4

设计与声明

Designs and Declarations

所谓软件设计，是“令软件做出你希望它做的事情”的步骤和做法，通常以颇为一般性的构想开始，最终演变成十足的细节，以允许特殊接口（interfaces）的开发。这些接口而后必须转换为 C++ 声明式。本章中我将对良好 C++ 接口的设计和声明发起攻势。我以或许最重要、适合任何接口设计的一个准则作为开端：“让接口容易被正确使用，不容易被误用”。这个准则设立了一个舞台，让其他更专精的准则对付一大范围的题目，包括正确性、高效性、封装性、维护性、延展性，以及协议的一致性。

以下准备的材料并不覆盖你需要知道的优良接口设计的每一件事，但它强调某些最重要的考虑，对某些最频繁出现的错误提出警告，为 class、function 和 template 设计者经常遭遇的问题提供解答。

条款 18：让接口容易被正确使用，不易被误用

Make interfaces easy to use correctly and hard to use incorrectly.

C++ 在接口之海漂浮。function 接口、class 接口、template 接口……每一种接口都是客户与你的代码互动的手段。假设你面对的是一群“讲道理的人”，那些客户企图把事情做好。他们想要正确使用你的接口。这种情况下如果他们对任何其中一个接口的用法不正确，你至少也得负一部分责任。理想上，如果客户企图使用某个接口而却没有获得他所预期的行为，这个代码不该通过编译；如果代码通过了编译，它的作为就该是客户所想要的。

欲开发一个“容易被正确使用，不容易被误用”的接口，首先必须考虑客户可能做出什么样的错误。假设你为一个用来表现日期的 class 设计构造函数：

```
class Date {
public:
   Date(int month, int day, int year);
   ...
};
```

乍见之下这个接口通情达理（至少在美国如此），但它的客户很容易犯下至少两个错误。第一，他们也许会以错误的次序传递参数：

```
Date d(30, 3, 1995);    //喔欧! 应该是 "3, 30" 而不是 "30, 3"
```

第二，他们可能传递一个无效的月份或天数：

```
Date d(2, 30, 1995);    //喔欧! 应该是 "3, 30" 而不是 "2, 30"
```

（上个例子也许看起来很蠢，但别忘了，键盘上的 2 就在 3 旁边。打岔一个键的情况并不是太罕见。）

许多客户端错误可以因为导入新类型而获得预防。真的，在防范“不值得拥有的代码”上，类型系统（type system）是你的主要同盟国。既然这样，就让我们导入简单的外覆类型（wrapper types）来区别天数、月份和年份，然后于 Date 构造函数中使用这些类型：

```
struct Day {
explicit Day(int d)
  : val(d) { }
int val;
};

struct Month {
explicit Month(int m)
  : val(m) { }
int val;
};

struct Year {
explicit Year(int y)
  : val(y){ }
int val;
};

class Date {
public:
   Date(const Month& m, const Day& d, const Year& y);
   ...
};
Date d(30, 3, 1995);                          //错误!不正确的类型
Date d(Day(30), Month(3), Year(1995));        //错误!不正确的类型
Date d(Month(3), Day(30), Year(1995));        //OK, 类型正确
```

令 Day, Month 和 Year 成为成熟且经充分锻炼的 classes 并封装其内数据，比简单使用上述的 structs 好（见条款 22）。但即使 structs 也已经足够示范：明智而审慎地导入新类型对预防“接口被误用”有神奇疗效。

一旦正确的类型就定位，限制其值有时候是通情达理的。例如一年只有 12 个有效月份，所以 Month 应该反映这一事实。办法之一是利用 enum 表现月份，但 enums 不具备我们希望拥有的类型安全性，例如 enums 可被拿来当一个 ints 使用（见条款 2）。比较安全的解法是预先定义所有有效的 Months：

```
class Month {
public:
   static Month Jan() { return Month(1); }      //函数，返回有效月份。
   static Month Feb() { return Month(2); }      //稍后解释为什么。
   ...                                          //这些是函数而非对象。
   static Month Dec() { return Month(12); }
   ...                                          //其他成员函数
private:
   explicit Month(int m);                       //阻止生成新的月份。
   ...                                          //这是月份专属数据。
};
Date d(Month::Mar(), Day(30), Year(1995));
```

如果“以函数替换对象，表现某个特定月份”让你觉得诡异，或许是因为你忘记了 non-local static 对象的初始化次序有可能出问题。建议阅读条款 4 恢复记忆。

预防客户错误的另一个办法是，限制类型内什么事可做，什么事不能做。常见的限制是加上 const。例如条款 3 曾经说明为什么“以 const 修饰 operator* 的返回类型”可阻止客户因“用户自定义类型”而犯错：

```
if (a * b = c) ...      //喔欧，原意其实是要做一次比较动作！
```

下面是另一个一般性准则“让 types 容易被正确使用，不容易被误用”的表现形式：“除非有好理由，否则应该尽量令你的 types 的行为与内置 types 一致”。客户已经知道像 int 这样的 type 有些什么行为，所以你应该努力让你的 types 在合样合理的前提下也有相同表现。例如，如果 a 和 b 都是 ints，那么对 a*b 赋值并不合法，所以除非你有好的理由与此行为分道扬镳，否则应该让你的 types 也有相同的表现。是的，一旦怀疑，就请拿 ints 做范本。

避免无端与内置类型不兼容，真正的理由是为了提供行为一致的接口。很少有其他性质比得上“一致性”更能导致“接口容易被正确使用”，也很少有其他性质比得

上“不一致性”更加剧接口的恶化。STL 容器的接口十分一致（虽然不是完美地一致），这使它们非常容易被使用。例如每个 STL 容器都有一个名为 size 的成员函数，它会告诉调用者目前容器内有多少对象。与此对比的是 Java，它允许你针对数组使用 length *property*，对 Strings 使用 length *method*，而对 Lists 使用 size *method*；.NET 也一样混乱，其 Arrays 有个 *property* 名为 Length，其 ArrayLists 有个 *property* 名为 Count。有些开发人员会以为整合开发环境（integrated development environments, IDEs）能使这般不一致性变得不重要，但他们错了。不一致性对开发人员造成的心理和精神上的摩擦与争执，没有任何一个 IDE 可以完全抹除。

任何接口如果要求客户必须记得做某些事情，就是有着“不正确使用”的倾向，因为客户可能会忘记做那件事。例如条款 13 导入了一个 factory 函数，它返回一个指针指向 Investment 继承体系内的一个动态分配对象：

```
Investment* createInvestment();          //来自条款 13; 为求简化暂略参数。
```

为避免资源泄漏，createInvestment 返回的指针最终必须被删除，但那至少开启了两个客户错误机会：没有删除指针，或删除同一个指针超过一次。

条款 13 表明客户如何将 createInvestment 的返回值存储于一个智能指针如 auto_ptr 或 tr1::shared_ptr 内，因而将 delete 责任推给智能指针。但万一客户忘记使用智能指针怎么办？许多时候，较佳接口的设计原则是先发制人，就令 factory 函数返回一个智能指针：

```
std::tr1::shared_ptr<Investment> createInvestment();
```

这便实质上强迫客户将返回值存储于一个 tr1::shared_ptr 内，几乎消弭了忘记删除底部 Investment 对象（当它不再被使用时）的可能性。

实际上，返回 tr1::shared_ptr 让接口设计者得以阻止一大群客户犯下资源泄漏的错误，因为就如条款 14 所言，tr1::shared_ptr 允许当智能指针被建立起来时指定一个资源释放函数（所谓删除器，"deleter"）绑定于智能指针身上（auto_ptr 就没有这种能耐）。

假设 class 设计者期许那些“从 createInvestment 取得 Investment* 指针”的客户将该指针传递给一个名为 getRidOfInvestment 的函数，而不是直接在它身上动刀（使用 delete）。这样一个接口又开启通往另一个客户错误的大门，该错误是“企图使用错误的资源析构机制”（也就是拿 delete 替换 getRidOfInvestment）。

createInvestment 的设计者可以针对此问题先发制人：返回一个“将 getRidOfInvestment 绑定为删除器（deleter）”的 tr1::shared_ptr。

tr1::shared_ptr 提供的某个构造函数接受两个实参：一个是被管理的指针，另一个是引用次数变成 0 时将被调用的“删除器”。这启发我们创建一个 null tr1::shared_ptr 并以 getRidOfInvestment 作为其删除器，像这样：

```
std::tr1::shared_ptr<Investment>          //企图创建一个 null shared_ptr
  pInv(0, getRidOfInvestment);            //并携带一个自定的删除器。
                                          //此式无法通过编译。
```

啊呀，这不是有效的 C++。tr1::shared_ptr 构造函数坚持其第一参数必须是个指针，而 0 不是指针，是个 int。是的，它可被转换为指针，但在此情况下并不够好，因为 tr1::shared_ptr 坚持要一个不折不扣的指针。转型（cast）可以解决这个问题：

```
std::tr1::shared_ptr<Investment>          //建立一个 null shared_ptr 并以
  pInv( static_cast<Investment*>(0),      //getRidOfInvestment 为删除器;
    getRidOfInvestment);                  //条款 27 提到 static_cast
```

因此，如果我们要实现 createInvestment 使它返回一个 tr1::shared_ptr 并夹带 getRidOfInvestment 函数作为删除器，代码看起来像这样：

```
std::tr1::shared_ptr<Investment> createInvestment()
{
  std::tr1::shared_ptr<Investment> retVal(static_cast<Investment* >(0),
                                          getRidOfInvestment);
  retVal = ... ;   //令 retVal 指向正确对象
  return retVal;
}
```

当然啦，如果被 pInv 管理的原始指针（raw pointer）可以在建立 pInv 之前先确定下来，那么“将原始指针传给 pInv 构造函数”会比“先将 pInv 初始化为 null 再对它做一次赋值操作”为佳。至于其原因，请见条款 26。

tr1::shared_ptr 有一个特别好的性质是：它会自动使用它的“每个指针专属的删除器”，因而消除另一个潜在的客户错误：所谓的 "cross-DLL problem"。这个问题发生于“对象在动态连接程序库（DLL）中被 new 创建，却在另一个 DLL 内被 delete 销毁”。在许多平台上，这一类“跨 DLL 之 new/delete 成对运用”会导致运行期错误。tr1::shared_ptr 没有这个问题，因为它缺省的删除器是来自“tr1::shared_ptr 诞生所在的那个 DLL”的 delete。这意思是……唔……让我举个例子，如果 Stock 派生自 Investment 而 createInvestment 实现如下：

```
std::tr1::shared_ptr<Investment> createInvestment()
{
    return std::tr1::shared_ptr<Investment>(new Stock);
}
```

返回的那个 tr1::shared_ptr 可被传递给任何其他 DLLs，无需在意 "cross-DLL problem"。这个指向 Stock 的 tr1::shared_ptrs 会追踪记录“当 Stock 的引用次数变成 0 时该调用的那个 DLL's delete”。

本条款并非特别针对 tr1::shared_ptr，而是为了“让接口容易被正确使用，不容易被误用”而设。但由于 tr1::shared_ptr 如此容易消除某些客户错误，值得我们核计其使用成本。最常见的 tr1::shared_ptr 实现品来自 Boost（见条款 55）。Boost 的 shared_ptr 是原始指针（raw pointer）的两倍大，以动态分配内存作为簿记用途和“删除器之专属数据”，以 virtual 形式调用删除器，并在多线程程序修改引用次数时蒙受线程同步化（thread synchronization）的额外开销。（只要定义一个预处理器符号就可以关闭多线程支持）。总之，它比原始指针大且慢，而且使用辅助动态内存。在许多应用程序中这些额外的执行成本并不显著，然而其“降低客户错误”的成效却是每个人都看得到。

请记住

- 好的接口很容易被正确使用，不容易被误用。你应该在你的所有接口中努力达成这些性质。
- “促进正确使用”的办法包括接口的一致性，以及与内置类型的行为兼容。
- “阻止误用”的办法包括建立新类型、限制类型上的操作，束缚对象值，以及消除客户的资源管理责任。
- tr1::shared_ptr 支持定制型删除器（custom deleter）。这可防范 DLL 问题，可被用来自动解除互斥锁（mutexes; 见条款 14）等等。

条款 19：设计 class 犹如设计 type

Treat class design as type design.

C++ 就像在其他 OOP（面向对象编程）语言一样，当你定义一个新 class，也就定义了一个新 type。身为 C++ 程序员，你的许多时间主要用来扩张你的类型系统（type system）。这意味你并不只是 class 设计者，还是 type 设计者。重载（overloading）函数和操作符、控制内存的分配和归还、定义对象的初始化和终结……全都在你手上。因此你应该带着和“语言设计者当初设计语言内置类型时”一样的谨慎来研讨 class 的设计。

设计优秀的 classes 是一项艰巨的工作，因为设计好的 types 是一项艰巨的工作。好的 types 有自然的语法，直观的语义，以及一或多个高效实现品。在 C++ 中，一个不良规划下的 class 定义恐怕无法达到上述任何一个目标。甚至 class 的成员函数的效率都有可能受到它们“如何被声明”的影响。

那么，如何设计高效的 classes 呢？首先你必须了解你面对的问题。几乎每一个 class 都要求你面对以下提问，而你的回答往往导致你的设计规范：

- **新 type 的对象应该如何被创建和销毁？** 这会影响到你的 class 的构造函数和析构函数以及内存分配函数（`operator new`, `operator new[]`, `operator delete` 和 `operator delete[]`——见第 8 章）的设计，当然前提是如果你打算撰写它们。
- **对象的初始化和对象的赋值该有什么样的差别？** 这个答案决定你的构造函数和赋值（*assignment*）操作符的行为，以及其间的差异。很重要的是别混淆了“初始化”和“赋值”，因为它们对应于不同的函数调用（见条款 4）。
- **新 type 的对象如果被 *passed by value*（以值传递），意味着什么？** 记住，*copy* 构造函数用来定义一个 type 的 *pass-by-value* 该如何实现。
- **什么是新 type 的“合法值”？** 对 class 的成员变量而言，通常只有某些数值集是有效的。那些数值集决定了你的 class 必须维护的约束条件（invariants），也就决定了你的成员函数（特别是构造函数、赋值操作符和所谓 "setter" 函数）必须进行的错误检查工作。它也影响函数抛出的异常、以及（极少被使用的）函数异常明细列（exception specifications）。

- **你的新 type 需要配合某个继承图系（inheritance graph）吗？**如果你继承自某些既有的 classes，你就受到那些 classes 的设计的束缚，特别是受到“它们的函数是 virtual 或 non-virtual”的影响（见条款 34 和条款 36）。如果你允许其他 classes 继承你的 class，那会影响你所声明的函数——尤其是析构函数——是否为 virtual（见条款 7）。

- **你的新 type 需要什么样的转换？**你的 type 生存于其他一海票 types 之间，因而彼此该有转换行为吗？如果你希望允许类型 `T1` 之物被隐式转换为类型 `T2` 之物，就必须在 class `T1` 内写一个类型转换函数（`operator T2`）或在 class `T2` 内写一个 non-explicit-one-argument（可被单一实参调用）的构造函数。如果你只允许 explicit 构造函数存在，就得写出专门负责执行转换的函数，且不得为类型转换操作符（type conversion operators）或 non-explicit-one-argument 构造函数。（条款 15 有隐式和显式转换函数的范例。）

- **什么样的操作符和函数对此新 type 而言是合理的？**这个问题的答案决定你将为你的 class 声明哪些函数。其中某些该是 member 函数，某些则否（见条款 23, 24, 46）。

- **什么样的标准函数应该驳回？**那些正是你必须声明为 private 者（见条款 6）。

- **谁该取用新 type 的成员？**这个提问可以帮助你决定哪个成员为 public，哪个为 protected，哪个为 private。它也帮助你决定哪一个 classes 和/或 functions 应该是 friends，以及将它们嵌套于另一个之内是否合理。

- **什么是新 type 的“未声明接口”（undeclared interface）？**它对效率、异常安全性（见条款 29）以及资源运用（例如多任务锁定和动态内存）提供何种保证？你在这些方面提供的保证将为你的 class 实现代码加上相应的约束条件。

- **你的新 type 有多么一般化？**或许你其实并非定义一个新 type，而是定义一整个 types 家族。果真如此你就不该定义一个新 class，而是应该定义一个新的 class template。

- **你真的需要一个新 type 吗？**如果只是定义新的 derived class 以便为既有的 class 添加机能，那么说不定单纯定义一或多个 non-member 函数或 templates，更能够达到目标。

这些问题不容易回答，所以定义出高效的 classes 是一种挑战。然而如果能够设计出至少像 C++ 内置类型一样好的用户自定义（user-defined）classes，一切汗水便都值得。

请记住

- Class 的设计就是 type 的设计。在定义一个新 type 之前，请确定你已经考虑过本条款覆盖的所有讨论主题。

条款 20：宁以 pass-by-reference-to-const 替换 pass-by-value

Prefer pass-by-reference-to-const to pass-by-value.

缺省情况下 C++ 以 *by value* 方式（一个继承自 C 的方式）传递对象至（或来自）函数。除非你另外指定，否则函数参数都是以实际实参的复件（副本）为初值，而调用端所获得的亦是函数返回值的一个复件。这些复件（副本）系由对象的 *copy* 构造函数产出，这可能使得 *pass-by-value* 成为昂贵的（费时的）操作。考虑以下 class 继承体系：

```
class Person {
public:
    Person();                              //为求简化，省略参数
    virtual ~Person();                     //条款 7 告诉你为什么它是 virtual
    ...
private:
    std::string name;
    std::string address;
};

class Student: public Person {
public:
    Student();                             //再次省略参数
    ~Student();
    ...
private:
    std::string schoolName;
    std::string schoolAddress;
};
```

现在考虑以下代码，其中调用函数 validateStudent，后者需要一个 Student 实参（*by value*）并返回它是否有效：

```
bool validateStudent(Student s);              //函数以 by value 方式接受学生
Student plato;                                //柏拉图,苏格拉底的学生
bool platoIsOK = validateStudent(plato);      //调用函数
```

当上述函数被调用时，发生什么事？

无疑地 Student 的 *copy* 构造函数会被调用，以 plato 为蓝本将 s 初始化。同样明显地，当 validateStudent 退回 s 会被销毁。因此，对此函数而言，参数的传递成本是“一次 Student *copy* 构造函数调用，加上一次 Student 析构函数调用”。

但那还不是整个故事喔。Student 对象内有两个 string 对象，所以每次构造一个 Student 对象也就构造了两个 string 对象。此外 Student 对象继承自 Person 对象，所以每次构造 Student 对象也必须构造出一个 Person 对象。一个 Person 对象又有两个 string 对象在其中，因此每一次 Person 构造动作又需承担两个 string 构造动作。最终结果是，以 *by value* 方式传递一个 Student 对象会导致调用一次 Student *copy* 构造函数、一次 Person *copy* 构造函数、四次 string *copy* 构造函数。当函数内的那个 Student 复件被销毁，每一个构造函数调用动作都需要一个对应的析构函数调用动作。因此，以 *by value* 方式传递一个 Student 对象，总体成本是“六次构造函数和六次析构函数”！

这是正确且值得拥有的行为，毕竟你希望你的所有对象都能够被确实地构造和析构。但尽管如此，如果有什么方法可以回避所有那些构造和析构动作就太好了。有的，就是 *pass by reference-to*-const：

```
bool validateStudent(const Student& s);
```

这种传递方式的效率高得多：没有任何构造函数或析构函数被调用，因为没有任何新对象被创建。修订后的这个参数声明中的 const 是重要的。原先的 validateStudent 以 *by value* 方式接受一个 Student 参数，因此调用者知道他们受到保护，函数内绝不会对传入的 Student 作任何改变；validateStudent 只能够对其复件（副本）做修改。现在 Student 以 *by reference* 方式传递，将它声明为 const 是必要的，因为不这样做的话调用者会忧虑 validateStudent 会不会改变他们传入的那个 Student。

以 *by reference* 方式传递参数也可以避免 *slicing*（对象切割）问题。当一个 derived class 对象以 *by value* 方式传递并被视为一个 base class 对象，base class 的 *copy* 构造函

数会被调用，而“造成此对象的行为像个 derived class 对象”的那些特化性质全被切割掉了，仅仅留下一个 base class 对象。这实在不怎么让人惊讶，因为正是 base class 构造函数建立了它。但这几乎绝不会是你想要的。假设你在一组 classes 上工作，用来实现一个图形窗口系统：

```
class Window {
public:
    ...
    std::string name() const;                    //返回窗口名称
    virtual void display() const;                //显示窗口和其内容
};

class WindowWithScrollBars: public Window {
public:
    ...
    virtual void display() const;
};
```

所有 Window 对象都带有一个名称，你可以通过 name 函数取得它。所有窗口都可显示，你可以通过 display 函数完成它。display 是个 virtual 函数，这意味简易朴素的 base class Window 对象的显示方式和华丽高贵的 WindowWithScrollBars 对象的显示方式不同（见条款 34 和条款 36）。

现在假设你希望写个函数打印窗口名称，然后显示该窗口。下面是错误示范：

```
void printNameAndDisplay(Window w)           //不正确! 参数可能被切割。
{
    std::cout << w.name();
    w.display();
}
```

当你调用上述函数并交给它一个 WindowWithScrollBars 对象，会发生什么事呢？

```
WindowWithScrollBars wwsb;
printNameAndDisplay(wwsb);
```

喔，参数 w 会被构造成为一个 Window 对象；它是 **passed by value**，还记得吗？而造成 wwsb “之所以是个 WindowWithScrollBars 对象”的所有特化信息都会被切除。在 printNameAndDisplay 函数内不论传递过来的对象原本是什么类型，参数 w 就像一个 Window 对象(因为其类型是 Window)。因此在 printNameAndDisplay 内调用 display 调用的总是 Window::display，绝不会是 WindowWithScrollBars::display。

解决切割（slicing）问题的办法，就是以 *by reference-to-*const 的方式传递 w：

```
void printNameAndDisplay(const Window& w)        //很好,参数不会被切割
{
   std::cout << w.name();
   w.display();
}
```

现在，传进来的窗口是什么类型，w 就表现出那种类型。

如果窥视 C++ 编译器的底层，你会发现，*references* 往往以指针实现出来，因此 *pass by reference* 通常意味真正传递的是指针。因此如果你有个对象属于内置类型（例如 int），*pass by value* 往往比 *pass by reference* 的效率高些。对内置类型而言，当你有机会选择采用 *pass-by-value* 或 *pass-by-reference-to-*const 时，选择 *pass-by-value* 并非没有道理。这个忠告也适用于 STL 的迭代器和函数对象，因为习惯上它们都被设计为 *passed by value*。迭代器和函数对象的实践者有责任看看它们是否高效且不受切割问题（slicing problem）的影响。这是“规则之改变取决于你使用哪一部分 C++（见条款 1）”的一个例子。

内置类型都相当小，因此有人认为，所有小型 types 都是 *pass-by-value* 的合格候选人，甚至它们是用户自定义的 class 亦然。这是个不可靠的推论。对象小并不就意味其 *copy* 构造函数不昂贵。许多对象——包括大多数 STL 容器——内含的东西只比一个指针多一些，但复制这种对象却需承担“复制那些指针所指的每一样东西”。那将非常昂贵。

即使小型对象拥有并不昂贵的 *copy* 构造函数，还是可能有效率上的争议。某些编译器对待“内置类型”和“用户自定义类型”的态度截然不同，纵使两者拥有相同的底层表述（underlying representation）。举个例子，某些编译器拒绝把只由一个 double 组成的对象放进缓存器内，却很乐意在一个正规基础上对光秃秃的 doubles 那么做。当这种事发生，你更应该以 *by reference* 方式传递此等对象，因为编译器当然会将指针（*references* 的实现体）放进缓存器内，绝无问题。

“小型的用户自定义类型不必然成为 *pass-by-value* 优良候选人”的另一个理由是，作为一个用户自定义类型，其大小容易有所变化。一个 type 目前虽然小，将来也许会变大，因为其内部实现可能改变。甚至当你改用另一个 C++ 编译器都有可能改变 type 的大小。举个例子，在我下笔此刻，某些标准程序库实现版本中的 string 类型比其他版本大七倍。

一般而言，你可以合理假设“*pass-by-value* 并不昂贵”的唯一对象就是内置类型和 STL 的迭代器和函数对象。至于其他任何东西都请遵守本条款的忠告，尽量以 *pass-by-reference-to-*const 替换 *pass-by-value*。

请记住

- 尽量以 *pass-by-reference-to-*const 替换 *pass-by-value*。前者通常比较高效，并可避免切割问题（slicing problem）。
- 以上规则并不适用于内置类型，以及 STL 的迭代器和函数对象。对它们而言，*pass-by-value* 往往比较适当。

条款 21：必须返回对象时，别妄想返回其 reference

Don't try to return a reference when you must return an object.

一旦程序员领悟了 *pass-by-value*（传值）的效率牵连层面（见条款 20），往往变成十字军战士，一心一意根除 *pass-by-value* 带来的种种邪恶。在坚定追求 *pass-by-reference* 的纯度中，他们一定会犯下一个致命错误：开始传递一些 *references* 指向其实并不存在的对象。这可不是件好事。

考虑一个用以表现有理数（rational numbers）的 class，内含一个函数用来计算两个有理数的乘积：

```
class Rational {
public:
   Rational(int numerator = 0,        //条款 24 说明为什么这个构造函数
            int denominator = 1);     //不声明为 explicit
   ...
private:
   int n, d;                          //分子（numerator）和分母（denominator）
 friend
   const Rational                     //条款 3 说明为什么返回类型是 const
     operator* (const Rational& lhs,
                const Rational& rhs);
};
```

这个版本的 operator* 系以 *by value* 方式返回其计算结果（一个对象）。如果你完全不担心该对象的构造和析构成本，你其实是明显逃避了你的专业责任。若非必要，没有人会想要为这样的对象付出太多代价，问题是需要付出任何代价吗？

唔，如果可以改而传递 reference，就不需付出代价。但是记住，所谓 reference 只是个名称，代表某个既有对象。任何时候看到一个 reference 声明式，你都应该立刻问自己，它的另一个名称是什么？因为它一定是某物的另一个名称。以上述 operator* 为例，如果它返回一个 reference，后者一定指向某个既有的 Rational 对象，内含两个 Rational 对象的乘积。

我们当然不可能期望这样一个（内含乘积的）Rational 对象在调用 operator* 之前就存在。也就是说，如果你有：

```
Rational a(1, 2);                    //a = 1/2
Rational b(3, 5);                    //b = 3/5
Rational c = a * b;                  //c 应该是 3/10
```

期望“原本就存在一个其值为 3/10 的 Rational 对象”并不合理。如果 operator* 要返回一个 reference 指向如此数值，它必须自己创建那个 Rational 对象。

函数创建新对象的途径有二：在 stack 空间或在 heap 空间创建之。如果定义一个 local 变量，就是在 stack 空间创建对象。根据这个策略试写 operator* 如下：

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{
   Rational result(lhs.n * rhs.n, lhs.d * rhs.d);    //警告！糟糕的代码！
   return result;
}
```

你可以拒绝这种做法，因为你的目标是要避免调用构造函数，而 result 却必须像任何对象一样地由构造函数构造起来。更严重的是：这个函数返回一个 reference 指向 result，但 result 是个 local 对象，而 local 对象在函数退出前被销毁了。因此，这个版本的 operator* 并未返回 reference 指向某个 Rational，它返回的 reference 指向一个“从前的”Rational；一个旧时的 Rational；一个曾经被当做 Rational 但如今已经成空、发臭、败坏的残骸，因为它已经被销毁了。任何调用者甚至只是对此函数的返回值做任何一点点运用，都将立刻坠入“无定义行为”的恶地。事情的真相是，任何函数如果返回一个 reference 指向某个 local 对象，都将一败涂地。（如果函数返回指针指向一个 local 对象，也是一样。）

于是，让我们考虑在 heap 内构造一个对象，并返回 reference 指向它。Heap-based 对象由 new 创建，所以你得写一个 heap-based operator* 如下：

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{                                                    //警告！更糟的写法
   Rational* result = new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
   return *result;
}
```

唔，你还是必须付出一个“构造函数调用”代价，因为分配所得的内存将以一个适当的构造函数完成初始化动作。但此外你现在又有了另一个问题：谁该对着被你 new 出来的对象实施 delete？

即使调用者诚实谨慎，并且出于良好意识，他们还是不太能够在这样合情合理的用法下阻止内存泄漏：

```
Rational w, x, y, z;
w = x * y * z;              //与 operator*(operator*(x, y), z) 相同
```

这里，同一个语句内调用了两次 operator*，因而两次使用 new，也就需要两次 delete。但却没有合理的办法让 operator* 使用者进行那些 delete 调用，因为没有合理的办法让他们取得 operator* 返回的 references 背后隐藏的那个指针。这绝对导致资源泄漏。

但或许你注意到了，上述不论 on-the-stack 或 on-the-heap 做法，都因为对 operator* 返回的结果调用构造函数而受惩罚。也许你还记得我们的最初目标是要避免如此的构造函数调用动作。或许你认为你知道有一种办法可以避免任何构造函数被调用。或许你心里出现下面这样的实现代码，此法奠基于“让 operator* 返回的 reference 指向一个被定义于函数内部的 static Rational 对象”：

```
const Rational& operator* (const Rational& lhs,
                           const Rational& rhs)
{                                                    //警告，又一堆烂代码。
   static Rational result;                           //static 对象，此函数将返回
                                                     //  其 reference。
   result = ... ;                                    //将 lhs 乘以 rhs，并将结果
                                                     //  置于 result 内。
   return result;
}
```

就像所有用上 static 对象的设计一样，这一个也立刻造成我们对多线程安全性的疑虑。不过那还只是它显而易见的弱点。如果想看看更深层的瑕疵，考虑以下面这些完全合理的客户代码：

```
bool operator==(const Rational& lhs,        //一个针对 Rationals
                const Rational& rhs); //  而写的 operator==
Rational a, b, c, d;
...
if ((a * b) == (c * d)) {
   当乘积相等时，做适当的相应动作;
} else {
   当乘积不等时，做适当的相应动作;
}
```

猜想怎么着？表达式((a*b) == (c*d))总是被核算为 true，不论 a, b, c 和 d 的值是什么！

一旦将代码重新写为等价的函数形式，很容易就可以了解出了什么意外：

```
if (operator==(operator*(a, b), operator*(c, d)))
```

注意，在 operator== 被调用前，已有两个 operator* 调用式起作用，每一个都返回 reference 指向 operator* 内部定义的 static Rational 对象。因此 operator== 被要求将“operator* 内的 static Rational 对象值”拿来和“operator* 内的 static Rational 对象值”比较，如果比较结果不相等，那才奇怪呢。（译注：这里我补充说明：两次 operator* 调用的确各自改变了 static Rational 对象值，但由于它们返回的都是 reference，因此调用端看到的永远是 static Rational 对象的“现值”。）

这应该足够说服你，欲令诸如 operator* 这样的函数返回 reference，只是浪费时间而已，但现在或许又有些人这样想：“唔，如果一个 static 不够，或许一个 static array 可以得分……”。

我不打算再次写出示例来驳斥这个想法以彰显自己多么厉害，但我可以简单描述为什么你该为了提出这个念头而脸红。首先你必须选择 array 大小 n。如果 n 太小，你可能会耗尽“用以存储函数返回值”的空间，那么情况就回到了我们刚才讨论过的单一 static 设计。但如果 n 太大，会因此降低程序效率，因为 array 内的每一个对象都会在函数第一次被调用时构造完成。那么将消耗 n 个构造函数和 n 个析构函数——即使我们所讨论的函数只被调用一次。如果所谓“最优化”是改善软件效率的过程，我们现在所谈的这些应该称为“恶劣化”。最后，想一想如何将你需要的值放进 array 内，

而那么做的成本又是多少。在对象之间搬移数值的最直接办法是通过赋值(*assignment*)操作，但赋值的成本几何？对许多 types 而言它相当于调用一个析构函数（用以销毁旧值）加上一个构造函数（用以复制新值）。但你的目标是避免构造和析构成本耶！面对现实吧，这个做法不会成功的。就算以 vector 替换 array 也不会让情况更好些。

一个“必须返回新对象”的函数的正确写法是：就让那个函数返回一个新对象呗。对 Rational 的 operator* 而言意味以下写法（或其他本质上等价的代码）：

```
inline const Rational operator * (const Rational& lhs, const Rational& rhs)
{
   return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

当然，你需得承受 operator* 返回值的构造成本和析构成本，然而长远来看那只是为了获得正确行为而付出的一个小小代价。但万一账单很恐怖，你承受不起，别忘了 C++ 和所有编程语言一样，允许编译器实现者施行最优化，用以改善产出码的效率却不改变其可观察的行为。因此某些情况下 operator* 返回值的构造和析构可被安全地消除。如果编译器运用这一事实（它们也往往如此），你的程序将继续保持它们该有的行为，而执行起来又比预期的更快。

我把以上的讨论浓缩总结为：当你必须在“返回一个 reference 和返回一个 object”之间抉择时，你的工作就是挑出行为正确的那个。就让编译器厂商为“尽可能降低成本”鞠躬尽瘁吧，你可以享受你的生活。

请记住

- 绝不要返回 pointer 或 reference 指向一个 local stack 对象，或返回 reference 指向一个 heap-allocated 对象，或返回 pointer 或 reference 指向一个 local static 对象而有可能同时需要多个这样的对象。条款 4 已经为“在单线程环境中合理返回 reference 指向一个 local static 对象”提供了一份设计实例。

条款 22：将成员变量声明为 private

Declare data members private.

OK，下面是我的规划。首先带你看看为什么成员变量不该是 public，然后让你看看所有反对 public 成员变量的论点同样适用于 protected 成员变量。最后导出一个结论：成员变量应该是 private。获得这个结论后，本条款也就大功告成了。

好，现在看看 public 成员变量。为什么不采用它呢？

让我们从语法一致性开始（同时请见条款 18）。如果成员变量不是 public，客户唯一能够访问对象的办法就是通过成员函数。如果 public 接口内的每样东西都是函数，客户就不需要在打算访问 class 成员时迷惑地试着记住是否该使用小括号（圆括号）。他们只要做就是了，因为每样东西都是函数。就生命而言，这至少可以省下许多搔首弄耳的时间。

或许你不认为一致性的理由足以令人信服，那么这个事实如何：使用函数可以让你对成员变量的处理有更精确的控制。如果你令成员变量为 public，每个人都可以读写它，但如果你以函数取得或设定其值，你就可以实现出“不准访问”、“只读访问”以及“读写访问”。见鬼了，你甚至可以实现“惟写访问”，如果你想要的话：

```
class AccessLevels {
public:
    ...
    int getReadOnly() const { return readOnly; }
    void setReadWrite(int value) { readWrite = value; }
    int getReadWrite( ) const { return readWrite; }
    void setWriteOnly(int value) { writeOnly = value; }
private:
    int noAccess;        //对此 int 无任何访问动作
    int readOnly;        //对此 int 做只读访问（read-only access）
    int readWrite;       //对此 int 做读写访问（read-write access）
    int writeOnly;       //对此 int 做惟写访问（write-only access）
};
```

如此细微地划分访问控制颇有必要，因为许多成员变量应该被隐藏起来。每个成员变量都需要一个 getter 函数和 setter 函数毕竟罕见。

还是不够说服你？是端出大口径武器的时候了：封装啦。如果你通过函数访问成员变量，日后可改以某个计算替换这个成员变量，而 class 客户一点也不会知道 class 的内部实现已经起了变化。

举个例子，假设你正在写一个自动测速程序，当汽车通过，其速度便被计算并填入一个速度收集器内：

```
class SpeedDataCollection {
    ...
public:
    void addValue(int speed);                    //添加一笔新数据
    double averageSoFar() const;                 //返回平均速度
    ...
};
```

现在让我们考虑成员函数 averageSoFar。做法之一是在 class 内设计一个成员变量，记录至今以来所有速度的平均值。当 averageSoFar 被调用，只需返回那个成员变量就好。另一个做法是令 averageSoFar 每次被调用时重新计算平均值，此函数有权力调取收集器内的每一笔速度值。

上述第一种做法（随时保持平均值）会使每一个 SpeedDataCollection 对象变大，因为你必须为用来存放目前平均值、累积总量、数据点数的每一个成员变量分配空间。然而 averageSoFar 却可因此而十分高效；它可以只是一个返回目前平均值的 inline 函数（见条款 30）。相反地，“被询问才计算平均值”会使得 averageSoFar 执行较慢，但每一个 SpeedDataCollection 对象比较小。

谁说得出哪一个比较好？在一部内存吃紧的机器上（例如一台嵌入式路边侦测装置），或是在一个并不常常需要平均值的应用程序中，“每次需要时才计算”或许是比较好的解法。但在一个频繁需要平均值的应用程序中，如果反应速度非常重要，内存不是重点，这时候“随时维持一个当下平均值”往往更好一些。重点是，由于通过成员函数来访问平均值（也就是封装了它），你得以替换不同的实现方式（以及其他你可能想到的东西），客户最多只需重新编译。（如果遵循条款 31 所描述的技术，你甚至可以消除重新编译的不便性。）

将成员变量隐藏在函数接口的背后，可以为“所有可能的实现”提供弹性。例如这可使得成员变量被读或被写时轻松通知其他对象、可以验证 class 的约束条件以及函

数的前提和事后状态、可以在多线程环境中执行同步控制……等等。来自 Delphi 和 C# 阵营的 C++ 程序员应该知道，这般能力等价于其他语言中的 "properties"，尽管额外需要一组小括号。

封装的重要性比你最初见到它时还重要。如果你对客户隐藏成员变量（也就是封装它们），你可以确保 class 的约束条件总是会获得维护，因为只有成员函数可以影响它们。犹有进者，你保留了日后变更实现的权利。如果你不隐藏它们，你很快会发现，即使拥有 class 原始码，改变任何 public 事物的能力还是极端受到束缚，因为那会破坏太多客户码。Public 意味不封装，而几乎可以说，不封装意味不可改变，特别是对被广泛使用的 classes 而言。被广泛使用的 classes 是最需要封装的一个族群，因为它们最能够从“改采用一个较佳实现版本”中获益。

protected 成员变量的论点十分类似。实际上它和 public 成员变量的论点相同，虽然或许最初看起来不是一回事。“语法一致性”和“细微划分之访问控制”等理由显然也适用于 protected 数据，就像对 public 一样适用。但封装呢？protected 成员变量的封装性是不是高过 public 成员变量？答案令人惊讶：并非如此。

条款 23 会告诉你，某些东西的封装性与“当其内容改变时可能造成的代码破坏量”成反比。因此，成员变量的封装性与“成员变量的内容改变时所破坏的代码数量”成反比。所谓改变，也许是从 class 中移除它（或许这有利于计算，就像上述的 `averageSoFar`）。

假设我们有一个 public 成员变量，而我们最终取消了它。多少代码可能会被破坏呢？唔，所有使用它的客户码都会被破坏，而那是一个不可知的大量。因此 public 成员变量完全没有封装性。假设我们有一个 protected 成员变量，而我们最终取消了它，有多少代码被破坏？唔，所有使用它的 derived classes 都会被破坏，那往往也是个不可知的大量。因此，protected 成员变量就像 public 成员变量一样缺乏封装性，因为在这两种情况下，如果成员变量被改变，都会有不可预知的大量代码受到破坏。虽然这个结论有点违反直观，但经验丰富的程序库作者会告诉你，它是真的。一旦你将一个成员变量声明为 public 或 protected 而客户开始使用它，就很难改变那个成员变量所涉及的一切。太多代码需要重写、重新测试、重新编写文档、重新编译。从封装的角度观

之，其实只有两种访问权限：private（提供封装）和其他（不提供封装）。

请记住

- 切记将成员变量声明为 private。这可赋予客户访问数据的一致性、可细微划分访问控制、允诺约束条件获得保证，并提供 class 作者以充分的实现弹性。
- protected 并不比 public 更具封装性。

条款 23：宁以 non-member、non-friend 替换 member 函数

Prefer non-member non-friend functions to member functions.

想象有个 class 用来表示网页浏览器。这样的 class 可能提供的众多函数中，有一些用来清除下载元素高速缓存区（cache of downloaded elements）、清除访问过的 URLs 的历史记录（history of visited URLs）、以及移除系统中的所有 cookies：

```
class WebBrowser {
public:
   ...
   void clearCache( );
   void clearHistory( );
   void removeCookies( );
   ...
};
```

许多用户会想一整个执行所有这些动作，因此 WebBrowser 也提供这样一个函数：

```
class WebBrowser {
public:
   ...
   void clearEverything( );          //调用 clearCache, clearHistory,
                                     //和 removeCookies
   ...
};
```

当然，这一机能也可由一个 non-member 函数调用适当的 member 函数而提供出来：

```
void clearBrowser(WebBrowser& wb)
{
   wb.clearCache();
   wb.clearHistory();
   wb.removeCookies( );
}
```

那么，哪一个比较好呢？是 member 函数 clearEverything 还是 non-member 函数 clearBrowser?

面向对象守则要求，数据以及操作数据的那些函数应该被捆绑在一块，这意味它建议 member 函数是较好的选择。不幸的是这个建议不正确。这是基于对面向对象真实意义的一个误解。面向对象守则要求数据应该尽可能被封装，然而与直观相反地，member 函数 clearEverything 带来的封装性比 non-member 函数 clearBrowser 低。此外，提供 non-member 函数可允许对 WebBrowser 相关机能有较大的包裹弹性（packaging flexibility），而那最终导致较低的编译相依度，增加 WebBrowser 的可延伸性。因此在许多方面 non-member 做法比 member 做法好。重要的是，我们必须了解其原因。

让我们从封装开始讨论。如果某些东西被封装，它就不再可见。愈多东西被封装，愈少人可以看到它。而愈少人看到它，我们就有愈大的弹性去变化它，因为我们的改变仅仅直接影响看到改变的那些人事物。因此，愈多东西被封装，我们改变那些东西的能力也就愈大。这就是我们首先推崇封装的原因：它使我们能够改变事物而只影响有限客户。

现在考虑对象内的数据。愈少代码可以看到数据（也就是访问它），愈多的数据可被封装，而我们也就愈能自由地改变对象数据，例如改变成员变量的数量、类型等等。如何量测“有多少代码可以看到某一块数据”呢？我们计算能够访问该数据的函数数量，作为一种粗糙的量测。愈多函数可访问它，数据的封装性就愈低。

条款 22 曾说过，成员变量应该是 private，因为如果它们不是，就有无限量的函数可以访问它们，它们也就毫无封装性。能够访问 private 成员变量的函数只有 class 的 member 函数加上 friend 函数而已。如果要你在一个 member 函数（它不只可以访问 class 内的 private 数据，也可以取用 private 函数、enums、typedefs 等等）和一个 non-member, non-friend 函数（它无法访问上述任何东西）之间做抉择，而且两者提供相同机能，那么，导致较大封装性的是 non-member non-friend 函数，因为它并不增加“能够访问

class 内之 private 成分"的函数数量。这就解释了为什么 clearBrowser(一个 non-member non-friend 函数）比 clearEverything（一个 member 函数）更受欢迎的原因：它导致 WebBrowser class 有较大的封装性。

在这一点上有两件事情值得注意。第一，这个论述只适用于 non-member non-friend 函数。friends 函数对 class private 成员的访问权力和 member 函数相同，因此两者对封装的冲击力道也相同。从封装的角度看，这里的选择关键并不在 member 和 non-member 函数之间，而是在 member 和 non-member non-friend 函数之间。（当然，封装并非唯一考虑。条款 24 解释当我们考虑隐式类型转换，应该在 member 和 non-member 函数之间抉择。）

第二件值得注意的事情是，只因在意封装性而让函数"成为 class 的 non-member"，并不意味它"不可以是另一个 class 的 member"。这对那些习惯于"所有函数都必须定义于 class 内"的语言（如 Eiffel, Java, C#）的程序员而言，可能是个温暖的慰藉。例如我们可以令 clearBrowser 成为某工具类（utility class）的一个 static member 函数。只要它不是 WebBrowser 的一部分（或成为其 friend），就不会影响 WebBrowser 的 private 成员封装性。

在 C++，比较自然的做法是让 clearBrowser 成为一个 non-member 函数并且位于 WebBrowser 所在的同一个 namespace（命名空间）内：

```
namespace WebBrowserStuff {
    class WebBrowser { ... };
    void clearBrowser(WebBrowser& wb);
    ...
}
```

然而这不只是为了看起来自然而已。要知道，namespace 和 classes 不同，前者可跨越多个源码文件而后者不能。这很重要，因为像 clearBrowser 这样的函数是个"提供便利的函数"，如果它既不是 members 也不是 friends，就没有对 WebBrowser 的特殊访问权力，也就不能提供"WebBrowser 客户无法以其他方式取得"的机能。举个例子，如果 clearBrowser 不存在，客户端就只好自行调用 clearCache, clearHistory 和 removeCookies。

一个像 WebBrowser 这样的 class 可能拥有大量便利函数，某些与书签（bookmarks）有关，某些与打印有关，还有一些与 cookie 的管理有关…… 通常大多数客户只对其中某些感兴趣。没道理一个只对书签相关便利函数感兴趣的客户却与……呃……例如一

个 cookie 相关便利函数发生编译相依关系。分离它们的最直接做法就是将书签相关便利函数声明于一个头文件，将 cookie 相关便利函数声明于另一个头文件，再将打印相关便利函数声明于第三个头文件，依此类推：

```
//头文件 "webbrowser.h" — 这个头文件针对 class WebBrowser 自身
//  及 WebBrowser 核心机能。
namespace WebBrowserStuff {
class WebBrowser { ... };
   ...              //核心机能，例如几乎所有客户都需要的
                    //  non-member 函数。
}
//头文件 "webbrowserbookmarks.h"
namespace WebBrowserStuff {
   ...              //与书签相关的便利函数
}
//头文件 "webbrowsercookies.h"
namespace WebBrowserStuff {
   ...              //与 cookie 相关的便利函数
}
...
```

注意，这正是 C++ 标准程序库的组织方式。标准程序库并不是拥有单一、整体、庞大的 <C++StandardLibrary> 头文件并在其中内含 std 命名空间内的每一样东西，而是有数十个头文件（<vector>, <algorithm>, <memory>等等），每个头文件声明 std 的某些机能。如果客户只想使用 vector 相关机能，他不需要 #include <memory>；如果客户不想使用 list，也不需要 #include <list>。这允许客户只对他们所用的那一小部分系统形成编译相依（见条款 31，其中讨论降低编译依存性的其他做法）。以此种方式切割机能并不适用于 class 成员函数，因为一个 class 必须整体定义，不能被分割为片片段段。

将所有便利函数放在多个头文件内但隶属同一个命名空间，意味客户可以轻松扩展这一组便利函数。他们需要做的就是添加更多 non-member non-friend 函数到此命名空间内。举个例子，如果某个 WebBrowser 客户决定写些与影像下载相关的便利函数，他只需要在 WebBrowserStuff 命名空间内建立一个头文件，内含那些函数的声明即可。新函数就像其他旧有的便利函数那样可用且整合为一体。这是 class 无法提供的另一

个性质，因为 class 定义式对客户而言是不能扩展的。当然啦，客户可以派生出新的 classes，但 derived classes 无法访问 base class 中被封装的（即 private）成员，于是如此的 "扩展机能" 拥有的只是次级身份。此外一如条款 7 所说，并非所有 classes 都被设计用来作为 base classes。

请记住

- 宁可拿 non-member non-friend 函数替换 member 函数。这样做可以增加封装性、包裹弹性（packaging flexibility）和机能扩充性。

条款 24：若所有参数皆需类型转换，请为此采用 non-member 函数

Declare non-member functions when type conversions should apply to all parameters.

我在导读中提过，令 classes 支持隐式类型转换通常是个糟糕的主意。当然这条规则有其例外，最常见的例外是在建立数值类型时。假设你设计一个 class 用来表现有理数，允许整数“隐式转换”为有理数似乎颇为合理。的确，它并不比 C++ 内置从 int 至 double 的转换来得不合理，而还比 C++ 内置从 double 至 int 的转换来得合理些。假设你这样开始你的 Rational class：

```
class Rational {
public:
   Rational(int numerator = 0,          //构造函数刻意不为 explicit;
            int denominator = 1);       //允许 int-to-Rational 隐式转换。
   int numerator() const;               //分子（numerator）和分母（denominator）
   int denominator() const;             //的访问函数（accessors）—见条款 22。
private:
   ...
};
```

你想支持算术运算诸如加法、乘法等等，但你不确定是否该由 member 函数、non-member 函数，或可能的话由 non-member friend 函数来实现它们。你的直觉告诉你，当你犹豫就该保持面向对象精神。你知道有理数相乘和 Rational class 有关，因此很自然地似乎该在 Rational class 内为有理数实现 operator*。条款 23 曾经反直觉地主张，将函数放进相关 class 内有时会与面向对象守则发生矛盾，但让我们先把那放在一

旁，先研究一下将 operator* 写成 Rational 成员函数的写法：

```
class Rational {
public:
   ...
   const Rational operator* (const Rational& rhs) const;
};
```

（如果你不确定为什么这个函数被声明为此种形式，也就是为什么它返回一个 const ***by-value*** 结果但接受一个 ***reference-to***-const 实参，请参考条款 3, 20 和 21。）

这个设计使你能够将两个有理数以最轻松自在的方式相乘：

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);
Rational result = oneHalf * oneEighth;      //很好
result = result * oneEighth;                //很好
```

但你还不满足。你希望支持混合式运算，也就是拿 Rationals 和……嗯……例如 ints 相乘。毕竟很少有什么东西会比两个数值相乘更自然的了——即使是两个不同类型的数值。

然而当你尝试混合式算术，你发现只有一半行得通：

```
result = oneHalf * 2;              //很好
result = 2 * oneHalf;              //错误!
```

这不是好兆头。乘法应该满足交换律，不是吗？

当你以对应的函数形式重写上述两个式子，问题所在便一目了然了：

```
result = oneHalf.operator*(2);          //很好
result = 2.operator*(oneHalf);          //错误!
```

是的，oneHalf 是一个内含 operator* 函数的 class 的对象，所以编译器调用该函数。然而整数 2 并没有相应的 class，也就没有 operator* 成员函数。编译器也会尝试寻找可被以下这般调用的 non-member operator*（也就是在命名空间内或在 global 作用域内）：

```
result = operator*(2, oneHalf);         //错误!
```

但本例并不存在这样一个接受 int 和 Rational 作为参数的 non-member operator*，因此查找失败。

再次看看先前成功的那个调用。注意其第二参数是整数 2，但 Rational::operator*需要的实参却是个 Rational 对象。这里发生了什么事？为什么 2 在这里可被接受，在另一个调用中却不被接受？

因为这里发生了所谓隐式类型转换（implicit type conversion）。编译器知道你正在传递一个 int，而函数需要的是 Rational；但它也知道只要调用 Rational 构造函数并赋予你所提供的 int，就可以变出一个适当的 Rational 来。于是它就那样做了。换句话说此一调用动作在编译器眼中有点像这样：

```
const Rational temp(2);          //根据 2 建立一个暂时性的 Rational 对象。
result = oneHalf * temp;         //等同于 oneHalf.operator*(temp);
```

当然，只因为涉及 non-explicit 构造函数，编译器才会这样做。如果 Rational 构造函数是 explicit，以下语句没有一个可通过编译：

```
result = oneHalf * 2;            //错误！（在 explicit 构造函数的情况下）
                                 //无法将 2 转换为一个 Rational。
result = 2 * oneHalf;            //一样的错误，一样的问题。
```

这就很难让 Rational class 支持混合式算术运算了，不过至少上述两个句子的行为从此一致☺。

然而你的目标不仅在一致性，也要支持混合式算术运算，也就是希望有个设计能让以上语句通过编译。这把我们带回到上述两个语句，为什么即使 Rational 构造函数不是 explicit，仍然只有一个可通过编译，另一个不可以：

```
result = oneHalf * 2;            //没问题（在 non-explicit 构造函数的情况下）
result = 2 * oneHalf;            //错误！（甚至在 non-explicit 构造函数的情况下）
```

结论是，只有当参数被列于参数列（*parameter list*）内，这个参数才是隐式类型转换的合格参与者。地位相当于“被调用之成员函数所隶属的那个对象”——即 this 对象——的那个隐喻参数，绝不是隐式转换的合格参与者。这就是为什么上述第一次调用可通过编译，第二次调用则否，因为第一次调用伴随一个放在参数列内的参数，第二次调用则否。

然而你一定也会想要支持混合式算术运算。可行之道终于拨云见日：让 operator* 成为一个 non-member 函数，俾允许编译器在每一个实参身上执行隐式类型转换：

```
class Rational {
   ...                                                    //不包括 operator*
};
const Rational operator*(const Rational& lhs,            //现在成了一个
                         const Rational& rhs)             //non-member 函数
{
   return Rational(lhs.numerator() * rhs.numerator( ),
                   lhs.denominator() * rhs.denominator());
}
Rational oneFourth(1, 4);
Rational result;
result = oneFourth * 2;                                   //没问题
result = 2 * oneFourth;                                   //万岁，通过编译了！
```

这当然是个快乐的结局，不过还有一点必须操心：`operator*` 是否应该成为 `Rational` class 的一个 friend 函数呢？

就本例而言答案是否定的，因为 `operator*` 可以完全藉由 `Rational` 的 public 接口完成任务，上面代码已表明此种做法。这导出一个重要的观察：member 函数的反面是 non-member 函数，不是 friend 函数。太多 C++ 程序员假设，如果一个“与某 class 相关”的函数不该成为一个 member（也许由于其所有实参都需要类型转换，例如先前的 `Rational` 的 `operator*` 函数），就该是个 friend。本例表明这样的理由过于牵强。无论何时如果你可以避免 friend 函数就该避免，因为就像真实世界一样，朋友带来的麻烦往往多过其价值。当然有时候 friend 有其正当性，但这个事实依然存在：不能够只因函数不该成为 member，就自动让它成为 friend。

本条款内含真理，但却不是全部的真理。当你从 Object-Oriented C++ 跨进 Template C++（见条款 1）并让 `Rational` 成为一个 class *template* 而非 class，又有一些需要考虑的新争议、新解法、以及一些令人惊讶的设计牵连。这些争议、解法和设计牵连形成了条款 46。

请记住

- 如果你需要为某个函数的所有参数（包括被 `this` 指针所指的那个隐喻参数）进行类型转换，那么这个函数必须是个 non-member。

条款 25：考虑写出一个不抛异常的 swap 函数

Consider support for a non-throwing swap.

swap 是个有趣的函数。原本它只是 STL 的一部分，而后成为异常安全性编程（exception-safe programming，见条款 29）的脊柱，以及用来处理自我赋值可能性（见条款 11）的一个常见机制。由于 swap 如此有用，适当的实现很重要。然而在非凡的重要性之外它也带来了非凡的复杂度。本条款探讨这些复杂度及因应之道。

所谓 *swap*（置换）两对象值，意思是将两对象的值彼此赋予对方。缺省情况下 *swap* 动作可由标准程序库提供的 swap 算法完成。其典型实现完全如你所预期：

```
namespace std {
    template<typename T>                    //std::swap 的典型实现;
    void swap( T& a, T& b)                  //置换 a 和 b 的值.
    {
        T temp(a);
        a = b;
        b = temp;
    }
}
```

只要类型 T 支持 *copying*（通过 *copy* 构造函数和 *copy assignment* 操作符完成），缺省的 swap 实现代码就会帮你置换类型为 T 的对象，你不需要为此另外再做任何工作。

这缺省的 swap 实现版本十分平淡，无法刺激你的肾上腺。它涉及三个对象的复制：a 复制到 temp，b 复制到 a，以及 temp 复制到 b。但是对某些类型而言，这些复制动作无一必要；对它们而言 swap 缺省行为等于是把高速铁路铺设在慢速小巷弄内。

其中最主要的就是“以指针指向一个对象，内含真正数据”那种类型。这种设计的常见表现形式是所谓“pimpl 手法”（pimpl 是 "pointer to implementation" 的缩写，见条款 31）。如果以这种手法设计 Widget class，看起来会像这样：

```
class WidgetImpl {                          //针对 Widget 数据而设计的 class;
public:                                     //细节不重要。
    ...
private:
    int a, b, c;                            //可能有许多数据，
    std::vector<double> v;                  //意味复制时间很长。
    ...
};
```

```
class Widget {                                    //这个 class 使用 pimpl 手法
public:
   Widget(const Widget& rhs);
   Widget& operator=(const Widget& rhs)           //复制 Widget 时，令它复制其
   {                                              // WidgetImpl 对象。
        ...                                       //关于 operator=的一般性实现
        *pImpl = *(rhs.pImpl);                    // 细节，见条款 10,11 和 12。
        ...
   }
   ...
private:
   WidgetImpl* pImpl;                             //指针，所指对象内含
};                                                // Widget 数据。
```

一旦要置换两个 Widget 对象值，我们唯一需要做的就是置换其 pImpl 指针，但缺省的 swap 算法不知道这一点。它不只复制三个 Widgets，还复制三个 WidgetImpl 对象。非常缺乏效率！一点也不令人兴奋。

我们希望能够告诉 std::swap：当 Widgets 被置换时真正该做的是置换其内部的 pImpl 指针。确切实践这个思路的一个做法是：将 std::swap 针对 Widget 特化。下面是基本构想，但目前这个形式无法通过编译：

```
namespace std {
   template<>                                     //这是 std::swap 针对
   void swap<Widget>( Widget& a,                  // “T 是 Widget”的特化版本。
                      Widget& b )                 //目前还不能通过编译。
   {
        swap(a.pImpl, b.pImpl);                   //置换 Widgets 时只要置换它们的
   }                                              // pImpl 指针就好。
}
```

这个函数一开始的 "template<>" 表示它是 std::swap 的一个全特化（*total template specialization*）版本，函数名称之后的 "<Widget>" 表示这一特化版本系针对“T 是 Widget”而设计。换句话说当一般性的 swap template 施行于 Widgets 身上便会启用这个版本。通常我们不能够（不被允许）改变 std 命名空间内的任何东西，但可以（被允许）为标准 templates（如 swap）制造特化版本，使它专属于我们自己的 classes（例如 Widget）。以上作为正是如此。

但是一如稍早我说，这个函数无法通过编译。因为它企图访问 a 和 b 内的 pImpl 指针，而那却是 private。我们可以将这个特化版本声明为 friend，但和以往的规矩不太

一样：我们令 Widget 声明一个名为 swap 的 public 成员函数做真正的置换工作，然后将 std::swap 特化，令它调用该成员函数：

```
class Widget {                              //与前同，唯一差别是增加 swap 函数
public:
    ...
    void swap(Widget& other)
    {
        using std::swap;                    //这个声明之所以必要，稍后解释。
        swap(pImpl, other.pImpl);           //若要置换 Widgets 就置换其 pImpl 指针。
    }
    ...
};

namespace std {
    template<>                              //修订后的 std::swap 特化版本
    void swap<Widget>( Widget& a,
                       Widget& b )
    {
        a.swap(b);                          //若要置换 Widgets，调用其
    }                                       //swap 成员函数。
}
```

这种做法不只能够通过编译，还与 STL 容器有一致性，因为所有 STL 容器也都提供有 public swap 成员函数和 std::swap 特化版本（用以调用前者）。

然而假设 Widget 和 WidgetImpl 都是 class templates 而非 classes，也许我们可以试试将 WidgetImpl 内的数据类型加以参数化：

```
template<typename T>
class WidgetImpl { ... };

template<typename T>
class Widget { ... };
```

在 Widget 内（以及 WidgetImpl 内，如果需要的话）放个 swap 成员函数就像以往一样简单，但我们却在特化 std::swap 时遇上乱流。我们想写成这样：

```
namespace std {
    template<typename T>
    void swap< Widget<T> >(Widget<T>& a,          //错误！不合法！
                           Widget<T>& b)
    { a.swap(b); }
}
```

看起来合情合理，却不合法。是这样的，我们企图偏特化（partially specialize）一个 function template（std::swap），但 C++ 只允许对 class templates 偏特化，在 function templates 身上偏特化是行不通的。这段代码不该通过编译（虽然有些编译器错误地接受了它）。

当你打算偏特化一个 function template 时，惯常做法是简单地为它添加一个重载版本，像这样：

```
namespace std {
    template<typename T>                    //std::swap的一个重载版本
    void swap(Widget<T>& a,                 //（注意 "swap" 之后没有 "<...>"）
              Widget<T>& b)                 //稍后我会告诉你，这也不合法。
    { a.swap(b); }
}
```

一般而言，重载 function templates 没有问题，但 std 是个特殊的命名空间，其管理规则也比较特殊。客户可以全特化 std 内的 templates，但不可以添加新的 templates（或 classes 或 functions 或其他任何东西）到 std 里头。std 的内容完全由 C++ 标准委员会决定，标准委员会禁止我们膨胀那些已经声明好的东西。啊呀，所谓“禁止”可能会使你沮丧，其实跨越红线的程序几乎仍可编译和执行，但它们的行为没有明确定义。如果你希望你的软件有可预期的行为，请不要添加任何新东西到 std 里头。

那该如何是好？毕竟我们总是需要一个办法让其他人调用 swap 时能够取得我们提供的较高效的 template 特定版本。答案很简单，我们还是声明一个 non-member swap 让它调用 member swap，但不再将那个 non-member swap 声明为 std::swap 的特化版本或重载版本。为求简化起见，假设 Widget 的所有相关机能都被置于命名空间 WidgetStuff 内，整个结果看起来便像这样：

```
namespace WidgetStuff {
    ...                                     //模板化的 WidgetImpl 等等。
    template<typename T>                    //同前，内含 swap 成员函数。
    class Widget { ... };

    ...
    template<typename T>                    //non-member swap 函数；
    void swap(Widget<T>& a,                 //这里并不属于 std 命名空间。
              Widget<T>& b)
    {
        a.swap(b);
    }
}
```

现在，任何地点的任何代码如果打算置换两个 Widget 对象，因而调用 swap，C++ 的名称查找法则（name lookup rules；更具体地说是所谓 *argument-dependent lookup* 或 *Koenig lookup* 法则）会找到 WidgetStuff 内的 Widget 专属版本。那正是我们所要的。

这个做法对 classes 和 class templates 都行得通，所以似乎我们应该在任何时候都使用它。不幸的是有一个理由使你应该为 classes 特化 std::swap（很快我会描述它），所以如果你想让你的“class 专属版” swap 在尽可能多的语境下被调用，你需得同时在该 class 所在命名空间内写一个 non-member 版本以及一个 std::swap 特化版本。

顺带一提，如果没有像上面那样额外使用某个命名空间，上述每件事情仍然适用（也就是说你还是需要一个 non-member swap 用来调用 member swap）。但，何必在 global 命名空间内塞满各式各样的 class, template, function, enum, enumerant 以及 typedef 名称呢？难道你对所谓“得体与适度”失去判断力了吗？

目前为止我所写的每一样东西都和 swap 编写者有关。换位思考，从客户观点看看事情也有必要。假设你正在写一个 function template，其内需要置换两个对象值：

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
    ...
    swap(obj1, obj2);
    ...
}
```

应该调用哪个 swap？是 std 既有的那个一般化版本？还是某个可能存在的特化版本？抑或是一个可能存在的 T 专属版本而且可能栖身于某个命名空间（但当然不可以是 std）内？你希望的应该是调用 T 专属版本，并在该版本不存在的情况下调用 std 内的一般化版本。下面是你希望发生的事：

```
template<typename T>
void doSomething(T& obj1, T& obj2)
{
    using std::swap;     //令 std::swap 在此函数内可用
    ...
    swap(obj1, obj2);    //为 T 型对象调用最佳 swap 版本
    ...
}
```

一旦编译器看到对 swap 的调用，它们便查找适当的 swap 并调用之。C++ 的名称查找法则（name lookup rules）确保将找到 global 作用域或 T 所在之命名空间内的任何 T 专属的 swap。如果 T 是 Widget 并位于命名空间 WidgetStuff 内，编译器会使用“实参取决之查找规则”（argument-dependent lookup）找出 WidgetStuff 内的 swap。如果没有 T 专属之 swap 存在，编译器就使用 std 内的 swap，这得感谢 using 声明式让 std::swap 在函数内曝光。然而即便如此编译器还是比较喜欢 std::swap 的 T 专属特化版，而非一般化的那个 template，所以如果你已针对 T 将 std::swap 特化，特化版会被编译器挑中。

因此，令适当的 swap 被调用是很容易的。需要小心的是，别为这一调用添加额外修饰符，因为那会影响 C++ 挑选适当函数。假设你以这种方式调用 swap：

```
std::swap(obj1, obj2);          //这是错误的 swap 调用方式
```

这便强迫编译器只认 std 内的 swap（包括其任何 template 特化），因而不再可能调用一个定义于它处的较适当 T 专属版本。啊呀，某些迷途程序员的确以此方式修饰 swap 调用式，而那正是“你的 classes 对 std::swap 进行全特化”的重要原因：它使得类型专属之 swap 实现版本也可被这些“迷途代码”所用（这样的代码出现在某些标准程序库实现版中，如果你有兴趣不妨帮助这些代码尽可能高效运作）。

此刻，我们已经讨论过 default swap、member swaps、non-member swaps、std::swap 特化版本、以及对 swap 的调用，现在让我把整个形势做个总结。

首先，如果 swap 的缺省实现码对你的 class 或 class template 提供可接受的效率，你不需要额外做任何事。任何尝试置换（swap）那种对象的人都会取得缺省版本，而那将有良好的运作。

其次，如果 swap 缺省实现版的效率不足（那几乎总是意味你的 class 或 template 使用了某种 pimpl 手法），试着做以下事情：

1. 提供一个 public swap 成员函数，让它高效地置换你的类型的两个对象值。稍后我将解释，这个函数绝不该抛出异常。
2. 在你的 class 或 template 所在的命名空间内提供一个 non-member swap，并令它调用上述 swap 成员函数。

3. 如果你正编写一个 class（而非 class template），为你的 class 特化 std::swap。并令它调用你的 swap 成员函数。

最后，如果你调用 swap，请确定包含一个 using 声明式，以便让 std::swap 在你的函数内曝光可见，然后不加任何 namespace 修饰符，赤裸裸地调用 swap。

唯一还未明确的是我的劝告：成员版 swap 绝不可抛出异常。那是因为 swap 的一个最好的应用是帮助 classes（和 class templates）提供强烈的异常安全性（exception-safety）保障。条款 29 对此主题提供了所有细节，但此技术基于一个假设：成员版的 swap 绝不抛出异常。这一约束只施行于成员版！不可施行于非成员版，因为 swap 缺省版本是以 *copy* 构造函数和 *copy assignment* 操作符为基础，而一般情况下两者都允许抛出异常。因此当你写下一个自定版本的 swap，往往提供的不只是高效置换对象值的办法，而且不抛出异常。一般而言这两个 swap 特性是连在一起的，因为高效率的 swaps 几乎总是基于对内置类型的操作（例如 pimpl 手法的底层指针），而内置类型上的操作绝不会抛出异常。

请记住

- 当 std::swap 对你的类型效率不高时，提供一个 swap 成员函数，并确定这个函数不抛出异常。
- 如果你提供一个 member swap，也该提供一个 non-member swap 用来调用前者。对于 classes（而非 templates），也请特化 std::swap。
- 调用 swap 时应针对 std::swap 使用 using 声明式，然后调用 swap 并且不带任何“命名空间资格修饰”。
- 为“用户定义类型”进行 std templates 全特化是好的，但千万不要尝试在 std 内加入某些对 std 而言全新的东西。