6

继承与面向对象设计

Inheritance and Object-Oriented Design

面向对象编程（OOP）几乎已经风靡两个年代了，所以关于继承、派生、virtual 函数等等，可能你已经有了一些经验。纵使你过去只以 C 编写程序，如今肯定也无法逃脱 OOP 的笼罩。

尽管如此，C++ 的 OOP 有可能和你原本习惯的 OOP 稍有不同：“继承”可以是单一继承或多重继承，每一个继承连接（link）可以是 public, protected 或 private，也可以是 virtual 或 non-virtual。然后是成员函数的各个选项：virtual？non-virtual？pure virtual？以及成员函数和其他语言特性的交互影响：缺省参数值与 virtual 函数有什么交互影响？继承如何影响 C++ 的名称查找规则？设计选项有哪些？如果 class 的行为需要修改，virtual 函数是最佳选择吗？

本章对这些题目全面宣战。此外我也解释 C++ 各种不同特性的真正意义，也就是当你使用某个特定构件你真正想要表达的意思。例如“public 继承”意味 "is-a"，如果你尝试让它带着其他意义，你会惹祸上身。同样道理，virtual 函数意味“接口必须被继承”，non-virtual 函数意味“接口和实现都必须被继承”。如果不能区分这些意义，会造成 C++ 程序员大量的苦恼。

如果你了解 C++ 各种特性的意义，你会发现，你对 OOP 的看法改变了。它不再是一项用来划分语言特性的仪典，而是可以让你通过它说出你对软件系统的想法。一旦你知道该通过它说些什么，移转至 C++ 世界也就不再是可怕的高要求了。

条款 32：确定你的 public 继承塑模出 is-a 关系

Make sure public inheritance models "is-a."

在《*Some Must Watch While Some Must Sleep*》（W. H. Freeman and Company, 1974）这本书中，作者 William Dement 说了一个故事，谈到他曾经试图让学生记下课程中最重要的一些教导。书上说，他告诉他的班级，一般英国学生对于发生在 1066 年的黑斯廷斯（Hastings）战役所知不多。如果有学生记得多一些，Dement 强调，无非也只是记得 1066 这个数字而已。然后 Dement 继续其课程，其中只有少数重要信息，包括“安眠药反而造成失眠症”这类有趣的事情。他一再要求学生，纵使忘了课程中的其他每一件事，也要记住这些数量不多的重要事情。Dement 在整个学期中不断耳提面命这样的话。

课程结束后，期末考的最后一道题目是：“写下你从本课程获得的一件永生不忘的事”。当 Dement 批改试卷，他目瞪口呆。几乎每一个人都写下 "1066"。

这就是为什么现在我要戒慎恐惧地对你声明，以 C++ 进行面向对象编程，最重要的一个规则是：public inheritance（公开继承）意味 "is-a"（是一种）的关系。把这个规则牢牢地烙印在你的心中吧！

如果你令 class D（"Derived"）以 public 形式继承 class B（"Base"），你便是告诉 C++ 编译器（以及你的代码读者）说，每一个类型为 D 的对象同时也是一个类型为 B 的对象，反之不成立。你的意思是 B 比 D 表现出更一般化的概念，而 D 比 B 表现出更特殊化的概念。你主张“B 对象可派上用场的任何地方，D 对象一样可以派上用场”（译注：此即所谓 Liskov Substitution Principle），因为每一个 D 对象都是一种（是一个）B 对象。反之如果你需要一个 D 对象，B 对象无法效劳，因为虽然每个 D 对象都是一个 B 对象，反之并不成立。

C++ 对于“public 继承”严格奉行上述见解。考虑以下例子：

```
class Person { ... };
class Student: public Person { ... };
```

根据生活经验我们知道，每个学生都是人，但并非每个人都是学生。这便是这个继承体系的主张。我们预期，对人可以成立的每一件事——例如每个人都有生日——对学生也都成立。但我们并不预期对学生可成立的每一件事——例如他或她

注册于某所学校——对人也成立。人的概念比学生更一般化，学生是人的一种特殊形式。

于是，承上所述，在 C++ 领域中，任何函数如果期望获得一个类型为 Person（或 pointer-to-Person 或 reference-to-Person）的实参，都也愿意接受一个 Student 对象（或 pointer-to-Student 或 reference-to-Student）：

```
void eat(const Person& p);          //任何人都会吃
void study(const Student& s);       //只有学生才到校学习
Person p;                           //p 是人
Student s;                          //s 是学生
eat(p);                             //没问题，p 是人
eat(s);                             //没问题，s 是学生，而学生也是（is-a）人
study(s);                           //没问题，s 是个学生
study(p);                           //错误！p 不是个学生
```

这个论点只对 public 继承才成立。只有当 Student 以 public 形式继承 Person，C++的行为才会如我所描述。private 继承的意义与此完全不同（见条款 39），至于 protected 继承，那是一种其意义至今仍然困惑我的东西。

public 继承和 **is-a** 之间的等价关系听起来颇为简单，但有时候你的直觉可能会误导你。举个例子，企鹅（penguin）是一种鸟，这是事实。鸟可以飞，这也是事实。如果我们天真地以 C++ 描述这层关系，结果如下：

```
class Bird {
public:
   virtual void fly();              //鸟可以飞
   ...
};
class Penguin: public Bird {        //企鹅是一种鸟
   ...
};
```

突然间我们遇上了乱流，因为这个继承体系说企鹅可以飞，而我们知道那不是真的。怎么回事？

在这个例子中，我们成了不严谨语言（英语）下的牺牲品。当我们说鸟会飞的时候，我们真正的意思并不是说所有的鸟都会飞，我们要说的只是一般的鸟都有飞行能力。如果谨慎一点，我们应该承认一个事实：有数种鸟不会飞。我们来到以下

继承关系，它塑模出较佳的真实性：

```
class Bird {
   ...                                  //没有声明 fly 函数
};

class FlyingBird: public Bird {
public:
   virtual void fly();
   ...
};

class Penguin: public Bird {
   ...                                  //没有声明 fly 函数
};
```

这样的继承体系比原先的设计更能忠实反映我们真正的意思。

即便如此，此刻我们仍然未能完全处理好这些鸟事，因为对某些软件系统而言，可能不需要区分会飞的鸟和不会飞的鸟。如果你的程序忙着处理鸟喙和鸟翅，完全不在乎飞行，原先的“双 classes 继承体系”或许就相当令人满足了。这反映出一个事实，世界上并不存在一个“适用于所有软件”的完美设计。所谓最佳设计，取决于系统希望做什么事，包括现在与未来。如果你的程序对飞行一无所知，而且也不打算未来对飞行“有所知”，那么不去区分会飞的鸟和不会飞的鸟，不失为一个完美而有效的设计。实际上它可能比“对两者做出区隔”更受欢迎，因为这样的区隔在你企图塑模的世界中并不存在。

另有一种思想派别处理我所谓“所有的鸟都会飞，企鹅是鸟，但是企鹅不会飞，喔欧”的问题，就是为企鹅重新定义 fly 函数，令它产生一个运行期错误：

```
void error(const std::string& msg); //定义于另外某处

class Penguin: public Bird {
public:
   virtual void fly() { error("Attempt to make a penguin fly!"); }
   ...
};
```

很重要的是，你必须认知这里所说的某些东西可能和你所想的不同。这里并不是说“企鹅不会飞”，而是说“企鹅会飞，但尝试那么做是一种错误”。

如何描述其间的差异？从错误被侦测出来的时间点观之，“企鹅不会飞”这一限制可由编译期强制实施，但若违反“企鹅尝试飞行，是一种错误”这一条规则，只有运行期才能检测出来。

为了表现“企鹅不会飞，就这样”的限制，你不可以为 Penguin 定义 fly 函数：

```
class Bird {
   ...                                  //没有声明 fly 函数
};
class Penguin: public Bird {
   ...                                  //没有声明 fly 函数
};
```

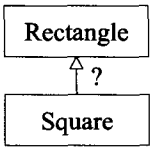
现在，如果你试图让企鹅飞，编译器会对你的背信加以谴责：

```
Penguin p;
p.fly( );                               //错误!
```

这和采取“令程序于运行期发生错误”的解法极为不同。若以那种做法，编译器不会对 p.fly 调用式发出任何抱怨。条款 18 说过：好的接口可以防止无效的代码通过编译，因此你应该宁可采取“在编译期拒绝企鹅飞行”的设计，而不是“只在运行期才能侦测它们”的设计。

或许你承认你对鸟类缺乏直觉，但基础几何学得不错。喔，是吗？那么我请问，正方形和矩形之间可能有多么复杂？

好，请回答这个简单的问题：class Square 应该以 public 形式继承 class Rectangle 吗？



“咄！”你说，“当然应该如此！每个人都知道正方形是一种矩形，反之则不一定”，这是真理，至少学校是这么教的。但是我不认为我们还在象牙塔内。

考虑这段代码：

```
class Rectangle {
public:
   virtual void setHeight(int newHeight);
   virtual void setWidth(int newWidth);
   virtual int height( ) const;          //返回当前值
   virtual int width() const;
   ...
};

void makeBigger(Rectangle& r)            //这个函数用以增加 r 的面积
{
   int oldHeight = r.height( );
   r.setWidth(r.width() + 10);           //为 r 的宽度加 10
   assert(r.height( ) == oldHeight);     //判断 r 的高度是否未曾改变
}
```

显然，上述的 assert 结果永远为真。因为 makeBigger 只改变 r 的宽度；r 的高度从未被更改。

现在考虑这段代码，其中使用 public 继承，允许正方形被视为一种矩形：

```
class Square: public Rectangle { ... };
Square s;
...
assert(s.width() == s.height());         //这对所有正方形一定为真。
makeBigger(s);                           //由于继承，s 是一种（is-a）矩形，
                                         //所以我们可以增加其面积。
assert(s.width() == s.height());         //对所有正方形应该仍然为真。
```

这也很明显，第二个 assert 结果也应该永远为真。因为根据定义，正方形的宽度和其高度相同。

但现在我们遇上了一个问题。我们如何调解下面各个 assert 判断式：

- 调用 makeBigger 之前，s 的高度和宽度相同；
- 在 makeBigger 函数内，s 的宽度改变，但高度不变；

- makeBigger 返回之后，s 的高度再度和其宽度相同。（注意 s 是以 *by reference* 方式传给 makeBigger，所以 makeBigger 修改的是 s 自身，不是 s 的副本。）

怎么样？

欢迎来到“public 继承”的精彩世界。你在其他领域（包括数学）学习而得的直觉，在这里恐怕无法如预期般地帮助你。本例的根本困难是，某些可施行于矩形身上的事情（例如宽度可独立于其高度被外界修改）却不可施行于正方形身上（宽度总是应该和高度一样）。但是 public 继承主张，能够施行于 base class 对象身上的每件事情，每件事情啃，也可以施行于 derived class 对象身上。在正方形和矩形例子中（另一个类似例子是条款 38 的 sets 和 lists），那样的主张无法保持，所以以 public 继承塑模它们之间的关系并不正确。编译器会让你通过，但是一如我们所见，这并不保证程序的行为正确。就像每一位程序员一定学过的（某些人也许比其他人更常学到）：代码通过编译并不表示就可以正确运作。

不要因为你发展经年的软件直觉在与面向对象观念打交道的过程中失去效用，便心慌意乱起来。那些知识还是有价值的，但现在你已经为你的“设计”军械库加上继承（inheritance）这门大炮，你也必须为你的直觉添加新的洞察力，以便引导你适当运用“继承”这一支神兵利器。当有一天有人展示一个长达数页的函数给你看，你终将回忆起“令 Penguin 继承 Bird，或是令 Square 继承 Rectangle”的概念和趣味；这样的继承有可能接近事实真象，但也有可能不。

is-a 并非是唯一存在于 classes 之间的关系。另两个常见的关系是 **has-a**（有一个）和 **is-implemented-in-terms-of**（根据某物实现出）。这些关系将在条款 38 和 39 讨论。将上述这些重要的相互关系中的任何一个误塑为 **is-a** 而造成的错误设计，在 C++ 中并不罕见，所以你应该确定你确实了解这些个“classes 相互关系”之间的差异，并知道如何在 C++ 中最好地塑造它们。

请记住

- “public 继承”意味 **is-a**。适用于 base classes 身上的每一件事情一定也适用于 derived classes 身上，因为每一个 derived class 对象也都是一个 base class 对象。

条款 33：避免遮掩继承而来的名称

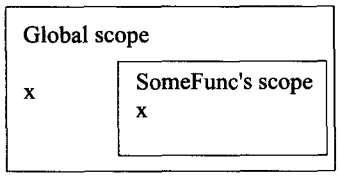
Avoid hiding inherited names.

关于“名称”，莎士比亚说过这样一句话：“名称是什么呢？”他问，“一朵玫瑰叫任何名字还是一样芬芳。”吟游诗人也写过这样的话：“偷了我的好名字的人呀……害我变得好可怜。”完全正确。这把我们引到了 C++“继承而来的名称”。

这个题材和继承其实无关，而是和作用域（scopes）有关。我们都知道在诸如这般的代码中：

```
int x;                          //global 变量
void someFunc()
{
    double x;                   //local 变量
    std::cin >> x;              //读一个新值赋予 local 变量 x
}
```

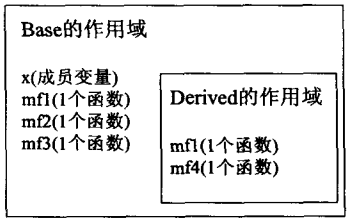
这个读取数据的语句指涉的是 local 变量 x，而不是 global 变量 x，因为内层作用域的名称会遮掩（遮蔽）外围作用域的名称。我们可以这样看本例的作用域形势：



当编译器处于 someFunc 的作用域内并遭遇名称 x 时，它在 local 作用域内查找是否有什么东西带着这个名称。如果找到就不再找其他作用域。本例的 someFunc 的 x 是 double 类型而 global x 是 int 类型，但那不要紧。C++ 的名称遮掩规则（name-hiding rules）所做的唯一事情就是：遮掩名称。至于名称是否应和相同或不同的类型，并不重要。本例中一个名为 x 的 double 遮掩了一个名为 x 的 int。

现在导入继承。我们知道，当位于一个 derived class 成员函数内指涉（*refer to*）base class 内的某物（也许是个成员函数、typedef、或成员变量）时，编译器可以找出我们所指涉的东西，因为 derived classes 继承了声明于 base classes 内的所有东西。实际运作方式是，derived class 作用域被嵌套在 base class 作用域内，像这样：

```
class Base {
private:
   int x;
public:
   virtual void mf1() = 0;
   virtual void mf2();
   void mf3();
   ...
};
class Derived: public Base {
public:
   virtual void mf1();
   void mf4();
   ...
};
```



此例内含一组混合了 public 和 private 名称，以及一组成员变量和成员函数名称。这些成员函数包括 pure virtual, impure virtual 和 non-virtual 三种，这是为了强调我们谈的是名称，和其他无关。这个例子也可以加入各种名称类型，例如 enums, nested classes 和 typedefs。整个讨论中唯一重要的是这些东西的名称，至于这些东西是什么并不重要。本例使用单一继承，然而一旦了解单一继承下发生的事，很容易就可以推想 C++ 在多重继承下的行为。

假设 derived class 内的 mf4 的实现码部分像这样：

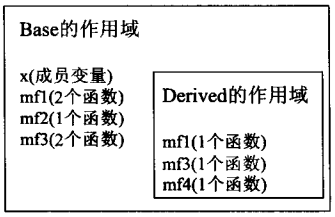
```
void Derived::mf4( )
{
   ...
   mf2();
   ...
}
```

当编译器看到这里使用名称 mf2，必须估算它指涉（*refer to*）什么东西。编译器的做法是查找各作用域，看看有没有某个名为 mf2 的声明式。首先查找 local 作用域（也就是 mf4 覆盖的作用域），在那儿没找到任何东西名为 mf2。于是查找其外围作用域，也就是 class Derived 覆盖的作用域。还是没找到任何东西名为 mf2，于是再往外围移动，本例为 base class。在那儿编译器找到一个名为 mf2 的东西了，于是停止查找。如果 Base 内还是没有 mf2，查找动作便继续下去，首先找内含 Base 的那个 namespace(s) 的作用域（如果有的话），最后往 global 作用域找去。

刚才我描述的程序虽然精确，但范围不够广。我们的目标并不是为了知道撰写编译器必须实践的名称查找规则，而是希望知道足够的信息，用以避免发生让人不快的惊讶。对于后者，现在我们有了丰富的信息。

再次考虑前一个例子，这次让我们重载 mf1 和 mf3，并且添加一个新版 mf3 到 Derived 去。如条款 36 所说，这里发生的事情是：Derived 重载了 mf3，那是一个继承而来的 non-virtual 函数。这会使整个设计立刻显得疑云重重，但为了充分认识继承体系内的“名称可视性”，我们暂时安之若素。

```
class Base {
private:
    int x;
public:
    virtual void mf1() = 0;
    virtual void mf1(int);
    virtual void mf2();
    void mf3();
    void mf3(double);
    ...
};

class Derived: public Base {
public:
    virtual void mf1();
    void mf3();
    void mf4();
    ...
};
```



这段代码带来的行为会让每一位第一次面对它的 C++ 程序员大吃一惊。以作用域为基础的“名称遮掩规则”并没有改变，因此 base class 内所有名为 mf1 和 mf3 的函数都被 derived class 内的 mf1 和 mf3 函数遮掩掉了。从名称查找观点来看，Base::mf1 和 Base::mf3 不再被 Derived 继承！

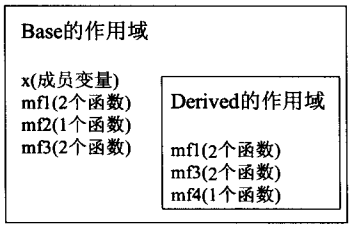
```
Derived d;
int x;
...
d.mf1();                //没问题，调用 Derived::mf1
d.mf1(x);               //错误！因为 Derived::mf1 遮掩了 Base::mf1
d.mf2();                //没问题，调用 Base::mf2
d.mf3();                //没问题，调用 Derived::mf3
d.mf3(x);               //错误！因为 Derived::mf3 遮掩了 Base::mf3
```

如你所见，上述规则都适用，即使 base classes 和 derived classes 内的函数有不同的参数类型也适用，而且不论函数是 virtual 或 non-virtual 一体适用。这和本条款一开始展示的道理相同，当时函数 someFunc 内的 double x 遮掩了 global 作用域内的 int x，如今 Derived 内的函数 mf3 遮掩了一个名为 mf3 但类型不同的 Base 函数。

这些行为背后的基本理由是为了防止你在程序库或应用框架（application framework）内建立新的 derived class 时附带地从疏远的 base classes 继承重载函数。不幸的是你通常会想继承重载函数。实际上如果你正在使用 public 继承而又不继承那些重载函数，就是违反 base 和 derived classes 之间的 **is-a** 关系，而条款 32 说过 **is-a** 是 public 继承的基石。因此你几乎总会想要推翻（override）C++ 对“继承而来的名称”的缺省遮掩行为。

你可以使用 using 声明式达成目标：

```
class Base {
private:
    int x;
public:
    virtual void mf1() = 0;
    virtual void mf1(int);
    virtual void mf2();
    void mf3();
    void mf3(double);
    ...
};

class Derived: public Base {
public:
    using Base::mf1;     //让 Base class 内名为 mf1 和 mf3 的所有东西
    using Base::mf3;     //在 Derived 作用域内都可见（并且 public）
    virtual void mf1();
    void mf3();
    void mf4();
    ...
};
```



现在，继承机制将一如往昔地运作：

```
Derived d;
int x;
...
d.mf1();      //仍然没问题，仍然调用 Derived::mf1
d.mf1(x);     //现在没问题了，调用 Base::mf1
d.mf2();      //仍然没问题，仍然调用 Base::mf2
d.mf3();      //没问题，调用 Derived::mf3
d.mf3(x);     //现在没问题了，调用 Base::mf3
```

这意味如果你继承 base class 并加上重载函数，而你又希望重新定义或覆写（推翻）其中一部分，那么你必须为那些原本会被遮掩的每个名称引入一个 using 声明式，否则某些你希望继承的名称会被遮掩。

有时候你并不想继承 base classes 的所有函数，这是可以理解的。在 public 继承下，这绝对不可能发生，因为它违反了 public 继承所暗示的“base 和 derived classes 之间的**is-a**关系”。（这也就是为什么上述 using 声明式被放在 derived class 的 public 区域的原因：base class 内的 public 名称在 publicly derived class 内也应该是 public。）然而在 private 继承之下（见条款 39）它却可能是有意义的。例如假设 Derived 以 private 形式继承 Base，而 Derived 唯一想继承的 mf1 是那个无参数版本。using 声明式在这里派不上用场，因为 using 声明式会令继承而来的某给定名称之所有同名函数在 derived class 中都可见。不，我们需要不同的技术，即一个简单的转交函数（forwarding function）：

```
class Base {
public:
   virtual void mf1() = 0;
   virtual void mf1(int);
   ...                    //与前同
};

class Derived: private Base {
public:
   virtual void mf1()  //转交函数（forwarding function）;
   { Base::mf1( ); }   //暗自成为 inline（见条款 30）
   ...
};

...
Derived d;
int x;
d.mf1();               //很好，调用的是 Derived::mf1
d.mf1(x);              //错误！Base::mf1()被遮掩了
```

inline 转交函数（forwarding function）的另一个用途是为那些不支持 using 声明式（注：这并非正确行为）的老旧编译器另辟一条新路，将继承而得的名称汇入 derived class 作用域内。

这就是继承和名称遮掩的完整故事。但是当继承结合 templates，我们又将面对“继承名称被遮掩”的一个全然不同的形式。关于“以角括号定界”的所有东西，详见条款 43。

请记住

- derived classes 内的名称会遮掩 base classes 内的名称。在 public 继承下从来没有人希望如此。
- 为了让被遮掩的名称再见天日，可使用 using 声明式或转交函数（forwarding functions）。

条款 34：区分接口继承和实现继承

Differentiate between inheritance of interface and inheritance of implementation.

表面上直截了当的 public 继承概念，经过更严密的检查之后，发现它由两部分组成：函数接口（function interfaces）继承和函数实现（function implementations）继承。这两种继承的差异，很像本书导读所讨论的函数声明与函数定义之间的差异。

身为 class 设计者，有时候你会希望 derived classes 只继承成员函数的接口（也就是声明）；有时候你又会希望 derived classes 同时继承函数的接口和实现，但又希望能够覆写（*override*）它们所继承的实现；又有时候你希望 derived classes 同时继承函数的接口和实现，并且不允许覆写任何东西。

为了更好地感觉上述选择之间的差异，让我们考虑一个展现绘图程序中各种几何形状的 class 继承体系：

```
class Shape {
public:
   virtual void draw( ) const = 0;
   virtual void error(const std::string& msg);
   int objectID( ) const;
   ...
};

class Rectangle: public Shape { ... };
class Ellipse: public Shape { ... };
```

Shape 是个抽象 class；它的 pure virtual 函数 draw 使它成为一个抽象 class。所以客户不能够创建 Shape class 的实体，只能创建其 derived classes 的实体。尽管如此，Shape 还是强烈影响了所有以 public 形式继承它的 derived classes，因为：

- 成员函数的接口总是会被继承。一如条款 32 所说，public 继承意味 **is-a**（是一种），所以对 base class 为真的任何事情一定也对其 derived classes 为真。因此如果某个函数可施行于某 class 身上，一定也可施行于其 derived classes 身上。

Shape class 声明了三个函数。第一个是 draw，于某个隐喻的视屏中画出当前对象。第二个是 error，准备让那些“需要报导某个错误”的成员函数调用。第三个是 objectID，返回当前对象的一个独一无二的整数识别码。每个函数的声明方式都不相同：draw 是个 pure virtual 函数；error 是个简朴的（非纯）impure virtual 函数；objectID 是个 non-virtual 函数。这些不同的声明带来什么样的暗示呢？

首先考虑 pure virtual 函数 draw：

```
class Shape {
public:
   virtual void draw( ) const = 0;
   ...
};
```

pure virtual 函数有两个最突出的特性：它们必须被任何“继承了它们”的具象 class 重新声明，而且它们在抽象 class 中通常没有定义。把这两个性质摆在一起，你就会明白：

- 声明一个 pure virtual 函数的目的是为了让 derived classes 只继承函数接口。

这对 Shape::draw 函数是再合理不过的事了，因为所有 Shape 对象都应该是可绘出的，这是合理的要求。但 Shape class 无法为此函数提供合理的缺省实现，毕竟椭圆形绘法迥异于矩形绘法。Shape::draw 的声明式乃是对具象 derived classes 设计者说，“你必须提供一个 draw 函数，但我不干涉你怎么实现它。”

令人意外的是，我们竟然可以为 pure virtual 函数提供定义。也就是说你可以为 Shape::draw 供应一份实现代码，C++ 并不会发出怨言，但调用它的唯一途径是“调用时明确指出其 class 名称”：

```
Shape* ps = new Shape;              //错误！Shape 是抽象的
Shape* ps1 = new Rectangle;         //没问题
ps1->draw( );                       //调用 Rectangle::draw
Shape* ps2 = new Ellipse;           //没问题
ps2->draw();                        //调用 Ellipse::draw
ps1->Shape::draw( );                //调用 Shape::draw
ps2->Shape::draw( );                //调用 Shape::draw
```

除了能够帮助你在鸡尾酒派对上留给大师级程序员一个深刻的印象，一般而言这项性质用途有限。但是一如稍后你将看到，它可以实现一种机制，为简朴的（非纯）impure virtual 函数提供更平常更安全的缺省实现。

简朴的 impure virtual 函数背后的故事和 pure virtual 函数有点不同。一如往常，derived classes 继承其函数接口，但 impure virtual 函数会提供一份实现代码，derived classes 可能覆写（*override*）它。稍加思索，你就会明白：

- 声明简朴的（非纯）impure virtual 函数的目的，是让 derived classes 继承该函数的接口和缺省实现。

考虑 `Shape::error` 这个例子：

```
class Shape {
public:
   virtual void error(const std::string& msg);
   ...
};
```

其接口表示，每个 class 都必须支持一个“当遇上错误时可调用”的函数，但每个 class 可自由处理错误。如果某个 class 不想针对错误做出任何特殊行为，它可以退回到 `Shape` class 提供的缺省错误处理行为。也就是说 `Shape::error` 的声明式告诉 derived classes 的设计者，“你必须支持一个 `error` 函数，但如果你不想自己写一个，可以使用 `Shape` class 提供的缺省版本”。

但是，允许 impure virtual 函数同时指定函数声明和函数缺省行为，却有可能造成危险。欲探讨原因，让我们考虑 XYZ 航空公司设计的飞机继承体系。该公司只有 A 型和 B 型两种飞机，两者都以相同方式飞行。因此 XYZ 设计出这样的继承体系：

```
class Airport { ... };                    //用以表现机场
class Airplane {
public:
   virtual void fly(const Airport& destination);
   ...
};
void Airplane::fly(const Airport& destination)
{
   缺省代码，将飞机飞至指定的目的地
}
class ModelA: public Airplane { ... };
class ModelB: public Airplane { ... };
```

为了表示所有飞机都一定能飞，并阐明“不同型飞机原则上需要不同的 fly 实现”，Airplane::fly 被声明为 virtual。然而为了避免在 ModelA 和 ModelB 中撰写相同代码，缺省飞行行为由 Airplane::fly 提供，它同时被 ModelA 和 ModelB 继承。

这是个典型的面向对象设计。两个 classes 共享一份相同性质（也就是它们实现 fly 的方式），所以共同性质被搬到 base class 中，然后被这两个 classes 继承。这个设计突显出共同性质，避免代码重复，并提升未来的强化能力，减缓长期维护所需的成本。所有这些都是面向对象技术如此受到欢迎的原因。XYZ 航空公司应该感到骄傲。

现在，假设 XYZ 盈余大增，决定购买一种新式 C 型飞机。C 型和 A 型以及 B 型有某些不同。更明确地说，它的飞行方式不同。

XYZ 公司的程序员在继承体系中针对 C 型飞机添加了一个 class，但由于他们急着让新飞机上线服务，竟忘了重新定义其 fly 函数：

```
class ModelC: public Airplane {
   ...                                    //未声明 fly 函数
};
```

然后代码中有一些诸如此类的动作：

```
Airport PDX(...);                         //PDX 是我家附近的机场
Airplane* pa = new ModelC;
...
pa->fly(PDX);                             //调用 Airplane::fly
```

这将酿成大灾难；这个程序试图以 ModelA 或 ModelB 的飞行方式来飞 ModelC。这不是一个可以公开鼓舞旅游信心的行为。

问题不在 Airplane::fly 有缺省行为，而在于 ModelC 在未明白说出“我要”的情况下就继承了该缺省行为。幸运的是我们可以轻易做到“提供缺省实现给 derived classes，但除非它们明白要求否则免谈”。此间技俩在于切断“virtual 函数接口”和其“缺省实现”之间的连接。下面是一种做法：

```
class Airplane {
public:
virtual void fly(const Airport& destination) = 0;
   ...
protected:
   void defaultFly(const Airport& destination);
};
void Airplane::defaultFly(const Airport& destination)
{
   缺省行为，将飞机飞至指定的目的地。
}
```

请注意，Airplane::fly 已被改为一个 pure virtual 函数，只提供飞行接口。其缺省行为也出现在 Airplane class 中，但此次系以独立函数 defaultFly 的姿态出现。若想使用缺省实现（例如 ModelA 和 ModelB），可以在其 fly 函数中对 defaultFly 做一个 inline 调用（但请注意条款 30 所言，inline 函数和 virtual 函数之间的交互关系）：

```
class ModelA: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }
   ...
};
class ModelB: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }
   ...
};
```

现在 ModelC class 不可能意外继承不正确的 fly 实现代码了，因为 Airplane 中的 pure virtual 函数迫使 ModelC 必须提供自己的 fly 版本：

```
class ModelC: public Airplane {
public:
   virtual void fly(const Airport& destination);
   ...
};
void ModelC::fly(const Airport& destination)
{
   将C型飞机飞至指定的目的地
}
```

这个方案并非安全无虞，程序员还是可能因为剪贴（copy-and-paste）代码而招来麻烦，但它的确比原先的设计值得倚赖。至于 Airplane::defaultFly，请注意它现在成了 protected，因为它是 Airplane 及其 derived classes 的实现细目。乘客应该只在意飞机能不能飞，不在意它们怎么飞。

Airplane::defaultFly 是个 non-virtual 函数，这一点也很重要。因为没有任何一个 derived class 应该重新定义此函数（见条款 36）。如果 defaultFly 是 virtual 函数，就会出现一个循环问题：万一某些 derived class 忘记重新定义 defaultFly，会怎样？

有些人反对以不同的函数分别提供接口和缺省实现，像上述的 fly 和 defaultFly 那样。他们关心因过度雷同的函数名称而引起的 class 命名空间污染问题。但是他们也同意，接口和缺省实现应该分开。这个表面上看起来的矛盾该如何解决？唔，我们可以利用“pure virtual 函数必须在 derived classes 中重新声明，但它们也可以拥有自己的实现”这一事实。下面便是 Airplane 继承体系如何给 pure virtual 函数一份定义：

```
class Airplane {
public:
   virtual void fly(const Airport& destination) = 0;
   ...
};
```

```
void Airplane::fly(const Airport& destination)  //pure virtual 函数实现
{
   缺省行为，将飞机飞至指定的目的地
}
class ModelA: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { Airplane::fly(destination); }
   ...
};
class ModelB: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { Airplane::fly(destination); }
   ...
};
class ModelC: public Airplane {
public:
   virtual void fly(const Airport& destination);
   ...
};
void ModelC::fly(const Airport& destination)
{
   将C型飞机飞至指定的目的地
}
```

这几乎和前一个设计一模一样，只不过 pure virtual 函数 `Airplane::fly` 替换了独立函数 `Airplane::defaultFly`。本质上，现在的 `fly` 被分割为两个基本要素：其声明部分表现的是接口（那是 derived classes 必须使用的），其定义部分则表现出缺省行为（那是 derived classes 可能使用的，但只有在它们明确提出申请时才是）。如果合并 `fly` 和 `defaultFly`，就丧失了“让两个函数享有不同保护级别”的机会：习惯上被设为 protected 的函数（`defaultFly`）如今成了 public（因为它在 `fly` 之中）。

最后，让我们看看 `Shape` 的 non-virtual 函数 `objectID`：

```
class Shape {
public:
   int objectID( ) const;
   ...
};
```

如果成员函数是个 non-virtual 函数，意味是它并不打算在 derived classes 中有不同的行为。实际上一个 non-virtual 成员函数所表现的不变性（*invariant*）凌驾其特异性（*specialization*），因为它表示不论 derived class 变得多么特异化，它的行为都不可以改变。就其自身而言：

- 声明 non-virtual 函数的目的是为了令 derived classes 继承函数的接口及一份强制性实现。

你可以把 `Shape::objectID` 的声明想做是："每个 `Shape` 对象都有一个用来产生对象识别码的函数；此识别码总是采用相同计算方法，该方法由 `Shape::objectID` 的定义式决定，任何 derived class 都不应该尝试改变其行为"。由于 non-virtual 函数代表的意义是不变性（*invariant*）凌驾特异性（*specialization*），所以它绝不该在 derived class 中被重新定义。这也是条款 36 所讨论的一个重点。

pure virtual 函数、simple (impure) virtual 函数、non-virtual 函数之间的差异，使你得以精确指定你想要 derived classes 继承的东西：只继承接口，或是继承接口和一份缺省实现，或是继承接口和一份强制实现。由于这些不同类型的声明意味根本意义并不相同的事情，当你声明你的成员函数时，必须谨慎选择。如果你确实履行，应该能够避免经验不足的 class 设计者最常犯的两个错误。

第一个错误是将所有函数声明为 non-virtual。这使得 derived classes 没有余裕空间进行特化工作。non-virtual 析构函数尤其会带来问题（见条款 7）。当然啦，设计一个并不想成为 base class 的 class 是绝对合理的，既然这样，将其所有成员函数都声明为 non-virtual 也很适当。但这种声明如果不是忽略了 virtual 和 non-virtual 函数之间的差异，就是过度担心 virtual 函数的效率成本。实际上任何 class 如果打算被用来当做一个 base class，都会拥有若干 virtual 函数（再次见条款 7）。

如果你关心 virtual 函数的成本，请容许我介绍所谓的 80-20 法则（也可见条款 30）。这个法则说，一个典型的程序有 80% 的执行时间花费在 20% 的代码身上。此一法则十分重要，因为它意味，平均而言你的函数调用中可以有 80% 是 virtual 而不冲击程序的大体效率。所以当你担心是否有能力负担 virtual 函数的成本之前，请先将心力放在那举足轻重的 20% 代码上头，它才是真正的关键。

另一个常见错误是将所有成员函数声明为 virtual。有时候这样做是正确的，例如条款 31 的 Interface classes。然而这也可能是 class 设计者缺乏坚定立场的前兆。某些函数就是不该在 derived class 中被重新定义，果真如此你应该将那些函数声明为 non-virtual。没有人有权利妄称你的 class 适用于任何人任何事任何物而他们只需花点时间重新定义你的函数就可以享受一切。如果你的不变性（*invariant*）凌驾特异性（*specialization*），别害怕说出来。

请记住

- 接口继承和实现继承不同。在 public 继承之下，derived classes 总是继承 base class 的接口。
- pure virtual 函数只具体指定接口继承。
- 简朴的（非纯）impure virtual 函数具体指定接口继承及缺省实现继承。
- non-virtual 函数具体指定接口继承以及强制性实现继承。

条款 35：考虑 virtual 函数以外的其他选择

Consider alternatives to virtual functions.

假设你正在写一个视频游戏软件，你打算为游戏内的人物设计一个继承体系。你的游戏属于暴力砍杀类型，剧中人物被伤害或因其他因素而降低健康状态的情况并不罕见。你因此决定提供一个成员函数 `healthValue`，它会返回一个整数，表示人物的健康程度。由于不同的人物可能以不同的方式计算他们的健康指数，将 `healthValue` 声明为 virtual 似乎是再明白不过的做法：

```
class GameCharacter {
public:
   virtual int healthValue() const;   //返回人物的健康指数;
   ...                                //derived classes 可重新定义它。
};
```

`healthValue` 并未被声明为 pure virtual，这暗示我们将会有个计算健康指数的缺省算法（见条款 34）。

这的确是再明白不过的设计，但是从某个角度说却反而成了它的弱点。由于这个设计如此明显，你可能因此没有认真考虑其他替代方案。为了帮助你跳脱面向对象设计路上的常轨，让我们考虑其他一些解法。

藉由 Non-Virtual Interface 手法实现 *Template Method* 模式

我们将从一个有趣的思想流派开始，这个流派主张 virtual 函数应该几乎总是 private。这个流派的拥护者建议，较好的设计是保留 healthValue 为 public 成员函数，但让它成为 non-virtual，并调用一个 private virtual 函数（例如 doHealthValue）进行实际工作：

```
class GameCharacter {
public:
    int healthValue() const                    //derived classes 不重新定义它，
    {                                          //见条款 36。
        ...                                    //做一些事前工作，详下。
        int retVal = doHealthValue();          //做真正的工作。
        ...                                    //做一些事后工作，详下。
        return retVal;
    }
    ...
private:
    virtual int doHealthValue() const          //derived classes 可重新定义它。
    {
        ...                                    //缺省算法，计算健康指数。
    }
};
```

在这段（以及本条款其余的）代码中，我直接在 class 定义式内呈现成员函数本体。一如条款 30 所言，那也就让它们全都暗自成了 inline。但其实我以这种方式呈现代码只是为了让你比较容易阅读。我所描述的设计与 inlining 其实没有关联，所以请不要认为成员函数在这里被定义于 classes 内有特殊用意。不，它没有。

这一基本设计，也就是“令客户通过 public non-virtual 成员函数间接调用 private virtual 函数”，称为 *non-virtual interface*（NVI）手法。它是所谓 ***Template Method*** 设计模式（与 C++ templates 并无关联）的一个独特表现形式。我把这个 non-virtual 函数（healthValue）称为 virtual 函数的外覆器（*wrapper*）。

NVI 手法的一个优点隐身在上述代码注释“做一些事前工作”和“做一些事后工作”之中。那些注释用来告诉你当时的代码保证在“virtual 函数进行真正工作之前和之后”被调用。这意味外覆器（wrapper）确保得以在一个 virtual 函数被调用之前设定好适当场景，并在调用结束之后清理场景。“事前工作”可以包括锁定互斥器（locking a mutex）、制造运转日志记录项（log entry）、验证 class 约束条件、验证函数先决条件等等。“事后工作”可以包括互斥器解除锁定（unlocking a mutex）、验证函数的事后条件、再次验证 class 约束条件等等。如果你让客户直接调用 virtual 函数，就没有任何好办法可以做这些事。

有件事实或许会妨碍你跃跃欲试的心：NVI 手法涉及在 derived classes 内重新定义 private virtual 函数。啊，重新定义若干个 derived classes 并不调用的函数！这里并不存在矛盾。“重新定义 virtual 函数”表示某些事“如何”被完成，“调用 virtual 函数”则表示它“何时”被完成。这些事情都是各自独立互不相干的。NVI 手法允许 derived classes 重新定义 virtual 函数，从而赋予它们“如何实现机能”的控制能力，但 base class 保留诉说“函数何时被调用”的权利。一开始这些听起来似乎诡异，但 C++ 的这种“derived classes 可重新定义继承而来的 private virtual 函数”的规则完全合情合理。

在 NVI 手法下其实没有必要让 virtual 函数一定得是 private。某些 class 继承体系要求 derived class 在 virtual 函数的实现内必须调用其 base class 的对应兄弟（例如 p.120 的程序），而为了让这样的调用合法，virtual 函数必须是 protected，不能是 private。有时候 virtual 函数甚至一定得是 public（例如具备多态性质的 base classes 的析构函数 — 见条款 7），这么一来就不能实施 NVI 手法了。

藉由 Function Pointers 实现 *Strategy* 模式

NVI 手法对 public virtual 函数而言是一个有趣的替代方案，但从某种设计角度观之，它只比窗饰花样更强一些而已。毕竟我们还是使用 virtual 函数来计算每个人物的健康指数。另一个更戏剧性的设计主张“人物健康指数的计算与人物类型无关”，这样的计算完全不需要“人物”这个成分。例如我们可能会要求每个人物的构造函数接受一个指针，指向一个健康计算函数，而我们可以调用该函数进行实际计算：

```
class GameCharacter;                    //前置声明（forward declaration）
//以下函数是计算健康指数的缺省算法。
int defaultHealthCalc(const GameCharacter& gc);
class GameCharacter {
public:
   typedef int (*HealthCalcFunc)(const GameCharacter&);
   explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
    : healthFunc(hcf)
   {}
   int healthValue() const
   { return healthFunc(*this); }
   ...
private:
   HealthCalcFunc healthFunc;
};
```

这个做法是常见的 ***Strategy*** 设计模式的简单应用。拿它和“植基于GameCharacter继承体系内之 virtual 函数”的做法比较，它提供了某些有趣弹性：

- 同一人物类型之不同实体可以有不同的健康计算函数。例如：

```
class EvilBadGuy: public GameCharacter {
public:
   explicit EvilBadGuy(HealthCalcFunc hcf = defaultHealthCalc)
    : GameCharacter(hcf)
   { ... }
   ...
};
int loseHealthQuickly(const GameCharacter&);     //健康指数计算函数 1
int loseHealthSlowly(const GameCharacter&);      //健康指数计算函数 2

EvilBadGuy ebg1(loseHealthQuickly);              //相同类型的人物搭配
EvilBadGuy ebg2(loseHealthSlowly);               // 不同的健康计算方式
```

- 某已知人物之健康指数计算函数可在运行期变更。例如 GameCharacter 可提供一个成员函数 setHealthCalculator，用来替换当前的健康指数计算函数。

换句话说，“健康指数计算函数不再是 GameCharacter 继承体系内的成员函数”这一事实意味，这些计算函数并未特别访问“即将被计算健康指数”的那个对象的内部成分。例如 defaultHealthCalc 并未访问 EvilBadGuy 的 non-public 成分。

如果人物的健康可纯粹根据该人物 public 接口得来的信息加以计算，这就没有问题，但如果需要 non-public 信息进行精确计算，就有问题了。实际上任何时候当你将 class 内的某个机能（也许取道自某个成员函数）替换为 class 外部的某个等价机能(也许取道自某个 non-member non-friend 函数或另一个 class 的 non-friend 成员函数），这都是潜在争议点。这个争议将持续至本条款其余篇幅，因为我们即将考虑的所有替代设计也都涉及使用 GameCharacter 继承体系外的函数。

一般而言，唯一能够解决“需要以 non-member 函数访问 class 的 non-public 成分”的办法就是：弱化 class 的封装。例如 class 可声明那个 non-member 函数为 friends，或是为其实现的某一部分提供 public 访问函数（其他部分则宁可隐藏起来）。运用函数指针替换 virtual 函数，其优点（像是“每个对象可各自拥有自己的健康计算函数”和“可在运行期改变计算函数”）是否足以弥补缺点（例如可能必须降低 GameCharacter 封装性），是你必须根据每个设计情况的不同而抉择的。

藉由 tr1::function 完成 *Strategy* 模式

一旦习惯了 templates 以及它们对隐式接口（见条款 41）的使用，基于函数指针的做法看起来便过分苛刻而死板了。为什么要求“健康指数之计算”必须是个函数，而不能是某种“像函数的东西”（例如函数对象）呢？如果一定得是函数，为什么不能够是个成员函数？为什么一定得返回 int 而不是任何可被转换为 int 的类型呢？

如果我们不再使用函数指针（如前例的 healthFunc），而是改用一个类型为 tr1::function 的对象，这些约束就全都挥发不见了。就像条款 54 所说，这样的对象可持有（保存）任何可调用物（*callable entity*，也就是函数指针、函数对象、或成员函数指针），只要其签名式兼容于需求端。以下将刚才的设计改为使用 tr1::function：

```
class GameCharacter;                                    //如前
int defaultHealthCalc(const GameCharacter& gc);         //如前
class GameCharacter {
public:
    //HealthCalcFunc 可以是任何“可调用物”（callable entity），可被调用并接受
    //任何兼容于 GameCharacter 之物，返回任何兼容于 int 的东西。详下。
    typedef std::tr1::function<int (const GameCharacter&)> HealthCalcFunc;
```

```
    explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
      : healthFunc(hcf )
    {}
    int healthValue() const
    { return healthFunc( * this); }
    ...
private:
    HealthCalcFunc healthFunc;
};
```

如你所见，HealthCalcFunc 是个 typedef，用来表现 tr1::function 的某个具现体，意味该具现体的行为像一般的函数指针。现在我们靠近一点瞧瞧 HealthCalcFunc 是个什么样的 typedef：

```
std::tr1::function<int (const GameCharacter&)>
```

这里我把 tr1::function 具现体（instantiation）的目标签名式（target signature）以不同颜色强调出来。那个签名代表的函数是“接受一个 reference 指向 const GameCharacter，并返回 int”。这个 tr1::function 类型（也就是我们所定义的 HealthCalcFunc 类型）产生的对象可以持有（保存）任何与此签名式兼容的可调用物（callable entity）。所谓兼容，意思是这个可调用物的参数可被隐式转换为 const GameCharacter&，而其返回类型可被隐式转换为 int。

和前一个设计（其 GameCharacter 持有的是函数指针）比较，这个设计几乎相同。唯一不同的是如今 GameCharacter 持有一个 tr1::function 对象，相当于一个指向函数的泛化指针。这个改变如此细小，我总说它没有什么外显影响，除非客户在“指定健康计算函数”这件事上需要更惊人的弹性：

```
short calcHealth(const GameCharacter&);     //健康计算函数；
                                            //注意其返回类型为 non-int

struct HealthCalculator {                   //为计算健康而设计的函数对象
    int operator()(const GameCharacter&) const
    { ... }
};

class GameLevel {
public:
    float health(const GameCharacter&) const;    //成员函数，用以计算健康；
    ...                                          //注意其 non-int 返回类型
};

class EvilBadGuy: public GameCharacter {         //同前
    ...
};
```

```
class EyeCandyCharacter: public GameCharacter {      //另一个人物类型；
   ...                                               //假设其构造函数与
};                                                   //EvilBadGuy 同

EvilBadGuy ebg1(calcHealth);                         //人物 1，使用某个
                                                     // 函数计算健康指数

EyeCandyCharacter ecc1(HealthCalculator());          //人物 2，使用某个
                                                     // 函数对象计算健康指数
GameLevel currentLevel;
...
EvilBadGuy ebg2(                                     //人物 3，使用某个
  std::tr1::bind(&GameLevel::health,                 // 成员函数计算健康指数
                 currentLevel,
                 _1)                                 //详见以下
);
```

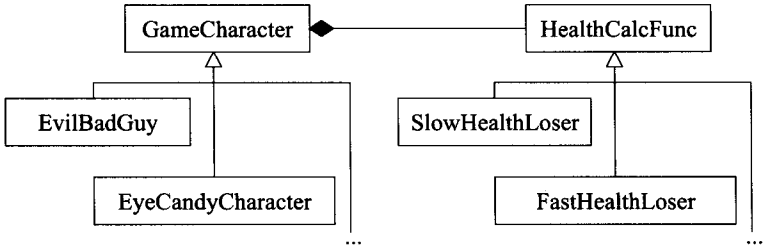
就我个人而言，当我发现 tr1::function 允许我们做的事时，非常吃惊。它让我浑身震颤。如果你没有这样的感觉，也许是你早已曾经惊叹 tr1::bind 所发生的事情。请允许我稍加解释。

首先我要表明，为计算 ebg2 的健康指数，应该使用 GameLevel class 的成员函数 health。好，GameLevel::health 宣称它自己接受一个参数（那是个 reference 指向 GameCharacter），但它实际上接受两个参数，因为它也获得一个隐式参数 GameLevel，也就是 this 所指的那个。然而 GameCharacters 的健康计算函数只接受单一参数：GameCharacter（这个对象将被计算出健康指数）。如果我们使用 GameLevel::health 作为 ebg2 的健康计算函数，我们必须以某种方式转换它，使它不再接受两个参数（一个 GameCharacter 和一个 GameLevel），转而接受单一参数（一个 GameCharacter）。在这个例子中我们必然会想要使用 currentLevel 作为“ebg2 的健康计算函数所需的那个 GameLevel 对象”，于是我们将 currentLevel 绑定为 GameLevel 对象，让它在“每次 GameLevel::health 被调用以计算 ebg2 的健康”时被使用。那正是 tr1::bind 的作为：它指出 ebg2 的健康计算函数应该总是以 currentLevel 作为 GameLevel 对象。

我跳过了一大堆细节，像是为什么 "_1" 意味“当为 ebg2 调用 GameLevel::health 时系以 currentLevel 作为 GameLevel 对象”。这样的细节不难阐述，但它们会分散我要说的根本重点：若以 tr1::function 替换函数指针，吾人将因此允许客户在计算人物健康指数时使用任何兼容的可调用物（*callable entity*）。如果这还不酷，什么是酷？

古典的 *Strategy* 模式

如果你对设计模式（design patterns）比对 C++ 的酷劲更有兴趣，我告诉你，传统(典型)的***Strategy***做法会将健康计算函数做成一个分离的继承体系中的virtual成员函数。设计结果看起来像这样：



如果你并未精通 UML 符号，别担心，这图只是告诉你 GameCharacter 是某个继承体系的根类，体系中的 EvilBadGuy 和 EyeCandyCharacter 都是 derived classes；HealthCalcFunc 是另一个继承体系的根类，体系中的 SlowHealthLoser 和 FastHealthLoser 都是 derived classes，每一个 GameCharacter 对象都内含一个指针，指向一个来自 HealthCalcFunc 继承体系的对象。

下面是对应的代码骨干：

```
class GameCharacter;          //前置声明（forward declaration）
class HealthCalcFunc {
public:
   ...
   virtual int calc(const GameCharacter& gc) const
   { ... }
   ...
};

HealthCalcFunc defaultHealthCalc;

class GameCharacter {
public:
   explicit GameCharacter(HealthCalcFunc* phcf = &defaultHealthCalc)
     : pHealthCalc(phcf )
   {}
   int healthValue() const
   { return pHealthCalc->calc(*this); }
   ...
private:
   HealthCalcFunc* pHealthCalc;
};
```

这个解法的吸引力在于，熟悉标准 ***Strategy*** 模式的人很容易辨认它，而且它还提供“将一个既有的健康算法纳入使用”的可能性——只要为 `HealthCalcFunc` 继承体系添加一个 derived class 即可。

摘要

本条款的根本忠告是，当你为解决问题而寻找某个设计方法时，不妨考虑 virtual 函数的替代方案。下面快速重点复习我们验证过的几个替代方案：

- 使用 non-virtual interface（NVI）手法，那是 ***Template Method*** 设计模式的一种特殊形式。它以 public non-virtual 成员函数包裹较低访问性（private 或 protected）的 virtual 函数。
- 将 virtual 函数替换为“函数指针成员变量”，这是 ***Strategy*** 设计模式的一种分解表现形式。
- 以 `tr1::function` 成员变量替换 virtual 函数，因而允许使用任何可调用物（callable entity）搭配一个兼容于需求的签名式。这也是 ***Strategy*** 设计模式的某种形式。
- 将继承体系内的 virtual 函数替换为另一个继承体系内的 virtual 函数。这是 ***Strategy*** 设计模式的传统实现手法。

以上并未彻底而详尽地列出 virtual 函数的所有替换方案，但应该足够让你知道的确有不少替换方案。此外，它们各有其相对的优点和缺点，你应该把它们全部列入考虑。

为避免陷入面向对象设计路上因常规而形成的凹洞中，偶而我们需要对着车轮猛推一把。这个世界还有其他许多道路，值得我们花时间加以研究。

请记住

- virtual 函数的替代方案包括 NVI 手法及 ***Strategy*** 设计模式的多种形式。NVI 手法自身是一个特殊形式的 ***Template Method*** 设计模式。
- 将机能从成员函数移到 class 外部函数，带来的一个缺点是，非成员函数无法访问 class 的 non-public 成员。
- `tr1::function` 对象的行为就像一般函数指针。这样的对象可接纳“与给定之目标签名式（target signature）兼容”的所有可调用物（callable entities）。

条款 36：绝不重新定义继承而来的 non-virtual 函数

Never redefine an inherited non-virtual function.

假设我告诉你，class D 系由 class B 以 public 形式派生而来，class B 定义有一个 public 成员函数 mf。由于 mf 的参数和返回值都不重要，所以我假设两者皆为 void。换句话说我的意思是：

```
class B {
public:
   void mf();
   ...
};

class D: public B { ... };
```

虽然我们对 B, D 和 mf 一无所知，但面对一个类型为 D 的对象 x：

```
D x;                    //x 是一个类型为 D 的对象
```

如果以下行为：

```
B* pB = &x;             //获得一个指针指向 x
pB->mf();               //经由该指针调用 mf
```

异于以下行为：

```
D* pD = &x;             //获得一个指针指向 x
pD->mf();               //经由该指针调用 mf
```

你可能会相当惊讶。毕竟两者都通过对象 x 调用成员函数 mf。由于两者所调用的函数都相同，凭借的对象也相同，所以行为也应该相同，是吗？

是的，理应如此，但事实可能不是如此。更明确地说，如果 mf 是个 non-virtual 函数而 D 定义有自己的 mf 版本，那就不是如此：

```
class D: public B {
public:
   void mf();              //遮掩（hides）了 B::mf；见条款 33
   ...
};
pB->mf();                  //调用 B::mf
pD->mf();                  //调用 D::mf
```

造成此一两面行为的原因是，non-virtual 函数如 B::mf 和 D::mf 都是静态绑定（statically bound，见条款 37）。这意思是，由于 pB 被声明为一个 pointer-to-B，通

过 pB 调用的 non-virtual 函数永远是 B 所定义的版本，即使 pB 指向一个类型为“B 派生之 class”的对象，一如本例。

但另一方面，virtual 函数却是动态绑定（dynamically bound，见条款 37），所以它们不受这个问题之苦。如果 mf 是个 virtual 函数，不论是通过 pB 或 pD 调用 mf，都会导致调用 D::mf，因为 pB 和 pD 真正指的都是一个类型为 D 的对象。

如果你正在编写 class D 并重新定义继承自 class B 的 non-virtual 函数 mf，D 对象很可能展现出精神分裂的不一致行径。更明确地说，当 mf 被调用，任何一个 D 对象都可能表现出 B 或 D 的行为；决定因素不在对象自身，而在于“指向该对象之指针”当初的声明类型。References 也会展现和指针一样难以理解的行径。

但那只是实务面上的讨论。我知道你真正想要的是理论层面的理由（关于“绝不重新定义继承而来的 non-virtual 函数”这回事）。我很乐意为你服务。

条款 32 已经说过，所谓 public 继承意味 **is-a**（是一种）的关系。条款 34 则描述为什么在 class 内声明一个 non-virtual 函数会为该 class 建立起一个不变性（invariant），凌驾其特异性（specialization）。如果你将这两个观点施行于两个 classes B 和 D 以及 non-virtual 成员函数 B::mf 身上，那么：

- 适用于 B 对象的每一件事，也适用于 D 对象，因为每个 D 对象都是一个 B 对象；
- B 的 derived classes 一定会继承 mf 的接口和实现，因为 mf 是 B 的一个 non-virtual 函数。

现在，如果 D 重新定义 mf，你的设计便出现矛盾。如果 D 真有必要实现出与 B 不同的 mf，并且如果每一个 B 对象——不管多么特化——真的必须使用 B 所提供的 mf 实现码，那么“每个 D 都是一个 B”就不为真。既然如此 D 就不该以 public 形式继承 B。另一方面，如果 D 真的必须以 public 方式继承 B，并且如果 D 真有需要实现出与 B 不同的 mf，那么 mf 就无法为 B 反映出“不变性凌驾特异性”的性质。既然这样 mf 应该声明为 virtual 函数。最后，如果每个 D 真的是一个 B，并且如果 mf 真的为 B 反映出“不变性凌驾特异性”的性质，那么 D 便不需要重新定义 mf，而且它也不应该尝试这样做。

不论哪一个观点，结论都相同：任何情况下都不该重新定义一个继承而来的 non-virtual 函数。

如果此条款使你感到枯燥乏味，或许是因为你已经读过条款 7，该条款解释为什么多态性（polymorphic）base classes 内的析构函数应该是 virtual。如果你违反那个准则(也就是说如果你在 polymorphic base class 内声明一个 non-virtual 析构函数)，你也就违反了本条款，因为 derived classes 绝对不该重新定义一个继承而来的 non-virtual 函数（此处指的是 base class 析构函数）。即使没有声明析构函数，此亦为真，因为条款 5 说，析构函数是“如果你没有为自己声明一个，编译器会为你生成一个”的数种成员函数之一。就本质而言，条款 7 只不过是本条款的一个特殊案例，尽管它也足够重要到单独成为一个条款。

请记住

- 绝对不要重新定义继承而来的 non-virtual 函数。

条款 37：绝不重新定义继承而来的缺省参数值

Never redefine a function's inherited default parameter value.

让我们一开始就将讨论简化。你只能继承两种函数：virtual 和 non-virtual 函数。然而重新定义一个继承而来的 non-virtual 函数永远是错误的（见条款 36），所以我们可以安全地将本条款的讨论局限于“继承一个带有缺省参数值的 virtual 函数”。

这种情况下，本条款成立的理由就非常直接而明确了：virtual 函数系动态绑定（dynamically bound），而缺省参数值却是静态绑定（statically bound）。

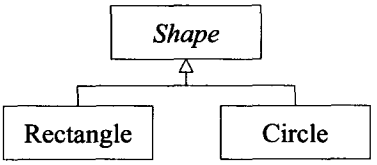
那是什么意思？你说你那负荷过重的脑袋早已忘记静态绑定和动态绑定之间的差异？（为了正式记录在案，容我再说一次，静态绑定又名前期绑定，*early binding*；动态绑定又名后期绑定，*late binding*。）现在让我们来一趟复习之旅吧！

对象的所谓静态类型（static type），就是它在程序中被声明时所采用的类型。考虑以下的 class 继承体系：

```
//一个用以描述几何形状的 class
class Shape {
public:
    enum ShapeColor { Red, Green, Blue };
    //所有形状都必须提供一个函数，用来绘出自己
    virtual void draw(ShapeColor color = Red) const = 0;
    ...
};
```

```
class Rectangle: public Shape {
public:
    //注意，赋予不同的缺省参数值。这真糟糕！
    virtual void draw(ShapeColor color = Green) const;
    ...
};

class Circle: public Shape {
public:
    virtual void draw(ShapeColor color) const;
    //译注：请注意，以上这么写则当客户以对象调用此函数，一定要指定参数值。
    //      因为静态绑定下这个函数并不从其 base 继承缺省参数值。
    //      但若以指针（或 reference）调用此函数，可以不指定参数值，
    //      因为动态绑定下这个函数会从其 base 继承缺省参数值。
    ...
};
```

这个继承体系图示如下：



现在考虑这些指针：

```
Shape* ps;                          //静态类型为 Shape*
Shape* pc = new Circle;             //静态类型为 Shape*
Shape* pr = new Rectangle;          //静态类型为 Shape*
```

本例中 ps, pc 和 pr 都被声明为 pointer-to-Shape 类型，所以它们都以它为静态类型。注意，不论它们真正指向什么，它们的静态类型都是 Shape*。

对象的所谓动态类型（dynamic type）则是指“目前所指对象的类型”。也就是说，动态类型可以表现出一个对象将会有什么行为。以上例而言，pc 的动态类型是 Circle*，pr 的动态类型是 Rectangle*。ps 没有动态类型，因为它尚未指向任何对象。

动态类型一如其名称所示，可在程序执行过程中改变（通常是经由赋值动作）：

```
ps = pc;                            //ps 的动态类型如今是 Circle*
ps = pr;                            //ps 的动态类型如今是 Rectangle*
```

Virtual 函数系动态绑定而来，意思是调用一个 virtual 函数时，究竟调用哪一份函数实现代码，取决于发出调用的那个对象的动态类型：

```
pc->draw(Shape::Red);                    //调用 Circle::draw(Shape::Red)
pr->draw(Shape::Red);                    //调用 Rectangle::draw(Shape::Red)
```

我知道这些都是老调重弹；是的，你当然已经了解 virtual 函数。但是当你考虑带有缺省参数值的 virtual 函数，花样来了，因为就如我稍早所说，virtual 函数是动态绑定，而缺省参数值却是静态绑定。意思是你可能会在“调用一个定义于 derived class 内的 virtual 函数”的同时，却使用 base class 为它所指定的缺省参数值：

```
pr->draw( );                             //调用 Rectangle::draw(Shape::Red)!
```

此例之中，pr 的动态类型是 Rectangle*，所以调用的是 Rectangle 的 virtual 函数，一如你所预期。Rectangle::draw 函数的缺省参数值应该是 GREEN，但由于 pr 的静态类型是 Shape*，所以此一调用的缺省参数值来自 Shape class 而非 Rectangle class！结局是这个函数调用有着奇怪并且几乎绝对没人预料得到的组合，由 Shape class 和 Rectangle class 的 draw 声明式各出一半力。

以上事实不只局限于“ps, pc 和 pr 都是指针”的情况；即使把指针换成 references 问题仍然存在。重点在于 draw 是个 virtual 函数，而它有个缺省参数值在 derived class 中被重新定义了。

为什么 C++ 坚持以这种乖张的方式来运作呢？答案在于运行期效率。如果缺省参数值是动态绑定，编译器就必须有某种办法在运行期为 virtual 函数决定适当的参数缺省值。这比目前实行的“在编译期决定”的机制更慢而且更复杂。为了程序的执行速度和编译器实现上的简易度，C++ 做了这样的取舍，其结果就是你如今所享受的执行效率。但如果你没有注意本条款所揭示的忠告，很容易发生混淆。

这一切都很好，但如果你试着遵守这条规则，并且同时提供缺省参数值给 base 和 derived classes 的用户，又会发生什么事呢？

```
class Shape {
public:
   enum ShapeColor { Red, Green, Blue };
   virtual void draw(ShapeColor color = Red) const = 0;
   ...
};
```

```
class Rectangle: public Shape {
public:
   virtual void draw(ShapeColor color = Red) const;
   ...
};
```

喔欧，代码重复。更糟的是，代码重复又带着相依性（with dependencies）：如果 Shape 内的缺省参数值改变了，所有“重复给定缺省参数值”的那些 derived classes 也必须改变，否则它们最终会导致“重复定义一个继承而来的缺省参数值”。怎么办？

当你想令 virtual 函数表现出你所想要的行为但却遭遇麻烦，聪明的做法是考虑替代设计。条款 35 列了不少 virtual 函数的替代设计，其中之一是 NVI（*non-virtual interface*）手法：令 base class 内的一个 public non-virtual 函数调用 private virtual 函数，后者可被 derived classes 重新定义。这里我们可以让 non-virtual 函数指定缺省参数，而 private virtual 函数负责真正的工作：

```
class Shape {
public:
   enum ShapeColor { Red, Green, Blue };
   void draw(ShapeColor color = Red) const       //如今它是 non-virtual
   {
        doDraw(color);                           //调用一个 virtual
   }
   ...
private:
   virtual void doDraw(ShapeColor color) const = 0; //真正的工作
};                                                  //在此处完成

class Rectangle: public Shape {
public:
   ...
private:
   virtual void doDraw(ShapeColor color) const;     //注意，不须指定
   ...                                              //缺省参数值。
};
```

由于 non-virtual 函数应该绝对不被 derived classes 覆写（见条款 36），这个设计很清楚地使得 draw 函数的 color 缺省参数值总是为 Red。

请记住

- 绝对不要重新定义一个继承而来的缺省参数值，因为缺省参数值都是静态绑定，而 virtual 函数——你唯一应该覆写的东西——却是动态绑定。

条款 38：通过复合塑模出 has-a 或"根据某物实现出"

Model "has-a" or "is-implemented-in-terms-of" through composition.

复合（composition）是类型之间的一种关系，当某种类型的对象内含它种类型的对象，便是这种关系。例如：

```
class Address { ... };                    //某人的住址
class PhoneNumber { ... };

class Person {
public:
   ...
private:
   std::string name;                      //合成成分物（composed object）
   Address address;                       //同上
   PhoneNumber voiceNumber;               //同上
   PhoneNumber faxNumber;                 //同上
};
```

本例之中 Person 对象由 string, Address, PhoneNumber 构成。在程序员之间复合（composition）这个术语有许多同义词，包括 *layering*（分层），*containment*（内含），*aggregation*（聚合）和 *embedding*（内嵌）。

条款 32 曾说，“public 继承”带有 **is-a**（是一种）的意义。复合也有它自己的意义。实际上它有两个意义。复合意味 **has-a**（有一个）或 is-implemented-in-terms-of（根据某物实现出）。那是因为你正打算在你的软件中处理两个不同的领域（domains）。程序中的对象其实相当于你所塑造的世界中的某些事物，例如人、汽车、一张张视频画面等等。这样的对象属于应用域（*application domain*）部分。其他对象则纯粹是实现细节上的人工制品，像是缓冲区（buffers）、互斥器（mutexes）、查找树（search trees）等等。这些对象相当于你的软件的实现域（*implementation domain*）。当复合发生于应用域内的对象之间，表现出 **has-a** 的关系；当它发生于实现域内则是表现 is-implemented-in-terms-of 的关系。

上述的 Person class 示范 **has-a** 关系。Person 有一个名称，一个地址，以及语音和传真两笔电话号码。你不会说“人是一个名称”或“人是一个地址”，你会说“人有一个名称”和“人有一个地址”。大多数人接受此一区别毫无困难，所以很少人会对 **is-a** 和 **has-a** 感到困惑。

比较麻烦的是区分 **is-a**（是一种）和 is-implemented-in-terms-of（根据某物实现出）这两种对象关系。假设你需要一个 template，希望制造出一组 classes 用来表现由不重复对象组成的 sets。由于复用（reuse）是件美妙无比的事情，你的第一个

直觉是采用标准程序库提供的 set template。是的，如果他人所写的 template 合乎需求，我们何必另写一个呢？

不幸的是 set 的实现往往招致"每个元素耗用三个指针"的额外开销。因为 sets 通常以平衡查找树（balanced search trees）实现而成，使它们在查找、安插、移除元素时保证拥有对数时间（logarithmic-time）效率。当速度比空间重要，这是个通情达理的设计，但如果你的程序却是空间比速度重要呢？那么标准程序库的 set 提供给你的是个错误决定下的取舍。似乎你终究还得写个自己的 template。

但是容我再说一次，复用（reuse）是件美好的事。如果你是一位数据结构专家，你就会知道，实现 sets 的方法太多了，其中一种便是在底层采用 linked lists。而你又刚好知道，标准程序库有一个 list template，于是你决定复用它。

更明确地说，你决定让你那个萌芽中的 Set template 继承 std::list。也就是让 Set<T>继承 list<T>。毕竟在你的实现理念中 Set 对象其实是个 list 对象。你于是声明 Set template 如下：

```
template<typename T>                    //将 list 应用于 Set。错误做法。
class Set: public std::list<T> { ... };
```

每件事看起来都很好，但实际上有些东西完全错误。一如条款 32 所说，如果 D 是一种 B，对 B 为真的每一件事情对 D 也都应该为真。但 list 可以内含重复元素，如果数值 3051 被安插到 list<int>两次，那个 list 将内含两笔 3051。Set 不可以内含重复元素，如果数值 3051 被安插到 Set<int>两次，这个 set 只内含一笔 3051。因此"Set 是一种 list"并不为真，因为对 list 对象为真的某些事情对 Set 对象并不为真。

由于这两个 classes 之间并非 **is-a** 的关系，所以 public 继承不适合用来塑模它们。正确的做法是，你应当了解，Set 对象可根据一个 list 对象实现出来：

```
template<class T>                       //将 list 应用于 Set。正确做法。
class Set {
public:
   bool member(const T& item) const;
   void insert(const T& item);
   void remove(const T& item);
   std::size_t size() const;
private:
   std::list<T> rep;                    //用来表述 Set 的数据
};
```

Set成员函数可大量倚赖list及标准程序库其他部分提供的机能来完成，所以其实现很直观也很简单，只要你熟悉以STL编写程序：

```
template<typename T>
bool Set<T>::member(const T& item) const
{
   return std::find(rep.begin(), rep.end(), item) != rep.end( );
}

template<typename T>
void Set<T>::insert(const T& item)
{
   if (!member(item)) rep.push_back(item);
}

template<typename T>
void Set<T>::remove(const T& item)
{
  typename std::list<T>::iterator it =              //见条款42对
    std::find(rep.begin(), rep.end(), item);        //"typename"的讨论
  if (it != rep.end()) rep.erase(it);
}

template<typename T>
std::size_t Set<T>::size( ) const
{
   return rep.size( );
}
```

这些函数如此简单，因此都适合成为inlining候选人。但请记住，在做出任何与inlining有关的决定之前，应该先看看条款30。

也许有人主张，如果Set接口遵循STL容器的协议，就更符合条款18对设计接口的警告："让它容易被正确使用，不易被误用"。但是这儿如果要遵循那些协议，需得为Set添加许多东西，那将模糊了它和list之间的关系。由于Set和list之间的关系是本条款的重点，所以我们以教学清澈度交换STL兼容性。此外，Set接口也不该造成"对Set而言无可置辩的权利"黯然失色，那个权利是指它和list间的关系。这关系并非**is-a**（虽然最初似乎是），而是is-implemented-in-terms-of。

请记住

- 复合（composition）的意义和public继承完全不同。
- 在应用域（application domain），复合意味**has-a**（有一个）。在实现域（implementation domain），复合意味is-implemented-in-terms-of（根据某物实现出）。

条款 39：明智而审慎地使用 private 继承

Use private inheritance judiciously.

条款 32 曾经论证过 C++ 如何将 public 继承视为 **is-a** 关系。在那个例子中我们有个继承体系，其中 class Student 以 public 形式继承 class Person，于是编译器在必要时刻（为了让函数调用成功）将 Students 暗自转换为 Persons。现在我再重复该例的一部分，并以 private 继承替换 public 继承：

```
class Person { ... };
class Student: private Person { ... };      //这次改用 private 继承
void eat(const Person& p);                  //任何人都会吃
void study(const Student& s);               //只有学生才在校学习

Person p;                                   //p 是人
Student s;                                  //s 是学生

eat(p);                                     //没问题，p 是人，会吃。
eat(s);                                     //错误！吓，难道学生不是人?!
```

显然 private 继承并不意味 **is-a** 关系。那么它意味什么？

“哇喔！”你说，“在我们探讨其意义之前，可否先搞清楚其行为。到底 private 继承的行为如何呢？”唔，统御 private 继承的首要规则你刚才已经见过了：如果 classes 之间的继承关系是 private，编译器不会自动将一个 derived class 对象（例如 Student）转换为一个 base class 对象（例如 Person）。这和 public 继承的情况不同。这也就是为什么通过 s 调用 eat 会失败的原因。第二条规则是，由 private base class 继承而来的所有成员，在 derived class 中都会变成 private 属性，纵使它们在 base class 中原本是 protected 或 public 属性。

够了，现在让我们开始讨论其意义。Private 继承意味 implemented-in-terms-of（根据某物实现出）。如果你让 class D 以 private 形式继承 class B，你的用意是为了采用 class B 内已经备妥的某些特性，不是因为 B 对象和 D 对象存在有任何观念上的关系。private 继承纯粹只是一种实现技术（这就是为什么继承自一个 private base class 的每样东西在你的 class 内都是 private：因为它们都只是实现枝节而已）。借用条款 34 提出的术语，private 继承意味只有实现部分被继承，接口部分应略去。如果 D 以 private 形式继承 B，意思是 D 对象根据 B 对象实现而得，再没有其他意涵了。Private 继承在软件“设计”层面上没有意义，其意义只及于软件实现层面。

Private 继承意味 is-implemented-in-terms-of（根据某物实现出），这个事实有点令人不安，因为条款 38 才刚指出复合（composition）的意义也是这样。你如何在两者之间取舍？答案很简单：尽可能使用复合，必要时才使用 private 继承。何时才是必要？主要是当 protected 成员和/或 virtual 函数牵扯进来的时候。其实还有一种激进情况，那是当空间方面的利害关系足以踢翻 private 继承的支柱时。稍后我们再来操这个心，毕竟它只是一种激进情况。

假设我们的程序涉及 Widgets，而我们决定应该较好地了解如何使用 Widgets。例如我们不只想要知道 Widget 成员函数多么频繁地被调用，也想知道经过一段时间后调用比例如何变化。要知道，带有多个执行阶段（execution phases）的程序，可能在不同阶段拥有不同的行为轮廓（behavioral profiles）。例如编译器在解析（parsing）阶段所用的函数，大大不同于在最优化（optimization）和代码生成（code generation）阶段所使用的函数。

我们决定修改 Widget class，让它记录每个成员函数的被调用次数。运行期间我们将周期性地审查那份信息，也许再加上每个 Widget 的值，以及我们需要评估的任何其他数据。为完成这项工作，我们需要设定某种定时器，使我们知道收集统计数据的时候是否到了。

我们宁可复用既有代码，尽量少写新代码，所以在自己的工具百宝箱中翻箱倒柜，并且很开心地发现了这个 class：

```
class Timer {
public:
   explicit Timer(int tickFrequency);
   virtual void onTick() const;        //定时器每滴答一次，
   ...                                 //此函数就被自动调用一次。
};
```

这就是我们找到的东西。一个 Timer 对象，可调整为以我们需要的任何频率滴答前进，每次滴答就调用某个 virtual 函数。我们可以重新定义那个 virtual 函数，让后者取出 Widget 的当时状态。完美！

为了让 Widget 重新定义 Timer 内的 virtual 函数，Widget 必须继承自 Timer。但 public 继承在此例并不适当，因为 Widget 并不是个 Timer。是呀，Widget 客户总不该能够对着一个 Widget 调用 onTick 吧，因为观念上那并不是 Widget 接口的一部分。如果允许那样的调用动作，很容易造成客户不正确地使用 Widget 接口，

那会违反条款 18 的忠告："让接口容易被正确使用，不易被误用"。在这里，public 继承不是个好策略。

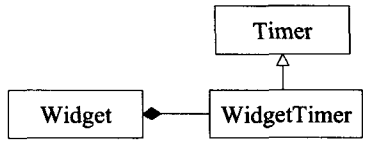
我们必须以 private 形式继承 Timer：

```
class Widget: private Timer {
private:
   virtual void onTick() const;        //查看 Widget 的数据...等等。
   ...
};
```

藉由 private 继承，Timer 的 public onTick 函数在 Widget 内变成 private，而我们重新声明（定义）时仍然把它留在那儿。再说一次，把 onTick 放进 public 接口内会误导客户端以为他们可以调用它，那就违反了条款 18。

这是个好设计，但不值几文钱，因为 private 继承并非绝对必要。如果我们决定以复合(composition)取而代之，是可以的。只要在 Widget 内声明一个嵌套式 private class，后者以 public 形式继承 Timer 并重新定义 onTick，然后放一个这种类型的对象于 Widget 内。下面是这种解法的草样：

```
class Widget {
private:
   class WidgetTimer: public Timer {
   public:
        virtual void onTick() const;
        ...
   };
   WidgetTimer timer;
   ...
};
```



这个设计比只使用 private 继承要复杂一些些，因为它同时涉及 public 继承和复合，并导入一个新 class（WidgetTimer）。坦白说我展示它主要是为了提醒你，解决一个设计问题的方法不只一种，而训练自己思考多种做法是值得的（看看条款 35）。尽管如此，我可以想出两个理由，为什么你可能愿意（或说应该）选择这样的 public 继承加复合，而不是选择原先的 private 继承设计。

首先，你或许会想设计 Widget 使它得以拥有 derived classes，但同时你可能会想阻止 derived classes 重新定义 onTick。如果 Widget 继承自 Timer，上面的想法就不可能实现，即使是 private 继承也不可能。（还记得吗，条款 35 曾说 derived classes 可以重新定义 virtual 函数，即使它们不得调用它。）但如果 WidgetTimer 是 Widget

内部的一个 private 成员并继承 Timer，Widget 的 derived classes 将无法取用 WidgetTimer，因此无法继承它或重新定义它的 virtual 函数。如果你曾经以 Java 或 C# 编程并怀念“阻止 derived classes 重新定义 virtual 函数”的能力（也就是 Java 的 final 和 C# 的 sealed），现在你知道怎么在 C++ 中模拟它了。

第二，你或许会想要将 Widget 的编译依存性降至最低。如果 Widget 继承 Timer，当 Widget 被编译时 Timer 的定义必须可见，所以定义 Widget 的那个文件恐怕必须#include Timer.h。但如果 WidgetTimer 移出 Widget 之外而 Widget 内含指针指向一个 WidgetTimer，Widget 可以只带着一个简单的 WidgetTimer 声明式，不再需要#include 任何与 Timer 有关的东西。对大型系统而言，如此的解耦（decouplings）可能是重要的措施。关于编译依存性的最小化，详见条款 31。

稍早我曾谈到，private 继承主要用于“当一个意欲成为 derived class 者想访问一个意欲成为 base class 者的 protected 成分，或为了重新定义一或多个 virtual 函数”。但这时候两个 classes 之间的概念关系其实是 **is-implemented-in-terms-of**（根据某物实现出）而非 **is-a**。然而我也说过，有一种激进情况涉及空间最优化，可能会促使你选择“private 继承”而不是“继承加复合”。

这个激进情况真是有够激进，只适用于你所处理的 class 不带任何数据时。这样的 classes 没有 non-static 成员变量，没有 virtual 函数（因为这种函数的存在会为每个对象带来一个 vptr，见条款 7），也没有 virtual base classes（因为这样的 base classes 也会招致体积上的额外开销，见条款 40）。于是这种所谓的 *empty classes* 对象不使用任何空间，因为没有任何隶属对象的数据需要存储。然而由于技术上的理由，C++ 裁定凡是独立（非附属）对象都必须有非零大小，所以如果你这样做：

```
class Empty { };                    //没有数据，所以其对象应该不使用任何内存

class HoldsAnInt {                  //应该只需要一个 int 空间
private:
   int x;
   Empty e;                         //应该不需要任何内存
};
```

你会发现 sizeof(HoldsAnInt) > sizeof(int)；喔欧，一个 Empty 成员变量竟然要求内存。在大多数编译器中 sizeof(Empty) 获得 1，因为面对“大小为零之独

立（非附属）对象”，通常 C++ 官方勒令默默安插一个 char 到空对象内。然而齐位需求（alignment，见条款 50）可能造成编译器为类似 HoldsAnInt 这样的 class 加上一些衬垫（padding），所以有可能 HoldsAnInt 对象不只获得一个 char 大小，也许实际上被放大到足够又存放一个 int。在我试过的所有编译器中，的确有这种情况发生。

但或许你注意到了，我很小心地说“独立（非附属）”对象的大小一定不为零。也就是说，这个约束不适用于 derived class 对象内的 base class 成分，因为它们并非独立（非附属）。如果你继承 Empty，而不是内含一个那种类型的对象：

```
class HoldsAnInt: private Empty {
private:
   int x;
};
```

几乎可以确定 sizeof(HoldsAnInt)==sizeof(int)。这是所谓的 EBO（*empty base optimization*；空白基类最优化），我试过的所有编译器都有这样的结果。如果你是一个程序库开发人员，而你的客户非常在意空间，那么值得注意 EBO。另外还值得知道的是，EBO 一般只在单一继承（而非多重继承）下才可行，统治 C++对象布局的那些规则通常表示 EBO 无法被施行于“拥有多个 base”的 derived classes 身上。

现实中的 "empty" classes 并不真的是 empty。虽然它们从未拥有 non-static 成员变量，却往往内含 typedefs, enums, static 成员变量，或 non-virtual 函数。STL 就有许多技术用途的 empty classes，其中内含有用的成员（通常是 typedefs），包括 base classes unary_function 和 binary_function，这些是“用户自定义之函数对象”通常会继承的 classes。感谢 EBO 的广泛实践，使这样的继承很少增加 derived classes 的大小。

尽管如此，让我们回到根本。大多数 classes 并非 empty，所以 EBO 很少成为 private 继承的正当理由。更进一步说，大多数继承相当于 **is-a**，这是指 public 继承，不是 private 继承。复合和 private 继承都意味 is-implemented-in-terms-of，但复合比较容易理解，所以无论什么时候，只要可以，你还是应该选择复合。

当你面对“并不存在 **is-a** 关系”的两个 classes，其中一个需要访问另一个的 protected 成员，或需要重新定义其一或多个 virtual 函数，private 继承极有可能成为正统设计策略。即便如此你也已经看到，一个混合了 public 继承和复合的设计，往往能够释出你要的行为，尽管这样的设计有较大的复杂度。“明智而审慎地使用

private 继承”意味，在考虑过所有其他方案之后，如果仍然认为 private 继承是“表现程序内两个 classes 之间的关系”的最佳办法，这才用它。

请记住

- Private 继承意味 is-implemented-in-terms of（根据某物实现出）。它通常比复合（composition）的级别低。但是当 derived class 需要访问 protected base class 的成员，或需要重新定义继承而来的 virtual 函数时，这么设计是合理的。
- 和复合（composition）不同，private 继承可以造成 empty base 最优化。这对致力于“对象尺寸最小化”的程序库开发者而言，可能很重要。

条款 40：明智而审慎地使用多重继承

Use multiple inheritance judiciously.

一旦涉及多重继承（multiple inheritance；**MI**），C++ 社群便分为两个基本阵营。其中之一认为如果单一继承（single inheritance；**SI**）是好的，多重继承一定更好。另一派阵营则主张，单一继承是好的，但多重继承不值得拥有（或使用）。本条款的主要目的是带领大家了解多重继承的两个观点。

最先需要认清的一件事是，当 MI 进入设计景框，程序有可能从一个以上的 base classes 继承相同名称（如函数、typedef 等等）。那会导致较多的歧义（ambiguity）机会。例如：

```
class BorrowableItem {                  //图书馆允许你借某些东西
public:
   void checkOut( );                    //离开时进行检查
   ...
};

class ElectronicGadget {
private:
   bool checkOut( ) const;              //执行自我检测，返回是否测试成功
   ...
};

class MP3Player:                        //注意这里的多重继承
   public BorrowableItem,               //（某些图书馆愿意借出 MP3 播放器）
   public ElectronicGadget
{ ... };                                //这里，class 的定义不是我们的关心重点

MP3Player mp;
mp.checkOut( );                         //歧义！调用的是哪个 checkOut?
```

注意此例之中对 checkOut 的调用是歧义（模棱两可）的，即使两个函数之中只有一个可取用（BorrowableItem 内的 checkOut 是 public，ElectronicGadget 内的却是 private）。这与 C++ 用来解析（resolving）重载函数调用的规则相符：在看到是否有个函数可取用之前，C++ 首先确认这个函数对此调用之言是最佳匹配。找出最佳匹配函数后才检验其可取用性。本例的两个 checkOuts 有相同的匹配程度（译注：因此才造成歧义），没有所谓最佳匹配。因此 ElectronicGadget::checkOut 的可取用性也就从未被编译器审查。

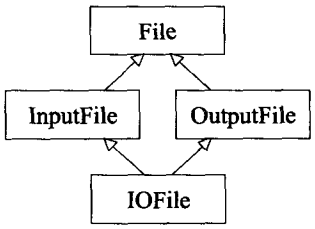
为了解决这个歧义，你必须明白指出你要调用哪一个 base class 内的函数：

```
mp.BorrowableItem::checkOut( );          //哎呀，原来是这个 checkOut...
```

你当然也可以尝试明确调用 ElectronicGadget::checkOut，但然后你会获得一个“尝试调用 private 成员函数”的错误。

多重继承的意思是继承一个以上的 base classes，但这些 base classes 并不常在继承体系中又有更高级的 base classes，因为那会导致要命的“钻石型多重继承”：

```
class File { ... };
class InputFile: public File { ... };
class OutputFile: public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```

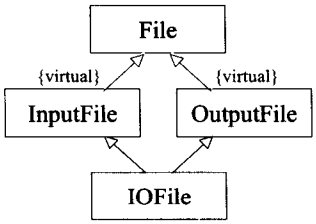


任何时候如果你有一个继承体系而其中某个 base class 和某个 derived class 之间有一条以上的相通路线（就像上述的 File 和 IOFile 之间有两条路径，分别穿越 InputFile 和 OutputFile），你就必须面对这样一个问题：是否打算让 base class 内的成员变量经由每一条路径被复制？假设 File class 有个成员变量 fileName，那么 IOFile 内该有多少笔这个名称的数据呢？从某个角度说，IOFile 从其每一个 base class 继承一份，所以其对象内应该有两份 fileName 成员变量。但从另一个角度说，简单的逻辑告诉我们，IOFile 对象只该有一个文件名称，所以它继承自两个 base classes 而来的 fileName 不该重复。

C++ 在这场辩论中并没有倾斜立场；两个方案它都支持——虽然其缺省做法是执行复制（也就是上一段所说的第一个做法）。如果那不是你要的，你必须令那个带有此数据的 class（也就是 File）成为一个 *virtual base class*。为了这样做，你必

须令所有直接继承自它的 classes 采用“virtual 继承”：

```
class File { ... };
class InputFile: virtual public File { ... };
class OutputFile: virtual public File { ... };
class IOFile: public InputFile,
              public OutputFile
{ ... };
```



C++ 标准程序库内含一个多重继承体系，其结构就如右图那样，只不过其 classes 其实是 class templates，名称分别是 basic_ios, basic_istream, basic_ostream 和 basic_iostream，而非这里的 File, InputFile, OutputFile 和 IOFile。

从正确行为的观点看，public 继承应该总是 virtual。如果这是唯一一个观点，规则很简单：任何时候当你使用 public 继承，请改用 virtual public 继承。但是，啊呀，正确性并不是唯一观点。为避免继承得来的成员变量重复，编译器必须提供若干幕后戏法，而其后果是：使用 virtual 继承的那些 classes 所产生的对象往往比使用 non-virtual 继承的兄弟们体积大，访问 virtual base classes 的成员变量时，也比访问 non-virtual base classes 的成员变量速度慢。种种细节因编译器不同而异，但基本重点很清楚：你得为 virtual 继承付出代价。

virtual 继承的成本还包括其他方面。支配“virtual base classes 初始化”的规则比起 non-virtual bases 的情况远为复杂且不直观。virtual base 的初始化责任是由继承体系中的最低层（*most derived*）class 负责，这暗示 (1) classes 若派生自 virtual bases 而需要初始化，必须认知其 virtual bases——不论那些 bases 距离多远，(2) 当一个新的 derived class 加入继承体系中，它必须承担其 virtual bases（不论直接或间接）的初始化责任。

我对 virtual base classes（亦相当于对 virtual 继承）的忠告很简单。第一，非必要不使用 virtual bases。平常请使用 non-virtual 继承。第二，如果你必须使用 virtual base classes，尽可能避免在其中放置数据。这么一来你就不需担心这些 classes 身上的初始化（和赋值）所带来的诡异事情了。Java 和 .NET 的 Interfaces 值得注意，它在许多方面兼容于 C++ 的 virtual base classes，而且也不允许含有任何数据。

现在让我们看看下面这个用来塑模“人”的 C++ Interface class（见条款 31）：

```
class IPerson {
public:
   virtual ~IPerson();
   virtual std::string name() const = 0;
   virtual std::string birthDate() const = 0;
};
```

IPerson 的客户必须以 Iperson 的 pointers 和 references 来编写程序，因为抽象 classes 无法被实体化创建对象。为了创建一些可被当做 IPerson 来使用的对象，IPerson 的客户使用 factory functions（工厂函数，见条款 31）将“派生自 IPerson 的具象 classes”实体化：

```
//factory function（工厂函数），根据一个独一无二的数据库 ID 创建一个 Person 对象。
//条款 18 告诉你为什么返回类型不是原始指针。
std::tr1::shared_ptr<IPerson> makePerson(DatabaseID personIdentifier);

//这个函数从使用者手上取得一个数据库 ID
DatabaseID askUserForDatabaseID();

DatabaseID id(askUserForDatabaseID());
std::tr1::shared_ptr<IPerson> pp(makePerson(id));
                                          //创建一个对象支持 Iperson 接口。
...                                       //藉由 Iperson 成员函数处理*pp。
```

但是 makePerson 如何创建对象并返回一个指针指向它呢？无疑地一定有某些派生自 IPerson 的具象 class，在其中 makePerson 可以创建对象。

假设这个 class 名为 CPerson。就像具象 class 一样，CPerson 必须提供“继承自 IPerson”的 pure virtual 函数的实现代码。我们可以从无到有写出这些东西，但更好的是利用既有组件，后者做了大部分或所有必要事情。例如，假设有个既有的数据库相关 class，名为 PersonInfo，提供 CPerson 所需要的实质东西：

```
class PersonInfo {
public:
   explicit PersonInfo(DatabaseID pid);
   virtual ~PersonInfo();
   virtual const char* theName() const;
   virtual const char* theBirthDate( ) const;
   ...
private:
   virtual const char* valueDelimOpen() const;       //详下
   virtual const char* valueDelimClose() const;
   ...
};
```

你可以说这是个旧式 class，因为其成员函数返回 const char* 而不是 string 对象。尽管如此，如果鞋子合脚，干嘛不穿它？这个 class 的成员函数的名称已经暗示我们其结果有可能很令人满意。

你会发现，PersonInfo 被设计用来协助以各种格式打印数据库字段，每个字段值的起始点和结束点以特殊字符串为界。缺省的头尾界限符号是方括号（中括号），所以（例如）字段值 "Ring-tailed Lemur" 将被格式化为：

```
[Ring-tailed Lemur]
```

但由于方括号并非放之四海人人喜爱的界限符号，所以两个 virtual 函数 valueDelimOpen 和 valueDelimClose 允许 derived classes 设定它们自己的头尾界限符号。PersonInfo 成员函数将调用这些 virtual 函数，把适当的界限符号添加到它们的返回值上。以 PersonInfo::theName 为例，代码看起来像这样：

```
const char* PersonInfo::valueDelimOpen() const
{
   return "[";                          //缺省的起始符号
}
const char* PersonInfo::valueDelimClose() const
{
   return "]";                          //缺省的结尾符号
}
const char* PersonInfo::theName() const
{
   //保留缓冲区给返回值使用；由于缓冲区是 static，因此会被自动初始化为“全部是 0”
   static char value[Max_Formatted_Field_Value_Length];

   //写入起始符号
   std::strcpy(value, valueDelimOpen());

   现在，将 value 内的字符串添附到这个对象的 name 成员变量中
   （小心，避免缓冲区超限）

   //写入结尾符号
   std::strcat(value, valueDelimClose());
   return value;
}
```

或许有人质疑 PersonInfo::theName 的老旧设计（特别是它竟然使用固定大小的 static 缓冲区，那将充斥超限问题和线程问题，见条款 21），但是不妨暂时把这样的疑问放两旁，把以下焦点摆中间：theName 调用 valueDelimOpen 产生字符串起始符号，然后产生 name 值，然后调用 valueDelimClose。由于 valueDelimOpen

和 valueDelimClose 都是 virtual 函数，theName 返回的结果不仅取决于 PersonInfo 也取决于从 PersonInfo 派生下去的 classes。

身为 CPerson 实现者，这是个好消息，因为仔细阅读 IPerson 文档后，你发现 name 和 birthDate 两函数必须返回未经装饰（不带起始符号和结尾符号）的值。也就是说如果有人名为 Homer，调用其 name 函数理应获得 "Homer" 而不是 "[Homer]"。

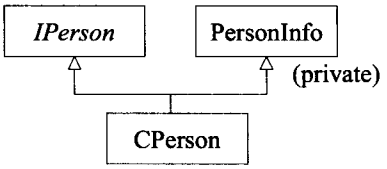
CPerson 和 PersonInfo 的关系是，PersonInfo 刚好有若干函数可帮助 CPerson 比较容易实现出来。就这样。它们的关系因此是 **is-implemented-in-terms-of**（根据某物实现出），而我们知道这种关系可以两种技术实现：复合（见条款 38）和 private 继承（见条款 39）。条款 39 指出复合通常是较受欢迎的做法，但如果需要重新定义 virtual 函数，那么继承是必要的。本例之中 CPerson 需要重新定义 valueDelimOpen 和 valueDelimClose，所以单纯的复合无法应付。最直接的解法就是令 CPerson 以 private 形式继承 PersonInfo，虽然条款 39 也说过，只要多加一点工作，CPerson 也可以结合“复合 + 继承”技术以求有效重新定义 PersonInfo 的 virtual 函数。此处我将使用 private 继承。

但 CPerson 也必须实现 IPerson 接口，那需得以 public 继承才能完成。这导致多重继承的一个通情达理的应用：将“public 继承自某接口”和“private 继承自某实现”结合在一起：

```
class IPerson {                                    //这个 class 指出需实现的接口
public:
   virtual ~IPerson();
   virtual std::string name() const = 0;
   virtual std::string birthDate() const = 0;
};

class DatabaseID { ... };                          //稍后被使用；细节不重要。

class PersonInfo {                                 //这个 class 有若干有用函数，
public:                                            //可用以实现 IPerson 接口。
   explicit PersonInfo(DatabaseID pid);
   virtual ~PersonInfo();
   virtual const char* theName() const;
   virtual const char* theBirthDate( ) const;
   virtual const char* valueDelimOpen() const;
   virtual const char* valueDelimClose() const;
   ...
};
```

```
class CPerson: public IPerson, private PersonInfo { //注意，多重继承
public:
    explicit CPerson(DatabaseID pid): PersonInfo(pid) { }
    virtual std::string name() const          //实现必要的 IPerson 成员函数
    { return PersonInfo::theName(); }

    virtual std::string birthDate() const   //实现必要的 IPerson 成员函数
    { return PersonInfo::theBirthDate(); }

private:                                                          //重新定义
    const char* valueDelimOpen() const { return ""; }            //继承而来的
    const char* valueDelimClose() const { return ""; }           // virtual
                                                                  //"界限函数"
};
```

在 UML 图中这个设计看起来像这样：



这个例子告诉我们，多重继承也有它的合理用途。

故事结束前，请容我说，多重继承只是面向对象工具箱里的一个工具而已。和单一继承比较，它通常比较复杂，使用上也比较难以理解，所以如果你有个单一继承的设计方案，而它大约等价于一个多重继承设计方案，那么单一继承设计方案几乎一定比较受欢迎。如果你唯一能够提出的设计方案涉及多重继承，你应该更努力想一想——几乎可以说一定会有某些方案让单一继承行得通。然而多重继承有时候的确是完成任务之最简洁、最易维护、最合理的做法，果真如此就别害怕使用它。只要确定，你的确是在明智而审慎的情况下使用它。

请记住

- 多重继承比单一继承复杂。它可能导致新的歧义性，以及对 virtual 继承的需要。
- virtual 继承会增加大小、速度、初始化（及赋值）复杂度等等成本。如果 virtual base classes 不带任何数据，将是最具实用价值的情况。
- 多重继承的确有正当用途。其中一个情节涉及“public 继承某个 Interface class”和“private 继承某个协助实现的 class”的两相组合。