   explicit Transaction(const std::string& logInfo);
   void logTransaction(const std::string& logInfo) const; //如今是个
                                                          //non-virtual 函数
   ...
};

Transaction::Transaction(const std::string& logInfo)
{
   ...
   logTransaction(logInfo);                               //如今是个
                                                          //non-virtual 调用
}

class BuyTransaction: public Transaction {
public:
   BuyTransaction( parameters )
     : Transaction(createLogString( parameters ))        //将 log 信息
   { ... }                                                //传给 base class 构造函数
   ...
private:
   static std::string createLogString( parameters );
};
```

换句话说由于你无法使用 virtual 函数从 base classes 向下调用，在构造期间，你可以藉由“令 derived classes 将必要的构造信息向上传递至 base class 构造函数”替换之而加以弥补。

请注意本例之 BuyTransaction 内的 private static 函数 createLogString 的运用。是的，比起在成员初值列（member initialization list）内给予 base class 所需数据，利用辅助函数创建一个值传给 base class 构造函数往往比较方便（也比较可读）。令此函数为 static，也就不可能意外指向“初期未成熟之 BuyTransaction 对象内尚未初始化的成员变量”。这很重要，正是因为“那些成员变量处于未定义状态”，所以“在 base class 构造和析构期间调用的 virtual 函数不可下降至 derived classes”。

请记住

- 在构造和析构期间不要调用 virtual 函数，因为这类调用从不下降至 derived class（比起当前执行构造函数和析构函数的那层）。

条款 10：令 operator= 返回一个 *reference to* *this

Have assignment operators return a reference to *this.

关于赋值，有趣的是你可以把它们写成连锁形式：

```
int x, y, z;
x = y = z = 15;                    //赋值连锁形式
```

同样有趣的是，赋值采用右结合律，所以上述连锁赋值被解析为：

```
x = (y = (z = 15));
```

这里 15 先被赋值给 z，然后其结果（更新后的 z）再被赋值给 y，然后其结果（更新后的 y）再被赋值给 x。

为了实现“连锁赋值”，赋值操作符必须返回一个 reference 指向操作符的左侧实参。这是你为 classes 实现赋值操作符时应该遵循的协议：

```
class Widget {
public:
   ...
```

```
    Widget& operator=(const Widget& rhs)          //返回类型是个 reference，
    {                                             //  指向当前对象。
        ...
        return* this;                             //返回左侧对象
    }
    ...
};
```

这个协议不仅适用于以上的标准赋值形式，也适用于所有赋值相关运算，例如：

```
class Widget {
public:
    ...
    Widget& operator+=(const Widget& rhs)         //这个协议适用于
    {                                             //  +=,-=,*=, 等等。
        ...
        return *this;
    }
    Widget& operator=(int rhs)                    //此函数也适用，即使
    {                                             //  此一操作符的参数类型
        ...                                       //  不符协定。
        return *this;
    }
    ...
};
```

注意，这只是个协议，并无强制性。如果不遵循它，代码一样可通过编译。然而这份协议被所有内置类型和标准程序库提供的类型如 string, vector, complex, tr1::shared_ptr 或即将提供的类型（见条款 54）共同遵守。因此除非你有一个标新立异的好理由，不然还是随众吧。

请记住

- 令赋值（*assignment*）操作符返回一个 reference to *this。

条款 11：在 operator= 中处理“自我赋值”

Handle assignment to self in operator=.

“自我赋值”发生在对象被赋值给自己时：

```
class Widget { ... };
Widget w;
...
w = w;                                  //赋值给自己
```

这看起来有点愚蠢，但它合法，所以不要认定客户绝不会那么做。此外赋值动

作并不总是那么可被一眼辨识出来，例如：

```
a[i] = a[j];        //潜在的自我赋值
```

如果 i 和 j 有相同的值，这便是个自我赋值。再看：

```
*px = *py;          //潜在的自我赋值
```

如果 px 和 py 恰巧指向同一个东西，这也是自我赋值。这些并不明显的自我赋值，是“别名”（*aliasing*）带来的结果：所谓“别名”就是“有一个以上的方法指称（指涉）某对象”。一般而言如果某段代码操作 pointers 或 references 而它们被用来“指向多个相同类型的对象”，就需考虑这些对象是否为同一个。实际上两个对象只要来自同一个继承体系，它们甚至不需声明为相同类型就可能造成“别名”，因为一个 base class 的 reference 或 pointer 可以指向一个 derived class 对象：

```
class Base { ... };
class Derived: public Base { ... };
void doSomething(const Base& rb,      //rb 和*pd 有可能其实是同一对象
                 Derived* pd);
```

如果遵循条款 13 和条款 14 的忠告，你会运用对象来管理资源，而且你可以确定所谓“资源管理对象”在 *copy* 发生时有正确的举措。这种情况下你的赋值操作符或许是“自我赋值安全的”（self-assignment-safe），不需要额外操心。然而如果你尝试自行管理资源（如果你打算写一个用于资源管理的 class 就得这样做），可能会掉进“在停止使用资源之前意外释放了它”的陷阱。假设你建立一个 class 用来保存一个指针指向一块动态分配的位图（bitmap）：

```
class Bitmap { ... };
class Widget {
   ...
private:
   Bitmap* pb;          //指针，指向一个从 heap 分配而得的对象
};
```

下面是 operator= 实现代码，表面上看起来合理，但自我赋值出现时并不安全（它也不具备异常安全性，但我们稍后才讨论这个主题）。

```
Widget&
Widget::operator=(const Widget& rhs)  //一份不安全的 operator= 实现版本.
{
   delete pb;                         //停止使用当前的 bitmap,
   pb = new Bitmap(*rhs.pb);          //使用 rhs's bitmap 的副本（复件）。
   return *this;                      //见条款 10。
}
```

这里的自我赋值问题是，operator= 函数内的 *this（赋值的目的端）和 rhs 有可能是同一个对象。果真如此 delete 就不只是销毁当前对象的 bitmap，它也销毁 rhs 的 bitmap。在函数末尾，Widget——它原本不该被自我赋值动作改变的——发现自己持有一个指针指向一个已被删除的对象！

欲阻止这种错误，传统做法是藉由 operator=最前面的一个“证同测试（identity test）”达到“自我赋值”的检验目的：

```
Widget& Widget::operator=(const Widget& rhs)
{
    if (this == &rhs) return *this;    //证同测试（identity test）：
                                       //如果是自我赋值，就不做任何事。
    delete pb;
    pb = new Bitmap(*rhs.pb);
    return *this;
}
```

这样做行得通。稍早我曾经提过，前一版 operator= 不仅不具备“自我赋值安全性”，也不具备“异常安全性”，这个新版本仍然存在异常方面的麻烦。更明确地说，如果 "new Bitmap" 导致异常（不论是因为分配时内存不足或因为 Bitmap 的 *copy* 构造函数抛出异常），Widget 最终会持有一个指针指向一块被删除的 Bitmap。这样的指针有害。你无法安全地删除它们，甚至无法安全地读取它们。唯一能对它们做的安全事情是付出许多调试能量找出错误的起源。

令人高兴的是，让 operator= 具备“异常安全性”往往自动获得“自我赋值安全”的回报。因此愈来愈多人对“自我赋值”的处理态度是倾向不去管它，把焦点放在实现“异常安全性”（exception safety）上。条款 29 深度探讨了异常安全性，本条款只要你注意“许多时候一群精心安排的语句就可以导出异常安全（以及自我赋值安全）的代码”，这就够了。例如以下代码，我们只需注意在复制 pb 所指东西之前别删除 pb：

```
Widget& Widget::operator=(const Widget& rhs)
{
    Bitmap* pOrig = pb;              //记住原先的 pb
    pb = new Bitmap(*rhs.pb);        //令 pb 指向 *pb 的一个复件（副本）
    delete pOrig;                    //删除原先的 pb
    return *this;
}
```

现在，如果 "new Bitmap" 抛出异常，pb（及其栖身的那个 Widget）保持原状。即使没有证同测试（identity test），这段代码还是能够处理自我赋值，因为我们对原 bitmap 做了一份复件、删除原 bitmap、然后指向新制造的那个复件。它或许不是处理“自我赋值”的最高效办法，但它行得通。

如果你很关心效率，可以把“证同测试”（identity test）再次放回函数起始处。然而这样做之前先问问自己，你估计“自我赋值”的发生频率有多高？因为这项测试也需要成本。它会使代码变大一些（包括原始码和目标码）并导入一个新的控制流（control flow）分支，而两者都会降低执行速度。Prefetching、caching 和 pipelining 等指令的效率都会因此降低。

在 operator= 函数内手工排列语句（确保代码不但“异常安全”而且“自我赋值安全”）的一个替代方案是，使用所谓的 copy and swap 技术。这个技术和“异常安全性”有密切关系，所以由条款 29 详细说明。然而由于它是一个常见而够好的 operator= 撰写办法，所以值得看看其实现手法像什么样子：

```
class Widget {
...
void swap(Widget& rhs);      //交换*this 和 rhs 的数据；详见条款 29
...
};
Widget& Widget::operator=(const Widget& rhs)
{
    Widget temp(rhs);        //为 rhs 数据制作一份复件（副本）
    swap(temp);              //将*this 数据和上述复件的数据交换。
    return *this;
}
```

这个主题的另一个变奏曲乃利用以下事实：(1) 某 class 的 *copy assignment* 操作符可能被声明为“以 *by value* 方式接受实参”；(2) 以 *by value* 方式传递东西会造成一份复件/副本（见条款 20）：

```
Widget& Widget::operator=(Widget rhs) //rhs 是被传对象的一份复件（副本）
{                                     //注意这里是 pass by value.
    swap(rhs);                        //将*this 的数据和复件/副本的数据互换
    return *this;
}
```

我个人比较忧虑这个做法，我认为它为了伶俐巧妙的修补而牺牲了清晰性。然而将“*copying*动作”从函数本体内移至“函数参数构造阶段”却可令编译器有时生成更高效的代码。

请记住

- 确保当对象自我赋值时 operator= 有良好行为。其中技术包括比较“来源对象”和“目标对象”的地址、精心周到的语句顺序、以及 copy-and-swap。
- 确定任何函数如果操作一个以上的对象，而其中多个对象是同一个对象时，其行为仍然正确。

条款 12：复制对象时勿忘其每一个成分

Copy all parts of an object.

设计良好之面向对象系统（OO-systems）会将对象的内部封装起来，只留两个函数负责对象拷贝（复制），那便是带着适切名称的 *copy* 构造函数和 *copy assignment* 操作符，我称它们为 *copying* 函数。条款 5 观察到编译器会在必要时候为我们的 classes 创建 *copying* 函数，并说明这些“编译器生成版”的行为：将被拷对象的所有成员变量都做一份拷贝。

如果你声明自己的 *copying* 函数，意思就是告诉编译器你并不喜欢缺省实现中的某些行为。编译器仿佛被冒犯似的，会以一种奇怪的方式回敬：当你的实现代码几乎必然出错时却不告诉你。

考虑一个 class 用来表现顾客，其中手工写出（而非由编译器创建）*copying* 函数，使得外界对它们的调用会被志记（logged）下来：

```
void logCall(const std::string& funcName);        //制造一个 log entry
class Customer {
public:
   ...
   Customer(const Customer& rhs);
   Customer& operator=(const Customer& rhs);
   ...
private:
   std::string name;
};
```

```
Customer::Customer(const Customer& rhs)
  : name(rhs.name)                                    //复制 rhs 的数据
{
   logCall("Customer copy constructor");
}
Customer& Customer::operator=(const Customer& rhs)
{
   logCall("Customer copy assignment operator");
   name = rhs.name;                                   //复制 rhs 的数据
   return *this;                                      //见条款 10
}
```

这里的每一件事情看起来都很好，而实际上每件事情也的确都好，直到另一个成员变量加入战局：

```
class Date { ... };                  //日期
class Customer {
public:
   ...                               //同前
private:
   std::string name;
   Date lastTransaction;
};
```

这时候既有的 *copying* 函数执行的是局部拷贝（partial copy）：它们的确复制了顾客的 name，但没有复制新添加的 lastTransaction。大多数编译器对此不出任何怨言——即使在最高警告级别中（见条款 53）。这是编译器对“你自己写出 *copying* 函数”的复仇行为：既然你拒绝它们为你写出 *copying* 函数，如果你的代码不完全，它们也不告诉你。结论很明显：如果你为 class 添加一个成员变量，你必须同时修改 *copying* 函数。（你也需要修改 class 的所有构造函数（见条款 4 和条款 45）以及任何非标准形式的 operator=（条款 10 有个例子）。如果你忘记，编译器不太可能提醒你。）

一旦发生继承，可能会造成此一主题最暗中肆虐的一个潜藏危机。试考虑：

```
class PriorityCustomer: public Customer {        //一个 derived class
public:
   ...
   PriorityCustomer(const PriorityCustomer& rhs);
   PriorityCustomer& operator=(const PriorityCustomer& rhs);
   ...
private:
   int priority;
};
```

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
  : priority(rhs.priority)
{
   logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
   logCall("PriorityCustomer copy assignment operator");
   priority = rhs.priority;
   return *this;
}
```

PriorityCustomer 的 ***copying*** 函数看起来好像复制了 PriorityCustomer 内的每一样东西，但是请再看一眼。是的，它们复制了 PriorityCustomer 声明的成员变量，但每个 PriorityCustomer 还内含它所继承的 Customer 成员变量复件（副本），而那些成员变量却未被复制。PriorityCustomer 的 ***copy*** 构造函数并没有指定实参传给其 base class 构造函数（也就是说它在它的成员初值列（member initialization list）中没有提到 Customer），因此 PriorityCustomer 对象的 Customer 成分会被不带实参之 Customer 构造函数（即 ***default*** 构造函数——必定有一个否则无法通过编译）初始化。***default*** 构造函数将针对 name 和 lastTransaction 执行缺省的初始化动作。

以上事态在 PriorityCustomer 的 ***copy assignment*** 操作符身上只有轻微不同。它不曾企图修改其 base class 的成员变量，所以那些成员变量保持不变。

任何时候只要你承担起“为 derived class 撰写 ***copying*** 函数”的重责大任，必须很小心地也复制其 base class 成分。那些成分往往是 private（见条款 22），所以你无法直接访问它们，你应该让 derived class 的 ***copying*** 函数调用相应的 base class 函数：

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
  : Customer(rhs),                          //调用 base class 的 copy 构造函数
    priority(rhs.priority)
{
   logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
   logCall("PriorityCustomer copy assignment operator");
   Customer::operator=(rhs);                //对 base class 成分进行赋值动作
   priority = rhs.priority;
   return *this;
}
```

本条款题目所说的“复制每一个成分”现在应该很清楚了。当你编写一个 *copying* 函数，请确保 (1) 复制所有 local 成员变量，(2) 调用所有 base classes 内的适当的 *copying* 函数。

这两个 *copying* 函数往往有近似相同的实现本体，这可能会诱使你让某个函数调用另一个函数以避免代码重复。这样精益求精的态度值得赞赏，但是令某个 *copying* 函数调用另一个 *copying* 函数却无法让你达到你想要的目标。

令 *copy assignment* 操作符调用 *copy* 构造函数是不合理的，因为这就像试图构造一个已经存在的对象。这件事如此荒谬，乃至于根本没有相关语法。是有一些看似如你所愿的语法，但其实不是；也的确有些语法背后真正做了它，但它们在某些情况下会造成你的对象败坏，所以我不打算将那些语法呈现给你看。单纯地接受这个叙述吧：你不该令 *copy assignment* 操作符调用 *copy* 构造函数。

反方向——令 *copy* 构造函数调用 *copy assignment* 操作符——同样无意义。构造函数用来初始化新对象，而 *assignment* 操作符只施行于已初始化对象身上。对一个尚未构造好的对象赋值，就像在一个尚未初始化的对象身上做“只对已初始化对象才有意义”的事一样。无聊嘛！别尝试。

如果你发现你的 *copy* 构造函数和 *copy assignment* 操作符有相近的代码，消除重复代码的做法是，建立一个新的成员函数给两者调用。这样的函数往往是 private 而且常被命名为 `init`。这个策略可以安全消除 *copy* 构造函数和 *copy assignment* 操作符之间的代码重复。

请记住

- *Copying* 函数应该确保复制“对象内的所有成员变量”及“所有 base class 成分”。
- 不要尝试以某个 *copying* 函数实现另一个 *copying* 函数。应该将共同机能放进第三个函数中，并由两个 *coping* 函数共同调用。