继承和面向对象设计

条款32: 确定你的 public 继承塑膜出 is-a 关系、

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

以 c++ 进行面向对象编程，最重要的一个规则是: public inheritance 意味着 is-a 的关系。也就 是说，当你令 class D public inheritance class B 时，D 的对象也是一个 B 的对象，但是 B 的对象不一定是个 D 的对象。

但是这种关系有时候会违反直觉，例如 企鹅 和 鸟 的关系:

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

class Penguin : public Bird {};
```

这种继承关系看起来没有问题，但是 Penguin 根本不会飞，这意味着 Bird 类在设计时发生了失误。因 为部分 fly 并不是 Bird 的共性。通常有两种修改方法:

```
void error(const std::string& msg);
class Penguin : public Bird {
public:
  virtual void fly() { error("..."); }
}
```

这种修改方式，当用户尝试调用 penguin.fly 时会产生错误。这并不是最佳的处理方案，因为这会在运 行期产生错误。正如 条款18 所建议的那样，应当将这种错误扼杀在 编译期 内。

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

class Penguin : public Bird {};
```

第二种方法是可以按照这种结构修改继承关系。但是这种代码的问题在于，随着维护，未来可能对是否能够 fly 并不感兴趣，这也意味着这种结构不再适用。当需要进行更改时就会变得十分麻烦。

另外一个例子就是矩形和正方形的关系:

```
class Rectangle {
public:
  virtual void setHeight(int newHeight);
  virtual void setWidth(int newWidth);
  virtual int height() const;
  virtual int int width() const;
};

class Square : public Rectangle {};

void makeBigger(Rectangle& r) {
  int oldHeight = r.height();
  r.setWidth(r.width() + 10);
  assert(r.height() == oldHeight);
}

Square s;
assert(s.width() == s.height());
makeBigger(s);

assert(s.width() == s.height());
```

显然，直观的想法是两个 assert 结果应该相同，因为正方形总是长宽相等的。但是 makeBigger 后， 长宽不相等了，但是后面 assert 又为 true。这显然是不合理的。

is-a 并非唯一的 classes 之间的关系，除此之外还有 has-a 和 is-implemented-in-terms-of 两种关系。需要了解这些 class 相互关系之间的差异，才能设计好的继承类型。

条款33: 避免遮掩继承而来的名称

- 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。
- 为了让遮掩的名称再见天日，可使用 using 声明式或转交函数。

这个条款主要用来解释作用域的相关内容。

```
int x;
void func() {
  double x;
  std::cin >> x;
}
```

上面这段代码， global x 被 local x 名称遮掩了，因此实际上操作的是 double x。在继承关系里 也会发生这种事情。

```
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() {
    mf2();
  }
};
```

上述代码中，mf4 调用 mf2，编译器会先在 local 域内查找，没有的话查找 Drived 域，然后 Base 域，紧接着 namespace，最后 global。因此实际上调用的是 Base::mf2。如果修改一下上面的代码， 就可能发生 遮掩名称 的问题。

```
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() {
    mf2();
  }
};

Derived d;
int x;

d.mf1();  // 正确
d.mf1(x); // 错误
d.mf2();  // 正确
d.mf3();  // 正确
d.mf3(x); // 错误
```

上述代码在调用过程中，就会发生名称掩盖。可以看到，当调用 mf1 和 mf3 时都会选择 Derived:: 内 的内容，而没有选择 Base 内的方法。

想要正确运行，有两种方式完成:

1. 使用 using 声明式

```
class Derived : public Base {
public:
  using Base::mf1;
  using Base::mf3;
  virtual void mf1();
  void mf3();
  void mf4() {
    mf2();
  }
};

Derived d;
int x;

d.mf1();  // 正确
d.mf1(x); // 正确
d.mf2();  // 正确
d.mf3();  // 正确
d.mf3(x); // 正确
```

使用 using 声明式来重新声明那些被掩盖的名称，可以实现调用 base function 的功能。

2. 使用转交函数

但是有时候你又不想继承 Base 中的所有函数(这显然不能使用 public 继承)，而是想要继承部分函数。 那么就可以使用 转交函数 来实现。

```
class Derived : private Base {
public:
  virtual void mf1() {
    Base::mf1();
  }
};
```

这样做既不会像 using 那样将所有名称完全暴露，又能继承部分的函数。

条款34: 区分接口继承和实现继承

- 接口继承和实现继承不同。在 public 继承之下，derived class 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 impure virtual 函数具体指定接口继承和缺省实现继承

- non-virtual 函数具体指定接口继承以及强制性实现继承。

在 public 继承概念的基础上，可以发现它还由两部分组成: 函数接口继承 和 函数实现继承。身为 class 的设计者有时你会希望 derived class：1. 只继承函数接口，2. 同时继承接口和实现，但是希望能够 覆写，3. 同时继承接口和实现，并且不允许覆写。

```
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 {};
```

- 成员接口总是会被继承。

Shape 对 derived class 的影响十分深远，正如 public 继承意味着 is-a 关系那样。所有能够施 用于 Shape 的方法，也同样适用 derived class。正是如此，成员接口必须被继承。

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

在 Shape 中，draw 是一个 pure virtual 函数，它具有两个突出特点: 1. 必须被 继承了他们 的 具象 class 重新声明，2。abstract class 中通常没有 pure virtual 的定义。这意味着所有继承 自 Shape 的 derived classes 都必须提供一个 draw 函数，但是 Shape 不干涉如何实现。

pure virtual 可以提供定义，但是需要指明 class 才可以调用，例如: Shape::draw()。它可以用 来实现一个机制，为简朴的 impure virtual 函数提供更平常更安全的缺省实现。

- impure virtual 函数的目的是，让 derived classes 继承该函数的接口和缺省实现。

适用 impure virtual 意味着，derived class 的设计者必须支持一个 error 函数，如果不想写， 可以适用 Shape class 的缺省版本。

但是这种 impure virtual 函数的写法很容易因为疏忽而产生问题，特别是在代码维护过程中。

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

void Airplane::fly(const Airport& destination) {
  缺省代码
}

class ModelA: public Airplane {};
class ModelB: public Airplane {};
```

```
class ModelC: public Airplane {};
```

上述代码的本意是 ModelA 和 ModelB 采用默认飞行方式，而 ModelC 则适用自己的飞行方式。但是在 程序维护过程中，忘记为 ModelC 实现了新的飞行方式，所以在调用时仍然采用了默认方案。这就产生了 错误。

为了防止这种事情的发生，必须时刻提醒 derived class 的设计者，在 缺省方案 和 自定义方案 中进 行选择。

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


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

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

通过 pure virtual 来提醒 derived class 设计者必须进行选择，能够很好的防止疏忽产生问题。同 时又通过 defaultFly 来提供默认飞行方式。

上面的代码将接口和缺省实现分开实现了，但是很多人反对这样的事情发生。一种很好的手段就是为 pure virtual 函数定义自己的实现，来省去 defaultFly。

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

class ModelA : public Airplane {
```

```
public:
  virtual void fly(const Airport& destination) {
    Airplane::fly(destinatino);
  }
};

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

- 声明 non-virtual 函数的目的是为了令 derived classes 继承函数的接口以及一份强制性实现 non-virtual 函数意味着，不打算在 derived classes 中有不同的行为，它所表现出的不变性凌驾于 特异性之上。

如果你能够履行上面的这些差异，那么你应该能够避免两个错误:

1. 将所有函数声明为 non-virtual。这让 dervied classes 没有足够的空间进行特化工作。特别是 non-virtual 析构函数。实际上任何 class 如果打算被用来当作一个 base class，它就会拥有若干个 virtual 函数。
2. 将所有成员函数声明为 virtual 除了部分 interface classes，大多数 class 都会有某些函数就是不该在 derived class 中被重 新定义，那么你应该将这些函数声明为 non-virtual。

条款35: 考虑 virtual 函数以外的其他选择

- virtual 函数的替代方案包括 NVI 手法以及 Strategy 涉及模式的多种形式。NVI 手法自身一个特殊形式的 Template Method 设计模式。
- 将机能从成员函数转移到 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兼容” 的所有可调用物。

```
class GameCharacter {
public:
  virtual int healthValue() const;
};
```

上面是一个游戏的角色类型，他具有一个 non-pure virtual 函数，用来返回当前生命值。这意味着， 每个不同的角色可以实现不同的 healthValue 方案，同时还具有一个默认的方案。

如何替代 virtual 呢?

借由 non-virtual interface 实现 Template Method 模式

```
class GameCharacter {
public:
  int healthValue() const {
    ...
    int retVal = doHealthValue();
    ...
    return retVal;
  }
private:
  virtual int doHealthValue() const {
    缺省代码
  }
};
```

上面代码的思想是，virtual 函数应该几乎总是 private 的。而较好的设计是保留 healthValue 为 public 成员函数，但是让它成为 non-virtual，并调用一个 private virtual 函数。这种手法被 称为 non-virtual interface (NVI)。它是所谓 Template Method 设计模式的一个独特表现形式。

NVI 手法的优点在于，调用 doHealthValue 的前后可以添加部分处理工作，例如锁定/解锁 mutex，制 造运转日志记录项、验证 class 约束条件、验证函数先决/事后条件等等。

NVI 手法下其实没必要让 virtual 一定得是 private。例如某些函数可能 derived class 必须调用 base class 的函数，此时就可以适用 protected。而部分要求必须适用 public 修饰的函数，例如 具有多态性质的 base classes 的析构函数，这么依赖就不能使用 NVI 手法了。

借由 Function Pointers 实现 Strategy 模式

```
class GameCharacter;
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;
};

class EvilBadGuy : public GameCharacter {
public:
  explicit EvilBadGuy(HealthCalcFunc hcf = defaultHealthCalc) : GameCharacter(hcf)
{}
};

int loseHealthQuickly(const GameCharacter&);
int loseHealthSlowly(const GameCharacter&);
```

```
EvilBadGuy ebg1(loseHealthQuickly);
EvilBadGuy ebg1(loseHealthSlowly);
```

通过上面这种方式，借助 Function Pointers 实现 strategy 模式，可以看到有两点好处:

- 同一类型的对象，可以使用不同的运算方法。
- 已知的某个对象，计算过程可以在运行期进行更改。

但是这样设计程序也会带来问题，因为你将一个 member function 替换成为了 non-member function， 这就意味着，如果这个函数需要用到 private member 相关的内容就没有办法了。除非，你尝试通过 firends 或者为其中一部分提供 public 访问方法，但是这些行为又会降低 GameCharacter 的封装性。因此， 你需要在 non-member funtion pointer 与 封装性 之间进行取舍。

借由 function 完成 Strategy

将上面的这种实现方式改为通过 std::function 对象来实现。

```
#include <functional>

class GameCharacter;
int defaultHealthCalc(const GameCharacter& gc);
class GameCharacter {
public:
  typedef std::function<int (const GameCharacter&)> HealthCalcFunc;
  explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc) : healthFunc(hcf)
{}
  int healthValue() const {
    return healthFunc(*this);
  }
private:
  HealthCalcFunc healthFunc;
};
```

这种改变是为非常细小的，只是将原有的函数指针转化为了一个 std::function 对象，相当于指向函数 的泛化指针。但是有时可以提供更高的弹性。这主要来自于 std::function 可以封装普通函数、lambda 函数、仿函数(函数对象)以及成员函数。

```
// 普通函数，但是返回值为 short, std::function 允许隐式转型
short calcHealth(const GameCharacter& gc);

// 函数对象
struct HealthCalculator {
  int operator()(const GameCharacter& gc) const;
};

// 成员函数
class GameLevel {
public:
  float health(const GameCharacter& gc) const;
```

```
};

class EvilBadGuy : public GameCharacter {
public:
  EvilBadGuy(HealthCalcFunc hcf) : GameCharacter(hcf) {}
};

EvilBadGuy ebg1(calcHealth); // 封装普通函数
EvilBadGuy ebg2(HealthCalculator()); // 封装函数对象

GameLevel currentLevel;
EvilBadGuy ebg3(std::bind(
  &GameLevel::health,
  currentLevel,
  std::placeholders::_1
)); // 封装成员函数
```

上述代码为用户提供了更多的设计选择。

古典的 strategy 模式

传统的 Strategy 做法是采用一个分离的继承体系来完成的。其中包括两个根系，一个是用来表示角色的 GameCharacter(EvilBadGuy, EyeCandyCharacter)，另一个则是用来表示生命计算的 HealthCalcFunc (SlowHealthLoser, FastHealthLoser)

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

private:
  HealthCalcFunc* pHealthCalc;
};
```

后续需要更更多的人物，以及生命值计算方法，只需要在这两个类的基础上进行继承就可以了。

条款36: 绝不重新定义继承而来的 non-virtual 函数

- 绝对不要重新定义继承而来的 non-virtual 函数。

加入存在下面的继承关系:

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

class D : public B {
public:
  void mf(void);
};

D x;
B* pb = &x;
pb->mf();
D* pd = &x;
pd->mf();
```

上面的代码中，两次调用结果不一样。这主要是因为 non-virtual 函数是 静态绑定 的，这也就意味着 使用 B pointer 来调用就会调用 B::mf，而用 D pointer 力调用则是 D::mf。另一方面 virtual 却是 动态绑定 的，所以 virtual function 并不会产生这个问题。

那么为什么不要重新定义继承而来的 non-virtual 函数呢？两点原因:

- 如果 D 需要重新定义 mf，而每个 B 对象又必须调用 B::mf，而非 D::mf，那么这就意味着这并非 public 继承关系。
- 如果 D 需要 public 继承 B，并且 D 需要实现不同的 mf，那么就应该使用 virtual 函数，而不应该反映出 “不变性凌驾特异性” 的性质

条款37: 绝不重新定义继承而来的缺省参数值

- 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——你唯一可以覆写的东西——确实动态绑定。

这条条款的范围可以缩小，因为条款36已经讨论了不应该重新定义 non-virtual 函数，那实际上这里强 调的就是 继承带有缺省数值的 virtual 函数。

本条条款成立的主要原因就是: virtual 函数是动态绑定的，而缺省参数值则是静态绑定的。

```
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 = Green) const;
```

```
};

class Circle : public Shape {
public:
  virtual void draw(ShapeColor color) const;
};

Shape* p = new Rectangle();
p->draw();
```

像上面的 Rectangle 类，重定义了缺省参数值，这可能会带来与预期相违背的结果。正如上面的调用方 式，由于 p 的静态类型是 Shape*，而缺省参数是静态绑定的，因此实际上进行的是 p->draw(Red) 操 作而非 p->draw(Green)。

那如果并不重定义，只是将缺省传给 derived class 呢？

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

这种做法也是不可取的，因为代码重合。并且不好维护，比如后续将 Shape 的缺省参数改为 Green，那 就又会出现上面同样的问题。一个好的解决方案是，使用条款35中的手段来更改这种代码，比如 NVI。

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

class Rectangle {
private:
  virtual void doDraw(ShapeColor color) const {
    ...
  }
}
```

条款38: 通过复合塑模出 has-a 或 “根据某物实现出”

- 复合的意义和 public 继承完全不同。
- 在应用域，复合意味着 has-a。在实现域，复合意味 has-a。在实现域，复合意味着 is-implemented-in-terms-of。

复合关系是类型之间的一种关系，当某种类型的对象内含其他类型对象，就是这种关系。

复合关系也和 public 继承一样具有现实意义，它主要包括两种:

- has-a: 复合发生在应用域内的对象时，表现出 has-a 的关系。应用域就是为了塑造现实世界中的某些事物而设计的类。
- is-implemented-in-terms-of: 复合发生在实现域内，表现出这种关系。实现域是为了实现某些细节而设计的类，比如缓冲区、互斥器、查找树等等。

has-a

has-a 的关系很好理解，比如下面的代码就是 “人有一个地址” 这样的关系。

```
class Address {};
class PhoneNumber {};
class Person {
public:
private:
  std::string name;
  Address address;
  PhoneNumber voiceNumber;
  PhoneNumber foxNumber;
};
```

is-implemented-in-terms-of

这种 根据某物实现出 的意义则不太好理解。

这里以 set 为例，假如你希望设计自己的 set，并且你知道可以使用底层的 linked lists 来实现。 于是你开始尝试让 set 继承 list。

```
template<typename T>
class Set : public std::list<T> {};
```

看上去很完美，但是存在重大问题。我们已经知道了 public 是 is-a 的关系，那也就意味着 Set 一定 是一个 list。但是 list 允许多个重复数据，而 Set 不允许。那显然两者并不是 is-a 的关系。正确 的做法是，将 list 应用于 Set。

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

```
  }
  bool remove(const T& item) {
    typename std::list<T>::iterator it = std::find(rep.begin(), rep.end(), item);
    if (it != rep.end()) rep.erase(it);
  }
private:
  std::list<T> rep;
}
```

条款39: 明智而审慎的使用 private 继承

- private 继承意味着 is-implemented-in-terms of。它通常比复合级别耕地。但是当 derived class 需要访问 protected base class 的成员，或者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” 的程序开发者而言，可能很重要。

---

private 继承有两条规则:

1. private 继承关系，编译器不会自动将一个 derived class 对象转换为 base class。
2. private 继承将从 base class 继承而来的所有成员在 derived class 中变成 private 属性。

private 继承的含义是 implemented-in-terms-of。private 继承也意味着只有 实现部分 被继承， 接口部分 被自动略去。而 private 仅是一种实现计数，没有设计意义。

和前面提出的复合相比，应该尽可能的使用复合，必要时才使用 private 继承。这种必要源自当 protected 成员或者 virtual 函数被牵扯进来时。

一个例子是使用 timer。下面是一个 Timer 对象，它用来对时间进行计时操作。

```
class Timer {
public:
  explicit Timer(int tickFrequency);
  virtual void onTick() const;
};
```

现在我们需要为 Widget 对象实现一个功能，让它记录每个成员函数被调用的次数。我们可以使用 private 继承来实现，因为只有这样才能够重新定义 Timer::onTick 方法。

```
class Widget : private Timer {
private:
  virtual void onTick() const;
};
```

但是使用 private 继承并不是必要的，我们可以使用复合来重构上面的代码。

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

使用下面这种复合方式来撰写代码有两点好处:

- 如果你希望 Widget 可以拥有子类，但是又向阻止子类重新定义 onTick，就需要使用复用。因为继承无法实现这种手段。
- 如果你希望将 Widget 的编译依存性降至最低，就需要使用复用。因为这种形式不需要 include 任何东西。

空白基类优化

Empty Base Optimization(EOB)。这种行为源自对空白类大小的期待。通常，空白类的独立对象大小 会不为0, 因为 c++ 可能会安插一个 char 到空对象中。有的时候为了位对齐的需求，可能会更大。但是， 非独立对象却不是，非独立对象则很有可能大小为 0。

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

// sizeof(HoldsAnInt) > sizeof(int)
```

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

// sizeof(HoldsAnInt) == sizeof(int)
```

可以看到使用继承方案，可以消除空白类的体积负担。而这里的 empty 类，可能并不是 empty 的，他们 往往内涵 typedefs, enums, static 成员变量，或者 non-virtual 函数。这里的 empty 实际上 是指，不含 non-static 成员变量。

总结

总的来说当你面对 “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 成员时，或者重新定义 virtual 函数，private 继承极有可能称为正统设计。当然，你也可以使用 public 继承和复合技术来替代这个过程，尽管有更高的复杂度。

条款40: 明智而审慎的使用多重继承

- 多重继承比单一继承更复杂。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、初始化复杂度等等成本。如果 virtual base class 不带任何数据，僵尸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。

语义歧义

当涉及多重继承，程序可能从一个以上的 base classes 继承相同的名称，这会导致歧义。

```
class BorrowableItem {
public:
  void checkOut(); // 离开时进行检查
};
class ElectronicGadget {
private:
  bool checkOut() const; // 执行自我检测，返回测试是否通过
};

class MP3Player : public BorrowableItem, public ElectronicGadget {};

MP3Player mp3;
mp3.checkOut(); // 究竟调用的哪个 checkOut。
```

虽然上面的 ElectronicGadget::checkOut 是 private 的，但是编译器在开始阶段只在乎调用的名 称是否解析正确。而在此时会发生歧义，因为按照最佳匹配，两者具有相同的匹配程度。

正确的做法是:

```
mp3.BorrowableItem::checkOut()
```

钻石型多重继承

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

上面就是一个钻石型的多重继承，其中存在的问题在于当 File 中有一个对象 A 时，IOFile 中应该有 几分 A 对象。c++ 支持两种选择。

- 多份 这是一个缺省做法，就是从 InputFile 和 OutputFile 中都复制对象 A。上面的代码就是这样的效果。
- 一份 如果你希望只存在一份，那么你就需要令那个 base class 成为一个 virtual base class，也就是你 必须使用 virtual 继承。

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

从正确行为的观点来看，public 继承总应该时 virtual 的。但是这又涉及到成本问题，编译器为了避免 成员变量重复，它必须在幕后做一些工作，这就导致几个结果: 1. virtual public 继承的体积更大，2. virtual public 继承的访问速度更慢，3. classes 若派生自 virtual base，当需要初始化时，必 须认知 virtual bases，不论距离 base 多远，4. 当一个新的 derived class 加入继承体系中，它 必须承担其 virtual bases 的初始化责任。

因此关于 virtual bases classes 的忠告很简单:

1. 非必须使用 virtual base，就使用 non-virtual 继承。
2. 如果必须使用 virtual base，那就尽量避免在其中放置数据。

public 继承 interface，private 继承协助类

多重继承也有很多正常的实用场景。这里举例一个 public 和 private 继承并用的场景。

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

IPerson 是一个接口，并且使用 工厂函数 产生 Person 对象。这也意味着，想要产生 Person 对象， 首先需要设计一个类继承并实现 IPerson。

另外一个工具类 PersonInfo，用来实现数据库相关操作，它为后续对象提供了一些好用的方法。

```
class PersonInfo {
public:
  explicit PersonInfo(DatabaseId id);
  virtual ~PersonInfo();
  virtual const char* theName() const;
  virtual const char* theBirthDate() const;
private:
  virtual const char* valueDelimOpen() const;
```

```
  virtual const char* valueDElimClose() const;
};
```

现在希望实现一个类，它既可以被生产为 IPerson，也可以依靠 PersonInfo 来实现 IPerson 其中的 操作。

```
class CPerson: public IPerson, private PersonInfo {
public:
  explicit CPerson(DatabaseId id): PersonInfo(id) {}
  virtual std::string name() const { // 借用 PersonInfo 方法实现 IPerson 的方法
    return PersonInfo::theName();
  }
  virtual std::string birthDate const {
    return PersonInfo::theBirthDate();
  }
private:
  virtual const char* valueDelimOpen() const { return "{"; } // 重写从 PersonInfo
继承来的 virtual 限界字符函数
  virtual const char* valueDElimClose() const { return "}"; }
}
```