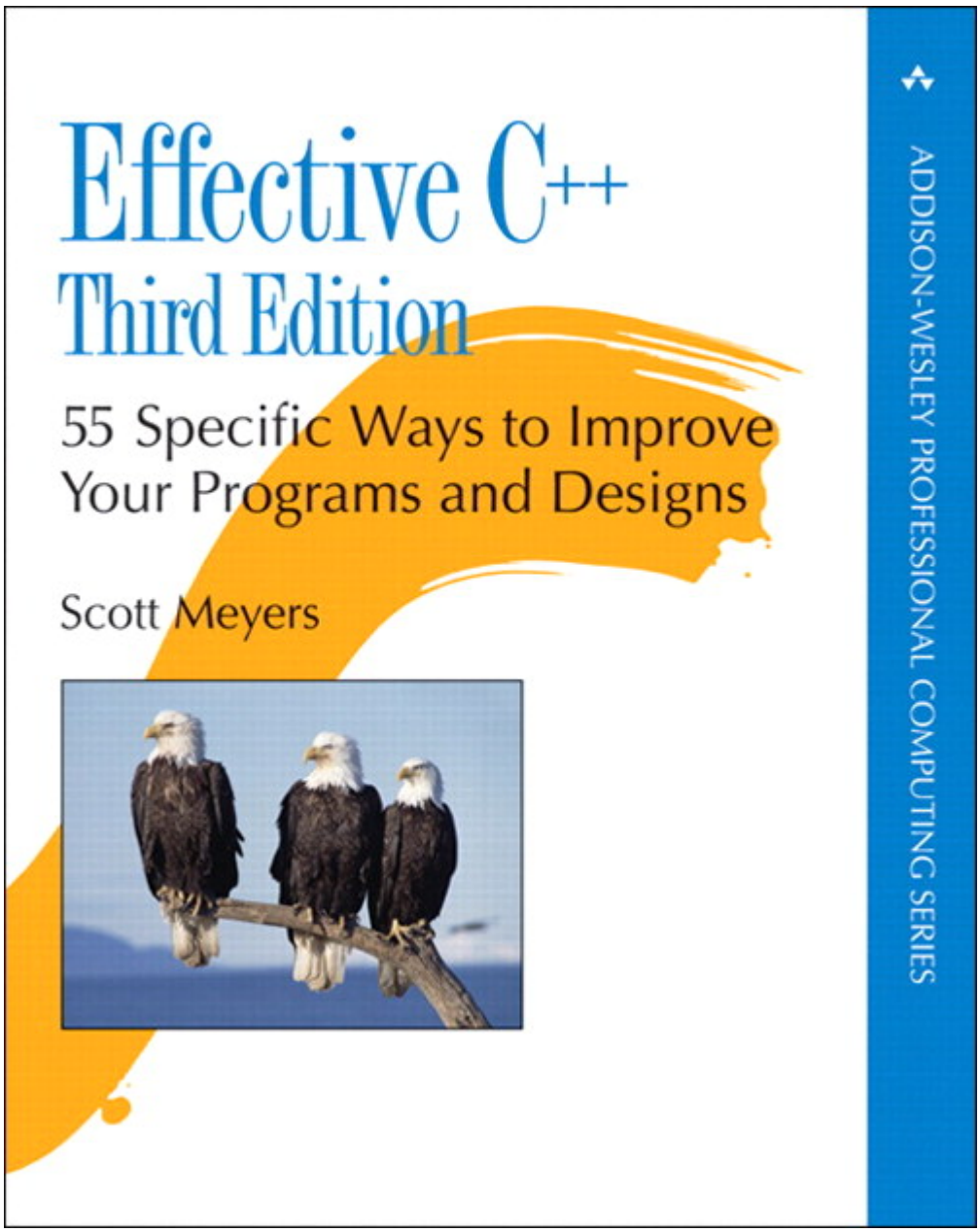
Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs - Cover

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/



- # Terminology（术语）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

这是一个所有程序员都应该了解的小型的 C++ vocabulary（词汇表）。下面的术语都足够重要，对它们的含义取得完全一致对于我们来说是完全必要的。

declaration（声明）告诉编译器关于某物的 name（名字）和 type（类型），但它省略了某些细节。以下这些都是 declaration（声明）：

```
extern int x;                                   ?// object declaration

std::size_t numDigits(int number);                // function declaration

class Widget;                                   ?// class declaration

template<typename T>                              // template declaration
class GraphNode;                                  // (see Item 42 for info
on
                                                  // the use of "typename")
```

注意即使是 built-in type（内建类型），我还是更喜欢将整数 x 看作一个 "object"，某些人将 "object" 这个名字保留给 user-defined type（用户定义类型），但我不是他们中的一员。再有就是注意函数 numDigits 的返回类型是 std::size_t，也就是说，namespace（命名空间）std 中的 size_t 类型。这个 namespace（命名空间）是 C++ 标准库中每一样东西实际所在的地方。但是，因为 C 标准库（严谨地说，来自于 C89）在 C++ 中也能使用，从 C 继承来的符号（诸如 size_t ）可能存在于全局范围，或 std 内部，或两者都有，这依赖于哪一个头文件被 #include 。在本书中，我假设 C++ 头文件被 #include，这也就是为什么我用 std::size_t 代替 size_t 的原因。当文字讨论中涉及到标准库组件时，我一般不再提及 std，这依赖于你认可类似 size_t，vector，以及 cout 之类的东西都在 std 中，在示例代码中，我总是包含 std ，因为真正的代码没有它将无法编译。

顺便说一下，size_t 仅仅是供 C++ 对某物计数时使用的某些 unsigned 类型的 typedef（例如，一个 char*-based string（基于 char* 的 string）中字符的个数，一个 STL container（容器）中元素的个数，等等）。它也是 vector，deque，和 string 的 operator[] 函数所持有的类型，这是一个在 Item 3 中定义我们自己的 operator[] 函数时将要遵守的惯例。

每一个函数的 declaration（声明）都表明了它的 ***signature***（识别特征），也就是它的参数和返回类型。一个函数的 signature（识别特征）与它的类型相同。对于 numDigits 的情况，signature（识别特征）是 std::size_t (int)，也就是说，“函数取得一个 int，并返回一个 std::size_t”。官方的 "signature"（识别特征）的 C++ 定义排除了函数的返回类型，但是在本书中，将返回类型考虑为 signature（识别特征）的一部分更加有用。

definition（定义）为编译器提供在 declaration（声明）时被省略的细节。对于一个 object（对象），definition（定义）是编译器为 object（对象）留出内存的地方。对于一个 function（函数）或一个 function template（函数模板），definition（定义）提供代码本体。对于一个 class（类）或一个 class template（类模板），definition（定义）列出了 class（类）或者 template（模板）的 members（成员）：

```
int x;                                              // object definition

std::size_t numDigits(int number)                  ?// function definition.
{                                                  ?// (This function
returns
?std::size_t digitsSoFar = 1;                       // the number of digits
                                                    // in its parameter.)
?while ((number /= 10) != 0) ++digitsSoFar;

?return digitsSoFar;
}

class Widget {                                      // class definition
public:
?Widget();
?~Widget();
?...
};

template<typename T>                                // template definition
class GraphNode {          ?
public:
?GraphNode();
?~GraphNode();
?...
};
```

initialization（初始化）是设定一个 object（对象）的第一个值的过程。对于由 structs 或 classes 产生的 objects（对象）来说，initialization（初始化）通过 constructors（构造函数）完成。***default constructor***（缺省构造函数）就是不需要任何 arguments（实参）就可以调用的那一个。这样的一个 constructor（构造函数）既可以没有 parameters（形参），也可以每一个 parameter（形参）都有缺省值（此段原文有误，根据作者网站勘误更改——译者注。）：

```
class A {
public:
?A();                                          // default constructor
};

class B {
public:
?explicit B(int x = 0, bool b = true);  ?// default constructor; see
below
};                                               // for info on "explicit"

class C {
public:
?explicit C(int x);                            // not a default constructor
};
```

这里 classes `B` 和 `C` 的 constructors（构造函数）都被声明为 `explicit`（显式）。这是为了防止它们被用来执行 implicit type conversions（隐式类型转换），虽然他们还可以被用于 explicit type conversions（显示类型转换）：

```
void doSomething(B bObject);          // a function taking an object of
                                      // type B
```

```
B bObj1;                                  // an object of type B

doSomething(bObj1);                     ?// fine, passes a B to doSomething

B bObj2(28);                              // fine, creates a B from the int
28
                                          // (the bool defaults to true)

doSomething(28);                          // error! doSomething takes a B,
                                          // not an int, and there is no
                                          // implicit conversion from int to
B

doSomething(B(28));                     ?// fine, uses the B constructor to
                                          // explicitly convert (i.e., cast)
the
                                          // int to a B for this call. (See
                                          // Item 27 for info on casting.)
```

constructors（构造函数）被声明为 explicit（显式）通常比 non-explicit（非显式）更可取，因为它们可以防止编译器执行意外的（常常是无意识的）type conversions（类型转换）。除非我有一个好的理由允许一个 constructor（构造函数）被用于 implicit type conversions（隐式类型转换），否则我就将它声明为 explicit（显式）。我希望你能遵循同样的方针。

请注意我是如何突出上面的示例代码中的 cast（强制转换）的。贯穿本书，我用这样的突出引导你注意那些应该注意的材料。（我也突出章节号码?　墙鼜鏊且蛭　蚁肴盟　每匆恍　#?

copy constructor（拷贝构造函数）被用来以一个 object（对象）来初始化同类型的另一个 object（对象），***copy assignment operator***（拷贝赋值运算符）被用来将一个 object（对象）中的值拷贝到同类型的另一个 object（对象）中：

```
class Widget {
public:
?Widget();                                  // default constructor
?Widget(const Widget& rhs);               ?// copy constructor
?Widget& operator=(const Widget& rhs);      // copy assignment operator
?...
};
Widget w1;                                 ?// invoke default constructor

Widget w2(w1);                             ?// invoke copy constructor

w1 = w2;                                   ?// invoke copy
                                           ?// assignment operator
```

当你看到什么东西看起来像一个 assignment（赋值）的话，要仔细阅读，因为 "=" 在语法上还可以被用来调用 copy constructor（拷贝构造函数）：

```
Widget w3 = w2;                            // invoke copy constructor!
```

幸运的是，copy constructor（拷贝构造函数）很容易从 copy assignment（拷贝赋值）中区别出来。如果一个新的 object（对象）被定义（就象上面那行代码中的 w3），一个 constructor（构造函数）必须被调用；它不可能是一个 assignment（赋值）。如果没有新的 object（对象）被定义（就象上面那行 "w1 = w2" 代码中），没有

constructor（构造函数）能被调用，所以它就是一个 assignment（赋值）。

copy constructor（拷贝构造函数）是一个特别重要的函数，因为它定义一个 object（对象）如何 passed by value（通过传值的方式被传递）。例如，考虑这个：

```
bool hasAcceptableQuality(Widget w);

...
Widget aWidget;
if (hasAcceptableQuality(aWidget)) ...
```

参数 w 通过传值的方式被传递给 hasAcceptableQuality，所以在上面的调用中，aWidget 被拷贝给 w。拷贝动作通过 Widget 的 copy constructor（拷贝构造函数）被执行。pass-by-value（通过传值方式传递）意味着 "call the copy constructor"（调用拷贝构造函数）。（然而，通过传值方式传递 user-defined types（用户定义类型）通常是一个不好的想法，pass-by-reference-to-const（传引用给 const）通常是更好的选择。关于细节，参见 Item 20。）

STL 是 Standard Template Library（标准模板库），作为 C++ 标准库的一部分，致力于 containers（容器）（例如，vector，list，set，map，等等），iterators（迭代器）（例如，vector<int>::iterator，set<string>::iterator，等等），algorithms（算法）（例如，for_each，find，sort，等等），以及相关机能。相关机能中的很多都通过 ***function objects***（函数对象）——行为表现类似于函数的 objects（对象）——提供。这样的 objects（对象）来自于重载了 operator() ——函数调用运算符——的 class（类），如果你不熟悉 STL，在读本书的时候，你应该有一本像样的参考手册备查，因为对于我来说 STL 太有用了，以至于不能不利用它。一但你用了一点点，你也会有同样的感觉。

从 Java 或 C# 那样的语言来到 C++ 的程序员可能会对 ***undefined behavior***（未定义行为）的概念感到吃惊。因为各种各样的原因，C++ 中的一些 constructs（结构成分）的行为没有确切的定义：你不能可靠地预知运行时会发生什么。这里是两个带有 undefined behavior（未定义行为）的代码的例子：

```
int *p = 0;                                  ?// p is a null pointer

std::cout << *p;                               // dereferencing a null pointer
                                               // yields undefined behavior

char name[] = "Darla";                         // name is an array of size 6
(don't
                                               // forget the trailing null!)

char c = name[10];                             // referring to an invalid array
index
                                               // yields undefined behavior
```

为了强调 undefined behavior（未定义行为）的结果是不可预言而且可能是令人讨厌的，有经验的 C++ 程序员常常说带有 undefined behavior（未定义行为）的程序 can（能）毁掉你的辛苦工作的成果。这是真的：一个带有 undefined behavior（未定义行为）的程序 *could*（可以）毁掉你的心血。只不过可能性不太大。更可能的是那个程序的表现反复无常，有时会运行正? # 惺被岢沟淄甑埃 褂惺被岵 砦蟮慕崩 S惺盗Φ? C++ 程序员能以最佳状态避开 undefined behavior（未定义行为）。本书中，我会指出许多你必须要注意它的地方。

另一个可能把从其它语言转到 C++ 的程序员搞糊涂的术语是 ***interface***（接口）。Java 和 .NET 的语言都将 Interfaces（接口）作为一种语言要素，但是在 C++ 中没有这种事，但是在 Item 31

讨论了如何模拟它。当我使用术语 "interface"（接口）时，一般情况下我说的是一个函数的 signature（识别特征），是一个 class（类）的可访问元素（例如，一个 class（类）的 "public interface"，"protected interface"，或 "private interface"），或者是对一个 template（模板）的 type parameter（类型参数）来说必须合法的 expressions（表达式）（参见 Item 41）。也就是说，我是作为一个相当普遍的设计概念来谈论 interface（接口）的。

client（客户）是使用你写的代码（一般是 interfaces（接口））的某人或某物。例如，一个函数的 clients（客户）就是它的使用者：调用这个函数（或持有它的地址）的代码的片段以及写出和维护这样的代?氲娜恕?class（类）或者 template（模板）的 clients（客户）是使用这个 class（类）或 template（模板）的软件的部件，以及写出和维护那些代码的程序员。在讨论 clients（客户）的时候，我一般指向程序员，因为程序员会被困扰和误导，或者因为不好的 interfaces（接口）而烦恼。但他们写的代码却不会。

你也许不习惯于为 clients（客户）着想，但是我会用大量的时间试图说服你：你应该尽你所能使他们的生活更轻松。记住，你?彩且桓鋈渌　丝　5.娜砑　？client（客户）。难道你不希望那些人为你把事情弄得轻松些吗？除此之外，你几乎肯定会在某个时候发现?阕约捍υ诹四阕约旱? client（客户）的位置上（也就是说，使用你写的代码），而这个时候，你会为你在开发你的 interfaces（接口）时在头脑中保持了对 client（客户）的关心而感到高兴。

在本书中，我常常掩盖 functions（函数）和 function templates（函数模板）之间以及 classes（类）和 class templates（类模板）之间的区别。那是因为对其中一个确定的事对另一个常常也可以确定。如果不是这样，我?峒　鸽源? classes（类），functions（函数），以及由 classes（类）和 functions（函数）产生的 templates（模板）。

在代码注释中提到 constructor（构造函数）和 destructors（析构函数）时，我有时使用缩写形式 ***ctor*** 和 ***dtor***。

Naming Conventions（命名惯例）

我试图为 objects（对象），classes（类），functions（函数），templates（模板）等选择意味深长的名字，但是在我的某些名字后面的含义可能不会立即显现出来。例如，我?毛鹣不兜牧礁? parameter names（参数名字）是 `lhs` 和 `rhs`。它们分别代表 "left-hand side" 和 "right-hand side"。我经常用它们作为实现 binary operators（二元运算符）的函数（例如，`operator==` 和 `operator*`）的 parameter names（参数名字）。例如，如果 `a` 和 `b` 是代表有理数的 objects（对象），而且如果 `Rational` objects（对象）能通过一个 non-member（非成员）的 `operator*` 函数相乘（Item 24 中解释的很可能就是这种情况），表达式

```
a * b
```

与函数调用

```
operator*(a,b)
```

就是等价的。

在 Item 24 中，我这样声明 `operator*`：

```
const Rational operator*(const Rational& lhs, const Rational& rhs);
```

你可以看到，left-hand operand（左手操作数）`a` 在函数内部以 `lhs` 的面目出现，而 right-hand operand（右手操作数）`b` 以 `rhs` 的面目出现。

对于 member functions（成员函数），left-hand argument（左手参数）表现为 `this` pointer（this 指针），所以有时候我单独使用 parameter name（参数名字） `rhs`。你可能已经在第 5 页中某些 `Widget` 的 member functions（成员函数）的 declarations（声明）（本文介绍 copy constructor（拷贝构造函数）的那一段中的例子——译者注）中注意到了这一点。这一点提醒了我。我经常在示例中使用 `Widget` class（类）。"Widget" 并不意味着什么东西。它仅仅是在我需要一个示例类的名字的时候不时地使用一下的名字。它和 GUI 工具包中的 widgets 没有任何关系。

我经常遵循这个规则为 pointers（指针）命名：一个指向 type（类型）`T` 的 object（对象）的 pointer（指针）被称为 `pt`，"pointer to T"。以下是例子：

```
Widget *pw;                             // pw = ptr to Widget

class Airplane;
Airplane *pa;                           // pa = ptr to Airplane

class GameCharacter;
GameCharacter *pgc;                     // pgc = ptr to GameCharacter
```

我对 references（引用）使用类似的惯例：`rw` 可以认为是一个 reference to a `Widget`（引向一个 `Widget` 的引用），而 `ra` 是一个 reference to an `Airplane`（引向一个 `Airplane` 的引用）。

在讨论 member functions（成员函数）的时候我偶尔会使用名字 `mf`。

Threading Considerations（对线程的考虑）

作为一种语言，C++ 没有 threads（线程）的概念——实际上，是没有任何一种 concurrency（并发）的概念。对于 C++ 标准库也是同样如此。就 C++ 涉及的范围而言，multithreaded programs（多线程编程）并不存在。

而且至今它们依然如此。我致力于让此书基于标准的，可移植的 C++，但我也不能对 thread safety（线程安全）已成为很多程序员所面临的一个问题的事实视而不见。我对付这个标准 C++ 和现实之间的裂痕的方法就是指出某个 C++ constructs（结构成分）以我的分析很可能在 threaded environment（线程环境）中引起问题的地方。这样不但不会使本书成为一本 multithreaded programming with C++（用 C++ 进行多线程编程）的书。反而，它更会使本书在相当程度上成为这样一本 C++ 编程的书：将自己在很大程度上限制于 single-threaded（单线程）思路，承认 multithreading（多线程）的存在，并试图指出有线程意识的程序员需要特别留心评估我提供的建议的地方。

如果你不熟悉 multithreading（多线程）或者不必为此担心，你可以忽略我关于线程的讨论。如果你正在编写一个多线程的应用或? 猓 蘼廴_绾危 爰亲〃业钠雷10.筒13. 魑 闶褂? C++ 时需要致力去解决的问题的起点。

TR1 和 Boost

你会发现提及 TR1 和 Boost 的地方遍及全书。它们每一个都有一个专门的 Item 在某些细节上进行描述（Item 54 是 TR1，Item 55 是 Boost），但是，不幸的是，这些 Item 在全书的最后。（他们在那里是因为那样更好一些，我确实试过很多其它的地方。）如果你愿意，你? 衷诰涂梢苑 ⑯亩聊切? Item，但是如果你更喜欢从本书的起始处而不是结尾处开始，以下摘要会对你有所帮助：

- TR1 ("Technical Report 1") 是被加入 C++ 标准库的新机能的 specification（规格说明书）。这些机能以新的 class（类）和 function templates（函数模板）的形式提供了诸如 hash tables（哈希表），reference-counting smart pointers（引用计数智能指针），regular expressions（正则表达式），等等。所有的 TR1 组件都位于嵌套在 namespace `std` 内部的 namespace `tr1` 内。

- Boost 是一个组织和一个网站 (http://boost.org)
  提供的可移植的，经过同行评审的，开源的 C++ 库。大多数 TR1 机能都基于 Boost 的工作，而且直到编译器厂商在他们的 C++ 库发行版中包含 TR1 之前，Boost 网站很可能会保持开发者寻找 TR1 实现的第一站的地位。Boost 提供的东西比用于 TR1 的更多，无论如何，在很多情况下，它还是值得去了解一下的。

窗体底端

Introduction（导言）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

学习一种编程语言的基础是一回事：学习如何用那种语言设计和实现高效率的程序完全是另外一回事。对于 C++ ——一种以拥有非同寻常的能力范围和表现力而自豪的语言——更是尤其如此。如果能正确使用，与 C++ 共事是一件令人快乐的事情。极多样的设计样式被直接表达并有效实现。对于 classes（类），functions（函数），以及 templates（模板）的明智选择和小心精巧的安排能使应用程序的编程更加简单，直观，高效，并基本无错。?绻 阒 廊绾稳ブ觯 闯龈咝丫?C++ 程序并不特别难。然而，如果不经训练就贸然使用，C++ 也会导致不可理解的，难以维护的，无法扩展的，低效率的，错误百出的代码。

本书的目的在于引导你如何高效使用 C++。我假设你已经熟悉了作为语言的 C++ 并有使用它的一些经验。我在此提供的是使用这种语言的指南，以使你的程序易于理解，可维护，易移植，可?┭梗 矢撸 倚形 夏愕脑て淞?

我提供的建议落在两个主要的范围中：普通的设计策略，以及特殊语言特性的具体细节。设计的讨论集中于如?卧?C++ 做某件事情的多种不同方法之间进行选择。如何在 inheritance（继承）和 templates（模板）之间选择？如何在 public（公有）和 private inheritance（私有继承）之间选择？如何在 private inheritance（私有继承）和 composition（复合）之间选择？如何在 member（成员）和 non-member functions（非成员函数）之间选择？如何在 pass-by-value（传值）和 pass-by-reference（传引用）之间选择？在一开始就做出正确的决定是很重要的，因为一个不好的选择可能会直到开发过程很晚的阶段才显现出来，在这时候再调整它常常是困难重重，极为耗时而且代价不菲的。

即使在你正确地知道你要做什么的时候，仅仅把事情做对也是需要技巧的。assignment operators（赋值运算符）的合适的返回类型是什么？destructor（析构函数）什么时候应该是 virtual（虚拟）的？当 operator new（运算符 new ）找不到足够的内存时它应该怎么办？类似这些的令人费神的细节是至关重要的，因为错误的做法几乎总是导?挛薹ィち系模 芸赡芰钊嗣曰蟮某绦蛐形 U獗臼檎 抢窗镏 惚范庶庑┫ 侍獾摹?

这不是一本全面的 C++ 手册。它收集了 55 个详细的提议（我将它们称为 Items ）告诉你怎样才能改善你的程序和设计。每一个 Item 都能 stands more or less on its own（独立成章），但大部分也包含对其它 Items 的参考。因而，读这本书的一个方法就是从你感兴趣的一个 Item 开始，然后顺着它的参考条目继续看下去。

这本书也不是一本 C++ 入门书。例如，在 Chapter 2（第二章），我希望能告诉你关于 constructors（构造函数），destructors（析构函数），以及 assignment operators（赋值运算符）正确实现的全部内容，但是我假设你已经知道或者能在别处找到这些函数做些什么?约八 歉迷踅 鞣淖柿稀 4 罅?C++ 书籍包含类似这样的信息。

这本书的目的是为了突出 C++ 编程中那些常常被忽略的方面。其它书描述了语言的各个部分。这本书则告诉你如何将这些部分结合起来以达?礁咝Ⅱ喑痰淖钪漳康摹Ｆ渌 楦夔叱闵绾问鼓愕某绦蚰芪槐嘁猾Ｕ獗臼樵蚋夔叱闵绾伪范饽切 ┍嘁肫鞑荒茇夔?你的问题。

与此同时，本书将自己限制在 standard C++（标准 C++）之内。在此仅仅使用官方的语言标准中的特性。可移植性是本书一个关键的关注点，所以如果你要寻找?教丁览档奶匾院筒考 谖铁锜换嵴业健?

另一个在本书中找不到的东西是 C++ Gospel（C++ 福音书）——通向完美的 C++ 软件的一条 One True Path（真理之路）。本书中的每一个 Item 都在如何生成更好的设计，如何避免一般的问题，以及如何得到更高的效率等方面提供指导，但没有一个 Item 是普遍适用的。软件设计和实现是一项复杂的任务，被 hardware（硬件），operating system（操作系统），和 application（应用程序）的限制所影响，所以我能做的最好的事情就是为创建更好的程序提供 guidelines（指导方针）。

如果你在所有的时候都遵守这些 guidelines（指导方针），你将不太可能落入环绕在 C++ 周围的大量陷阱中，但是作为 guidelines（指导方针）的固有限制，它总有例外。这就是为什么每一个 Item 都有一个详细的解释。这些解释是本书中最重要的部分。只有理解了一个 Item 背后的基本原理，你才能决定它是否适合你开发的软件以及你被困其下的特有的限制。

本书最好用来增加关于 C++ 如何运转，为什么它会这样运转，以及如何让它的行为为你服务等方面的知识。盲目运用本书的 Items 无疑是不适当的，但是与此同时，如果你没有更好的理由，或许也不应该违反这些

guidelines（指导方针）中的任何一条。

窗体底端

Preface（前言）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

我在 1991 年写出了 *Effective C++* 的最早版本，1997
年出了第二版，我更新了一些重要的方面的素材，但是，因为我不想使熟悉本书第一版的读者感到困惑，我尽?
畚罂赡鼙 3 至嗽　械慕剁埂W钤绲? 50 个 Item 标题中的 48
个原则上保持不变。如果把书看作一栋房子，第二版就相当于通过更换地毯、涂料和灯光设备等使其焕然一新?
£

对于第三版，我打散了原有的秩序。（很多次我希望我能做得更彻底。）1991 年以来，C++
世界经历了巨大的改变，而将近 15 年以前我制定的那些 Item 也不再契合本书的目标——将最重要的 C++
编程准则融入小而易读的建议中。1991 年，假设 C++ 程序员具有 C 语言背景是有一定道理的。现在，转到
C++ 的程序员很可能来自 Java 或 C#。1991 年，inheritance（继承）和 object-oriented programming
（面向对象编程）对于大多数程序员来说都是新鲜的。现在，这些已经是非常普遍的概念，而 exceptions
（异常），templates（模板），以及 generic programming（泛型编程）成为新的需要更多指导的领域。1991
年，没有人听说过 design patterns（设计模式）。现在，讨论软件系统时很难不涉及它们。1991 年，C++
正式标准化的工作刚刚开始。现在，标准化已经八年了，而下一个版本的工作也已经开始。

为应对这些变化，我尽己所能将写字板擦得一干二净，并不断地追问自己："在 2005 年，对于目前的 C++
程序员，什么才是最重要的建议？"结果就是这个新版本中的一组 Item。本书包括了关于 resource
management（资源管理）和 programming with templates（使用模板编程）的新的章节。实际上，
templates（模板）的考虑贯穿全书，因为它几乎影响了 C++ 的每个方面。本书也包括关于在 exceptions
（异常）存在的场合下编程，在应用 design patterns（设计模式），以及在使用新的 TR1 库程序（TR1 在
Item 54 中介绍）等方面的新的素材。本书还承认在 single-threaded systems
（单线程系统）中能很好地工作的技术和方法可能不适用于 multithreaded systems
（多线程系统）。噢，本书超过一半的素材都是新的。然而，第二版中基本原理方面的资料中的大部分依然是?
匾　模　　晕医　　且哉度　蚰茄　男问奖Ａ粥吕础＃　隰茉? Appendix B（附录 B）找到第二版和第三版
Item 的对照表。）

我尽我所能使本书趋于最好，但我并不幻想完美。如果你觉得本书中的一些 Item
作为常规的建议不太合适；或者有更好的方法实现本书中需要完成的任务；或者有一个或更多技术上的讨论不?
魅罚　煌耆　　蛉菀琢钊宋蠼猓　敫蒌呶摇Ｈ绻　惴⑾秩魏未砦?——
技术上的，文法上的，印刷上的，无论哪种——
也请告诉我。如果你是第一个让我注意到某个问题的人，我很高兴将你的名字加入到以后再次印刷的致谢中。

即使 Item 的数量增加到 55，本书中的这一组准则离完满无遗还差得很远。但是成为好的规则——
在几乎所有的应用程序几乎所有的时间中得到应用——
比它看上去更加困难。如果你有增加准则的建议，我很高兴能听到它。

我维护本书从第一次印刷起的变化列表，包括错误修正，进一步解释，和技术更新。这个列表可以从
Effective C++ Errata 网页得到，http://aristeia.com/BookErrata/ec++3e-errata.html
。如果你希望当我更新列表时，你能得到通报，我建议你参加我的 mailing list
（邮件列表）。我用它作为通告，发给那些对跟踪我的职业工作感兴趣的人们。关于细节问题，请参考
http://aristeia.com/MailingList/。

SCOTT DOUGLAS MEYERS　　STAFFORD, OREGON
http://aristeia.com/　　APRIL 2005

Item 1: 将 C++ 视为 federation of languages（语言联合体）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

最初，C++ 仅仅是在 C 的基础上附加了一些 object-oriented（面向对象）的特性。C++ 最初的名称—— "C with Classes" 就非常直观地表现了这一点。

作为一个语言的成熟过程，C++ 的成长大胆而充满冒险，它吸收的思想，特性，以至于编程策略与 C with Classes 越来越不同。exceptions（异常）要求不同的建构功能的途径（参见 Item 29），templates（模板）将设计思想提升到新的高度（参见 Item 41），而 STL 定义了一条前所未见的通向扩展性的道路。

今天的 C++ 已经成为一个 *multiparadigm programming language*（多范式的编程语言），一个囊括了 procedural（过程化），object-oriented（面向对象），functional（函数化），generic（泛型）以及 metaprogramming（元编程）特性的联合体。这些能力和弹性使 C++ 成为无可匹敌的工具，但也引起了一些混乱。所有的 "proper usage"（惯用法）规则似乎都有例外。我们该如何认识这样一个语言？

最简单的方法是不要将 C++ 视为一个单一的语言，而是一个亲族的语言的 federation（联合体）。在每一个特定的 sublanguage（子语言）中，它的特性趋向于直截了当，简单易记。但你从一个 sublanguage（子语言）转到另外一个，它的规则也许会发生变化。为了感受 C++，你必须将它的主要的 sublanguages（子语言）组织到一起。幸运的是，它只有 4 个：

- **C** ——归根结底，C++ 依然是基于 C 的。blocks（模块），statements（语句），preprocessor（预处理器），built-in data types（内建数据类型），arrays（数组），pointers（指针）等等，全都来自于 C。在很多方面。C++ 提出了比相应的 C 版本更高级的解决问题的方法（例如，参见 Item 2（取代 reprocessor（预处理器））和 13（使用 objects（对象）管理 resources（资源））），但是，当你发现你自己工作在 C++ 的 C 部分时，effective programming（高效编程）的规则表现了 C 的诸多限制范围：没有 templates（模板），没有 exceptions（异常），没有 overloading（重载）等等。
- **Object-Oriented C++** —— C++ 的这部分就是 C with Classes 涉及到的全部：classes（类）（包括构造函数和析构函数），encapsulation（封装），inheritance（继承），polymorphism（多态），virtual functions (dynamic binding)（虚拟函数（动态绑定））等。C++ 的这一部分直接适用于 object-oriented design（面向对象设计）的经典规则。
- **Template C++** ——这是 C++ 的 generic programming（泛型编程）部分，大多数程序员对此都缺乏经验。template（模板）的考虑已遍及 C++，而且好的编程规则中包含特殊的 template-only（模板专用）条款已不再不同寻常（参见 Item 46 通过调用 template functions（模板函数）简化 type conversions（类型转换））。实际上，templates（模板）极为强大，它提供了一种全新的 programming paradigm（编程范式）—— *template metaprogramming* (TMP)（模板元编程）。Item 48 提供了一个 TMP 的概述，但是，除非你是一个 hard-core template junkie（死心塌地的模板瘾君子），否则你不需在此费心，TMP 的规则对主流的 C++ 编程少有影响。
- **STL** —— STL 是一个 template library（模板库），但它一个非常特殊的 template library（模板库）。它将 containers（容器），iterators（迭代器），algorithms（算法）和 function objects（函数对象）非常优雅地整合在一起，但是。templates（模板）和 libraries（库）也可以围绕其它的想法建立起来。STL 有很多独特的处事方法，当你和 STL 一起工作，你需要遵循它的规则。

在头脑中保持这四种 sublanguages（子语言），当你从一种 sublanguage（子语言）转到另一种时，为了高效编程你需要改变你的策略，不要吃惊你遭遇到的情景。例如，使?Ã built-in（内建）（也就是说，C-like（类 C 的））类型时，pass-by-value（传值）通常比 pass-by-reference（传引用）更高效，但是当你从 C++ 的 C 部分转到 Object-Oriented C++（面向对象 C++），user-defined constructors（用户自定义构造函数）和 destructors（析构函数）意味着，通常情况下，更好的做法是 pass-by-reference-to-`const`（传引用给 `const`）。在 Template C++ 中工作时，这一点更加重要，因为，在这种情况下，你甚至不知道你的操作涉及到的 object（对象）的类型。然而，当你进入 STL，你知道 iterators（迭代器）和 function objects（函数对象）以 C 的 pointers（指针）为原型，对于 STL 中的 iterators（迭代器）和 function objects（函数对象），古老的 C 中的 pass-by-value（传值）规则又重新生效。（关于选择 parameter-passing（参数传递）方式的全部细节，参见 Item 20。）

C++ 不是使用一套规则的单一语言，而是 federation of four sublanguages（四种子语言的联合体），每一种都有各自的规则。在头脑中保持这些 sublanguages（子语言），你会发现对 C++ 的理解会容易得多。

Things to Remember

- effective C++ programming（高效 C++ 编程）规则的变化，依赖于你使用 C++ 的哪一个部分。

窗体底端

- # Item 2: 用 consts, enums 和 inlines 取代 #defines

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

这个 Item 改名为"用 compiler（编译器）取代 preprocessor（预处理器）"也许更好一些，因为 #define 根本就没有被看作是语言本身的一部分。这是它很多问题中的一个。当你像下面这样做：

```
#define ASPECT_RATIO 1.653
```

compiler（编译器）也许根本就没有看见这个符号名 ASPECT_RATIO，在 compiler（编译器）得到源代码之前，这个名字就已经被 preprocessor（预处理器）消除了。结果，名字 ASPECT_RATIO 可能就没有被加入 symbol table（符号表）。如果在编译的时候，发现一个 constant（常量）使用的错误，你可能会陷入混乱之中，因为错误信息中很可能用 1.653 取代了 ASPECT_RATIO。如果，ASPECT_RATIO 不是你写的，而是在头文件中定义的，你可能会对 1.653 的出处毫无头绪，你还会为了跟踪它而浪费时间。在 symbolic debugger（符号调试器）中也会遇到同样的问题，同样是因为这个名字可能并没有被加入 symbol table（符号表）。

解决方案是用 constant（常量）来取代 macro（宏）：

```
const double AspectRatio = 1.653;    // uppercase names are usually for
                                    ?// macros, hence the name change
```

作为一个 language constant（语言层面上的常量），AspectRatio 被 compilers（编译器）明确识别并确实加入 symbol table（符号表）。另外，对于 floating point constant（浮点常量）（比如本例）来说，使用 constant（常量）比使用 #define 能产生更小的代码。这是因为 preprocessor（预处理器）盲目地用 1.653 置换 macro name（宏名字）ASPECT_RATIO，导致你的 object code（目标代码）中存在多个 1.653 的拷贝，如果使用 constant（常量）AspectRatio，就绝不会产生多于一个的拷贝。

用 constant（常量）代替 #defines 时，有两个特殊情况值得提出。首先是关于 constant pointers（常量指针）的定义。因为 constant definitions（常量定义）通常被放在 header files（头文件）中（这样它们就可以被包含在多个 source files（源文件）中），除了 pointer（指针）指向的目标是常量外，*pointer*（指针）本身被声明为 const 更加重要。例如，在头文件中定义一个 constant char*-based string（基于 char* 的字符串常量）时，你必须写两次 const：

```
const char * const authorName = "Scott Meyers";
```

对于 const（特别是与 pointers（指针）相结合时）的意义和使用的完整讨论，请参见 Item 3。然而在此值的一提的是，string objects（对象）通常比它的 char*-based（基于 char*）的祖先更可取，所以，更好的 authorName 的定义方式如下：

```
const std::string authorName("Scott Meyers");
```

第二个特殊情况涉及到 class-specific constants（类属（类内部专用的）常量）。为了将一个 constant（常量）的作用范围限制在一个 class（类）内，你必须将它作为一个类的 member（成员），而且为了确保它最多只有一个 constant（常量）拷贝，你还必须把它声明为一个 *static* member（静态成员）。

```
class GamePlayer {

private:

?static const int NumTurns = 5;     ?// constant declaration

?int scores[NumTurns];                // use of constant
?...
};
```

你从上面只看到了 NumTurns 的 *declaration*（声明），而不是 definition（定义）。通常，C++ 要求你为你使用的任何东西都提供一个 definition（定义），但是一个 static（静态）的 integral type（整型族）（例如：integers（整型），chars，bools）的 class-specific constants（类属常量）是一个例外。只要你不去取得它们的 address（地址），你可以只声明并使用它，而不提供它的 definition（定义）。如果你要取得一个 class constant（类属常量）的 address（地址），或者你使用的 compiler（编译器）在你没有取得 address（地址）时也不正确地要求 definition（定义）的话，你可以提供如下这样一个独立的 definition（定义）：

```
const int GamePlayer::NumTurns;      // definition of NumTurns; see
                                    ?// below for why no value is given
```

你应该把它放在一个 implementation file（实现文件）而非 header file（头文件）中。因为 class constants（类属常量）的 initial value（初始值）在声明时已经提供（例如：NumTurns 在定义时被初始化为 5），因此在定义处允许没有 initial value（初始值）。

注意，顺便提一下，没有办法使用 #define 来创建一个 class-specific constant（类属常量），因为 #defines 不考虑 scope（作用范围）。一旦一个 macro（宏）被定义，它将大范围影响你的代码（除非在后面某处存在 #undefed）。这就意味着，#defines 不仅不能用于 class-specific constants（类属常量），而且不能提供任何形式的 encapsulation（封装），也就是说，没有类似 "private"（私有）#define 的东西。当然，const data members（const 数据成员）是能够被封装的，NumTurns 就是如此。

比较老的 compilers（编译器）可能不接受上面的语法，因为它习惯于将一个 static class member（静态类成员）在声明时就获得 initial value（初始值）视为非法。而且，in-class initialization（类内初始化）仅仅对于 integral types（整型族）和 constants（常量）才被允许。如果上述语法不能使用，你可以将 initial value（初始值）放在定义处：

```
class CostEstimate {

private:

?static const double FudgeFactor;    // declaration of static class
?...                                 ?// constant; goes in header file
};

const double                          // definition of static class

?CostEstimate::FudgeFactor = 1.35;?// constant; goes in impl. file
```

这就是你所要做的全部。仅有的例外是当在类的编译期需要 value of a class constant（一个类属常量的值）的情况，例如前面在声明 array（数组）GamePlayer::scores 时（compilers（编译器）必须在编译期知道 array（数组）的 size（大小））。如果 compilers（编译器）（不正确地）禁止这种关于 static integral class constants（静态整型族类属常量）的 initial values（初始值）的使用方法的 in-class specification（规范），一个可接受的替代方案被亲切地（并非轻蔑地）昵称为 "the enum hack"。这项技术获得以下事实的支持：一个 enumerated type（枚举类型）的值可以用在一个需要 ints

的地方。所以 GamePlayer 可以被如下定义：

```
class GamePlayer {

private:

?enum { NumTurns = 5 };              // "the enum hack" - makes
                                      // NumTurns a symbolic name for 5

?int scores[NumTurns];              ?// fine
?...
};
```

the enum hack 有几个值得被人所知的原因。首先，the enum hack 的行为在几个方面上更像一个 #define 而不是 const，而有时这正是你所需要的。例如：可以合法地取得一个 const 的 address（地址），但不能合法地取得一个 enum 的 address（地址），这正像同样不能合法地取得一个 #define 的 address（地址）。如果你不希望人们得到你的 integral constants（整型族常量）的 pointer（指针）或 reference（引用），enum（枚举）就是强制约束这一点的好方法。（关于更多的通过编码的方法强制执行设计约束的方法，参?û Item 18。）同样，使用好的 compilers（编译器）不会为 integral types（整型族类型）的 const objects（const 对象）分配多余的内存（除非你创建了这个对象的指针或引用），即使拖泥带水的 compilers（编译器）乐意，你也决不会乐意为这样的 objects（对象）分配多余的内存。像 #defines 和 enums（枚举）就绝不会导致这种 unnecessary memory allocation（不必要的内存分配）。

需要知道 the enum hack 的第二个理由是纯粹实用主义的，大量的代码在使用它，所以当你看到它时，你要认识它。实际上，the enum hack 是 template metaprogramming（模板元编程）的一项基本技术（参见 Item 48）。

回到 preprocessor（预处理器）上来，#define 指令的另一个普遍的（不好的）用法是实现看来像函数，但不会引起一个函数调用的开销的 macros（宏）。以下是一个用较大的宏参数调用函数 f 的 macro（宏）：

```
// call f with the maximum of a and b

#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))
```

这样的 macro（宏）有数不清的缺点，想起来就让人头疼。

无论何时，你写这样一个 macro（宏），都必须记住为 macro body（宏体）中所有的 arguments（参数）加上括号。否则，当其他人在表达式中调用了你的 macro（宏），你将陷入麻烦。但是，即使你确实做到了这一点，你还是会看到意想不到的事情发生：

```
int a = 5, b = 0;

CALL_WITH_MAX(++a, b);                // a is incremented twice

CALL_WITH_MAX(++a, b+10);            ?// a is incremented once
```

这里，调用 f 之前 a 递增的次数取决于它和什么进行比较！

幸运的是，你并不是必须要和这样不知所云的东西打交道。你可以通过一个 inline function（内联函数）的 template（模板）来获得 macro（宏）的效率，以及完全可预测的行为和常规函数的类型安全（参见 Item 30）：

```
template<typename T>                                    // because we don't

inline void callWithMax(const T& a, const T& b)   ?// know what T is, we

{                                                        ?// pass by
reference-to-

?f(a > b ? a : b);                                    ?// const - see Item 20

}
```

这个 template（模板）产生一组函数，每一个获得两个相同类型的对象并使用其中较大的一个调用 f。这样就不需要为函数体内部的参数加上括号，也不需要担心多余的参数解析次数，等等。此外，因?[a] callWithMax 是一个真正的函数，它遵循函数的作用范围和访问规则。例如，谈论一个类的私有的 inline function（内联函数）会获得正确的理解，但是用 macro（宏）就无法做到这一点。

为了得到 consts，enums 和 inlines 的可用性，你需要尽量减少 preprocessor（预处理器）（特别是 #define）的使用，但还不能完全消除。#include 依然是基本要素，而 #ifdef/#ifndef 也继续扮演着重要的角色。现在还不是让 preprocessor（预处理器）完全退休的时间，但你应该给它漫长而频繁的假期。

Things to Remember

- 对于 simple constants（简单常量），用 const objects（const 对象）或 enums（枚举）取代 #defines。
- 对于 function-like macros（类似函数的宏），用 inline functions（内联函数）取代 #defines。

窗体底端

- # Item 3: 只要可能就用 const

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

关于 const 的一件美妙的事情是它允许你指定一种 semantic（语义上的）约束：一个特定的 object（对象）不应该被修改。而 compilers（编译器）将执行这一约束。它允许你通知 compilers（编译器）和其他程序员，某个值应该保持不变。如果确实如此，你就应该明确地表示出来，因为这? 焕矗 憔涂梢阅鼻? compilers（编译器）的帮助，确保这个值不会被改变。

keyword（关键字）const 非常多才多艺。在 classes（类）的外部，你可以将它用于 global（全局）或 namespace（命名空间）范围的 constants（常量）（参见 Item 2），以及那些在 file（文件）、function（函数）或 block（模块）scope（范围）内被声明为 static（静态）的对象。在 classes（类）的内部，你可以将它用于 static（静态）和 non-static（非静态）data members（数据成员）上。对于 pointers（指针），你可以指定这个 pointer（指针）本身是 const，或者它所指向的数据是 const，或者两者都是，或者都不是：

```
char greeting[] = "Hello";

char *p = greeting;                    ?// non-const pointer,
                                         // non-const data

const char *p = greeting;              ?// non-const pointer,
                                         // const data

char * const p = greeting;               // const pointer,
                                         // non-const data

const char * const p = greeting;         // const pointer,
                                         // const data
```

这样的语法本身其实并不像表面上那样反复无常。如果 const 出现在星号左边，则指针 *pointed to*（指向）的内容为 constant（常量）；如果 const 出现在星号右边，则 *pointer itself*（指针自身）为 constant（常量）；如果 const 出现在星号两边，则两者都为 constant（常量）。

当指针指向的内容为 constant（常量）时，一些人将 const 放在类型之前，另一些人将它放在类型之后星号之前。两者在意义上并没有区别，所以，如下两个函? 哂邢嗤 ？parameter type（参数类型）：

```
void f1(const Widget *pw);          // f1 takes a pointer to a
                                    // constant Widget object

void f2(Widget const *pw);          // so does f2
```

因为它们都存在于实际的代码中，你应该习惯于这两种形式。

STL iterators（迭代器）以 pointers（指针）为原型，所以一个 iterator 在行为上非常类似于一个 T* pointer（指针）。声明一个 iterator 为 const 就类似于声明一个 pointer（指针）为 const（也就是说，声明一个 T* const pointer（指针））：不能将这个 iterator 指向另外一件不同的东西，但是它所指向的东西本身可以变化。如果你要一个 iterator 指向一个不能变化的东西（也就是一个 const T* pointer（指针）的 STL 对等物），你需要一个

const_iterator:

```
std::vector<int> vec;
...
const std::vector<int>::iterator iter =     // iter acts like a T* const
?vec.begin();
*iter = 10;                                  // OK, changes what iter
points to
++iter;                                  ?   // error! iter is const

std::vector<int>::const_iterator cIter =   ?/ cIter acts like a const T*
?vec.begin();
*cIter = 10;                                 // error! *cIter is const
++cIter;                                     // fine, changes cIter
```

对 const 最强有力的用法来自于它在 function declarations（函数声明）中的应用。在一个 function declaration（函数声明）中，const 既可以用在函数的 return value（返回值）上，也可以用在个别的 parameters（参数）上，对于 member functions（成员函数），还可以用于整个函数。

一个函数返回一个 constant value（常量值），常常可以在不放弃安全和效率的前提下尽可能减少客户的错误造成的影响。例如，考虑? Ú Item 24 中考察的 rational numbers（有理数）的 operator* 函数的声明。

```
class Rational { ... };

const Rational operator*(const Rational& lhs, const Rational& rhs);
```

很多第一次看到这些的程序员会不以为然。为什么 operator* 的结果应该是一个 const object（对象）？因为如果它不是，客户就可以犯下如此暴行：

```
Rational a, b, c;
...
(a * b) = c;                          ?     // invoke operator= on the
                                    ?       // result of a*b!
```

我不知道为什么一些程序员要为两个数的乘积赋值，但是我知道很多程序员这样做也并非不称职。所? 姓庑┛ 赡芾醋砸桓黾虻ㄐ氖淙氪砦螅丁　笳飧隼嘈湍芪灰　阶　偷? bool）：

```
if (a * b = c) ...                        // oops, meant to do a
comparison!
```

如果 a 和 b 是 built-in type（内建类型），这样的代码显而易见是非法的。一个好的 user-defined types（用户自定义类型）的特点就是要避免与 built-ins（内建类型）毫无理由的不和谐（参见 Item 18），而且对我来说允许给两个数的乘积赋值看上去正是毫无理由的。将 operator* 的返回值声明为 const 就可以避免这一点，这就是我们要这样做的理由。

关于 const parameters（参数）没什么特别新鲜之处——它们的行为就像 local（局部）的 const objects（对象），而且无论何时，只要你能，你就应该这样使用。除非你需要改变一个 parameter（参数）或 local object（本地对象）的能力，否则，确保将它声明为 const。它只需要你键入六个字符，就能将你从我们刚刚看到的这个恼人的错误中拯救出来："我想键入 '==' ，但我意外地键入了 '='?。

const member functions（const 成员函数）

member functions（成员函数）被声明为 const 的目的是标明这个 member functions（成员函数）可能会被 const objects（对象）调用。因为两个原因，这样的 member functions（成员函数）非常重要。首先，它使一个 class（类）的 interface（接口）更容易被理解。知道哪个函数可以改变 object（对象）而哪个不可以是很重要的。第二，它们可以和 const objects（对象）一起工作。因为，书写高效代码有一个很重要的方面，就像 Item 20 所解释的，提升一个 C++ 程序的性能的基本方法就是 pass objects by reference-to-const（以传引用给 const 的方式传递一个对象）。这个技术只有在 const member functions（成员函数）和作为操作结果的 const-qualified objects（被 const 修饰的对象）存在时才是可行的。

很多人没有注意到这样的事实，即 member functions（成员函数）在只有 constness（常量性）不同时是可以被 overloaded（重载）的，但这是 C++ 的一个重要特性。考虑一个代表文本块的类：

```
class TextBlock {
public:
?...
?const char& operator[](std::size_t position) const   // operator[] for
?{ return text[position]; }                           // const objects

?char& operator[](std::size_t position)               // operator[] for
?{ return text[position]; }                           // non-const
objects

private:
   std::string text;
};
```

TextBlock 的 operator[]s 可能会这样使用：

```
TextBlock tb("Hello");
std::cout << tb[0];                ??// calls non-const
                                       // TextBlock::operator[]
const TextBlock ctb("World");
std::cout << ctb[0];                   // calls const
TextBlock::operator[]
```

顺便提一下，const objects（对象）在实际程序中最经常出现的是作为这样一个操作的结果：passed by pointer- or reference-to-const（以传指针或者引用给 const 的方式传递）。上面的 ctb 的例子是人工假造的。下面这个例子更真实一些：

```
void print(const TextBlock& ctb)       // in this function, ctb is const
{
?std::cout << ctb[0];                ?/ calls const
TextBlock::operator[]
?...
}
```

通过 overloading（重载） operator[]，而且给不同的版本不同的返回类型，你能对 const 和 non-const 的 TextBlocks 做不同的操作：

```
std::cout << tb[0];                    // fine — reading a
                                ?      // non-const TextBlock

tb[0] = 'x';                           // fine — writing a
                                      ?// non-const TextBlock
```

```
std::cout << ctb[0];                    ?// fine — reading a
                                        ?// const TextBlock

ctb[0] = 'x';                           ?// error! — writing a
                                      ?   // const TextBlock
```

请注意这里的错误只与被调用的 operator[] 的 *return type*（返回类型）有关，而调用 operator[] 本身总是正确的。错误出现在企图为 const char& 赋值的时候，因为它是 const 版本的 operator[] 的 return type（返回类型）。

再请注意 non-const 版本的 operator[] 的 return type（返回类型）是 *reference* to a char（一个 char 的引用）而不是一个 char 本身。如果 operator[] 只是返回一个简单的 char，下面的语句将无法编译：

```
tb[0] = 'x';
```

因为改变一个返回 built-in type（内建类型）的函数的返回值总是非法的。即使它合法，C++ returns objects by value（以传值方式返回对象）这一事实（参见 Item 20）也意味着被改变的是 tb.text[0] 的一个 *copy*（拷贝），而不是 tb.text[0] 自己，这不会是你想要的行为。

让我们为哲学留一点时间。看看一个 member function（成员函数）是 const 意味着什么？有两个主要的概念：*bitwise constness*（二进制位常量性）（也称为 *physical constness*（物理常量性））和 *logical constness*（逻辑常量性）。

bitwise（二进制位）const 派别坚持认为，一个 member function（成员函数），当且仅当它不改变 object（对象）的任何 data members（数据成员）（static（静态的）除外），也就是说如果不改变 object（对象）内的任何 bits（二进制位），则这个 member function（成员函数）就是 const。bitwise constness（二进制位常量性）的一个好处是比较容易监测违例：编译器只需要寻找对 data members（数据成员）的 assignments（赋值）。实际上，bitwise constness（二进制位常量性）就是 C++ 对 constness（常量性）的定义，一个 const member function（成员函数）不被允许改变调用它的 object（对象）的任何 non-static data members（非静态数据成员）。

不幸的是，很多效果上并不是完全 const 的 member functions（成员函数）通过了 bitwise（二进制位）的检验。特别是，一个经常改变某个 pointer（指针）指向的内容的 member function（成员函数）效果上不是 const 的。除非这个 *pointer*（指针）在这个 object（对象）中，否则这个函数就是 bitwise（二进制位）const 的，编译器也不会提出异议。例如，假设我们有一个 TextBlock-like class（类似 TextBlock 的类），因为它需要与一个不知 string objects（对象）为何物的 C API 打交道，所以它需要将它的数据存储为 char* 而不是 string。

```
class CTextBlock {
public:
?...

?char& operator[](std::size_t position) const   // inappropriate (but
bitwise

?{ return pText[position]; }                    ?// const) declaration of
                                                   // operator[]
private:
?char *pText;
};
```

尽管 operator[] 返回 a reference to the object's internal data（一个引向对象内部数据的引用），这个 class（类）还是（不适当地）将它声明为一个 const member function（成员函数）（Item 28 将谈论一个深入的主题）。先将它放到一边，看看 operator[] 的实现，它并没有使用任何手段改变 pText。结果，编译器愉快地生成了 operator[] 的代码，因为毕竟对所有编译器而言，它都是 bitwise（二进制位）const 的，但是我们看看会发生什么：

```
const CTextBlock cctb("Hello");      ?// declare constant object

char *pc = &cctb[0];                   // call the const operator[] to
get a
                                       // pointer to cctb's data

*pc = 'J';                             // cctb now has the value "Jello"
```

这里确实出了问题，你用一个 particular value（确定值）创建一个 constant object（常量对象），然后你只是用它调用了 const member functions（成员函数），但是你还是改变了它的值！

这就引出了 logical constness（逻辑常量性）的概念。这一理论的信徒认为：一个 const member function（成员函数）可能会改变调用它的 object（对象）中的一些 bits（二进制位），但是只能用客户无法察觉的方法。例如，你的 CTextBlock class（类）在需要的时候可以储存文本块的长度：

```
class CTextBlock {
public:

?...

?std::size_t length() const;

private:
?char *pText;
?std::size_t textLength;          ??/ last calculated length of
textblock
?bool lengthIsValid;                 // whether length is currently
valid
};

std::size_t CTextBlock::length() const
{
?if (!lengthIsValid) {
  ?textLength = std::strlen(pText);?// error! can't assign to textLength
  ?lengthIsValid = true;             // and lengthIsValid in a const
?}                                   // member function

?return textLength;
}
```

length 的实现当然不是 bitwise（二进制位）const 的——textLength 和 lengthIsValid 都可能会被改变——但是它还是被看作对 const CTextBlock 对象有效。但编译器不同意，它还是坚持 bitwise constness（二进制位常量性），怎么办呢？

解决方法很简单：利用以 mutable 闻名的 C++ 的 const-related（const 相关）的灵活空间。mutable 将 non-static data members（非静态数据成员）从 bitwise constness（二进制位常量性）的约束中解放出来：

```
class CTextBlock {
public:

?...

?std::size_t length() const;

private:
?char *pText;

?mutable std::size_t textLength;          // these data members may
?mutable bool lengthIsValid;              // always be modified, even in
};                                       ?// const member functions

std::size_t CTextBlock::length() const
{
?if (!lengthIsValid) {
  ?textLength = std::strlen(pText);    ?// now fine
  ?lengthIsValid = true;                 // also fine
?}

?return textLength;
}
```

避免 **const** 和 **non-const member functions**（成员函数）的重复

mutable 对于解决 bitwise-constness-is-not-what-I-had-in-mind（二进制位常量性不太合我的心意）的问题是一个不错的解决方案，但它不能解决全部的 const -related（const 相关）难题。例如，假设 TextBlock（包括 CTextBlock）中的 operator[] 不仅要返回一个适当的字符的 reference（引用），它还要进行 bounds checking（边界检查），logged access information（记录访问信息），甚至 data integrity validation（数据完整性确认），将这些功能都加入到 const 和 non-const 的 operator[] 函数中（不必为我们现在有着非凡长度的 implicitly inline functions（隐含内联函数）而烦恼，参见 Item 30），使它们变成如下这样的庞然大物：

```
class TextBlock {
public:

?...

?const char& operator[](std::size_t position) const
?{
  ?...                                  // do bounds checking
  ?...                                  // log access data
  ?...                                  // verify data integrity
  ?return text[position];
?}

?char& operator[](std::size_t position)
?{
  ?...                                  // do bounds checking
  ?...                                  // log access data
  ?...                                  // verify data integrity
  ?return text[position];
?}
```

```
private:
   std::string text;
};
```

哎呀！你是说 code duplication（重复代码）？还有随之而来的额外的编译时间，维护成本以及代码膨胀等令人头痛之类的事情吗？？比唬　部梢越? bounds checking（边界检查）等全部代码转移到一个单独的 member function（成员函数）（自然是 private（私有）的）中，并让两个版本的 operator[] 来调用它，但是，你还是要重复写出调用那个函数和 return 语句的代码。

你真正要做的是只实现一次 operator[] 的功能，而使用两次。换句话说，你可以用一个版本的 operator[] 去调用另一个版本。并可以为我们 casting away（通过强制转型脱掉）constness（常量性）。

作为一个通用规则，casting（强制转型）是一个非常坏的主意，我会投入整个一个 Item 的篇幅来告诉你不要使用它（Item 27），但是 code duplication（重复代码）也不是什么好事。在当前情况下，const 版本的 operator[] 所做的事也正是 non-const 版本所做的，仅有的不同是它有一个 const-qualified return type（被 const 修饰的返回类型）。在这种情况下，casting away（通过强制转型脱掉）return value（返回类型）的 const 是安全的，因为，无论谁调用 non-const operator[]，首先要有一个 non-const object（对象）。否则，它不能调用一个 non-const 函数。所以，即使需要一个 cast（强制转型），让 non-const operator[] 调用 const 版本也是避免重复代码的安全方法。代码如下，你读了后面的解释后对它的理解可能会更加清晰：

```
class TextBlock {
public:

?...

?const char& operator[](std::size_t position) const     // same as
before
?{
  ?...
  ?...
  ?...
  ?return text[position];
?}

?char& operator[](std::size_t position)         // now just calls const
op[]
?{
  ?return
    ?const_cast<char&>(                         // cast away const on
                                                 // op[]'s return type;
      ?static_cast<const TextBlock&>(*this)     // add const to *this's
type;
        ?[position]                             // call const version of
op[]
    ?);
?}

...

};
```

正如你看到的，代码中有两处 casts（强制转型），而不是一处。我们让 non-const operator[]

调用 const 版本，但是，如果在 non-const operator[] 的内部，我们仅仅是调用 operator[]，那我们将递归调用我们自己。它会进行一百万次甚至更多。为了避免 infinite recursion（无限递归），我们必须明确指出我们要调用 const operator[] ，但是没有直接的办法能做到这一点，于是我们将 *this 从 TextBlock& 的自然类型强制转型到 const TextBlock&。是的，我们使用 cast（强制转型）为它加上了 const！所以我们有两次 casts（强制转型）：第一次是为 *this 加上 const（以便在我们调用 operator[] 时调用它的 const 版本），第二次是从 const operator[] 的 return value（返回值）之中去掉 const。

加上 const 的 cast（强制转型）仅仅是强制施加一次安全的转换（从一个 non-const object（对象）到一个 const object（对象）），所以我们用一个 static_cast 来做。去掉 const 只能经由 const_cast 来完成，所以在这里我们没有别的选择。（在技术上，我们有。一个 C-style cast（C 风格的强制转型）也能工作，但是，就像我在 Item 27 中解释的，这样的 casts（强制转型）很少是一个正确的选择。如果你不熟悉 static_cast 或 const_cast，Item 27 中包含有一个概述。）

在完成其它事情的基础上，我们在此例中调用了一个 operator（操作符），所以，语法看上去有些奇怪。导致其不会赢得选美比赛，但是它根据 const 版本的 operator[] 实现其 non-const 版本而避免 code duplication（代码重复）的方法达到了预期的效果。使用丑陋的语法达到目标是否值得最好由你自己决定，但是?庵指 ？const member function（成员函数）实现它的 non-const 版本的技术却非常值得掌握。

更加值得掌握的是做这件事的反向方法——通过用 const 版本调用 non-const 版本来避免重复——是你不能做的。记住，一个 const member function（成员函数）承诺绝不会改变它的 object（对象）的逻辑状态，但是一个 non-const member function（成员函数）不会做这样的承诺。如果你从一个 const member function（成员函数）调用一个 non-const member function（成员函数），你将面临你承诺不会变化的 object（对象）被改变的风险。这就是为什么使用一个 const member function（成员函数）调用一个 non-const member function（成员函数）是错误的，object（对象）可能会被改变。实际上，那样的代码如果想通过编译，你必须用一个 const_cast 来去掉 *this 的 const，这是一个显而易见的麻烦。而反向的调用——就像我在上面用的——是安全的：一个 non-const member function（成员函数）对一个 object（对象）能够为所欲为，所以调用一个 const member function（成员函数）也没有任何风险。这就是为什么 static_cast 在这种情况下可以工作在 *this 上的原因：这里没有 const-related 危险。

就像在本 Item 开始我所说的，const 是一件美妙的东西。在 pointers（指针）和 iterators（迭代器）上，在 pointers（指针），iterators（迭代器）和 references（引用）涉及到的 object（对象）上，在 function parameters（函数参数）和 return types（返回值）上，在 local variables（局部变量）上，在 member functions（成员函数）上，const 是一个强有力的盟友。只要可能就用它，你会为你所做的感到高兴。

Things to Remember

- 将某些东西声明为 const 有助于编译器发现使用错误。const 能被用于任何 scope（范围）中的 object（对象），用于 function parameters（函数参数）和 return types（返回类型），用于整个 member functions（成员函数）。
- 编译器坚持 bitwise constness（二进制位常量性），但是你应该用 conceptual constness（概念上的常量性）来编程。（此处原文有误，conceptual constness 为作者在本书第二版中对 logical constness 的称呼，正文中的称呼改了，此处却没有改。其实此处还是作者新加的部分，却使用了旧?氖跄铮 郑?——译者注）
- 当 const 和 non-const member functions（成员函数）具有本质上相同的实现的时候，使用 non-const 版本调用 const 版本可以避免 code duplication（代码重复）。

窗体底端

- # Item 4: 确保 objects（对象）在使用前被初始化

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

C++ 看上去在对象的值的初始化方面变化莫测。例如，如果你这样做，

```
int x;
```

在某些情形下，x 会被初始化（为 0），但是在其它情形下，也可能没有。如果你这样做，

```
class Point {
?int x, y;
};

...

Point p;
```

p 的 data members（数据成员）有时会被初始化（为 0），但有时没有。如果你从一个不存在 uninitialized objects（未初始化对象）的语言来到 C++，请注意这个问题，因为它非常重要。

读取一个 uninitialized values（未初始化值）会引起 undefined behavior（未定义行为）。在一些平台上，读一个 uninitialized value（未初始化值）会引起程序中止，更可能的情况是得到一个你所读的那个位置上的 semi-random bits（半随机二进制位），最终导致不可预测的程序行为和恼人的调试。

现在，有一些描述关于什么时候能保证 object initialization（对象初始化）会发生什么时候不能保证的规则。不幸的是，这些规则很复杂——我觉得它复杂得无法记住。通常，如果你使用 C++ 的 C 部分（参见 Item 1），而且 initialization（初始化）可能会花费一些运行时间，它就不能保证发生。如果你使用 C++ 的 non-C 部分，事情会有些变化。这就是为什么一个 array（数组）（来自 C++ 的 C 部分）不能确保它的元素被初始化，但是一个 vector（来自 C++ 的 STL 部分）就能够确保。

处理这种事情的表面不确定状态的最好方法就是总是在使用之前初始化你的对象。对于 built-in types（内建类型）的 non-member objects（非成员对象），需要你手动做这件事。例如：

```
int x = 0;                                  ?// manual initialization of an
int

const char * text = "A C-style string";   // manual initialization of a
                                           ?// pointer (see also Item 3)

double d;                                    // "initialization" by reading
from

std::cin >> d;                              ?// an input stream
```

除此之外的几乎全部情况，initialization（初始化）的重任就落到了 constructors（构造函数）的身上。这里的规则很简单：确保 all constructors（所有的构造函数）都初始化了 object（对象）中的每一样东西。

这个规则很容易遵守，但重要的是不要把 assignment（赋值）和 initialization（初始化）搞混。考虑下面这个表现一个通讯录条目的 class（类）的 constructor（构造函数）：

```
class PhoneNumber { ... };

class ABEntry {                          ?// ABEntry = "Address Book Entry"

public:

?ABEntry(const std::string& name, const std::string& address,
```

```
        ?const std::list<PhoneNumber>& phones);

private:

?std::string theName;

?std::string theAddress;

?std::list<PhoneNumber> thePhones;

?int num TimesConsulted;

};

ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)

{

?theName = name;                          // these are all assignments,

?theAddress = address;                    // not initializations

?thePhones = phones;

?numTimesConsulted = 0;

}
```

这样做虽然使得 ABEntry objects（对象）具有了你所期待的值，但还不是最好的做法。C++ 的规则规定一个 object（对象）的 data members（数据成员）在进入 constructor（构造函数）的函数体之前被初始化。在 ABEntry 的 constructor（构造函数）内，theName，theAddress 和 thePhones 不是 being initialized（被初始化），而是 being assigned（被赋值）。initialization（初始化）发生得更早——在进入 ABEntry 的 constructor（构造函数）的函数体之前，它们的 default constructors（缺省的构造函数）已经被自动调用。但不包括 numTimesConsulted，因为它是一个 built-in type（内建类型）。不能保证它在被赋值之前被初始化。

一个更好的写 ABEntry constructor（构造函数）的方法是用 member initialization list（成员初始化列表）来代替 assignments（赋值）：

```
ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)

: theName(name),
?theAddress(address),                  ?// these are now all
initialization
?thePhones(phones),
?numTimesConsulted(0)

{}                                      ?// the ctor body is now empty
```

这个 constructor（构造函数）的最终结果和前面那个相同，但是通常它有更高的效率。assignment-based（基于赋值）的版本会首先调用 default constructors（缺省构造函数）初始化 theName，theAddress 和 thePhones，然而很快又在 default-constructed（缺省构造）的值之上赋予新值。那些 default constructions（缺省构造函数）所做的工作被浪费了。而 member initialization list（成员初始化列表）的方法避免了这个问题，因为 initialization list（初始化列表）中的 arguments（参数）就可以作为各种 data members（数据成员）的 constructor（构造函数）所使用的 arguments（参数）。在这种情况下，theName 从 name 中 copy-constructed（拷贝构造），theAddress 从 address 中 copy-constructed（拷贝构造），thePhones 从 phones 中 copy-constructed（拷贝构造）。对于大多数类型来说，只调用一次 copy

constructor（拷贝构造函数）的效率比先调用一次 default constructor（缺省构造函数）再调用一次 copy assignment operator（拷贝赋值运算符）的效率要高（有时会高很多）。

对于 numTimesConsulted 这样的 built-in type（内建类型）的 objects（对象），initialization（初始化）和 assignment（赋值）没有什么不同，但为了统一性，最好是经由 member initialization（成员初始化）来 initialize（初始化）每一件东西。类似地，当你只想 default-construct（缺省构造）一个 data member（数据成员）时也可以使用 member initialization list（成员初始化列表），只是不必指定 initialization argument（初始化参数）而已。例如，如果 ABEntry 有一个不取得 parameters（参数）的 constructor（构造函数），它可以像这样实现：

```
ABEntry::ABEntry()
:theName(),                              // call theName's default ctor;
  heAddress(),                          ?// do the same for theAddress;
  hePhones(),                            // and for thePhones;
  umTimesConsulted(0)                    // but explicitly initialize

{}                                       ?// numTimesConsulted to zero
```

因为对于那些在 member initialization list（成员初始化列表）中的，没有 initializers（初始化器）的，user-defined types（用户自定义类型）的 data members（数据成员），编译器会自动调用其 default constructors（缺省构造函数），所以一些程序员会认为上面的方法有些过分。这也不难理解，但是一个方针是：在 initialization list（初始化列表）中总是列出每一个 data member（数据成员），这就可以避免一旦发生疏漏就必须回忆起可能是哪一个 data members（数据成员）没有被初始化。例如，因为 numTimesConsulted 是一个 built-in type（内建类型），如果将它从 member initialization list（成员初始化列表）中删除，就为 undefined behavior（未定义行为）打开了方便之门。

有时，即使是 built-in types（内建类型），initialization list（初始化列表）也必须使用。比如，const 或 references（引用）data members（数据成员）是必须 be initialized（被初始化）的，它们不能 be assigned（被赋值）（参见 Item 5）。为了避免记忆什么时候 data members（数据成员）必须在 member initialization list（成员初始化列表）中初始化，而什么时候又是可选的，最简单的方法就是总是使用 initialization list（初始化列表）。它有时是必须的，而且它通常都比 assignments（赋值）更有效率。

很多 classes（类）有多个 constructors（构造函数），而每一个 constructor（构造函数）都有自己的 member initialization list（成员初始化列表）。如果有很多 data members（数据成员）和/或 base classes（基类），成倍增加的 initialization lists（初始化列表）的存在引起令人郁闷的重复（在列表中）和厌烦（在程序员中）。在这种情况下，不能不讲道理地从列表中删除那些 assignment（赋值）和 true initialization（真正的初始化）一样工作的 data members（数据成员）项目，而是将 assignments（赋值）移到一个单独的（当然是 private（私有）的）函数中，以供所有 constructors（构造函数）调用。这个方法对于那些 true initial values（真正的初始值）是从文件中读入或从数据库中检索出来的 data members（数据成员）特别有帮助。然而，通常情况下，true member initialization（真正的成员初始化）（经由一个 initialization list（初始化列表））比经由 assignment（赋值）来进行的 pseudo-initialization（假初始化）更可取。

C++ 并非变幻莫测的方面是一个 object（对象）的数据被初始化的顺序。这个顺序总是相同的：base classes（基类）在 derived classes（派生类）之前被初始化（参见 Item 12），在一个 class（类）内部，data members（数据成员）按照它们被声明的顺序被初始化。例如，在 ABEntry 中，theName 总是首先被初始化，theAddress 是第二个，thePhones 第三，numTimesConsulted 最后。即使它们在 member initialization list（成员初始化列表）中以一种不同的顺序排列（这不幸合法），这依然是成立的。为了避免读者?煜　约耙恍├：　磺宓男形　鹆砻蟮目赡苄裕琲itialization list（初始化列表）中的 members（成员）的排列顺序应该总是与它们在 class（类）中被声明的顺序保持一致。

一旦处理了 built-in types（内建类型）的 non-member objects（非成员对象）的显式初始化，而且确保你的 constructors（构造函数）使用 member initialization list（成员初始化列表）初始化了它的 base classes（基类）和 data members（数据成员），那就只剩下一件事情需要费心了。那就是——深呼吸先——定义在不同 translation units（转换单元）中的 non-local static objects（非局部静态对象）的 initialization（初始化）的顺序。

让我们一片一片地把这个词组拆开。

一个 static object（静态对象）的生存期是从它创建开始直到程序结束。stack and heap-based objects（基于堆栈的对象）就被排除在外了。包括 global objects（全局对象），objects defined at namespace scope（定义在命名空间范围内的对象），objects declared static inside classes（在类内部声明为静态的对象），objects declared static inside functions（在函数内部声明为静态的对象）和 objects declared static at file scope（在文件范围内被声明为静态的对象）。static objects inside functions（在函数内部的静态对象）以 local static objects（局部静态对象）（因为它局部于函数）为人所知，其它各种 static objects（静态对象）以 non-local static objects（非局部静态对象）为人所知。程序结束时 static objects（静态对象）会自动销毁，也就是当 main 停止执行时会自动调用它们的 destructors（析构函数）。

一个 translation unit（转换单元）是可以形成一个单独的 object file（目标文件）的 source code（源代码）。基本上是一个单独的 source file（源文件），再加上它全部的 #include 文件。

我们关心的问题是这样的：包括至少两个分别编译的 source files（源文件），每一个中都至少包含一个 non-local static object（非局部静态对象）（也就是说，global（全局）的，at namespace scope（命名空间范围）的，static in a class（类内）的或 at file scope（文件范围）的 object（对象））。实际的问题是这样的：如果其中一个 translation unit（转换单元）内的一个 non-local static object（非局部静态对象）的 initialization（初始化）用到另一个 translation unit（转换单元）内的non-local static object（非局部静态对象），它所用到的 object（对象）可能没有被初始化，因为 the relative order of initialization of non-local static objects defined in different translation units is undefined（定义在不同转换单元内的非局部静态对象的初始化的相对顺序是没有定义的）。

一个例子可以帮助我们。假设你有一个 FileSystem class（类），可以使 Internet 上的文件看起来就像在本地。因为你的 class（类）使得世界看起来好像只有一个单独的 file system（文件系统），你可以在 global（全局）或 namespace（命名空间）范围内创建一个专门的 object（对象）来代表这个单独的 file system（文件系统）：

```
class FileSystem {                  ?// from your library

public:
?...
?std::size_t numDisks() const;        // one of many member functions
?...
};

extern FileSystem tfs;              ?// object for clients to use;
                                    ?// "tfs" = "the file system"
```

一个 FileSystem object（对象）绝对是举足轻重的，所以在 tfs object（对象）被创建之前就使用将会损失惨重。（此段和下段原文有误，根据作者网站勘误更改——译者注。）

现在假设一些客户为一个 file system（文件系统）中的目录创建了一个 class（类），他们的 class（类）使用了 tfs object（对象）：

```
class Directory {                       // created by library client

public:

   Directory( params );
?...
};

Directory::Directory( params )

{
?...
?std::size_t disks = tfs.numDisks();    // use the tfs object
?...
```

```
}
```

更进一步，假设这个客户决定为临时文件创建一个单独的 Directory object（对象）：

```
Directory tempDir( params );       ?  ?// directory for temporary files
```

现在 initialization order（初始化顺序）的重要性变得明显了：除非 tfs 在 tempDir 之前初始化，否则，tempDir 的 constructor（构造函数）就会在 tfs 被初始化之前试图使用它。但是，tfs 和 tempDir 是被不同的人于不同的时间在不同的 source files（源文件）中创建的——它们是定义在不同 translation units（转换单元）中的 non-local static objects（非局部静态对象）。你怎么能确保 tfs 一定会在 tempDir 之前被初始化呢？

你不能。重申一遍，the relative order of initialization of non-local static objects defined in different translation units is undefined（定义在不同转换单元内的非局部静态对象的初始化的相对顺序是没有定义的）。这是有原因的。决?
?non-local static objects（非局部静态对象）的“恰当的”±
初始化顺序是困难的，非常困难，以至于无法完成。在最常见的形式下——多个 translation units（转换单元）和 non-local static objects（非局部静态对象）通过 implicit template instantiations（隐式模板实例化）产生（这本身可能也是经由 implicit template instantiations（隐式模板实例化）引起的）——不仅不可能确定一个正确的 initialization（初始化）顺序，甚至不值得去寻找可能确定正确顺序的特殊情况。

幸运的是，一个小小的设计改变从根本上解决了这个问题。全部要做的就是将每一个 non-local static object（非局部静态对象）移到它自己的函数中，在那里它被声明为 static（静态）。这些函数返回它所包含的 objects（对象）的引用。客户可以调用这些函数来代替直接涉及那些 objects（对象）。换一种说法，就是用 local static objects（局部静态对象）取代 non-local static objects（非局部静态对象）。（aficionados of design patterns（设计模式迷们）会认出这是 Singleton 模式的通用实现）。（实际上，这只不过是一个 Singleton 实现的一部分。我在这个 Item 中忽略的 Singleton 的核心部分是为了预防创建一个特定类型的多个 objects。）
（括号中的文字原文无，参照作者网站勘误添加——译者注。）

这个方法建立在 C++ 保证 local static objects（局部静态对象）的初始化发生在因为调用那个函数而第一次遇到那个 object（对象）的 definition（定义）时候。所以，如果你用调用返回 references to local static objects（局部静态对象的引用）的函数的方法取代直接访问 non-local static objects（非局部静态对象）的方法，你将确保你取回的 references（引用）引向 initialized objects（已初始化的对象）。作为一份额外收获，如果你从不调用这样一个仿效 non-local static object（非局部静态对象）的函数，你就不会付出创建和销毁这个 object（对象）的代价，而一个 true non-local static objects（真正的非局部静态对象）则不会有这样的效果。

以下就是这项技术在 tfs 和 tempDir 上的应用：

```
class FileSystem { ... };           // as before

FileSystem& tfs()                   // this replaces the tfs object; it
could be
{                                   // static in the FileSystem class
?static FileSystem fs;             // define and initialize a local
static object
?return fs;                       ?// return a reference to it
}

class Directory { ... };           ?// as before

Directory::Directory( params )     ?// as before, except references to
tfs are
{                                   // now to tfs()
?...
?std::size_t disks = tfs().numDisks();
?...
}

Directory& tempDir()               ?// this replaces the tempDir object;
```

```
it
{                                       // could be static in the Directory
class
?static Directory td;                ?// define/initialize local static
object
?return td;                          ?// return reference to it
}
```

这个改良系统的客户依然可以按照他们已经习惯的方法编程，只是他们现在应该用 tfs() 和 tempDir() 来代替 tfs 和 tempDir。也就是说，他们应该使用返回 references to objects（对象引用）的函数来代替使用 objects themselves（对象自身）。

按照以下步骤来写 reference-returning functions（返回引用的函数）总是很简单：在第 1 行定义并初始化一个 local static object（局部静态对象），在第 2 行返回它。这样的简单使它们成为 inlining（内联化）的完美的候选者，特别是在它们被频繁调用的时候（参见 Item 30 ）。在另一方面，这些函数包含 static object（静态对象）的事实使它们在 multithreaded systems（多线程系统）中会出现问题。更进一步，任何种类的 non-const static object（非常量静态对象）—— local（局部）的或 non-local（非局部）的——在 multiple threads（多线程）存在的场合都会发生麻烦。解决这个麻烦的方法之一是在程序的 single-threaded（单线程）的启动部分手动调用所有的 reference-returning functions（返回引用的函数）。以此来避免 initialization-related（与初始化相关）的混乱环境。

当然，用 reference-returning functions（返回引用的函数）来防止 initialization order problems（初始化顺序问题）的想法首先依赖于你的 objects（对象）有一个合理的 initialization order（初始化顺序）。如果你有一个系统，其中 object A 必须在 object B 之前初始化，但是 A 的初始化又依赖于 B 已经被初始化，你将遇到问题，坦白地讲，你遇到大麻烦了。然而，如果你避开了这种病态的境遇，?佚锸枋龅姆椒a岷芎玫匚 惴 瘢 辽僭?single-threaded applications（单线程应用）中是这样。

避免在初始化之前使用 objects（对象），你只需要做三件事。首先，手动初始化 built-in types（内建类型）的 non-member objects（非成员对象）。第二，使用 member initialization lists（成员初始化列表）初始化一个 object（对象）的所有部分。最后，在设计中绕过搞乱定义在分离的 translation units（转换单元）中的 non-local static objects（非局部静态对象）initialization order（初始化顺序）的不确定性。

Things to Remember

- 手动初始化 built-in type（内建类型）的 objects（对象），因为 C++ 只在某些时候才会自己初始化它们。
- 在 constructor（构造函数）中，用 member initialization list（成员初始化列表）代替函数体中的 assignment（赋值）。initialization list（初始化列表）中 data members（数据成员）的排列顺序要与它们在 class（类）中被声明的顺序相同。
- 通过用 local static objects（局部静态对象）代替 non-local static objects（非局部静态对象）来避免跨 translation units（转换单元）的 initialization order problems（初始化顺序问题）。

窗体底端

Chapter 2. Constructors（构造函数），Destructors（析构函数）与 Assignment Operators（赋值运算符）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

几乎每一个你自己写的 class（类）都会有一个或多个 constructors（构造函数），一个 destructor（析构函数）和一个 copy assignment operator（拷贝赋值运算符）。不要惊奇，那是些就像你的面包黄油一样的函数，它们控制着基本的操作，如? 唇丁桓鲂碌? object（对象）并确保已被初始化，消除一个 object（对象）并确保它被完全清除，以及为 object（对象）赋予一个新值。这些函数中出现错误，将导致你的 classes（类）出现影响深远的，而且令人不快的反弹，所以保证它们正确是生死攸关的事情。本章中，我将? 匀绾巫樽罢庑┲魑　桓? well-formed classes（良好的类）的中枢骨干的函数提供一些指导。

本章包括以下内容，点击打开：

由于程序的原因，本文件未被完整保存。

- # Item 5: 了解 C++ 为你偷偷地加上和调用了什么函数

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

一个 empty class（空类）什么时候将不再是 empty class（空类）？答案是当 C++ 搞定了它。如果你自己不声明一个 copy constructor（拷贝构造函数），一个 copy assignment operator（拷贝赋值运算符）和一个 destructor（析构函数），编译器就会为这些东西声明一个它自己的版本。此外，如果你自己根本没有声明 constructor（构造函数），编译器就会为你声明一个 default constructor（缺省构造函数）。所有这些函数都被声明为 public 和 inline（参见 Item 30）。作为结果，如果你写

```
class Empty{};
```

在本质上和你这样写是一样的：

```
class Empty {

public:

?Empty() { ... }                                ?// default constructor

?Empty(const Empty& rhs) { ... }                ?// copy constructor

?~Empty() { ... }                                 // destructor — see below
                                                   // for whether it's virtual

?Empty& operator=(const Empty& rhs) { ... } // copy assignment operator

};
```

这些函数只有在它们被需要的时候才会生成，但是并不需要做太多的事情，就会用到它们。下面的代?
牖崇偈姑恳桓龌　　桑?

```
Empty e1;                                     // default constructor;
                                            ?// destructor

Empty e2(e1);                                 // copy constructor

e2 = e1;                                    ?// copy assignment operator
```

假设编译器为你写了这些函数，那么它们做些什么呢？default constructor（缺省构造函数）和 destructor（析构函数）主要是给编译器一个地方放置 "behind the scenes" code（"°幕后"±代码）的，诸如 base classes（基类）和 non-static data members（非静态数据成员）的 constructors（构造函数）和 destructor（析构函数）的调用。注意，生成的 destructor（析构函数）是 non-virtual（非虚拟）的（参见 Item 7），除非它所在的 class（类）是从一个 base class（基类）继承而来，而 base class（基类）自己声明了一个 virtual destructor（虚拟析构函数）（这种情况下，函数的 virtualness（虚拟性）来自 base class（基类））。

对于 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符），compiler-generated versions（编译器生成版本）只是简单地从 source object（源对象）拷贝每一个 non-static data member（非静态数据成员）到 target object（目标对象）。例如，考虑一个 NamedObject template（模板），它允许你将名字和类型为 T 的 objects（对象）联系起来的：

```
template<typename T>
class NamedObject {
public:
?NamedObject(const char *name, const T& value);
?NamedObject(const std::string& name, const T& value);
```

```
?...

private:
?std::string nameValue;
?T objectValue;
};
```

因为 NamedObject 中声明了一个 constructors（构造函数），编译器就不会再生成一个 default constructor（缺省构造函数）。这一点很重要，它意味着如果你小心地设计一个 class（类），使它需要 constructor arguments（构造函数参数），你就不必顾虑编译器会不顾你的决定，轻率地增加一个不需要参数的 constructors（构造函数）。

NamedObject 既没有声明 copy constructor（拷贝构造函数）也没有声明 copy assignment operator（拷贝赋值运算符），所以编译器将生成这些函数（如果需要它们的话）。看，这就是 copy constructor（拷贝构造函数）的用法：

```
NamedObject<int> no1("Smallest Prime Number", 2);

NamedObject<int> no2(no1);                  // calls copy constructor
```

编译器生成的 copy constructor（拷贝构造函数）一定会用 no1.nameValue 和 no1.objectValue 分别初始化 no2.nameValue 和 no2.objectValue。nameValue 的类型是 string，标准 string 类型有一个 copy constructor（拷贝构造函数），所以将通过以 no1.nameValue 作为参数调用 string 的 copy constructor（拷贝构造函数）初始化 no2.nameValue。而另一方面，NamedObject<int>::objectValue 的类型是 int（因为在这个 template instantiation（模板实例化）中 T 是 int），而 int 是 built-in type（内建类型），所以将通过拷贝 no1.objectValue 的每一个二进制位初始化 no2.objectValue。

编译器为 NamedObject<int> 生成的 copy assignment operator（拷贝赋值运算符）本质上也会有同样的行为，但是，通常情况下，只有在结果代码合法而?矣幸桓龌侠淼目衫斫獾那珊鲜保噩ompiler-generated（编译器生成）的 copy assignment operator（拷贝赋值运算符）才会有我所描述的行为方式。如果这两项检测中的任一项失败了，编译?鹘 芫 愕?class（类）生成一个 operator=。

例如，假设 NamedObject 如下定义，nameValue 是一个 reference to a string（引向一个字符串的引用），而 objectValue 是一个 *const* T：

```
template<class T>
class NamedObject {
public:
?// this ctor no longer takes a const name, because nameValue
?// is now a reference-to-non-const string. The char* constructor
?// is gone, because we must have a string to refer to.
?NamedObject(std::string& name, const T& value);

?...                                  // as above, assume no
                                     ?// operator= is declared

private:
?std::string& nameValue;              // this is now a reference
?const T objectValue;                ?// this is now const
};
```

现在，考虑这里会发生什么：

```
std::string newDog("Persephone");

std::string oldDog("Satch");

NamedObject<int> p(newDog, 2);                // when I originally wrote
```

```
this, our
                                                    // dog Persephone was about
to
                                                    // have her second birthday
NamedObject<int> s(oldDog, 36);                  ?// the family dog Satch
(from my
                                                    // childhood) would be 36
if she
                                                    // were still alive

p = s;                                              // what should happen to
                                                    // the data members in p?
```

assignment（赋值）之前，p.nameValue 和 s.nameValue 都引向 string objects（对象），虽然并非同一个。那个 assignment（赋值）对 p.nameValue 产生了什么影响呢？assignment（赋值）之后，p.nameValue 所引向的 string 是否就是 s.nameValue 所引向的那一个呢，也就是说，reference（引用）本身被改变了？如果是这样，就违反了常规，因为 C++ 并没有提供使一个 reference（引用）引向另一个 objects（对象）的方法。换一种思路，是不是 p.nameValue 所引向的那个 string objects（对象）被改变了，从而影响了其他 objects（对象）—— pointers（指针）或 references（引用）持续指向的那个 string，也就是，赋值中并没有直接涉及到的对象？这是 compiler-generated（编译器生成）的 copy assignment operator（拷贝赋值运算符）应该做的事情吗？

面对这个难题，C++ 拒绝编译代码。如果你希望一个包含 reference member（引用成员）的 class（类）支持 assignment（赋值），你必须自己定义 copy assignment operator（拷贝赋值运算符）。对于含有 const members（const 成员）的 classes（类），编译器会有类似的行为（就像上面那个改变后的 class（类）中的 objectValue）。改变 const members（const 成员）是不合法的，所以编译器隐式生成的 assignment function（赋值函数）无法确定该如何对待它们。最后，如果 base classes（基类）将 copy assignment operator（拷贝赋值运算符）声明为 private，编译器拒绝为从它继承的 derived classes（派生类）生成 implicit copy assignment operators（隐式拷贝赋值运算符）。毕竟，编译器为派生类生成的 copy assignment operator（拷贝赋值运算符）也要处理其 base class parts（基类构件）（参见 Item 12），但如果这样做，它们当然无法调用那些 derived classes（派生类）无权调用的 member functions（成员函数）。

Things to Remember

- 编译器可以隐式生成一个 class（类）的 default constructor（缺省构造函数），copy constructor（拷贝构造函数），copy assignment operator（拷贝赋值运算符）和 destructor（析构函数）。

窗体底端

- # Item 6: 如果你不想使用 compiler-generated functions（编译器生成函数），就明确拒绝

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

房地产代理商出售房屋，服务于这样的代理商的软件系统自然要有一个 class（类）来表示被出售的房屋：

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

每一个房地产代理商都会很快指出，每一件房产都是独特的——没有两件是完全一样的。在这种情况下，为 HomeForSale object（对象）做一个 *copy*（拷贝）的想法就令人不解了。你怎么能拷贝一个独一无二的东西呢？因此最好让类似这种企图拷贝 HomeForSale object（对象）的行为不能通过编译：

```
HomeForSale h1;

HomeForSale h2;

HomeForSale h3(h1);                  // attempt to copy h1 — should
                                    ?// not compile!

h1 = h2;                            ?// attempt to copy h2 — should
                                    ?// not compile!
```

唉，防止这种编译的方法并非那么简单易懂。通常，如果你不希望一个 class（类）支持某种功能，你可以简单地不声明赋予它这种功能的函数。这个策略对于 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）不起作用，因为，就象 Item 5 中指出的，如果你不声明它们，而有人又想调用它们，编译器就会替你声明它们。

这就限制了你。如果你不声明 copy constructor（拷贝构造函数）或 copy assignment operator（拷贝赋值运算符），编译器也可以替你生成它们。你的 class（类）还是会支持 copying（拷贝）。另一方面，如果你声明了这些函数，你的 class（类）依然会支持 copying（拷贝）。而我们此时的目的却是 *prevent* copying（防止拷贝）！

解决这个问题的关键是所有的编译器生成的函数都是 public（公有）的。为了防止生成这些函数，你必须自己声明它们，但是你没有理由把它们声明为 public（公有）的。相反，应该将 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）声明为 *private*（私有）的。通过显式声明一个 member function（成员函数），可以防止编译器生成它自己的版本，而且将这个函数声明为 private（私有）的，可以防止别人调用它。

通常，这个方案并不十分保险，因为 member（成员）和 friend functions（友元函数）还是能够调用你的 private 函数。换句话说，除非你十分聪明地不 *define*（定义）它们。那么，当有人不小心地调用了它们，在 link-time（连接时）会出现错误。这个窍门——声明 member functions（成员函数）为 private 却故意不去实现它——确实很好，在 C++ 的 iostreams 库里，就有几个类用此方法 prevent copying（防止拷贝）。比如，看一下你用的标准库的实现中 ios_base，basic_ios 和 sentry 的 definitions（定义），你就会看到 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）被声明为 private 而且没有被定义的情况。

将这个窍门用到 HomeForSale 上，很简单：

```
class HomeForSale {
public:
?...

private:
?...
?HomeForSale(const HomeForSale&);           ?// declarations only
?HomeForSale& operator=(const HomeForSale&);
};
```

你会注意到，我省略了 functions' parameters
（函数参数）的名字。这不是必须的，只是一个普通的惯例。毕竟，函数不会被实现，更少会被用到?
惺裁幢匾　付ú问　　兀?

对于上面的 class definition（类定义），编译器将阻止客户拷贝 HomeForSale objects
（对象）的企图，如果你不小心在 member（成员）或 friend function
（友元函数）中这样做了，连接程序会提出抗议。

将 link-time error
（连接时错误）提前到编译时间也是可行的（早发现错误毕竟比晚发现好），通过不在
HomeForSale 本身中声明 copy constructor（拷贝构造函数）和 copy assignment operator
（拷贝赋值运算符）为 private，而是在一个为 prevent copying（防止拷贝）而特意设计的
base class（基类）中声明。这个 base class（基类）本身非常简单：

```
class Uncopyable {
protected:                                  // allow construction
?Uncopyable() {}                          ?// and destruction of
?~Uncopyable() {}                           // derived objects...

private:
?Uncopyable(const Uncopyable&);             // ...but prevent copying
?Uncopyable& operator=(const Uncopyable&);
};
```

为了阻止拷贝 HomeForSale objects（对象），我们现在必须让它从 Uncopyable 继承：

```
class HomeForSale: private Uncopyable {       // class no longer
?...                                       ?  ?// declares copy ctor or
};                                            ?// copy assign. operator
```

这样做是因为，如果有人——甚至是 member（成员）或 friend function（友元函数）——
试图拷贝一个 HomeForSale objects（对象），编译器将试图生成一个 copy constructor
（拷贝构造函数）和一个 copy assignment operator（拷贝赋值运算符）。就象 Item 12
解释的，这些函数的 compiler-generated versions（编译器生成版）会试图调用 base class
（基类）的相应函数，而这些调用将被拒绝，因为在 base class（基类）中，拷贝操作是 private
（私有）的。

Uncopyable 的实现和使用包含一些微妙之处，比如，从 Uncopyable 继承不必是 public
（公有）的（参见 Item 32 和 39），而且 Uncopyable 的 destructor（析构函数）不必是
virtual（虚拟）的（参见 Item 7）。因为 Uncopyable 不包含数据，所以它符合 Item 39 描述的
empty base class optimization（空基类优化）的条件，但因为它是 base class
（基类），此项技术的应用不能引入 multiple inheritance（多继承）（参见 Item 40
）。反过来说，multiple inheritance（多继承）有时会使 empty base class optimization
（空基类优化）失效（还是参见 Item 39

）。通常，你可以忽略这些微妙之处，而且仅仅像此处演示的这样来使用 Uncopyable，因为它的工作就像在做广告。你还可以使用在 Boost（参见 Item 55）中的一个可用版本。那个 class（类）名为 noncopyable。那是一个好东西，我只是发现那个名字有点儿 un-（不 ）嗯 *non*natural（非自然）。

Things to Remember

- 为了拒绝编译器自动提供的机能，将相应的 member functions（成员函数）声明为 private，而且不要给出 implementations（实现）。使用一个类似 Uncopyable 的 base class（基类）是方法之一。

窗体底端

由于程序的原因，本文件未被完整保存。

Item 7: 在 polymorphic base classes（多态基类）中将 destructors（析构函数）声明为 virtual（虚拟）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

有很多方法取得时间，所以有必要建立一个 TimeKeeper base class（基类），并为不同的计时方法建立 derived classes（派生类）：

```
class TimeKeeper {
public:
?TimeKeeper();
?~TimeKeeper();
?...
};

class AtomicClock: public TimeKeeper { ... };

class WaterClock: public TimeKeeper { ... };

class WristWatch: public TimeKeeper { ... };
```

很多客户只是想简单地取得时间而不关心如何计算的细节，所以一个 factory function（工厂函数）——返回 a base class pointer to a newly-created derived class object（一个指向新建派生类对象的基类指针）的函数——可以被用来返回一个指向 timekeeping object（计时对象）的指针：

```
TimeKeeper* getTimeKeeper();        // returns a pointer to a dynamic-
                                    // ally allocated object of a class
                                    // derived from TimeKeeper
```

与 factory function（工厂函数）的惯例一致，getTimeKeeper 返回的 objects（对象）是建立在 heap（堆）上的，所以为了避免泄漏内存和其它资源，每一个返回的 objects（对象）被完全 deleted 是很重要的。

```
TimeKeeper *ptk = getTimeKeeper();?// get dynamically allocated object
                                  ?// from TimeKeeper hierarchy

...                                 // use it

delete ptk;                        ?// release it to avoid resource leak
```

Item 13 解释了为什么依赖客户执行删除任务是 error-prone（错误倾向），Item 18 解释了 factory function（工厂函数）的 interface（接口）应该如何改变以防止普通的客户错误，但这些在这里都是次要的，因为在这个 Item 中，我们将精力集中于上面的代码中一个更基本的缺陷：即使客户做对了每一件事，也无法预知程序将如何运?
?

问题在于 getTimeKeeper 返回一个 pointer to a derived class object（指向派生类对象的指针）（比如 AtomicClock），那个 object（对象）经由一个 base class pointer（基类指针）（也就是一个 TimeKeeper* pointer）被删除，而且这个 base class（基类）（TimeKeeper）有一个 non-virtual destructor（非虚拟析构函数）。祸端就在这里，因为 C++ 规定：当一个 derived class object（派生类对象）通过使用一个 pointer to a base class with a non-virtual destructor（指向带有非虚拟析构函数的基类的指针）被删除，则结果是未定义的。运行时比较典型的后果是 derived part of the object（这个对象的派生部分）不会被销毁。如果 getTimeKeeper 返回一个指向 AtomicClock object（对象）的指针，则 object（对象）的 AtomicClock 部分（也就是在 AtomicClock class 中声明的 data members（数据成员））很可能不会被销毁，AtomicClock 的 destructor（析构函数）也不会运行。然而，base class part（基类部分）（也就是

TimeKeeper 部分）很可能已被销毁，这就导致了一个古怪的 "partially destroyed" object（"部分被销毁"±对象）。这是一个导致泄漏资源，破坏数据结构以及消耗大量调试时间的绝妙方法。

消除这个问题很简单：给 base class（基类）一个 virtual destructor（虚拟析构函数）。于是，删除一个 derived class object（派生类对象）的时候就有了你所期望的行为。将销毁 entire object（整个对象），包括全部的 derived class parts（派生类构件）：

```
class TimeKeeper {
public:
?TimeKeeper();
?virtual ~TimeKeeper();
?...
};

TimeKeeper *ptk = getTimeKeeper();

...

delete ptk;                          ?   // now behaves correctly
```

类似 TimeKeeper 的 base classes（基类）一般都包含除了 destructor（析构函数）以外的其它 virtual functions（虚拟函数），因为 virtual functions（虚拟函数）的目的就是允许 derived class implementations（派生类实现）的定制化（参见 Item 34）。例如，TimeKeeper 可以有一个 virtual functions（虚拟函数）getCurrentTime，它在各种不同的 derived classes（派生类）中有不同的实现。几乎所有拥有 virtual functions（虚拟函数）的 class（类）差不多都应该有一个 virtual destructor（虚拟析构函数）。

如果一个 class（类）不包含 virtual functions（虚拟函数），这经常预示不打算将它作为 base class（基类）使用。当一个 class（类）不打算作为 base class（基类）时，将 destructor（析构函数）虚拟通常是个坏主意。考虑一个表现二维空间中的点的 class（类）：

```
class Point {                            // a 2D point
public:
?Point(int xCoord, int yCoord);
?~Point();

private:
?int x, y;
};
```

如果一个 int 占用 32 bits，一个 Point object 正好适用于 64-bit 的寄存器。而且，这样一个 Point object 可以被作为一个 64-bit 的量传递给其它语言写的函数，比如 C 或者 FORTRAN。如果 Point 的 destructor（析构函数）被虚拟，情况就完全不一样了。

virtual functions（虚拟函数）的实现要求 object（对象）携带额外的信息，这些信息用于在运行时确定该 object（对象）应该调用哪一个 virtual functions（虚拟函数）。典型情况下，这一信息具有一种被称为 vptr（"virtual table pointer"）（虚拟函数表指针）的指针的形式。vptr 指向一个被称为 vtbl（"virtual table"）（虚拟函数表）的 array of function pointers（函数指针数组），每一个带有 virtual functions（虚拟函数）的 class（类）都有一个相关联的 vtbl。当在一个 object（对象）上调用 virtual functions（虚拟函数）时，实际的被调用函数通过下面的步骤确定：找到 object（对象）的 vptr 指向的 vtbl，然后在 vtbl 中寻找合适的 function pointer（函数指针）。

virtual functions（虚拟函数）是如何实现的细节并不重要。重要的是如果 Point class 包含一个 virtual functions（虚拟函数），这个类型的 object（对象）的大小就会增加。在一个 32-bit 架构中，它们将从 64 bits（相当于两个 ints）长到 96 bits（两个 ints 加上 vptr）；在一个 64-bit 架构中，它们可能从 64 bits 长到 128 bits，因为在这样的架构中指针的大小是 64 bits 的。为 Point 加上 vptr 将会使它的大小增长 50-100%！Point object（对象）不再适合 64-bit 寄存器。而且，Point object（对象）在 C++ 和其它语言（比如 C）中，看起来不再具有相同的结构，因为它们在其它语言中的对应物没有 vptr。结果，Points 不再可能传入其它语言写成的函数或从其中传出，除非你为 vptr 做出明确的补偿，而这是它自己的实现细节并因此失去可移植性。

最终结果就是无故地将所有 destructors（析构函数）声明为 virtual（虚拟），和从不把它们声明为 virtual（虚拟）一样是错误的。实际上，很多人总结过这条规则：declare a virtual destructor in a class if and only if that class contains at least one virtual function（当且仅当一个类中包含至少一个虚拟函数时，则在类中声明一个虚拟析构函数）。

甚至在完全没有 virtual functions（虚拟函数）时，也有可能纠缠于 non-virtual destructor（非虚拟析构函数）问题。例如，标准 string 类型不包含 virtual functions（虚拟函数），但是被误导的程序员有时将它当作 base class（基类）使用：

```
class SpecialString: public std::string {    // bad idea! std::string has a
?...                                          // non-virtual destructor
};
```

一眼看上去，这可能无伤大雅，但是，如果在程序的某个地方因为某种原因，你将一个 pointer-to-SpecialString（指向 SpecialString 的指针）转型为一个 pointer-to-string（指向 string 的指针），然后你将 delete 施加于这个 string pointer（指针），你就立刻被放逐到 undefined behavior（未定义行为）的领地：

```
SpecialString *pss =   new SpecialString("Impending Doom");

std::string *ps;
...
ps = pss;                                  // SpecialString* => std::string*
...
delete ps;                                ?// undefined! In practice,
                                          ?// *ps's SpecialString resources
                                          ?// will be leaked, because the
                                          ?// SpecialString destructor won't
                                          ?// be called.
```

同样的分析可以适用于任何缺少 virtual destructor（虚拟析构函数）的 class（类），包括全部的 STL container（容器）类型（例如，vector，list，set，tr1::unordered_map（参见 Item 54）等）。如果你受到从 standard container（标准容器）或任何其它带有 non-virtual destructor（非虚拟析构函数）的 class（类）继承的诱惑，一定要挺住！（不幸的是，C++ 不提供类似 Java 的 final classes（类）或 C# 的 sealed classes（类）的 derivation-prevention mechanism（防派生机制）。）

有时候，给一个 class（类）提供一个 pure virtual destructor（纯虚拟析构函数）能提供一些便利。回想一下，pure virtual functions（纯虚拟函数）导致 abstract classes（抽象类）——不能被实例化的 classes（类）（也就是说你不能创建这个类型的 objects（对象））。然而，有时候，你有一个 class（类），你希望它是抽象的，但没有任何 pure virtual functions（纯虚拟函数）。怎么办呢？因为一个 abstract classes（抽象类）注定要被用作 base class（基类），又因为一个 base class（基类）应该有一个 virtual destructor（虚拟析构函数），还因为一个 pure virtual functions（纯虚拟函数）产生一个 abstract classes（抽象类），好了，解决方案很简单：在你想要变成抽象的 class（类）中声明一个 pure virtual destructor（纯虚拟析构函数）。这是一个例子：

```
class AWOV {                              ?// AWOV = "Abstract w/o Virtuals"
public:
?virtual ~AWOV() = 0;                   ?// declare pure virtual destructor
};
```

这个 class（类）有一个 pure virtual functions（纯虚拟函数），所以它是抽象的，又因为它有一个 virtual destructor（虚拟析构函数），所以你不必担心析构函数问题。这是一个螺旋。然而，你必须为 pure virtual destructor（纯虚拟析构函数）提供一个 definition（定义）：

```
AWOV::~AWOV() {}                          ?// definition of pure virtual  ?dtor
```

destructors（析构函数）的工作方式是：most derived class（层次最低的派生类）的 destructor（析构函数）最先被调用，然后调用每一个 base class（基类）的 destructors（析构函数）。编译器会生成一个从它的 derived classes（派生类）的 destructors（析构函数）对 ~AWOV 的调用，所以你不得不确保为函数提供一个函数体。如果你不这样做，连接程序会提出抗议。

为 base classes（基类）提供 virtual destructor（虚拟析构函数）的规则仅仅适用于 polymorphic base classes（多态基类）—— base classes（基类）被设计成允许通过 base class

interfaces（基类接口）对 derived class types（派生类类型）进行操作。TimeKeeper 就是一个 polymorphic base classes（多态基类），因为即使我们只有类型为 TimeKeeper 的 pointers（指针）指向它们的时候，我们也期望能够操作 AtomicClock 和 WaterClock objects（对象）。

并非所有的 base classes（基类）都被设计用于 polymorphically（多态）。例如，无论是 standard string type（标准 string 类型），还是 STL container types（STL 容器类型）全被设计成 base classes（基类），可没有哪个是 polymorphic（多态）的。一些 classes（类）虽然被设计用于 base classes（基类），但并非被设计用于 polymorphically（多态）。这样的 classes（类）——例如 Item 6 中的 Uncopyable 和标准库中的 input_iterator_tag（参见 Item 47）——没有被设计成允许经由 base class interfaces（基类接口）对 derived class objects（派生类对象）进行操作。所以它们就不需要 virtual destructor（虚拟析构函数）。

Things to Remember

- polymorphic base classes（多态基类）应该声明 virtual destructor（虚拟析构函数）。如果一个 class（类）有任何 virtual functions（虚拟函数），它就应该有一个 virtual destructor（虚拟析构函数）。
- 不是设计用来作为 base classes（基类）或不是设计用于 polymorphically（多态）的 classes（类）就不应该声明 virtual destructor（虚拟析构函数）。

窗体底端

- # Item 8: 防止因为 exceptions（异常）而离开 destructors（析构函数）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

C++ 并不禁止从 destructors（析构函数）中引发 exceptions（异常），但是它坚决地阻止这样的实践。至于有什么好的理由，考虑：

```
class Widget {
public:
?...
?~Widget() { ... }              // assume this might emit an exception
};

void doSomething()
{
?std::vector<Widget> v;
?...
}                                ?// v is automatically destroyed here
```

当 vector v 被销毁时，它有责任销毁它包含的所有 Widgets。假设 v 中有十个 Widgets，在第一个的析构过程中，抛出一个 exception（异常）。其它 9 个 Widgets 仍然必须被销毁（否则它们持有的所有资源将被泄漏），所以 v 应该调用它们的 destructors（析构函数）。但是假设在这个调用期间，第二个 Widget 的 destructors（析构函数）又抛出一个 exception（异常）。现在同时有两个活动的 exceptions（异常），对于 C++ 来说，这太多了。在非常巧合的条件下产生这样两个同时活动的异常，程序的执行会终止或者引发 undefined behavior（未定义行为）。在本例中，将引发 undefined behavior（未定义行为）。使用任何其它的标准库 container（容器）（比如，list，set），任何 TR1（参见 Item 54）中的 container（容器），甚至是一个 array（数组），都可能会引发同样的 undefined behavior（未定义行为）。也并非必须是 containers（容器）或 arrays（数组）才会陷入麻烦。程序过早终止或 undefined behavior（未定义行为）是 destructors（析构函数）引发 exceptions（异常）的结果，即使没有使用 containers（容器）和 arrays（数组）也会如此。C++ 不喜欢引发 exceptions（异常）的 destructors（析构函数）。

这比较容易理解，但是如果你的 destructor（析构函数）需要执行一个可能失败而抛出一个 exception（异常）的操作，该怎么办呢？例如，假设你与一个数据库连接类一起工作：

```
class DBConnection {
public:
?...

?static DBConnection create();      ?// function to return
                                       // DBConnection objects; params
                                       // omitted for simplicity

?void close();                      ?// close connection; throw an
};                                     // exception if closing fails
```

为了确保客户不会忘记在 DBconnection objects（对象）上调用 close，一个合理的主意是为 DBConnection 建立一个 resource-managing class（资源管理类），在它的 destructor（析构函数）中调用 close。这样的 resource-managing classes（资源管理类）将在 Chapter 3（第三章）中一探究竟，但在这里，只要认为这样一个 class（类）的 destructor

（析构函数）看起来像这样就足够了：

```
class DBConn {                            ?// class to manage DBConnection
public:                                     // objects
?...
?~DBConn()                               ?// make sure database connections
?{                                          // are always closed
   db.close();
?}
private:
?DBConnection db;
};
```

它允许客户像这样编程：

```
{                                           // open a block
   DBConn dbc(DBConnection::create());?// create DBConnection object
                                          ?// and turn it over to a DBConn
                                          ?// object to manage
?..                                       ?// use the DBConnection object
                                          ?// via the DBConn interface
}                                           // at end of block, the DBConn
                                          ?// object is destroyed, thus
                                          ?// automatically calling close on
                                          ?// the DBConnection object
```

只要能成功地调用 close 就可以了，但是如果这个调用导致一个 exception（异常），DBConn 的 destructor（析构函数）将传播那个 exception（异常），也就是说，它将离开 destructor（析构函数）。这就产生了问题，因为 destructor（析构函数）抛出了一个烫手的山芋。

有两个主要的方法避免这个麻烦。DBConn 的 destructor（析构函数）能：

- **Terminate the program** if close tHRows（如果 close 抛出异常就终止程序），一般是通过调用 abort：

```
DBConn::~DBConn()
{
 try { db.close(); }
 catch (...) {
   make log entry that the call to close failed;
   std::abort();
 }
}
```

如果在析构的过程遭遇到错误后程序不能继续运行，这就是一个合理的选择。它有一个好处是：如果?市棠? destructor（析构函数）传播 exception（异常）可能会导致 undefined behavior（未定义行为），这样就能防止它发生。也就是说，调用 abort 就可以预先防止 undefined behavior（未定义行为）。

- **Swallow the exception** arising from the call to close（抑制这个对 close 的调用造成的异常）：

```
DBConn::~DBConn()
{
 try { db.close(); }
 catch (...) {
    ?make log entry that the call to close failed;
 }
}
```

通常，swallowing exceptions（抑制异常）是一个不好的主意，因为它会隐瞒重要的信息——*something failed*（某事失败了）！然而，有些时候，swallowing exceptions（抑制异常）比冒程序过早终止或 undefined behavior（未定义行为）的风险更可取。程序必须能够在遭遇到一个错误并忽略之后还能继续可靠地运行，这?拍艹晌　桓佳尚械难═瘪?

这些方法都不太吸引人。它们的问题首先在于程序无法对引起 close 抛出 exception（异常）的条件做出回应。

一个更好的策略是设计 DBConn 的 interface（接口），以使它的客户有机会对可能会发生的问题做出回应。例如，DBConn 能够自己提供一个 close 函数，从而给客户一个机会去处理从那个操作中发生的 exception（异常）。它还能保持对它的 DBConnection 是否已被 closed 的跟踪，如果没有就在 destructor（析构函数）中自己关闭它。这样可以防止连接被泄漏。如果在 DBConnection（原文如此，严重怀疑此处应为 DBConn ——译者注）的 destructor（析构函数）中对 close 的调用失败，无论如何，我们还可以再返回到终止或者抑制。

```
class DBConn {
public:
?...

?void close()                                    // new function for
?{                                              ?// client use
  ?db.close();
  ?closed = true;
?}

?~DBConn()
?{
   if (!closed) {
   try {                                          ?// close the
connection
     db.close();                                  ?// if the client
didn't
   }
   catch (...) {                                  ?// if closing fails,
     make log entry that call to close failed;    ?// note that and
     ...                                          ?// terminate or
swallow
   }
?}

private:
?DBConnection db;
?bool closed;
};
```

将调用 close 的责任从 DBConn 的 destructor（析构函数）移交给 DBConn 的客户（同时在 DBConn 的 destructor（析构函数）中包含一个"候补"调用）可能会作为一种肆无忌惮地推卸责任的做法而使你吃惊。你甚至可以把它看作对 Item 18 中关于使 interfaces（接口）易于正确使用的建议的违背。实际上，这都不正确。如果一个操作可能失败而抛出一个 exception（异常），而且可能有必要处理这个 exception（异常），这个 exception（异常）就 *has to come from some non-destructor function*（必须来自非析构函数）。这是因为 destructor（析构函数）引发 exception（异常）是危险的，永远都要冒着程序过早终止或 undefined behavior（未定义行为）的风险。在本例中，让客户自己调用 close 并不是强加给他们的负担，而是给他们一个时机去应付错误，否则他们将没有机会做出回应。如果他?

钦也坏娇捎玫交　幔α蛐硪蛭　　窍嘈挪换嵊写砦笳娴姆⑸　　强梢院睢运　　揽? DBConn 的 destructor（析构函数）为他们调用 close。如果一个错误恰恰在那时发生——如果由 close 抛出——如果 DBConn 抑制了那个 exception（异常）或者终止了程序，他们将无处诉苦。毕竟，他们无处着手处理问题，他们将不再使用它。

Things to Remember

- destructor（析构函数）应该永不引发 exceptions（异常）。如果 destructor（析构函数）调用了可能抛出异常的函数，destructor（析构函数）应该捕捉所有异常，然后抑制它们或者终止程序。
- 如果 class（类）客户需要能对一个操作抛出的 exceptions（异常）做出回应，则那个 class（类）应该提供一个常规的函数（也就是说，non-destructor（非析构函数））来完成这个操作。

窗体底端

- # Item 9:?绝不要在 construction（构造）或 destruction（析构）期间调用 virtual functions（虚拟函数）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

我以这个概述开始：你不应该在 construction（构造）或 destruction（析构）期间调用 virtual functions
（虚拟函数），因为这样的调用不会如你想象那样工作，而且它们做的事情保证会让你很郁闷。如果?阕 ？Java 或 C# 程序员，也请你密切关注本 Item，因为在 C++
急转弯的地方，那些语言也紧急转了一个弯。

假设你有一套模拟股票交易的 class hierarchy
（类继承体系），例如，购入订单，出售订单等。对于这样的交易来说可供审查是非常重要的，所以?看我桓鼋灰锥韵蟊淮唇ǎ 谝桓錾蟛槿罩局芯托枰 唇T桓鲅嘤Φ奶跄俊O旅媸且桓佳雌鹄此坪龌侠淼慕饩鑫侍獾姆椒ǎ?

```
class Transaction {                                    // base class for all
public:                                                // transactions
?Transaction();

?virtual void logTransaction() const = 0;       ?// make type-dependent
                                                        ?// log entry
?...
};

Transaction::Transaction()                              ?// implementation of
{                                                         // base class ctor
?...
?logTransaction();                                       // as final action, log
this
}                                                         // transaction

class BuyTransaction: public Transaction {              ?// derived class
public:
?virtual void logTransaction() const;                  ?// how to log trans-
                                                        ?// actions of this type
?...
};

class SellTransaction: public Transaction {               // derived class
public:
  irtual void logTransaction() const;                     // how to log trans-
                                                        ?// actions of this type
?...
};
```

考虑执行这行代码时会发生什么：

```
BuyTransaction b;
```

很明显一个 BuyTransaction 的 constructor（构造函数）会被调用，但是首先，一个 Transaction 的 constructor（构造函数）必须先被调用，derived class objects

（派生类对象）中的 base class parts（基类构件）先于 derived class parts（派生类构件）被构造。Transaction 的 constructor（构造函数）的最后一行调用 virtual functions（虚拟函数） logTransaction，但是结果会让你大吃一惊，被调用的 logTransaction 版本是在 Transaction 中的那一个，而不是 BuyTransaction 中的那一个——即使被创建的 object （对象）类型是 BuyTransaction。base class construction（基类构造）期间，virtual functions（虚拟函数）从来不会 go down（向下匹配）到 derived classes（派生类）。取而代之的是，那个 object （对象）的行为好像它就是 base type（基类型）。非正式地讲，base class construction（基类构造）期间，virtual functions（虚拟函数）被禁止。

这个表面上看起来匪夷所思的行为存在一个很好的理由。因为 base class constructors（基类构造函数）在 derived class constructors（派生类构造函数）之前执行，当 base class constructors（基类构造函数）运行时，derived class data members（派生类数据成员）还没有被初始化。如果 base class construction（基类构造）期间 virtual functions（虚拟函数）的调用 went down（向下匹配）到 derived classes（派生类），derived classes（派生类）的函数差不多总会涉及到 local data members（局部数据成员），但是那些 data members（数据成员）至此还没有被初始化。这就会为 undefined behavior（未定义行为）和通宵达旦的调试噩梦开了一张通行证。调用涉及到一个 object（对象）还没有被初始化的构件自然是危险的，所以 C++ 告诉你此路不通。

实际上还有比这更基本的原理。在一个 derived class object（派生类对象）的 base class construction（基类构造）期间，object（对象）的类型是 base class（基类）的类型。不仅 virtual functions（虚拟函数）会解析到 base class（基类），而且用到 runtime type information（运行时类型信息）的语言构件（例如，dynamic_cast（参见 Item 27）和 typeid），也会将那个 object（对象）视为 base class type（基类类型）。在我们的例子中，当 Transaction 的 constructor（构造函数）运行到初始化一个 BuyTransaction object（对象）的 base class（基类）部分时，那个 object（对象）的是 Transaction 类型。C++ 的每一个构件将以如下眼光来看待它，而且这种看法是合理的：这个 object（对象）的 BuyTransaction-specific 的构件还没有被初始化，所以对它们视若无睹是最安全的。直到 derived class constructor（派生类构造函数）的执行开始之前，一个 object（对象）不会成为一个 derived class object（派生类对象）。

同样的推理也适用于 destruction（析构）。一旦 derived class destructor（派生类析构函数）运行，这个 object（对象）的 derived class data members（派生类数据成员）就呈现为未定义的值，所以 C++ 就将它们视为不再存在。在进入 base class destructor（基类析构函数）时，这个 object（对象）就成为一个 base class object（基类对象），C++ 的所有构件—— virtual functions（虚拟函数），dynamic_casts 等——都以此看待它。

在上面的示例代码中，Transaction 的 constructor（构造函数）造成了对一个 virtual functions（虚拟函数）的一次直接调用，是对本 Item 的指导建议的显而易见的违背。这一违背是如此显见，以致一些编译器会给出一个关于它的警告。（?硪恍┬蚤换帷 2 渭? Item 53 对于警告的讨论。）即使没有这样的一个警告，这个问题也几乎肯定会在运行之前暴露出来，因为 logTransaction 函数在 Transaction 中是 pure virtual（纯虚拟）的。除非它被定义（不太可能，但确实可能——参见 Item 34），否则程序将无法连接：连接程序无法找到 Transaction::logTransaction 的必要的实现。

在 construction（构造）或 destruction（析构）期间调用 virtual functions（虚拟函数）的问题并不总是如此容易被察觉。如果 Transaction 有多个 constructors（构造函数），每一个都必须完成一些相同的工作，软件工程为避免代码重复，将共通的 initialization（初始化）代码，包括对 logTransaction 的调用，放入一个 private non-virtual initialization function（私有非虚拟初始化函数）中，叫做 init：

```
class Transaction {
public:
?Transaction()
```

```
?{ init(); }                                          // call to
non-virtual...

?virtual void logTransaction() const = 0;
?...

private:
?void init()
?{
  ?...
  ?logTransaction();                                  // ...that calls a
virtual!
?}
};
```

这个代码在概念上和早先那个版本相同，但是它更阴险，因为一般来说它会躲过编译器和连接程序的?r埂T谖庵榭鲋拢 蛭? logTransaction 在 Transaction 中是 pure virtual（纯虚的），在 pure virtual（纯虚）被调用时，大多数 runtime systems（运行时系统）会异常中止那个程序（一般会对此结果给出一条消息）。然而，如果 logTransaction 在 Transaction 中是一个 "normal" virtual function（“常规”虚拟函数）（也就是说，not pure virtual（非纯虚拟的）），而且带有一个实现，那个版本将被调用，程序会继续一路小跑，让你想象不出为?裁丛? derived class object（派生类对象）被创建的时候会调用 logTransaction 的错误版本。避免这个问题的唯一办法就是确保你的 constructors（构造函数）或 destructors（析构函数）决不在被创建或销毁的 object（对象）上调用 virtual functions（虚拟函数），它们所调用的全部函数也要服从同样的约束。

但是，你如何确保在每一次 Transaction hierarchy（继承体系）中的一个 object（对象）被创建时，都会调用 logTransaction 的正确版本呢？显然，在 Transaction constructor(s)（构造函数）中在这个 object（对象）上调用 virtual functions（虚拟函数）的做法是错误的。

有不同的方法来解决这个问题。其中之一是将 Transaction 中的 logTransaction 转变为一个 non-virtual function（非虚拟函数），这就需要 derived class constructors（派生类构造函数）将必要的日志信息传递给 Transaction constructor（构造函数）。那个函数就可以安全地调用 non-virtual（非虚拟）的 logTransaction。如下：

```
class Transaction {
public:
?explicit Transaction(const std::string& logInfo);

?void logTransaction(const std::string& logInfo) const;    // now a non-
                                                            // virtual
func
?...
};

Transaction::Transaction(const std::string& logInfo)
{
?...
?logTransaction(logInfo);                                   // now a non-
}                                                          ?// virtual
call

class BuyTransaction: public Transaction {
public:
  uyTransaction( parameters )
```

```
 : Transaction(createLogString( parameters ))              ?// pass log
info
?{ ... }                                                  ?// to base
class
   ...                                                      //
constructor

private:
?static std::string createLogString( parameters );
};
```

换句话说，由于你不能在 base classes（基类）的 construction（构造）过程中使用 virtual functions（虚拟函数）向下匹配，你可以改为让 derived classes（派生类）将必要的构造信息上传给 base class constructors（基类构造函数）作为补偿。

在此例中，注意 `BuyTransaction` 中那个 (private) static 函数 `createLogString` 的使用。使用一个辅助函数创建一个值传递给 base class constructors（基类构造函数），通常比通过在 member initialization list（成员初始化列表）给 base class（基类）它所需要的东西更加便利（也更加具有可读性）。将那个函数做成 `static`，就不会有偶然触及到一个新生的 `BuyTransaction` object（对象）的 as-yet-uninitialized data members（仍未初始化的数据成员）的危险。这很重要，因为实际上那些 data members（数据成员）处在一个未定义状态，这就是为什么在 base class（基类）construction（构造）和 destruction（析构）期间调用 virtual functions（虚拟函数）不能首先向下匹配到 derived classes（派生类）的原因。

Things to Remember

- 在 construction（构造）或 destruction（析构）期间不要调用 virtual functions（虚拟函数），因为这样的调用不会转到比当前执行的 constructor（构造函数）或 destructor（析构函数）所属的 class（类）更深层的 derived class（派生类）。

窗体底端

- # Item 11: 在 operator= 中处理 assignment to self（自赋值）

  作者：Scott Meyers

  译者：fatalerror99 (iTePub's Nirvana)

  发布：http://blog.csdn.net/fatalerror99/

  当一个 object（对象）赋值给自己的时候就发生了一次 assignment to self（自赋值）：

```
class Widget { ... };

Widget w;
...

w = w;                                  // assignment to self
```

  这看起来很愚蠢，但它是合法的，所以应该确信客户会这样做。另外，assignment（赋值）也并不总是那么容易辨别。例如，

```
a[i] = a[j];                            // potential assignment to self
```

  如果 i 和 j 有同样的值就是一个 assignment to self（自赋值），还有

```
*px = *py;                              // potential assignment to self
```

  如果 px 和 py 碰巧指向同一个东西，这也是一个 assignment to self（自赋值）。这些不太明显的 assignments to self（自赋值）是由 aliasing（别名）（有不止一个方法引用一个 object（对象））造成的。通常，使用 references（引用）或者 pointers（指针）操作相同类型的多个 objects（对象）的代码需要考虑那些 objects（对象）可能相同的情况。实际上，如果两个 objects（对象）来自同一个 hierarchy（继承体系），甚至不需要公开声明，它们就是相同类型的，因为一个 base class（基类）的 reference（引用）或者 pointer（指针）也能够引向或者指向一个 derived class（派生类）类型的 object（对象）：

```
class Base { ... };

class Derived: public Base { ... };

void doSomething(const Base& rb,                    // rb and *pd might
actually be
                 Derived* pd);                      // the same object
```

  如果你遵循 Items 13 和 14 的建议，你应该总是使用 objects（对象）来管理 resources（资源），而且你应该确保那些 resource-managing objects（资源管理对象）被拷贝时行为良好。在这种情况下，你的 assignment operators（赋值运算符）在你没有考虑自赋值的时候可能也是 self-assignment-safe（自赋值安全）的。然而，如果你试图自己管理 resources（资源）（如果你正在写一个 resource-managing class（资源管理类），你当然必须这样做），你可能会落入在你用完一个 resource（资源）之前就已意外地将它释放的陷阱。例如，假设你创建了一个 class（类），它持有一个指向动态分配 bitmap（位图）的 raw pointer（裸指针）：

```
class Bitmap { ... };

class Widget {
?...

private:
?Bitmap *pb;                                    // ptr to a
heap-allocated object
};
```

  下面是一个表面上看似合理 operator= 的实现，但如果出现 assignment to

self（自赋值）则是不安全的。（它也不是
exception-safe（异常安全）的，但我们要过一会儿才会涉及到它。）

```
Widget&
Widget::operator=(const Widget& rhs)             ?// unsafe impl. of
operator=
{
?delete pb;                                     ?// stop using current
bitmap
?pb = new Bitmap(*rhs.pb);                        // start using a copy
of rhs's bitmap

?return *this;                                    // see Item 10
}
```

这里的 self-assignment（自赋值）问题在 operator= 的内部，*this（赋值的目标）和 rhs 可能是同一个 object（对象）。如果它们是，则那个 delete 不仅会销毁 current object（当前对象）的 bitmap（位图），也会销毁 rhs 的 bitmap（位图）。在函数的结尾，Widget ——通过 assignment to self（自赋值）应该没有变化——发现自己持有一个指向已删除 object（对象）的指针！

防止这个错误的传统方法是在 operator= 的开始处通过 identity test（一致性检测）来阻止 assignment to self（自赋值）：

```
Widget& Widget::operator=(const Widget& rhs)
{
?if (this == &rhs) return *this;    // identity test: if a
self-assignment,
                                   ?// do nothing
?delete pb;
?pb = new Bitmap(*rhs.pb);

?return *this;
}
```

这个也能工作，但是我在前面提及那个 operator= 的早先版本不仅仅是
self-assignment-unsafe（自赋值不安全）的，它也是
exception-unsafe（异常不安全）的，而且这个版本还有异常上的麻烦。详细地说，如果 "new Bitmap" 表达式引发一个 exception（异常）（可能因为供分配的内存不足或者因为 Bitmap 的 copy constructor（拷贝构造函数）抛出一个异常），Widget 将以持有一个指向被删除的 Bitmap
的指针而告终。这样的指针是有毒的，你不能安全地删除它们。你甚至不能安全地读取它们。你对它?俏丅荒茏龅陌踩　氖虑榇蟾啪褪腔ǖ汛罅康牡魇跃　　炊隙ㄋ　瞧鹨蛴谗睦铎?

幸亏，使 operator= exception-safe（异常安全）一般也同时弥补了它的
self-assignment-safe（自赋值安全）。这就导致了更加通用的处理
self-assignment（自赋值）问题的方法就是忽略它，而将焦点集中于达到 exception safety（异常安全）。Item 29 更加深入地探讨了 exception safety（异常安全），但是在本 Item 中，已经足以看出，在很多情况下，仔细地调整一下语句的顺序就可以得到
exception-safe（异常安全）（同时也是
self-assignment-safe（自赋值安全））的代码。例如，在这里，我们只要注意不要删除 pb，直到我们拷贝了它所指向的目标之后：

```
Widget& Widget::operator=(const Widget& rhs)
{
?Bitmap *pOrig = pb;                  // remember original pb
?pb = new Bitmap(*rhs.pb);            // make pb point to a copy of *pb
?delete pOrig;                        // delete the original pb

?return *this;
}
```

现在，如果 "new Bitmap" 抛出一个 exception（异常），pb（以及它所在的 Widget

）的遗迹没有被改变。甚至不需要 identity test（一致性检测），这里的代码也能处理 assignment to self（自赋值），因为我们做了一个原始 bitmap（位图）的拷贝，删除原始 bitmap（位图），然后指向我们作成的拷贝。这可能不是处理 self-assignment（自赋值）的最有效率的做法，但它能够工作。

如果你关心效率，你可以在函数开始处恢复 identity test（一致性检测）。然而，在这样做之前，先问一下自己，你认为 self-assignments（自赋值）发生的频率是多少，因为这个检测不是免费午餐。它将使代码（源代码和目标代码）有少量增大，而且它将在控制流中引入一个分支，这两点都会降低运行速度。例如，instruction prefetching（指令预读），caching（缓存）和 pipelining（流水线操作）的效力都将被降低。

另一个可选的手动排列 operator= 中语句顺序以确保实现是 exception- and self-assignment-safe（异常和自赋值安全）的方法是使用被称为 "copy and swap" 的技术。这一技术和 exception safety（异常安全）关系密切，所以将在 Item 29 中描述。然而，这是一个写 operator= 的足够通用的方法，值得一看，这样一个实现看起来通常就像下面这样：

```
class Widget {
?...
?void swap(Widget& rhs);        // exchange *this's and rhs's data;
?...                            // see Item 29 for details
};

Widget& Widget::operator=(const Widget& rhs)
{
?Widget temp(rhs);              // make a copy of rhs's data

?swap(temp);                    // swap *this's data with the copy's
?return *this;
}
```

在这个主题上的一个变种利用了如下事实：（1）一个 clsaa（类）的 copy assignment（拷贝赋值运算符）可以被声明为 take its argument by value（以传值方式取得它的参数）；（2）通过传值方式传递某些东西以做出它的一个 copy（拷贝）（参见 Item 20）：

```
Widget& Widget::operator=(Widget rhs)    // rhs is a copy of the object
{                                        // passed in — note pass by val

?swap(rhs);                             ?// swap *this's data with
                                         ?// the copy's
?return *this;
}
```

对我个人来说，我担心这个方法在灵活的祭坛上牺牲了清晰度，但是通过将拷贝操作从函数体中转移?讲问　墓乖熘校　惺蹦苁贡嘁肫鞑　　行　实拇　氲挂彩鞘率怠?

Things to Remember

- 当一个 object（对象）被赋值给自己的时候，确保 operator= 是行为良好的。技巧包括比较 source（源）和 target objects（目标对象）的地址，关注语句顺序，和 copy-and-swap。
- 如果两个或更多 objects（对象）相同，确保任何操作多于一个 object（对象）的函数行为正确。

- # Item 12: 拷贝一个 object（对象）的所有 parts（构件）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

在设计良好的 object-oriented systems（面向对象系统）中，封装了 object（对象）的内部构件，只有两个拷贝 objects（对象）的函数：被恰当地称为 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）。我们将它们统称为 copying functions（拷贝函数）。Item 5 讲述了如果需要，编译器会生成 copying functions（拷贝函数），而且阐明了编译器生成的版本正像你所期望的：它们拷贝被拷贝 object（对象）的全部数据。

当你声明了你自己的 copying functions（拷贝函数），你就是在告诉编译器你不喜欢缺省实现中的某些东西。编译器对此好像怒发冲冠，而且它们会用一种古怪的方式报复：当你的实现几乎可以确定是错误的时，它们偏偏不告诉你?£

考虑一个代表 customers（顾客）的 class（类），这里的 copying functions（拷贝函数）是手写的，以便将对它们的调用记入日志：

```
void logCall(const std::string& funcName);        // make a log entry

class Customer {
public:
?...
?Customer(const Customer& rhs);
?Customer& operator=(const Customer& rhs);
?...

private:
?std::string name;
};
Customer::Customer(const Customer& rhs)
: name(rhs.name)                                  // copy rhs's data
{
?logCall("Customer copy constructor");
}

Customer& Customer::operator=(const Customer& rhs)
{
?logCall("Customer copy assignment operator");

?name = rhs.name;                                   // copy rhs's data

?return *this;                                   ?// see Item 10
}
```

这里的每一件事看起来都不错，实际上也确实不错——直到 Customer 中加入了其它 data member（数据成员）：

```
class Date { ... };        // for dates in time

class Customer {
public:
?...                      // as before

private:
?std::string name;
?Date lastTransaction;
};
```

在这里，已有的 copying functions（拷贝函数）只进行了 partial copy（部分拷贝）：它们拷贝了 customers（顾客）的 name，但没有拷贝他的 lastTransaction。然而，大多数编译器即使是在 maximal warning level（最高警告级别）（参见 Item 53），对此也是一声不吭。这是它们在对你自己写 copying functions（拷贝函数）进行报复。你拒绝了它们写的 copying functions（拷贝函数），所以即使你的代码是不完善的，它们也不告诉你。结论显而易见：如果你为一个 class（类）增加了一个 data member（数据成员），你必须确保你也更新了 copying functions（拷贝函数）。（你还必须更新 class（类）中的全部 constructors（构造函数）（参见 Items 4 和 45）以及任何非标准形式的 operator=（Item 10 给出了一个例子）。如果你忘记了，编译器未必会提醒你。）

这个问题最为迷惑人的情形之一是它会通过 inheritance（继承）发生。考虑：

```
class PriorityCustomer: public Customer {                ?// a derived
class
public:
   ...
   PriorityCustomer(const PriorityCustomer& rhs);
   PriorityCustomer& operator=(const PriorityCustomer& rhs);
   ...

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

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
?logCall("PriorityCustomer copy assignment operator");

?priority = rhs.priority;

?return *this;

}
```

PriorityCustomer 的 copying functions（拷贝函数）看上去好像是拷贝了 PriorityCustomer 中的每一样东西，但是再看一下。是的，它拷贝了 PriorityCustomer 声明的 data member（数据成员），但是每个 PriorityCustomer 还包括一份它从 Customer 继承来的 data members（数据成员）的拷贝，而那些 data members（数据成员）根本没有被拷贝！PriorityCustomer 的 copy constructor（拷贝构造函数）没有指定传递给它的 base class constructor（基类构造函数）的参数（也就是说，在它的 member initialization list（成员初始化列表）中没有提及 Customer），所以，PriorityCustomer object（对象）的 Customer 部分被 Customer 的不取得 arguments（实参）的 constructor（构造函数）—— default constructor（缺省构造函数）初始化。（假设它有，如果没有，代码将无法编译。）那个 constructor（构造函数）为 name 和 lastTransaction 进行一次 default initialization（缺省初始化）。

对于 PriorityCustomer 的 copy assignment operator（拷贝赋值运算符），情况只是稍微有些不同。它不会试图用任何方法改变它的 base class data members（基类数据成员），所以它们将保持不变。

无论何时，你打算自己为一个 derived class（派生类）写 copying functions（拷贝函数）时，你必须注意同时拷贝 base class parts（基类构件）。当然，那些构件一般是 private（私有）的（参见 Item 22），所以你不能直接访问它们。derived class（派生类）的 copying functions（拷贝函数）必须调用与它们相对应的 base class functions（基类函数）：

```
PriorityCustomer::PriorityCustomer(const PriorityCustomer& rhs)
:   ?Customer(rhs),                     // invoke base class copy ctor
?priority(rhs.priority)
{
?logCall("PriorityCustomer copy constructor");
}

PriorityCustomer&
PriorityCustomer::operator=(const PriorityCustomer& rhs)
{
?logCall("PriorityCustomer copy assignment operator");

?Customer::operator=(rhs);            // assign base class parts
?priority = rhs.priority;

?return *this;
}
```

本 Item 标题中的 "copy all parts" 的含义现在应该清楚了。当你写一个 copying functions（拷贝函数），需要保证（1）拷贝所有 local data members（局部数据成员）以及（2）调用所有 base classes（基类）中的适当的 copying function（拷贝函数）。

实际上，两个 copying functions（拷贝函数）经常有相似的函数体，而这一点可能吸引你试图通过用一个函数调用另一个来避免 code duplication（代码重复）。你希望避免 code duplication（代码重复）的愿望值得肯定，但是用一个 copying functions（拷贝函数）调用另一个来实现它是错误的。

用 copy assignment operator（拷贝赋值运算符）调用 copy constructor（拷贝构造函数）是没有意义的，因为你这样做就是试图去构造一个已经存在的 object（对象）。这太荒谬了，甚至没有一种语法来支持它。有一种语法看起来好像能让你这样做，？ 导噬夏阕霾坏剑 褂幸恢钟锃ú捎糜聋氏姆椒尸庑 觯 窃谀持痔登 禄岫云苹的愕? object（对象）。所以我不打算给你看任何那样的语法。无条件地接受这个观点：不要用 copy assignment operator（拷贝赋值运算符）调用 copy constructor（拷贝构造函数）。

尝试一下另一种相反的方法——用 copy constructor（拷贝构造函数）调用 copy assignment operator（拷贝赋值运算符）——这同样是荒谬的。一个 constructor（构造函数）初始化新的 objects（对象），而一个 assignment operator（赋值运算符）只能应用于已经初始化过的 objects（对象）。借助构造过程给一个 object（对象）assignment（赋值）将意味着对一个 not-yet-initialized object（尚未初始化的对象）做一些事，而这些事只有用于 initialized object（已初始化对象）才有意义。简直是胡搞！禁止尝试。

作为一种代替，如果你发现你的 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）有相似的代码，通过创建一个供两者调用的第三方 member function（成员函数）来消除重复。这样一个函数一般是 private（私有）的，而且经常叫做 init。这一策略是在 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）中消除 code duplication（代码重复）的安全的，被证实过的方法。

Things to Remember

- copying functions（拷贝函数）应该保证拷贝一个 object（对象）的所有 data members（数据成员）以及所有的 base class parts（基类构件）。
- 不要试图依据一个 copying functions（拷贝函数）实现另一个。作为代替，将通用功能放入一个供双方调用的第三方函数。

窗体底端

- # Item 13: 使用 objects（对象）管理资源

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

假设我们和一个模拟投资（例如，股票，债券等）的库一起工作，各种各样的投资形式从一个 root class（根类）Investment 继承出来：

```
class Investment { ... };           ?// root class of hierarchy of
                                      // investment types
```

进一步假设这个库为我们提供特定 Investment objects（对象）的方法是通过一个 factory function（工厂函数）（参见 Item 7）达成的：

```
Investment* createInvestment();      ?/ return ptr to dynamically
allocated
                                ?     // object in the Investment
hierarchy;
                                ?     // the caller must delete it
                                ?     // (parameters omitted for
simplicity)
```

就像注释指出的，当 createInvestment 函数返回的 object（对象）不再使用时，由 createInvestment 的调用者负责删除它。那么，考虑，写一个函数 f 来履行这个职责：

```
void f()
{
?Investment *pInv = createInvestment();          // call factory function
?...                                            ?// use pInv
?delete pInv;                                     // release object
}
```

这个看上去没什么问题，但是有几种 f 在删除它从 createInvestment 得到的 investment object（对象）时失败的情况。有可能在这个函数的 "..." 部分的某处有一个提前出现的 return 语句。如果这样一个 return 被执行，控制流程就再也无法到达 delete 语句。还可能发生的一个类似情况是如果 createInvestment 的使用和 delete 在一个循环里，而这个循环以一个 continue 或 goto 语句提前退出。还有，"..." 中的一些语句可能抛出一个 exception（异常）。如果这样，控制流程也不会到达那个 delete 。无论那个 delete 被如何跳过，我们泄漏的不仅仅是容纳 investment object（对象）的内存，还包括那个 object（对象）持有的任何资源。

当然，小心谨慎地编程能防止诸如此类的错误，但考虑到这些代码可能会随着时间的流逝而发生变化? N 硕匀砑 形 ぃ 恍┫丝赡芑嵩诿挥型耆 盐斩哉飧龌 淖试垂芾聿呗缘钠渌 糠值挠跋斓? 情况下增加一个 return 或 continue 语句。尤有甚者，f 的 "..." 部分可能调用了一个从不惯于抛出 exception（异常）的函数，但是在它被"改良"后突然这样做了。信赖 f 总能到达它的 delete 语句根本靠不住。

为了确保 createInvestment 返回的资源总能被释放，我们需要将那些资源放入一个 object（对象）中，这个 object（对象）的 destructor（析构函数）在控制流程离开 f 的时候会自动释放资源。实际上，这只是本 Item 背后的思想的一半：通过将资源放到 objects（对象）的内部，我们可以依赖 C++ 的 automatic destructor invocation（自动的析构函数调用）来确保资源被释放。（过一会儿我们要讨论本 Item 的思想的另一半。）

多数资源都是在堆上 dynamically allocated（动态分配）的，在一个单独的 block（块）或 function（函数）内使用，而且应该在控制流程离开那个 block（块）或 function（函数）的时候被释放。标准库的 auto_ptr 正是为这种情形量体裁衣的。auto_ptr 是一个 pointer-like object（类指针对象）（一个 *smart pointer*（智能指针）），它的 destructor（析构函数）自动在它指向的东西上调用 delete。下面就是如何使用 auto_ptr 来预防 f 的潜在的资源泄漏：

```
void f()
{
?std::auto_ptr<Investment> pInv(createInvestment());?// call factory
                                                      // function
?...                                                ?// use pInv as
                                                      // before
}                                                    ?// automatically
                                                      // delete pInv
via
                                                      // auto_ptr's
dtor
```

这个简单的例子示范了使用 objects（对象）管理资源的两个重要的方面：

- 获取资源后立即移交给 **resource-managing objects**（资源管理对象）。前面，createInvestment 返回的资源被用来初始化即将用来管理它的 auto_ptr 。实际上，因为获取一个资源并在同一个语句中初始化一个 resource-managing object（资源管理对象）是如此常见，所以使用 objects（对象）管理资源的观念常常被称为 *Resource Acquisition Is Initialization* (RAII) 。有时被获取的资源是被赋值给 resource-managing object（资源管理对象），而不是初始化它们的，但这两种方法都是在获取资源的同时就立即将? 平桓 桓? resource-managing object（资源管理对象）。
- **resource-managing objects**（资源管理对象）使用它们的 **destructors**（析构函数）确保资源被释放。因为当一个 object（对象）被销毁时（例如，当一个 object（对象）离开其活动范围）会自动调用 destructors（析构函数），无论控制流程是怎样离开一个 block（块）的，资源都会被正确释放。如果释放资源的动作会导致 exceptions（异常）被抛出，事情就会变得棘手，但是那是 Item 8 讲述的内容，所以在这里我们不必担心它。

因为当一个 auto_ptr 被销毁的时候，会自动删除它所指向的东西，所以不要让超过一个的 auto_ptr 指向一个 object（对象）非常重要。如果发生了这种事情，那个 object（对象）就会被删除超过一次，而且会让你的程序通过捷径进入 undefined behavior（未定义行为）。为了防止这样的问题，auto_ptrs 具有不同寻常的特性：拷贝它们（通过 copy constructor（拷贝构造函数）或者 copy assignment operator（拷贝赋值运算符））就是将它们置为空，而拷贝的指针取得资源的唯一所有权。

```
std::auto_ptr<Investment>                 // pInv1 points to the
?pInv1(createInvestment());              ?// object returned from
                                          ?// createInvestment

std::auto_ptr<Investment> pInv2(pInv1);   // pInv2 now points to the
                                         ?// object; pInv1 is now null

pInv1 = pInv2;                           ?// now pInv1 points to the
                                         ?// object, and pInv2 is null
```

这个奇怪的拷贝行为，增加了潜在的需求，就是通过 auto_ptrs 管理的资源必须绝对没有超过一个 auto_ptr 指向它们，这也就意味着 auto_ptrs 不是管理所有 dynamically allocated resources（动态分配资源）的最好方法。例如，STL containers（容器）要求其内含物能表现出"正常的"拷贝行为，所以 auto_ptrs 的容器是不被允许的。

相对于 auto_ptrs，另一个可选方案是一个 *reference-counting smart pointer* (RCSP)（引用计数智能指针）。一个 RCSP 是一个能持续跟踪有多少 objects（对象）指向一个特定的资源，并能够在不再有任何东西指向那个资源的时候自动删除它的 smart pointer（智能指针）。就这一点而论，RCSPs 提供的行为类似于 garbage collection（垃圾收集）的行为。然而，与 garbage collection（垃圾收集）不同的是， RCSPs 不能打破循环引用（例如，两个没有其它使用者的 objects（对象）互相指向对方）。

TR1 的 tr1::shared_ptr（参见 Item 54）是一个 RCSP，所以你可以这样写 f：

```
void f()
{
?...

?std::tr1::shared_ptr<Investment>
  ?pInv(createInvestment());              // call factory function
?...                                      // use pInv as before
}                                          // automatically delete
                                        ?// pInv via shared_ptr's dtor
```

这里的代码看上去和使用 auto_ptr 的几乎相同，但是拷贝 shared_ptrs 的行为却自然得多：

```
void f()
{
?...

?std::tr1::shared_ptr<Investment>         ?// pInv1 points to the
  ?pInv1(createInvestment());             ?// object returned from
                                           ?// createInvestment


?std::tr1::shared_ptr<Investment>         ?// both
pInv1 and pInv2 now
  ?pInv2(pInv1);                            // point to the object

?pInv1 = pInv2;                           ?// ditto — nothing has
                                           ?// changed
?...
}                                            // pInv1 and pInv2 are
                                          ?// destroyed, and the
                                          ?// object they point to is
                                          ?// automatically deleted
```

因为拷贝 tr1::shared_ptrs 的工作"符合预期"，它们能被用于 STL containers（容器）以及其它和 auto_ptr 的非正统的拷贝行为不相容的环境中。

可是，不要误解，本 Item 不是关于 auto_ptr，tr1::shared_ptr 或任何其它种类的 smart pointer（智能指针）的。而是关于使用 objects（对象）管理资源的重要性的。auto_ptr 和 tr1::shared_ptr 仅仅是做这些事的 objects（对象）的例子。（关于 tr1::shared_ptr 的更多信息，请参考 Item 14，18 和 54。）

auto_ptr 和 tr1::shared_ptr 都在它们的 destructors（析构函数）中使用 delete，而不是 delete []。（Item 16 描述两者的差异。）这就意味着将 auto_ptr 或 tr1::shared_ptr 用于 dynamically allocated arrays（动态分配数组）是个馊主意，可是，很遗憾，那居然可以编译：

```
std::auto_ptr<std::string>                    // bad idea! the wrong
?aps(new std::string[10]);                 ?// delete form will be
```

used

```
std::tr1::shared_ptr<int> spi(new int[1024]);  ?// same problem
```

你可能会吃惊地发现 C++ 中没有可用于 dynamically allocated arrays（动态分配数组）的类似 auto_ptr 或 tr1::shared_ptr 这样的东西，甚至在 TR1 中也没有。那是因为 vector 和 string 几乎总是能代替 dynamically allocated arrays（动态分配数组）。如果你依然觉得有可用于数组的类似 auto_ptr 和 tr1::shared_ptr 的 classes（类）更好一些的话，可以去看看 Boost（参见 Item 55）。在那里，你将如愿以偿地找到 boost::scoped_array 和 boost::shared_array 两个 classes（类）提供你在寻找的行为。

本 Item 的关于使用 objects（对象）管理资源的指导间接表明：如果你手动释放资源（例如，使用 delete，而不是在一个 resource-managing class（资源管理类）中），你就是在自找麻烦。像 auto_ptr 和 tr1::shared_ptr 这样的 pre-canned resource-managing classes（预制资源管理类）通常会使遵循本 Item 的建议变得容易，但有时，你使用了一个资源，而这些 pre-fab classes（预加工类）不能如你所愿地做事。如果碰上这种情况，你就需要精心打造你自己的 resource-managing classes（资源管理类）。那也并非困难得可怕，但它包含一些需要你细心考虑的微妙之处。那些需要考虑的？孪钍？ Item 14 和 15 的主题。

作为最后的意见，我必须指出 createInvestment 的 raw pointer（裸指针）的返回形式就是一份 resource leak（资源泄漏）的请帖，因为调用者忘记在他们取回来的指针上调用 delete 实在是太容易了。（即使他们使用一个 auto_ptr 或 tr1::shared_ptr 来实行 delete，他们仍然必须记住将 createInvestment 的返回值存储到一个 smart pointer object（智能指针对象）中。）对付这个问题调用需要改变 createInvestment 的接口，这是我在 Item 18 中安排的主题。

Things to Remember

- 为了防止 resource leaks（资源泄漏），使用 RAII objects（对象），在 RAII objects（对象）的 constructors（构造函数）中获取资源并在它们的 destructors（析构函数）中释放它们。
- 两个通用的 RAII classes（类）是 tr1::shared_ptr 和 auto_ptr。tr1::shared_ptr 通常是更好的选择，因为它的拷贝时的行为是符合直觉的。拷贝一个 auto_ptr 是将它置为空。

窗体底端

- # Item 14: 谨慎考虑 resource-managing classes（资源管理类）中的拷贝行为

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

Item 13 介绍了作为 resource-managing classes（资源管理类）支柱的 *Resource Acquisition Is Initialization* (RAII) 原则，并描述了 `auto_ptr` 和 `tr1::shared_ptr` 在 heap-based（基于堆）的资源上运用这一原则的表现。然而，并非所有的资源都是 heap-based（基于堆）的，而对于这样的资源，像 `auto_ptr` 和 `tr1::shared_ptr` 这样的 smart pointers（智能指针）通常就不像 resource handlers（资源句柄）那样合适。在这种情况下，有时，你很可能要根据你自己的需要去创建你自己的 resource-managing classes（资源管理类）。

例如，假设你使用一个 C API 提供的 `lock` 和 `unlock` 函数去操纵 `Mutex` 类型的 mutex objects（互斥体对象）：

```
void lock(Mutex *pm);                 // lock mutex pointed to by pm

void unlock(Mutex *pm);               // unlock the mutex
```

为了确保你从不会忘记解锁一个被你加了锁的 `Mutex`，你案打算创建一个 class 来管理锁。这样一个 class 的基本结构被 RAII 原则规定，通过 construction（构造函数）获取资源并通过 destruction（析构函数）释放它：

```
class Lock {
public:
?explicit Lock(Mutex *pm)
?: mutexPtr(pm)
?{ lock(mutexPtr); }                          ?// acquire resource

?~Lock() { unlock(mutexPtr); }                ?// release resource

private:
?Mutex *mutexPtr;
};
```

客户按照惯常的 RAII 风格来使用 `Lock`：

```
Mutex m;                    ?// define the mutex you need to use
...
{                             // create block to define critical section
 Lock ml(&m);                 // lock the mutex
...                           // perform critical section operations

}                             // automatically unlock mutex at end
                            ?// of block
```

这没什么问题，但是如果一个 `Lock` object 被拷贝应该发生什么？

```
Lock ml1(&m);                       ?// lock m
```

```
Lock ml2(ml1);                          // copy ml1 to ml2—what should
                                        // happen here?
```

这是一个更一般的问题的特定实例，每一个 RAII class 的作者都必须面对它：当一个 RAII object 被拷贝的时候应该发生什么？大多数情况下，你需要从下面各种可能性中挑选一个：

- **prohibit copying**（禁止拷贝）。在很多情况下，允许 RAII objects 被拷贝是没有意义的。这对于像 Lock 这样 class 很可能是正确的，因为同步原本的"拷贝"很少有什么意义。当拷贝对一个 RAII class 没有什么意义的时候，你应该禁止它。Item 6 解释了如何做到这一点：声明拷贝操作为私有。对于 Lock，看起来可能就像这样：

```
class Lock: private Uncopyable {          ?// prohibit copying — see
public:                                     // Item 6
?..                                      ?// as before
};
```

- **reference-count the underlying resource**（对底层的资源引用计数）。有时人们需要的是保持一个资源直到最后一个使用它的 object 被销毁。在这种情况下，拷贝一个 RAII object 应该增加引用这一资源的 objects 的数目。这也就是被 tr1::shared_ptr 使用的 "copy"（"拷贝"）的含意。

通常，RAII classes 只需要包含一个 tr1::shared_ptr data member（数据成员）就能够实现 reference-counting（引用计数）的拷贝行为。例如，如果 Lock 要使用 reference counting（引用计数），他可能要将 mutexPtr 的类型从 Mutex* 变为 tr1::shared_ptr<Mutex>。不幸的是，tr1::shared_ptr 的缺省行为是当它所指向的东西的 reference count（引用计数）变为零的时候将它删除，但这不是我们想要的。当我们使用完一个 Mutex 后，我们想要将它解锁，而不是将它删除。

幸运的是，tr1::shared_ptr 允许有一个对 "deleter"（当 reference count（引用计数）变为零时调用的一个 function（函数）或者 function object（函数对象））的特殊说明。（这一功能是 auto_ptr 所没有的，auto_ptr 总是删除它的 pointer（指针）。）deleter 是 tr1::shared_ptr 的 constructor（构造函数）的可选的第二个参数，所以，代码看起来就像这样：

```
class Lock {
public:
?explicit Lock(Mutex *pm)          // init shared_ptr with the Mutex
?: mutexPtr(pm, unlock)            // to point to and the unlock func
?{                                ?// as the deleter

  ?lock(mutexPtr.get());         ?// see Item 15 for info on "get"
?}
private:
?std::tr1::shared_ptr<Mutex> mutexPtr;  ?// use shared_ptr
};                                         // instead of raw pointer
```

在这个例子中，注意 Lock class 是如何不再声明一个 destructor（析构函数）的。那是因为它不需要。Item 5 解释了一个 class 的 destructor（析构函数）（无论它是 compiler-generated（编译器生成）的还是 user-defined（用户定义）的）会自动调用这个 class 的 non-static data members（非静态数据成员）的 destructors（析构函数）。在本例中，就是 mutexPtr。但是，当 mutex（互斥体）的 reference count（引用计数）变为零时，mutexPtr 的 destructor（析构函数）会自动调用 tr1::shared_ptr 的 deleter ——在此就是 unlock。（看过这个 class 的源代码的人多半意识到需要增加一条注释表明你并非忘记了 destruction（析构），而只是依赖 compiler-generated（编译器生成）的缺省行为。）

- **copy the underlying resource**（拷贝底层的资源）。有时就像你所希望的你可以拥有一个资源的多个拷贝，你需要一个 resource-managing class

（资源管理类）的唯一理由就是确保当你使用完每一个拷贝之后，它都会被释放。在这种? 榭鲅拢　奖匆桓? resource-managing object
（资源管理对象）也应该同时拷贝它所包覆的资源。也就是说，拷贝一个 resource-managing object（资源管理对象）需要实行一次 "deep copy"（"深层拷贝"）。

某些标准 `string` 类型的实现是由 pointers to heap memory（指向堆内存的指针）构成，那里存储着组成那个 string（字符串）的字符。这样的 `string`s 的 objects 包含一个 pointers to heap memory（指向堆内存的指针）。当一个 `string` object 被拷贝，一个拷贝应该由那个指针和它所指向的内存两者组成。这样的 `string`s 表现为 deep copying（深层拷贝）。

- **transfer ownership of the underlying resource**
  （传递底层资源的所有权）。在非常特殊的场合，你可能希望确保只有一个 RAII object 引用一个 raw resource（裸资源），而当这个 RAII object 被拷贝的时候，资源的所有权从 copied object（被拷贝对象）传递到 copying object（拷贝对象）。就像 Item 13 所说明的，这就是被 `auto_ptr` 使用的 "copy"（"拷贝"）的含意。

copying functions（拷贝函数）（copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符））可能由编译器生成，所以除非 compiler-generated（编译器生成）的版本所做的事正是你想要的（Item 5 说明了这些缺省行为），否则你应该自己编写它们。在某些情况下，你还要支持这些函数的泛型化版? 尽U庋　陌姥驹? Item 45 中描述。

Things to Remember

- 拷贝一个 RAII object 必须拷贝它所管理的资源，所以资源的拷贝行为决定了 RAII object 的拷贝行为。
- 通常的 RAII class 的拷贝行为是 disallowing copying（不允许拷贝）和 performing reference counting（实行引用计数），但是其它行为也是有可能的。

窗体底端

- # Item 16: 成对使用的 new 和 delete 要使用相同的形式

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

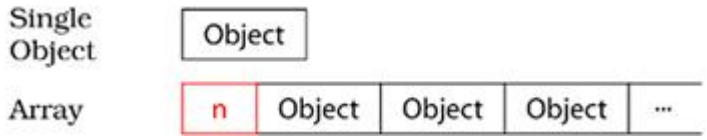
下面这段代码有什么问题？

```
std::string *stringArray = new std::string[100];
...
delete stringArray;
```

每一样东西看起来都很正常。也为 new 搭配了一个 delete 。但是，仍然有某件事情彻底错了。程序的行为是未定义的。至少，stringArray 指向的 100 个 string objects 中的 99 个不太可能被完全销毁，因为它们的 destructors （析构函数）或许根本没有被调用。

当你使用了一个 *new expression*（new 表达式）（也就是说，通过使用 new 动态创建一个 object ），有两件事情会发生。首先，内存被分配（通过一个名为 operator new 的函数——参见 Item 49 和 51）。第二，为这些内存调用一个或多个 constructors（构造函数）。当你使用一个 *delete expression*（delete 表达式）（也就是说，使用 delete ），有另外的两件事情会发生：为这些内存调用一个或多个 destructors （析构函数），然后内存被回收（通过一个名为 operator delete 的函数——参见 Item 51 ）。对于 delete 来说有一个大问题：在要被删除的内存中到底驻留有多少个 objects ？这个问题的答案将决定有多少个 destructors（析构函数）必须被调用。

事实上，问题很简单：将要被删除的指针是指向一个 single object（单一对象）还是一个 array of objects（对象的数组）？这是一个关键的问题，因为 single object （单一对象）的内存布局通常不同于数组的内存布局。详细地说，一个数组的内存布局通常包含数组? 拇笮。　庋　梢允沟? delete 更容易知道有多少个 destructors（析构函数）要被调用。而一个 single object（单一对象）的内存中缺乏这个信息。你可以认为不同的内存布局看起来如下图，那个 n 就是数组的大小：



这当然只是一个例子。编译器并不是必须这样实现，虽然很多是这样的。

当你对一个指针使用 delete，delete 知道数组大小信息是否存在的唯一方法就是由你来告诉它。如果你在你的 delete 使用中加入了方括号，delete 就假设那个指针指向的是一个数组。否则，就假设指向一个 single object（单一对象）。

```
std::string *stringPtr1 = new std::string;

std::string *stringPtr2 = new std::string[100];

...

delete stringPtr1;                      // delete an object
```

```
delete [] stringPtr2;                 ?// delete an array of objects
```

如果你对 stringPtr1 使用了 "[]" 形式会发生什么呢？结果是未定义的，但不太可能是什么好事。假设如上图的布局，delete 将读入某些内存的内容并将其看作是一个数组的大小，然后开始调用那么多 destructors（析构函数），全然不顾它在其上工作的内存不仅不是数组，而且还可能根本没有持有它能够析构的？糟偷？ objects（对象）。

如果你对 stringPtr2 没有使用 "[]" 形式会发生什么呢？也是未定义的，只不过你会看到它会引起过少的 destructors（析构函数）被调用。此外，对于类似 ints 的 built-in types（内建类型）其结果也是未定义的（而且有时是有害的），即使这样的类型没有 destructors（析构函数）。

规则很简单：如果你在一个 new 表达式中使用了 []，你也必须在相应的 delete 表达式中使用 []。如果你在一个 new 表达式中没有使用 []，在匹配的 delete 表达式中就不要使用 []。

当你写的一个 class 中包含一个指向 dynamically allocated memory（动态分配内存）的指针，而且还提供了多个 constructors（构造函数）的时候，把这条规则牢记于心就显得尤其重要，因为那时你必须小心地在所有的 constructors（构造函数）中使用 new 的 *same form*（同一种形式）初始化那个指针成员。如果你不这样做，你怎么知道在你的 destructor（析构函数）中使用 delete 的哪种形式呢？

这个规则对于有 typedef-inclined（typedef 爱好）的人也很值得注意，因为这意味着一个 typedef 的作者必须在文档中记录：当用 new 生成 typedef 类型的 objects（对象）时，应该使用 delete 的哪种形式。例如，考虑这个 typedef：

```
typedef std::string AddressLines[4];   // a person's address has 4
lines,
                                       // each of which is a string
```

因为 AddressLines 是一个 array（数组），这里对 new 的使用，

```
std::string *pal = new AddressLines;   // note that "new AddressLines"
                                       // returns a string*, just like
                                       // "new string[4]" would
```

必须被 delete 的 array（数组）形式匹配：

```
delete pal;                          ?// undefined!

delete [] pal;                         // fine
```

为了避免这种混淆，要避免对 array types（数组类型）使用 typedef。那很简单，因为标准 C++ 库（参见 Item 54）包含 string 和 vector，而且那些 templates（模板）将对 dynamically allocated arrays（动态分配数组）的需要减少到几乎为零。例如，这里，AddressLines 可以被定义为一个 strings 的 vector，也就是说，类型为 vector<string>。

Things to Remember

- 如果你在 new 表达式中使用了 []，就必须在对应的 delete 表达式中也使用 []。如果你在 new 表达式中没有使用 []，你也不必在对应的 delete 表达式中使用 []。

Item 17: 在 standalone statements（独立语句）中将 new 出来的 objects（对象）存入 smart pointers（智能指针）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

假设我们有一个函数用于取得我们的 processing priority（处理优先级），还有第二个函数用于根据 priority（优先级）对一个 dynamically allocated（动态分配）的 Widget 做一些处理：

```
int priority();
void processWidget(std::tr1::shared_ptr<Widget> pw, int priority);
```

不要忘记 using objects to manage resources（使用对象管理资源）的至理名言（参见 Item 13），processWidget 为 dynamically allocated（动态分配）的 Widget 使用了一个 smart pointer（智能指针）（在此，是一个 tr1::shared_ptr）。

现在考虑一个对 processWidget 的调用：

```
processWidget(new Widget, priority());
```

且慢，别想这样调用。它不能编译。tr1::shared_ptr 的取得一个 raw pointer（裸指针）的 constructor（构造函数）是 explicit（显式）的，所以没有从表达式 "new Widget" 返回的 raw pointer（裸指针）到 processWidget 所需的 tr1::shared_ptr 之间的隐式转换。然而，下面的代码，是可以编译的：

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

令人惊讶的是，尽管我们这里到处都在使用 object-managing resources（由对象管理的资源），这个调用还是可能泄漏资源。看看这是如何发生的会得到一些启示。

在编译器可以生成一个对 processWidget 的调用之前，它们必须计算被作为 parameters（形参）传递的 arguments（实参）的值。第二个 arguments（实参）不过是对函数 priority 的一个调用，但是第一个 arguments（实参）（"std::tr1::shared_ptr<Widget>(new Widget)"），由两部分组成：

- 表达式 "new Widget" 的执行。
- 对 tr1::shared_ptr constructor（构造函数）的一个调用。

因此，在 processWidget 可以被调用之前，编译器必须生成代码来做这三件事：

- 调用 priority。
- 执行 "new Widget"。
- 调用 tr1::shared_ptr constructor（构造函数）。

C++ 编译器允许在一个相当大的范围内决定这三件事被完成的顺序。（这里与 Java 和 C# 等语言的处理方式不同，那些语言里 function parameters（函数参数）总是按照一个特定的顺序被计算。）"new Widget" 表达式一定在 tr1::shared_ptr constructor（构造函数）能被调用之前执行，因为这个表达式的结果要作为一个 argument（实参）传递给 tr1::shared_ptr constructor（构造函数），但是对 priority 的调用可以在第一个，第二个或第三个执行。如果编译器选择第二个执行它（大概这样能使它们生成更有效率? 拇　耄　　颐亲钪盏玫秸庋　桓霾僮魑承颍?

1. 执行 "new Widget"。
2. 调用 priority。

3. 调用 tr1::shared_ptr constructor（构造函数）。

但是请考虑，如果对 priority 的调用引发一个 exception（异常），将发生什么。在这种情况下，从 "new Widget" 返回的 pointer（指针）被丢失，因为它没有被存入我们期望能阻止 resource leaks（资源泄漏）的 tr1::shared_ptr。由于一个 exception（异常）可能会插入到一个资源被创建的时间和那个资源被交给一个 resource-managing object（资源管理对象）的时间之间，所以在对 processWidget 的调用中可能会发生一次泄漏。

避免类似问题的方法很简单：用一个单独的语句创建 Widget 并将它存入一个 smart pointer（智能指针），然后将这个 smart pointer（智能指针）传递给 processWidget：

```
std::tr1::shared_ptr<Widget> pw(new Widget);?// store newed object
                                            ?// in a smart pointer in a
                                            ?// standalone statement

processWidget(pw, priority());              ?// this call won't leak
```

这样能工作是因为编译器在不同的语句之间重新安排操作顺序的活动余地比在一个语句之内要小得多。在这个? 薷牡拇 胫校?"new Widget" 表达式和对 tr1::shared_ptr constructor（构造函数）的调用，与对 priority 的调用在不同的语句中，所以编译器不会允许对 priority 的调用被插入到它们中间。

Things to Remember

- 在 standalone statements（独立语句）中将 new 出来的对象存入 smart pointers（智能指针）。如果疏忽了这一点，当 exceptions（异常）被抛出时，可能引起微妙的 resource leaks（资源泄漏）。

窗体底端

Item 19: 视 class design（类设计）为 type design（类型设计）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

在 C++ 中，就像其它 object-oriented programming languages（面向对象编程语言），通过定义一个新 class（类）来定义一个新 type（类型）。作为一个 C++ 开发者，你的大量时间就这样花费在加大你的 type system（类型系统）上。这意味着你不仅仅是一个 class designer（类设计者），而且是一个 *type* designer（类型设计者）。重载函数和运算符，控制内存分配和回收，定义 object 的初始化和终结过程——这些全在你的掌控之中。因此你应该在 class design（类设计）中倾注大量心血，就像语言设计者在语言的 built-in types（内建类型）的设计中所倾注的心血。

设计良好的 classes 是具有挑战性的，因为设计良好的 types 是具有挑战性的。良好的 types 拥有简单自然的语法，符合直觉的语义，以及一个或更多高效的实现。在 C++ 中，一个缺乏计划的 class definition（类设计），不可能达到上述任何一个目标。甚至一个 class 的 member functions（成员函数）的执行特征可能受到它们被声明的方式的影响。

那么，如何才能设计高效的 classes 呢？首先，你必须理解你所面对的问题。实际上每一个 class 都需要你面对下面这些问题，其答案通常就导向你的设计的限制因素：

- 你的新 **type**（类型）的 **objects** 应该如何创建和销毁？这些如何做将影响到你的 class 的 constructors（构造函数）和 destructor（析构函数），以及内存分配和回收的函数（`operator new`，`operator new[]`，`operator delete`，和 `operator delete[]` ——参见 Chapter 8 ）的设计，只要你写了它们。
- **object initialization**（对象初始化）和 **object assignment**（对象赋值）应该有什么不同？这个问题的答案决定了你的 constructors（构造函数）和你的 assignment operators（赋值运算符）的行为和它们之间的不同。这对于不混淆 initialization（初始化）和 assignment（赋值）是很重要的，因为它们相当于不同的函数调用（参见 Item 4）。
- **passed by value**（传值）对于你的新 **type**（类型）的 **objects** 意味着什么？记住，copy constructor（拷贝构造函数）定义了一个 type（类型）的 pass-by-value（传值）是如何实现的。
- 你的新 **type**（类型）的合法值的限定条件是什么？通常，对于一个 class 的 data members（数据成员）来说，仅有某些值的组合是合法的。那些组合决定了你的 class 必须维持的 invariants（不变量）。这些 invariants（不变量）决定了你必须在 member functions（成员函数）内部进行错误检查，特别是你的 constructors（构造函数），assignment operators（赋值运算符），和 "setter" 函数。它可能也会影响你的函数抛出的 exceptions（异常），和你的函数的exception specifications（异常规范）（你用到它的可能性很小）。
- 你的新 **type**（类型）是否适合放进一个 **inheritance graph**（继承图）中？如果你从已经存在的 classes 继承，你将被那些 classes 的设计所约束，特别是它们的函数是 virtual（虚拟）还是 non-virtual（非虚拟）（参见 Items 34 和 36）。如果你希望允许其它 classes 从你的 class 继承，将影响到你声明的函数是否为 virtual（虚拟），特别是你的 destructor（析构函数）（参见 Item 7）。
- 你的新 **type**（类型）允许哪种 **type conversions**（类型转换）？你的 type（类型）身处其它 types（类型）的海洋中，所以是否要在你的 type 和其它 types 之间有一些转换？如果你希望允许 type `T1` 的 objects *implicitly*（隐式）转换到 type `T2` 的 objects，你就要么在 class `T1` 中写一个 type conversion function（类型转换函数）（例如，`operator T2`），要么在 class `T2` 中写一个 non-`explicit` constructor（非显式构造函数），而且它们都要能够以单一 argument

（实参）调用。如果你希望仅仅允许 *explicit* conversions（显示转换），你就要写执行这个转换的函数，而且你还需要避免使它们的 type conversion operators（类型转换运算符）或 non-`explicit` constructor（非显式构造函数）能够以一个 argument（实参）调用。（作为一个既是隐式又是显式转换函数的例子，参见 Item 15。）

- 对于新 **type**（类型）哪些运算符和函数有意义？这个问题的答案决定你应该为你的 class 声明哪些函数。其中一些是 member functions（成员函数），另一些不是（参见 Items 23，24 和 46）。
- 哪些标准函数不应该被接受？你需要将那些都声明为 `private`（私有）的（参见 Item 6）。
- 你的新 **type**（类型）中哪些成员可以被访问？这个问题的可以帮助你决定哪些成员是 public（公有）的，哪些是 protected（保护）的，以及哪些是 private（私有）的。它也可以帮助你决定哪些 classes 和 / 或 functions 应该是 friends（友元），以及一个 class 嵌套在另一个 class 内部是否有意义。
- 什么是你的新 **type**（类型）的 **"undeclared interface"**？出于对 performance（性能），exception safety（异常安全）（参见 Item 29），以及 resource usage（资源使用）（例如，锁和动态内存）的考虑，它提供哪种保证？你在这些领域提供的保? そ 啃性际 愕? classes 的实现。
- 你的新 **type**（类型）有多大程度的通用性？也许你并非真的要定义一个新的 type（类型）。也许你要定义一个整个的类型家族。如果是这样，你不需要定义一个新的 class，而是需要定义一个新的 class *template*（类模板）。
- 一个新的 **type**（类型）真的是你所需要的吗？如果你只是要定义一个新的 derived class（派生类），以便让你可以为一个已存在的 class 增加一些功能，也许通过简单地定义一个或更多 non-member functions（非成员函数）或 templates（模板）能更好地达成你的目标。

回答这些问题是困难的，所以定义高效的 classes 是具有挑战性的。然而，如果做好了，在 C++ 中 user-defined classes（用户定义类）生成的 types（类型）至少可以和 built-in types（内建类型）一样好用，它会使一切努力都变的有价值。

Things to Remember

- class design（类设计）就是 type design（类型设计）。定义一个新 type（类型）之前，确保考虑了本 Item 讨论的所有问题。

窗体底端

Item 20: 用 pass-by-reference-to-const（传引用给 const）取代 pass-by-value（传值）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

缺省情况下，C++ 以 by value（传值）方式将 objects 传入或传出函数（这是一个从 C 继承来的特性）。除非你特别指定其它方式，否则 function parameters（函数形参）就会以 *copies* of the actual arguments（当前实参的拷贝）进行初始化，而且函数的调用者会取回函数返回值的一个 *copy*（拷贝）。这些拷贝由 objects 的 copy constructors（拷贝构造函数）生成。这就使得 pass-by-value（传值）成为一个代价不菲的操作。例如，考虑如下 class hierarchy（类层级结构）：

```
class Person {
public:
?Person();                          ?// parameters omitted for simplicity
?virtual ~Person();                   // see Item 7 for why this is virtual
?...

private:
?std::string name;
?std::string address;
};

class Student: public Person {
public:
?Student();                           // parameters again omitted
?~Student();
?...

private:
?std::string schoolName;
?std::string schoolAddress;
};
```

现在，考虑以下代码，在此我们调用一个函数 validateStudent，它得到一个 Student argument（实参）（以 by value（传值）方式），并返回它是否有效的结果：

```
bool validateStudent(Student s);            // function taking a Student
                                            // by value

Student plato;                              // Plato studied under Socrates

bool platoIsOK = validateStudent(plato);    // call the function
```

当这个函数被调用时会发生什么呢？

很明显，Student copy constructor（拷贝构造函数）被调用，用 plato 来初始化 parameter（形参）s。同样明显的是，当 validateStudent 返回时，s 就会被销毁。所以这个函数的 parameter-passing cost（参数传递成本）是一次 Student copy constructor（拷贝构造函数）的调用和一次 Student destructor（析构函数）的调用。

但这还不是全部。一个 Student object 内部包含两个 string objects，所以每次你构造一个 Student object 的时候，你也必须构造两个 string objects。一个 Student object 还要从一个 Person object 继承，所以每次你构造一个 Student object 的时候，你也必须构造一个 Person object。一个 Person object 内部又包含另外两个 string objects，所以每个 Person 的构造也承担着另外两个 string 的构造。最终，以 by value（传值）方式传递一个 Student object 的后果就是导致一次 Student copy constructor（拷贝构造函数）的调用，一次 Person copy constructor（拷贝构造函数）的调用，以及四次 string copy constructor（拷贝构造函数）调用。当 Student object 的拷贝被销毁时，每一次 constructor call（构造函数调用）都有一次 destructor call（析构函数调用）对应，所以以 by value（传值）方式传递一个 Student 的全部成本是六次 constructors（构造函数）和六次 destructors（析构函数）！

好了，这是正确的和值得的行为。毕竟，你想要你的所有 objects 都得到可靠的初始化和销毁。尽管如此，如果有一种办法可以绕过所有这些构造和析构过程，那就更好了，这?褪牵?pass by reference-to-const（传引用给 const）：

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

这样做非常有效：没有 constructors（构造函数）和 destructors（析构函数）被调用，因为没有新的 objects 被创建。被修改的 parameter declaration（形参声明）中的 const 是非常重要的。validateStudent 的最初版本以 by value（传值）方式取得一个 Student parameter（形参），所以调用者知道它们屏蔽了函数对它们传入的 Student 的任何可能的改变；validateStudent 只能改变它的一个 *copy*（拷贝）。现在 Student 以 by reference（传引用）方式传递，将它声明为 const 也是必要的，否则调用者必然担心 validateStudent 改变了它们传入的 Student。

以 by reference（传引用）方式传递 parameter（形参）还可以避免 *slicing problem*（切断问题）。当一个 derived class object（派生类对象）作为一个 base class object（基类对象）（以传值）方式被传递，base class copy constructor（基类拷贝构造函数）被调用，而那些使得 object 的行为像一个 derived class object（派生类对象）的特殊特性被 "sliced"（“切断”）了。只给你留下一个 simple base class object（纯粹基类对象）——这没什么可吃惊的，因为是一个 base class constructor（基类构造函数）创建了它。这几乎绝不是你想要的。例如，假设你在一组实现一个图形窗口系统的 classes 上工作：

```
class Window {
public:
?...
?std::string name() const;           // return name of window
?virtual void display() const;       // draw window and contents
};

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

所有 Window objects 都有一个名字，你能通过 name 函数得到它，而且所有的窗口都可以显示，你可一个通过调用 display 函数来做到这一点。display 为 virtual（虚拟）的的事实告诉你：一个 simple base class（纯粹基类）的 Window objects 的显示方法有可能不同于专门的 WindowWithScrollBars objects 的显示方法（参见 Items 34 和 36）。

现在，假设你想写一个函数打印出一个窗口的名字，并随后显示这个窗口。以下这个函数的写法是错误的：

```
void printNameAndDisplay(Window w)          // incorrect! parameter
{                                          ?// may be sliced!
```

```
?std::cout << w.name();
?w.display();
}
```

考虑当你用一个 WindowWithScrollBars object 调用这个函数时会发生什么：

```
WindowWithScrollBars wwsb;

printNameAndDisplay(wwsb);
```

parameter（形参）w 将被作为一个 *Window* object 构造——它是以 by value（传值）方式被传递的，记得吗？而且使 wwsb 表现得像一个 WindowWithScrollBars object 的特殊信息都被切断了。在 printNameAndDisplay 中，全然不顾传递给函数的那个 object 的类型，w 将始终表现得像一个 class Window 的 object（因为它就是一个 class Window 的 object）。特别是，在 printNameAndDisplay 中对 display 的调用将 *always*（总是）调用 Window::display，绝不会是 WindowWithScrollBars::display。

绕过 slicing problem（切断问题）的方法就是以 by reference-to-const（传引用给 const）方式传递 w：

```
void printNameAndDisplay(const Window& w)    // fine, parameter won't
{                                            // be sliced
?std::cout << w.name();
?w.display();
}
```

现在 w 将表现得像实际传入的那种窗口。

窗体底端

翻译：Effective C++, 3rd Edition, Item 20: 用 pass-by-reference-to-const（传引用给 const）取代 pass-by-value（传值）（下）

（点击此处，接上篇）

如如果你掀开编译器的盖头偷看一下，你会发现 references（引用）一般是作为 pointers（指针）实现的，所以以 by reference（传引用）方式传递某物实际上通常意味着传递一个 pointer（指针）。由此可以得出结论，如果你有一个 built-in typ（内建类型）的 object（例如，一个 `int`），以 by value（传值）方式传递它常常比 by reference（传引用）方式更高效。那么，对于 built-in typ（内建类型），当你需要在 pass-by-value（传值）和 pass-by-reference-to-`const`（传引用给 `const`）之间做一个选择时，没有道理不选择 pass-by-value（传值）。同样的建议也适用于 STL 中的 iterators（迭代器）和 function objects（函数对象），因为，作为惯例，它们就是为 passed by value（传值）设计的。iterators（迭代器）和 function objects（函数对象）的实现有责任保证拷贝的高效并且不受切断问题的影响。（这是一个"规则如何变化，依赖于你使用 C++ 的哪一个部分"的实例——参见 Item 1。）

built-in type（内建类型）很小，所以有人就断定所有的小类型都是 pass-by-value（传值）的上等候选者，即使它们是 user-defined（用户定义）的。这样的推理是不可靠的。仅仅因为一个 object 小，并不意味着调用它的 copy constructor（拷贝构造函数）就是廉价的。很多 objects ——大多数 STL containers（STL 容器）也在其中——容纳的东西比指针多不了什么，但是拷贝这样的 objects 必须同时拷贝它们指向的每一样东西。那可能是 very expensive（非常昂贵）的。

即使当一个小 objects 有廉价的 copy constructors（拷贝构造函数）时，也可能存在性能问题。一些编译器对 built-in（内建）的和 user-defined types（用户定义类型）并不一视同仁，即使他们有同样的底层表示。例如，一些编译器拒绝将仅由一个 `double` 组成的 objects 放入一个寄存器中，即使在常规上它们非常愿意将纯粹的 `double`s 放入那里。如果发生了这种事情，你以 by reference（传引用）方式传递这样的 objects 更好一些，因为编译器理所当然会将一个指针（references（引用）的实现）放入寄存器。

小的 user-defined types（用户定义类型）不一定是 pass-by-value（传值）的上等候选者的另一个原因是：作为 user-defined（用户定义）的，它的大小常常会变化。一个现在较小的类型在将来版本中可能变得较大，因为它的内部实现? 赡芑岜浠　I 踔恋蹦憾涣艘桓霾煌　? C++ 实现时，事情都可能会变化。例如，就在我这样写的时候，一些标准库的 `string` 类型的实现的大小就是另外一些实现的 *seven times*（七倍）。

通常情况下，你能合理地假设 pass-by-value（传值）廉价的类型仅有 built-in types（内建类型）及 STL iterator（迭代器）和 function object（函数对象）类型。对其他任何类型，请遵循本 Item 的建议，并用 pass-by-reference-to-`const`（传引用给 `const`）取代 pass-by-value（传值）。

Things to Remember

- 用 pass-by-reference-to-`const`（传引用给 `const`）取代 pass-by-value（传值）。典型情况下它更高效而且可以避免 slicing problem（切断问题）。
- 这条规则并不适用于 built-in types（内建类型）及 STL iterator（迭代器）和 function object（函数对象）类型。对于它们，pass-by-value（传值）通常更合适。

Item 21: 当你必须返回一个对象时不要试图返回一个引用

一旦程序员抓住对象传值的效率隐忧（参见 Item 20
），很多人就会成为狂热的圣战分子，誓要根除传值的罪恶，无论它隐藏多深。他们不屈不挠地追求传引用的?
慷龋　　侨　挤噶艘桓鲋旅　拇砦螅核　强　即　莶 2.淮燮诘亩韵蟮囊　谩Ｕ钰刹皇鞘裁春檬隆?

考虑一个代表有理数的类，包含一个将两个有理数相乘的函数：

```
class Rational {
public:
?Rational(int numerator = 0,                // see Item 24 for why this
          int denominator = 1);           ?// ctor isn't declared explicit

?...

private:
?int n, d;                                   // numerator and denominator

friend
   const Rational                            // see Item 3 for why the
     operator*(const Rational& lhs,          // return type is const
               const Rational& rhs);
};
```

operator*
的这个版本以传值方式返回它的结果，而且如果你没有担心那个对象的构造和析构的代价，你就是在推卸你的?
Ｔ抵霸稹Ｈ绻　悴皇瞧炔坏靡眩　悴挥Ω梦　庋　囊桓韶韵蟾冻氩杀尽Ｋ　晕侍饩驮谖铁铮耗闶瞧炔坏靡崖穑?

哦，如果你能用返回一个引用来作为代替，你就不是迫不得已。但是，请记住一个引用仅仅是一个名字，一个?
导蚀燮诘亩韵蟮拿　帧Ｎ蘼酆问敝灰　憧吹揭桓鲆　玫纳　鳎　阌Ω昧15.涛首约核　鞘裁炊　鞯牧硪桓雒　郑?
因为它必定是某物的另一个名字。在这个 operator*
的情况下，如果函数返回一个引用，它必须返回某个已存在的而且其中包含两个对象相乘的产物的 Rational
对象的引用。

当然没有什么理由期望这样一个对象在调用 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 should be 3/10
```

似乎没有理由期望那里碰巧已经存在一个值为十分之三的有理数。不是这样的，如果 operator*
返回这样一个数的引用，它必须自己创建那个数字对象。

一个函数创建一个新对象仅有两种方法：在栈上或者在堆上。栈上的生成物通过定义一个局部变量而生成。使?
谜飧霾呗裕　憧梢杂谜庵址椒去孕? operator*：

```
const Rational& operator*(const Rational& lhs,    // warning! bad code!
                          ?const Rational& rhs)
{
?Rational result(lhs.n * rhs.n, lhs.d * rhs.d);
?return result;
}
```

你可以立即否决这种方法，因为你的目标是避免调用构造函数，而 result

正像任何其它对象一样必须被构造。一个更严重的问题是这个函数返回一个引向 result 的引用，但是 result 是一个局部对象，而局部对象在函数退出时被销毁。那么，这个 operator* 的版本不会返回引向一个 Rational 的引用——它返回引向一个前 Rational；一个曾经的 Rational；一个空洞的、恶臭的、腐败的，从前是一个 Rational 但永不再是的尸体的引用，因为它已经被销毁了。任何调用者甚至于没有来得及匆匆看一眼这个函数的返回值?土15.探　肓宋炊丁逍形　牧斓于U馑鞘率担　魏畏祷匾桓鲆　蚓植勘淞康囊　玫暮　　际谴砦蟮摹#ú杂谌魏畏祷匾桓鲋赶蚓植勘淞康闹刚氲暮　　闪ｉ#?

那么，让我们考虑一下在堆上构造一个对象并返回引向它的引用的可能性。基于堆的对象通过使用 new 而开始存在，所以你可以像这样写一个基于堆的 operator*：

```
const Rational& operator*(const Rational& lhs,    // warning! more bad
                          ?const Rational& rhs)    // code!
{
?Rational *result = new Rational(lhs.n * rhs.n, lhs.d * rhs.d);
?return *result;
}
```

哦，你还是必须要付出一个构造函数调用的成本，因为通过 new 分配的内存要通过调用一个适当的构造函数进行初始化，但是现在你有另一个问题：谁是删除你用 new 做出来的对象的合适人选?

即使调用者尽职尽责且一心向善，它们也不太可能是用这样的方案来合理地预防泄漏：

```
Rational w, x, y, z;

w = x * y * z;                        // same as operator*(operator*(x, y), z)
```

这里，在同一个语句中有两个 operator* 的调用，因此 new 被使用了两次，这两次都需要使用 delete 来销毁。但是 operator* 的客户没有合理的办法进行那些调用，因为他们没有合理的办法取得隐藏在通过调用 operator* 返回的引用后面的指针。这是一个早已注定的资源泄漏。

窗体底端

由于程序的原因，本文件未被完整保存。

翻译：Effective C++, 3rd Edition, Item 21:
当你必须返回一个对象时不要试图返回一个引用（下）

（点击此处，接上篇）

但是也许你注意到无论是在栈上的还是在堆上的方法，为了从 operator* 返回的每一个 result ，我们都不得不容忍一次构造函数的调用。也许你想起我们最初的目标是避免这样的构造函数调用。也许你认? 阒 酪恢址椒 鼙范獬 淮我酝饧负跺 康墓乖旌 饕谩R残硐旅瓣飧鏊迪质悄阕龋 模 桓龌 ? operator* 返回一个引向 static Rational 对象的引用的实现，而这个 static Rational 对象定义在函数内部：

```
const Rational& operator*(const Rational& lhs,  ?// warning! yet more
                          ?const Rational& rhs)  ?// bad code!
{
?static Rational result;              // static object to which a
                                     ?// reference will be returned

?result = ... ;                      ?// multiply lhs by rhs and put the
                                      ?// product inside result
?return result;
}
```

就像所有使用了 static 对象的设计一样，这个也会立即引起我们的线程安全（thread-safety）的混乱，但那是它的比较明显的缺点? N 丝吹剿 母 畈愕娜毕苣 悸钦飧鏊耆 侠淼目突T 耄?

```
bool operator==(const Rational& lhs,           ?// an operator==
               ?const Rational& rhs);           // for Rationals

Rational a, b, c, d;

...
if ((a * b) == (c * d))?{
  ?do whatever's appropriate when the products are equal;
} else  ?{
  ?do whatever's appropriate when they're not;
}
```

猜猜会怎么样？不管 a，b，c，d 的值是什么，表达式 ((a*b) == (c*d)) 总是等于 true！

如果代码重写为功能完全等价的另一种形式，这一启示就很容易被理解了：

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

注意，当 operator== 被调用时，将同时存在两个起作用的对 operator* 的调用，每一个都将返回引向 operator* 内部的 static Rational 对象的引用。因此，operator== 将被要求比较 operator* 内部的 static Rational 对象的值和 operator* 内部的 static Rational 对象的值。如果它们不是永远相等，那才真的会令人大惊失色了。

这些应该足够让你信服试图从类似 operator* 这样的函数中返回一个引用纯粹是浪费时间，但是你们中的某些人可能会这样想“°好吧，就算一个 static 不够用，也许一个 static 的数组是一个窍门...-...-”±

我无法拿出示例代码来肯定这个设计，但我可以概要说明为什么这个想法应该让你羞愧得无地自容。首先，你? 匾胖≡褚桓?n 作为数组的大小。如果 n 太小，你可能会用完存储函数返回值的空间，与刚刚名誉扫地的 single-static 设计相比，在任何一个方面你都不会得到更多的东西。但是如果 n 太大，就会降低你的程序的性能，因为在函数第一次被调用的时候数组中的每一个对象都会被构造。即使这个? 颐钦 谔致鄯暮 霰坏饕昧艘淮危 步 媚愀冻?n 个构造函数和 n 个析构函数的成本。如果“°优化”± 是提高软件效率的过程，对于这种东西也只能是“°悲观主义”± 的。最后，考虑你怎样将你所需要的值放入数组的对象中，以及你做这些需要付出什么。在两个对象间移动值? 淖钪苯臃椒ň褪峭ü 持担 且淮胃持到 冻鏊裁矗慷杂诤芏啕嘈停 饩痛笞枷嗟庇诘饕靡淮挝龋购 厂 僭 吹闹担┘由系饕靡淮喂乖旌 ò研轮悼奖垂 ㄔ 5 悄愕哪勘晔潜范飧冻龋乖旌臀龋钩杀荆：娑缘慕 峁 褪牵赫飧龇椒ň 圆换岢晒Ａ＃ú唬 靡桓?vector 代替数组也不会让事情有多少改进。）

写一个必须返回一个新对象的函数的正确方法就是让那个函数返回一个新对象。对于 Rational 的 operator*，这就意味着下面这些代码或在本质上与其相当的某些东西：

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

当然，你可能付出了构造和析构 operator*
的返回值的成本，但是从长远看，这只是为正确行为付出的很小的代价。除此之外，这种令你感到恐怖的账单?
残碛涝抖疾换岬酱铎＞拖袼　械某绦蛏杓朴镅裕瑚++
允许编译器的实现者在不改变生成代码的可观察行为的条件下使用优化来提升它的性能，在某些条件下会产生?
缦陆崩　?operator*
的返回值的构造和析构能被安全地消除。如果编译器利用了这一点（编译器经常这样做），你的程序还是在它?
俣ū姆椒ㄉ霞绦　诵校　皇潜饶闫诖　囊　臁?

全部的焦点在这里：如果需要在返回一个引用和返回一个对象之间做出决定，你的工作就是让那个选择能提供?
　返男形　H媚愕谋嘁肫鞒　倘ソ示∧灾　鼓歉鲅＝裔】赡芳亓　邸?

Things to Remember

- 绝不要返回一个局部栈对象的指针或引用，绝不要返回一个被分配的堆对象的引用，如果存在需要一?鲆陨险庋　亩韵蟮目赡苄允保　　灰　祷匾桓鼍植?static 对象的指针或引用。（Item 4 提供的一个返回一个局部 static 的设计的例子是合理的，至少在单线程的环境中是这样。）

窗体底端

Item 22: 将数据成员声明为 private

好了，先公布一下计划。首先，我们将看看为什么数据成员不应该声明为 public 。然后，我们将看到所有反对 public 数据成员的理由同样适用于 protected 数据成员。这就导出了数据成员应该是 private 的结论，至此，我们就结束了。

那么，public 数据成员，为什么不呢？

我们从先从语法一致性开始（参见 Item 18）。如果数据成员不是 public 的，客户访问一个对象的唯一方法就是通过成员函数。如果在 public 接口中的每件东西都是一个函数，客户就不必绞尽脑汁试图记住当他们要访问一个类的成员时是否需要使用圆? ê拧K 侵灰 褂镁涂梢粤耍 蛭 考 鞠际且桓龌 R簧 岢终庖环秸耄 芙谴『芏嗄油返氖奔洹?

但是也许你不认为一致性的理由是强制性的。使用函数可以让你更加精确地控制成员的可存取性的事实又怎么? 兀咳绻 闵靡桓鏊 荸稍蔽? public ，每一个人都可以读写访问它，但是如果你使用函数去得到和设置它的值，你就能实现禁止访问，只读访问和? 列捶梦省：俸倪 绻 阈枰 闵踣量梢允迪种恍捶梦剩?

```
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;                        // no access to this int

?int readOnly;                        // read-only access to this int

?int readWrite;                      ?// read-write access to this int

?int writeOnly;                      ?// write-only access to this int
};
```

这种条分缕析的访问控制很重要，因为多数数据成员需要被隐藏。每一个数据成员都需要一个 getter 和 setter 的情况是很罕见的。

还不相信吗？那么该拿出一门重炮了：封装。如果你通过一个函数实现对数据成员的访问，你可以在以后用一? 黑扑懔刺婊徽飧鏊 荸稍保 褂媚愕睦嗟娜瞬换峄腥魏尾炀酢?

例如，假设你为一个监视通过的汽车的速度的自动设备写一个应用程序。每通过一辆汽车，它的速度就被计算? 夷歉鲋狄 尤氲狡 裎 故占 降乃 兴俣仁 兹募 现校?

```
class SpeedDataCollection {
?...
public:
?void addValue(int speed);        ?// add a new data value

?double averageSoFar() const;       // return average speed

?...
```

```
};
```

现在考虑成员函数 averageSoFar
的实现：实现它的办法之一是在类中用一个数据成员来实时变化迄今为止收集到的所有速度数据的平均值。无?
鄯问? averageSoFar 被调用，它只是返回那个数据成员的值。另一个不同的方法是在每次调用
averageSoFar 时重新计算它的值，通过分析集合中每一个数据值它能做成这些事情。

第一种方法（保持一个实时变化的值）使每一个 SpeedDataCollection
对象都比较大，因为你必须为持有实时变化的平均值，累计的和以及数据点的数量分配空间。可是，
averageSoFar 能实现得非常高效，它仅仅是一个返回实时变化的平均值的 inline 函数（参见 Item 30
）。反过来，无论何时被请求都要计算平均值使得 averageSoFar 的运行比较慢，但是每一个
SpeedDataCollection 对象都比较小。

谁能说哪一个最好？在内存非常紧张的机器（例如，一个嵌入式道旁设备）上，以及在一个很少需要平均值的?
τ贸绦蛑校　看味技扑闫骄　悼赡苁墙虾玫慕饩酏桨浮Ｔ谝桓銮捣毙枰　骄　档挠τ贸绦蛑校　俣仁腔　镜囊?
求，而且内存不成问题，保持一个实时变化的平均值更为可取。这里的重点在于通过经由一个成员函数访问平?
　担丁簿褪撬担　ü　　庾埃　　骢芑ㄝ徽饬礁霾煌　氖迪郑丁舶　夕渌　憧赡芟氲降模　　杂诓突Ｂ　疃嘁簿褪潜匾胫匾卤嘁舯＃　憧梢杂迷诤竺娴? Item 31 中记述的技术来消除这个麻烦。）

将数据成员隐藏在功能性的接口之后能为各种实现提供弹性。例如，它可以在读或者写的时候很简单地通报其?
　韵螅　梢约煅槔嗟牟槐淞恳约昂　　那爸没蚧笆锰邆　　梢栽诙嘞叱袒肪持兄葱型　饺挝瘢　鹊取 4 永嗨?
Delphi 和 C# 的语言来到 C++ 的程序员会认同这种类似那些语言中的"属性"
的等价物的功能，虽然需要附加一个带圆括号的额外的 set。

关于封装的要点可能比它最初显现出来的更加重要。如果你对你的客户隐藏你的数据成员（也就是说，封装它?
牵　　憔湍苋繁@嗟牟槐淞孔苣鼙晃　郑　蛭　挥谐稍焙　　苡跋焖　恰 4 送猓　阎ち袅艘院蟾谋湃愕氖迪志?
策的权力。如果你不隐藏这样的决策，你将很快发现，即使你拥有一个类的源代码，你改变任何一个 public
的东西的能力也是非常有限的，因为有太多的客户代码将被破坏。public
意味着没有封装，而且几乎可以说，没有封装意味着不可改变，尤其是被广泛使用的类。但是仍然被广泛使用?
睦啻蠖嗍　际切枰　庾暗模　蛭　　强梢源佑靡恢指　玫氖迪痔婊幌钟惺迪值哪芰ｘ谢竦米疃嗟囊娲Ａ?

反对 protected
数据成员的理由是类似的。实际上，它是一样的，虽然起先看起来似乎不那么清楚。关于语法一致性和条分缕?
龅姆梦士刂频穆壑ぞ拖裼糜? public 一样可以应用于 protected，但是关于封装又如何呢？难道
protected 数据成员不比 public 数据成员更具有封装性吗？实话实说，令人惊讶的答案是它们不。

Item 23

解释了如果某物发生了变化，某物的封装与可能被破坏的代码数量成反比。于是，如果数据成员发生了变化（?
　纾　绻　　淮永嘀幸瞥　ń赡苁俏　颂婊晃　扑悖　拖裨谏厦娴? averageSoFar
中）），数据成员的封装性与可能被破坏的代码数量成反比。

假设我们有一个 public
数据成员，随后我们消除了它。有多少代码会被破坏呢？所有使用了它的客户代码，其数量通常大得难以置信?
4 佣? public 数据成员就是完全未封装的。但是，假设我们有一个 protected
数据成员，随后我们消除了它。现在有多少代码会被破坏呢？所有使用了它的派生类，典型情况下，代码的数?
炕故谴蟮媚岩灾眯拧 4 佣? protected 数据成员就像 public
数据成员一样没有封装，因为在这两种情况下，如果数据成员发生变化，被破坏的客户代码的数量都大得难以?
眯拧Ｕ獠 2.环　现本酰　　歉挥芯　桐目馑迪终呋峋蒌吣悖　馑乔ㄩ嫱蛉返摹Ｒ坏├闵　饕桓鏊　荸稍蔽?
public 或 protected
，而且客户开始使用它，就很难再改变与这个数据成员有关的任何事情。有太多的代码不得不被重写，重测试?
　匚牡祷　　蛑毛嘁舜 4 臃庾暗墓鄯憷纯矗　导手挥辛礁龇梦什愦危?private
（提供了封装）与所有例外（没有提供封装）。

Things to Remember

- 声明数据成员为 private
  。它为客户提供了访问数据的语法层上的一致，提供条分缕析的访问控制，允许不变量被强制，而且为类的作者提供了实现上的弹性。
- protected 并不比 public 的封装性强。

Item 23: 用非成员非友元函数取代成员函数

想象一个象征 web
浏览器的类。在大量的函数中，这样一个类也许会提供清空已下载成分的缓存。清空已访问 URLs
的历史，以及从系统移除所有 cookies 的功能：

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

很多用户希望能一起执行全部这些动作，所以 WebBrowser 可能也会提供一个函数去这样做：

```
class WebBrowser {
public:
?...
?void clearEverything();                  // calls clearCache, clearHistory,
                                         ?// and removeCookies
?...
};
```

当然，这个功能也能通过非成员函数调用适当的成员函数来提供：

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

那么哪个更好呢，成员函数 clearEverything 还是非成员函数 clearBrowser?

面性对象原则指出：数据和对它们进行操作的函数应该被绑定到一起，而且建议成员函数是更好的选择。不幸?氖牵　飧鼋T槭遣徽　返摹K　　诙悦嫦蚴韵第鞘裁吹囊桓鑫蠼狻C嫦蚴韵笞　蛑赋鳖　萦Ω镁】赡鼙环庾?。与直觉不同，成员函数 clearEverything 居然会造成比非成员函数 clearBrowser
更差的封装性。此外，提供非成员函数允许 WebBrowser
相关功能的更大的包装弹性，而且，可以获得更少的编译依赖和 WebBrowser
扩展性的增进。因而，在很多方面非成员方法比一个成员函数更好。理解它的原因是非常重要的。

我们将从封装开始。如果某物被封装，它被从视线中隐藏。越多的东西被封装，就越少有东西能看见它。越少?卸　骼芸醇　　颐歉谋渌　牡　跃驮酱螅　蛭　颐堑母谋浣鼋鲋苯佑跋炷切├芸醇　颐潜淞耸裁吹亩　鳖D?物的封装性越强，那么我们改变它的能力就越强。这就是将封装的价值评价为第一的原因：它为我们提供一种?谋渲虑榈牡　裕　　鼋鲇跋煊邢薛目突Α?

结合一个对象考虑数据。越少有代码能看到数据（也就是说，访问它），数据封装性就越强，我们改变对象的?　茏奶甑缘淖杂梢簿驮酱螅　热纾　　荸稍钡氖　浚　　堑睦嘈停　鹊取W魑　嗌俅　肱芸吹揭豢槭　茏拇植?的尺度，我们可以计数能访问那块数据的函数的数量：越多函数能访问它，数据的封装性就越弱。

Item 22 说明了数据成员应该是 private
的，因为如果它们不是，就有无限量的函数能访问它们。它们根本就没有封装。对于 private
数据成员，能访问他们的函数的数量就是类的成员函数的数量加上友元函数的数量，因为只有成员和友元能访?Ê private 成员。假设在一个成员函数（能访问的不只是一个类的 private 数据，还有 private
函数，枚举，typedefs

，等等）和一个提供同样功能的非成员非友元函数（不能访问上述那些东西）之间有一个选择，能获得更强封?
靶缘难≡袷欠浅稍狈怯言　　蛭　　换嵩黾幽芊梦世嗟? private
部分的函数的数量。这就解释了为什么 clearBrowser（非成员非友元函数）比 clearEverything
（成员函数）更可取：它能为 WebBrowser 获得更强的封装性。

在这一点，有两件事值得注意。首先，这个论证只适用于非成员非友元函数。友元能像成员函数一样访问一个?
嗟? private
成员，因此同样影响封装。从封装的观点看，选择不是在成员和非成员函数之间，而是在成员函数和非成员非?
言　　洹＃ū比唬　庾安 2.皇墙鲇械墓鄯悖?Item 24
说明如果观点来自隐式类型转换，选择就是在成员和非成员函数之间。）

需要注意的第二件事是，如果仅仅是为了关注封装，则可以指出，一个函数是一个类的非成员并不意味着它不?
梢允橇硪桓隼嗟某稍薄U舛杂谙肮吡怂　泻　　甑胧粲诶嗟挠镅裕g　纾?Eiffel，Java，C#
，等等）的程序员是一个适度的安慰。例如，我们可以使 clearBrowser 成为一个 utility 类的 static
成员函数。只要它不是 WebBrowser 的一部分（或友元），它就不会影响 WebBrowser 的 private
成员的封装。

在 C++ 中，一个更自然的方法是使 clearBrowser 成为与 WebBrowser 在同一个 namespace
（名字空间）中的非成员函数：

```
namespace WebBrowserStuff {

  lass WebBrowser { ... };

  oid clearBrowser(WebBrowser& wb);

?...
}
```

相对于形式上的自然，这样更适用于它。无论如何，因为名字空间（不像类）能展开到多个源文件中。这是很?
匾　模　蛭　嗨? clearBrowser 的函数是方便性函数。作为既不是成员也不是友元，他们没有对
WebBrowser 进行专门的访问，所以他们不能提供任何一种 WebBrowser
的客户不能通过其它方法得到的功能。例如，如果 clearBrowser 不存在，客户可以直接调用
clearCache，clearHistory 和 removeCookies 本身。

一个类似 WebBrowser 的类可以有大量的方便性函数，一些是书签相关的，另一些打印相关的，还有一些是
cookie
管理相关的，等等。作为一个一般的惯例，多数客户仅对这些方便性函数的集合中的一些感兴趣。没有理由让?
桓鲋欢允榍┫喙氏姆奖阈院　　行巳さ目突ｒ诒嘁胧币览灯渌　　纾?cookie
相关的方便性函数。分隔它们的直截了当的方法就是在一个头文件中声明书签相关的方便性函数，在另一个不?
　耐肺募　猩　？cookie 相关的方便性函数，在第三个头文件声明打印相关的方便性函数，等等：

```
// header "webbrowser.h" - header for class WebBrowser itself
// as well as "core" WebBrowser-related functionality
namespace WebBrowserStuff {

   class WebBrowser { ... };

     ...                                  ?// "core" related functionality, e.g.
                                          ?// non-member functions almost
                                          ?// all clients need
}
// header "webbrowserbookmarks.h"
namespace WebBrowserStuff {
?...                                       // bookmark-related convenience
}                                          // functions
// header "webbrowsercookies.h"
```

```
namespace WebBrowserStuff {
?...                                    // cookie-related convenience
}                                        // functions

...
```

注意这里就像标准 C++ 库组织得一样严密。胜于有一个单独的一体式的 <C++StandardLibrary> 头文件包含 std namespace 中的所有东西，它们在许多头文件中（例如，<vector>，<algorithm>，<memory>，等等），每一个都声明了 std 中的一些机能。仅仅需要 vector 相关机能的客户不需要 #include <memory>，不用 list 的客户没有必要 #include <list> 。这就允许客户在编译时仅仅依赖他们实际使用的那部分系统。（参见 Item 31 对减少编译依赖的其它方法的讨论。）当机能来自一个类的成员函数时，用这种方法分割它是不可能的，因为? 桓隼啾匾胱魑　桓稣　謇炊丁澹　　荒芩姆治辶选?

将所有方便性函数放入多个头文件中——但是在一个 namespace 中——也意味着客户能容易地扩充方便性函数的集合。他们必须做的全部就是在 namespace 中加入更多的非成员非友元函数。例如，如果一个 WebBrowser 的客户决定写一个关于下载图像的方便性函数，他或她仅仅需要新建一个头文件，包含那些函数在 WebBrowserStuff namespace 中的声明。这个新的函数现在就像其它方便性函数一样可用并被集成。这是类不能提供的另一个特性，因为类? 丁宥杂诳突　抢 ┍泥獗盏摹 5 比唬　突 Э 梢耘缮　吕啵　　桥缮　嗖荒芊梦驶　嘀斜环庾暗模丁簿褪撬担? private 的）成员，所以这样的"扩充机能"只有二等身份。此外，就像 Item 7 中解释的，不是所有的类都是作为基类设计的。

Things to Remember

- 用非成员非友元函数取代成员函数。这样做可以提高封装性，包装弹性，和机能扩充性。

窗体底端

Item 24: 当类型转换应该用于所有参数时，声明为非成员函数

在此书的 Introduction
中我谈到让一个类支持隐式类型转换通常是一个不好的主意。当然，这条规则有一些例外，最普通的一种就是? 诖唇亡　道嘈褪薄@　纾　绻　闵杓埔桓鲇美幢硐钟欣硎　睦啵　市裳诱　　接欣硎　囊　阶　豢瓷先ゲ 7.遣? 合理。这的确不比 C++ 的内建类型从 int 到 double 的转换更不合理（而且比 C++ 的内建类型从 double 到 int 的转换合理得多）。在这种情况下，你可以用这种方法开始你的 Rational 类：

```
class Rational {
public:
?Rational(int numerator = 0,       ?// ctor is deliberately not explicit;
          int denominator = 1);      // allows implicit int-to-Rational
                                     // conversions

?int numerator() const;              // accessors for numerator and
?int denominator() const;            // denominator - see Item 22

private:
?...
};
```

你知道你应该支持算术运算，比如加法，乘法，等等，但是你不能确定是通过成员函数，非成员函数，还是非? 稍钡挠言　　词迪炙　恰D愕闹本醺蒌吆悖　蹦阋“诓欢ū氖焙颍　阌Ω眉岢置嫦蚨韵蟮脑　颉D懔私庶庖坏悖　谑嵌隙ǎ　蛭　欣硎　某朔业? Rational 类相关，所以在 Rational 类的内部实现有理数的 operator* 似乎更加正常。但是，与直觉不符的是，Item 23
指出将函数放在它们所关联的类的内部的主张有时候与面向对象的原则正好相反，但是让我们将它先放到一边? 囱芯恳幌氯? operator* 成为 Rational 的一个成员函数的想法究竟如何：

```
class Rational {
public:
?..

  onst Rational operator*(const Rational& rhs) const;
};
```

（如果你不能确定为什么这个函数声明为这个样子——返回一个 const by-value 的结果，却持有一个 reference-to-const 作为它的参数——请参考 Item 3，20 和 21。）

这个设计让你在有理数相乘时不费吹灰之力：

```
Rational oneEighth(1, 8);
Rational oneHalf(1, 2);

Rational result = oneHalf * oneEighth;          ?// fine

result = result * oneEighth;                    ?// fine
```

但是你并不感到满意。你还希望支持混合模式的操作，以便让 Rationals 能够和其它类型（例如，int ）相乘。毕竟，很少有事情像两个数相乘那么正常，即使它们碰巧是数字的不同类型。

当你试图做混合模式的算术运算时，可是，你发现只有一半时间它能工作：

```
result = oneHalf * 2;                           // fine

result = 2 * oneHalf;                           // error!
```

这是一个不好的征兆。乘法必须是可交换的，记得吗？

当你重写最后两个例子为功能等价的另一种形式时，问题的来源就变得很明显了：

```
result = oneHalf.operator*(2);                    ?// fine

result = 2.operator*(oneHalf);                    ?// error!
```

对象 oneHalf 是一个包含 operator* 的类的实例，所以编译器调用那个函数。然而，整数 2 与类没有关系，因而没有 operator* 成员函数。编译器同样要寻找能如下调用的非成员的 operator*s（也就是说，在 namespace 或全局范围内的 operator*s）：

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

但是在本例中，没有非成员的持有一个 int 和一个 Rational 的 operator*，所以搜索失败。

再看一眼那个成功的调用。你会发现它的第二个参数是整数 2，然而 Rational::operator* 却持有一个 Rational 对象作为它的参数。这里发生了什么呢？为什么 2 在一个位置能工作，在其它地方却不行呢？

发生的是隐式类型转换。编译器知道你传递一个 int 而那个函数需要一个 Rational ，但是它们也知道通过用你提供的 int 调用 Rational 的构造函数，它们能做出一个相配的 Rational ，这就是它们的所作所为。换句话说，它们将那个调用或多或少看成如下这样：

```
const Rational temp(2);             ?// create a temporary
                                      // Rational object from 2

result = oneHalf * temp;              // same as oneHalf.operator*(temp);
```

当然，编译器这样做仅仅是因为提供了一个非显性的构造函数。如果 Rational 的构造函数是显性的，这些语句都将无法编译：

```
result = oneHalf * 2;               ?// error! (with explicit ctor);
                                      // can't convert 2 to Rational

result = 2 * oneHalf;               ?// same error, same problem
```

支持混合模式操作失败了，但是至少两个语句的行为将步调一致。

然而，你的目标是既保持一致性又要支持混合运算，也就是说，一个能使上面两个语句都可以编译的设计。让? 颐欠祷卣饬礁鲇锞淇匆豢矗矗　　裁醇词? Rational 的构造函数不是显式的，也是一个可以编译而另一个不行：

```
result = oneHalf * 2;               ?// fine (with non-explicit ctor)

result = 2 * oneHalf;               ?// error! (even with non-explicit ctor)
```

其原因在于仅仅当参数列在参数列表中的时候，它们才有资格进行隐式类型转换。而对应于成员函数被调用的? 歉龆韵蟮囊　　问?—— this 指针指向的那个—— 根本没有资格进行隐式转换。这就是为什么第一个调用能编译而第二个不能。第一种情况包括一个参数被列在? 问　斜碇校　　诙　智榭雒挥小?

你还是希望支持混合运算，然而，现在做到这一点的方法或许很清楚了：让 operator* 作为非成员函数，因此就允许便一起将隐式类型转换应用于所有参数：

```
class Rational {
```

```
?...                                                  // contains no operator*
};
const Rational operator*(const Rational& lhs,        // now a non-member
                         const Rational& rhs)        // function
{
?return Rational(lhs.numerator() * rhs.numerator(),
                ?lhs.denominator() * rhs.denominator());
}
Rational oneFourth(1, 4);
Rational result;

result = oneFourth * 2;                               // fine
result = 2 * oneFourth;                               // hooray, it works!
```

这样的确使故事有了一个圆满的结局，但是有一个吹毛求疵的毛病。operator* 应该不应该作为 Rational 类的友元呢？

在这种情况下，答案是不，因为 operator* 能够根据 Rational 的 public 接口完全实现。上面的代码展示了做这件事的方法之一。这导出了一条重要的结论：与成员函数相对的是非成?焙　　皇怯言　　L　嗟某绦蛟奔偕枞绻　桓龌　　胍桓隼嘤泄囟　植挥Ω米魑　稍笔保g　纾　蛭　　械牟问　夹枰　嘈妥　唬　　Ω米魑　言　U飧鏊纠　っ髡庋　耐评硎怯腥毕茏摹N蘼酆问保　挥心隰鼙范?友元函数，你就避免它，因为，就像在现实生活中，朋友的麻烦通常多于他们的价值。当然，有时友谊是正当?模　　鞘率当砻鹘鼋鲆蛭　　挥Ω米魑　稍辈 2.蛔远　镶蹲潘　Ω米魑　言　？

本 Item 包含真理，除了真理一无所有，但它还不是完整的真理。当你从 Object-Oriented C++ 穿过界线进入 Template C++（参见 Item 1）而且将 Rational 做成一个类模板代替一个类，就有新的问题要考虑，也有新的方法来解决它们，以及一些令人惊讶的设计含义?U庋　奈侍猓　饩龇椒ê秃　迨? Item 46 的主题。

Things to Remember

- 如果你需要在一个函数的所有参数（包括被 this 指针所指向的那个）上使用类型转换，这个函数必须是一个非成员。

窗体底端

由于程序的原因，本文件未被完整保存。

Item 25: 考虑支持不抛异常的 swap

swap 是一个有趣的函数。最早作为 STL 的一部分被引入，后来它成为异常安全编程（exception-safe programming）的支柱（参见 Item 29）和压制自赋值可能性的通用机制（参见 Item 11）。因为 swap 太有用了，所以正确地实现它非常重要，但是伴随它的不同寻常的重要性而来的，是一系列不同寻常的复杂性? T论?Item 中，我们就来研究一下这些复杂性究竟是什么样的以及如何对付它们。

交换两个对象的值就是互相把自己的值送给对方。缺省情况下，通过标准的交换算法来实现交换是非常成熟的? 际酢 5 淞偷氖迪滞耆　　夏愕脑て冢?

```
namespace std {

?template<typename T>         ?// typical implementation of std::swap;
?void swap(T& a, T& b)          // swaps a's and b's values
?{
  ?T temp(a);
  ?a = b;
  ?b = temp;
?}
}
```

只要你的类型支持拷贝（通过拷贝构造函数和拷贝赋值运算符），缺省的 swap 实现就能交换你的类型的对象，而不需要你做任何特别的支持工作

可是，缺省的 swap 实现可能不那么酷。它涉及三个对象的拷贝：从 a 到 temp，从 b 到 a，以及从 temp 到 b。对一些类型来说，这些副本全是不必要的。对于这样的类型，缺省的 swap 就好像让你坐着快车驶入小巷。

这样的类型中最重要的就是那些主要由一个指针组成的类型，那个指针指向包含真正数据的另一种类型。这种? 杓品椒ū囊恢殖＜　谋硐中问绞?"pimpl idiom"（"pointer to implementation" ——参见 Item 31）。一个使用了这种设计的 Widget 类可能就像这样：

```
class WidgetImpl {                            ?// class for Widget data;
public:                                         // details are unimportant
?...

private:
?int a, b, c;                                ?// possibly lots of data -
?std::vector<double> v;                      ?// expensive to copy!
?...
};

class Widget {                                ?// class using the pimpl idiom
public:
?Widget(const Widget& rhs);

?Widget& operator=(const Widget& rhs)        ?// to copy a Widget, copy its
?{                                              // WidgetImpl object. For
   ...                                        ?// details on implementing
   *pImpl = *(rhs.pImpl);                      ?// operator= in general,
   ...                                          // see Items 10, 11, and 12.
?}
?...

private:
?WidgetImpl *pImpl;                             // ptr to object with this
};                                              // Widget's data
```

为了交换这两个 Widget 对象的值，我们实际要做的就是交换它们的 pImpl 指针，但是缺省的交换算法没有办法知道这些。它不仅要拷贝三个 Widgets，而且还有三个 WidgetImpl

对象，效率太低了。一点都不酷。

当交换 Widgets 的是时候，我们应该告诉 std::swap
我们打算做什么，执行交换的方法就是交换它们内部的 pImpl 指针。这种方法的正规说法是：针对 Widget
特化 std::swap（specialize std::swap for Widget
）。下面是一个基本的想法，虽然在这种形式下它还不能通过编译：

```
namespace std {

?template<>                              ?// this is a specialized version
?void swap<Widget>(Widget& a,            ?// of std::swap for when T is
                   ?Widget& b)           ?// Widget; this won't compile
?{
  ?swap(a.pImpl, b.pImpl);                 // to swap Widgets, just swap
?}                                         // their pImpl pointers
}
```

这个函数开头的 "template<>" 表明这是一个针对 std::swap 的完全模板特化（total template
specialization）（某些书中称为 "full template specialization" 或 "complete template
specialization" ——译者注），函数名后面的 "<Widget>" 表明特化是在 T 为 Widget
类型时发生的。换句话说，当通用的 swap 模板用于 Widgets 时，就应该使用这个实现。通常，我们改变
std namespace 中的内容是不被允许的，但允许为我们自己创建的类型（就像 Widget
）完全特化标准模板（就像 swap）。这就是我们现在在这里做的事情。

可是，就像我说的，这个函数还不能编译。那是因为它试图访问 a 和 b 内部的 pImpl 指针，而它们是
private 的。我们可以将我们的特化声明为友元，但是惯例是不同的：让 Widget 声明一个名为 swap
的 public 成员函数去做实际的交换，然后特化 std::swap 去调用那个成员函数：

```
class Widget {                        // same as above, except for the
public:                              ?// addition of the swap mem func
?...
?void swap(Widget& other)
?{
  ?using std::swap;                   // the need for this declaration
                                        // is explained later in this Item

  ?swap(pImpl, other.pImpl);        ?// to swap Widgets, swap their
?}                                     ?// pImpl pointers
?...
};

namespace std {

?template<>                           // revised specialization of
?void swap<Widget>(Widget& a,         // std::swap
                   ?Widget& b)
?{
  ?a.swap(b);                         // to swap Widgets, call their
?}                                   ?// swap member function
}
```

这个不仅能够编译，而且和 STL 容器保持一致，所有 STL 容器都既提供了 public swap
成员函数，又提供了 std::swap 的特化来调用这些成员函数。

可是，假设 Widget 和 WidgetImpl 是类模板，而不是类，或许因此我们可以参数化存储在
WidgetImpl 中的数据类型：

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

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

在 Widget 中加入一个 swap 成员函数（如果我们需要，在 WidgetImpl
中也加一个）就像以前一样容易，但我们特化 std::swap 时会遇到麻烦。这就是我们要写的代码：

```
namespace std {
?template<typename T>
?void swap<Widget<T> >(Widget<T>& a,     ?// error! illegal code!
                       ?Widget<T>& b)
?{ a.swap(b); }
}
```

这看上去非常合理，但它是非法的。我们试图部分特化（partially specialize）一个函数模板（
std::swap），但是尽管 C++ 允许类模板的部分特化（partial
specialization），但不允许函数模板这样做。这样的代码不能编译（尽管一些编译器错误地接受了它）。

当我们想要“部分特化”一个函数模板时，通常做法是简单地增加一个重载。看起来就像这样：

```
namespace std {

?template<typename T>              // an overloading of std::swap
?void swap(Widget<T>& a,         ?// (note the lack of "<...>" after
           ?Widget<T>& b)        ?// "swap"), but see below for
?{ a.swap(b); }                    // why this isn't valid code
}
```

通常，重载函数模板确实很不错，但是 std 是一个特殊的 namespace
，规则对它也有特殊的待遇。它认可完全特化 std 中的模板，但它不认可在 std
中增加新的模板（也包括类，函数，以及其它任何东西）。std 的内容由 C++
标准化委员会单独决定，并禁止我们对他们做出的决定进行增加。而且，禁止的方式使你无计可施。打破这条?
 畹某绦蚤畈欢嗟娜房梢员嘁牒驮诵校 堑男形 俏炊T宓摹H绻 阆M 愕娜砑 锌稍て诘男形 憔?
不应该向 std 中加入新的东西。

窗体底端

翻译：Effective C++, 3rd Edition, Item 25: 考虑支持不抛异常的 swap（下）

（点击此处，接上篇）

因此该怎么做呢？我们还是需要一个方法，既使其他人能调用 swap
，又能让我们得到更高效的模板特化版本。答案很简单。我们还是声明一个非成员 swap 来调用成员 swap
，只是不再将那个非成员函数声明为 std::swap 的特化或重载。例如，如果我们的 Widget 相关机能都在 namespace WidgetStuff 中，它看起来就像这个样子：

```
namespace WidgetStuff {
?...                                    // templatized WidgetImpl, etc.

?template<typename T>                 ?// as before, including the swap
?class Widget { ... };                  // member function

?...

?template<typename T>                 ?// non-member swap function;
?void swap(Widget<T>& a,                // not part of the std namespace
           ?Widget<T>& b)
?{
  ?a.swap(b);
?}
}
```

现在，如果某处有代码使用两个 Widget 对象调用 swap，C++ 的名字查找规则（以参数依赖查找（*argument-dependent lookup*）或 Koenig 查找（*Koenig lookup*）著称的特定规则）将找到 WidgetStuff 中的 Widget 专用版本。而这正是我们想要的。

这个方法无论对于类模板还是对于类都能很好地工作，所以看起来我们应该总是使用它。不幸的是，此处还是?嬖谝桓鰴杆 　嗵戲? std::swap 的动机（过一会儿我会讲到它），所以如果你希望你的 swap
的类专用版本在尽可能多的上下文中都能够调用（而你也确实这样做了），你就既要在你的类所在的 namespace 中写一个非成员版本，又要提供一个 std::swap 的特化版本。

顺便提一下，如果你不使用 namespaces，上面所讲的一切依然适用（也就是说，你还是需要一个非成员 swap 来调用成员 swap），但是你为什么要把你的类，模板，函数，枚举（此处作者连用了两个词（enum, enumerant），不知有何区别——译者注）和 typedef 名字都堆在全局 namespace 中呢？你觉得合适吗？

迄今为止我所写的每一件事情都适用于 swap
的作成者，但是有一种状况值得从客户的观点来看一看。假设你写了一个函数模板来交换两个对象的值：

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

哪一个 swap 应该被调用呢？std 中的通用版本，你知道它必定存在；std
中的通用版本的特化，可能存在，也可能不存在；T 专用版本，可能存在，也可能不存在，可能在一个 namespace 中，也可能不在一个 namespace 中（但是肯定不在 std 中）。究竟该调用哪一个呢？如果 T 专用版本存在，你希望调用它，如果它不存在，就回过头来调用 std
中的通用版本。如下这样就可以符合你的希望：

```
template<typename T>
```

```
void doSomething(T& obj1, T& obj2)
{
?using std::swap;              // make std::swap available in this function
?...
?swap(obj1, obj2);          ?// call the best swap for objects of type T
?...
}
```

当编译器看到这个 swap 调用，他会寻找正确的 swap 版本来调用。C++ 的名字查找规则确保能找到在全局 namespace 或者与 T 同一个 namespace 中的 T 专用的 swap。（例如，如果 T 是 namespace WidgetStuff 中的 Widget，编译器会利用参数依赖查找（argument-dependent lookup）找到 WidgetStuff 中的 swap。）如果 T 专用 swap 不存在，编译器将使用 std 中的 swap，这归功于此函数中的 using declaration 使 std::swap 在此可见。尽管如此，相对于通用模板，编译器还是更喜欢 T 专用的 std::swap 的特化，所以如果 std::swap 对 T 进行了特化，则特化的版本会被使用。

得到正确的 swap 调用是如此地容易。你需要小心的一件事是不要对调用加以限定，因为这将影响 C++ 确定该调用的函数，如果你这样写对 swap 的调用，

```
std::swap(obj1, obj2);          // the wrong way to call swap
```

这将强制编译器只考虑 std 中的 swap（包括任何模板特化），因此排除了定义在别处的更为适用的 T 专用版本被调用的可能性。唉，一些被误导的程序员就是用这种方法限定对 swap 的调用，这也就是为你的类完全地特化 std::swap 很重要的原因：它使得以这种被误导的方式写出的代码可以用到类型专用的 swap 实现。（这样的代码还存在于现在的一些标准库实现中，所以它将有利于你帮助这样的代码尽可能高效地工作?#?

到此为止，我们讨论了缺省的 swap，成员 swaps，非成员 swaps，std::swap 的特化版本，以及对 swap 的调用，所以让我们总结一下目前的状况。

首先，如果 swap 的缺省实现为你的类或类模板提供了可接受的性能，你不需要做任何事。任何试图交换你的类型的对象的人都?岬玫饺笔“姹镜闹C郑　　夷芫ぶ鞣煤芎谩?

第二，如果 swap 的缺省实现效率不足（这几乎总是意味着你的类或模板使用了某种 pimpl idiom 的变种），就按照以下步骤来做：

1. 提供一个能高效地交换你的类型的两个对象的值的 public 的 swap 成员函数。出于我过一会儿就要解释的动机，这个函数应该永远不会抛出异常。
2. 在你的类或模板所在的同一个 namespace 中提供一个非成员的 swap。用它调用你的 swap 成员函数。
3. 如果你写了一个类（不是类模板），就为你的类特化 std::swap。用它也调用你的 swap 成员函数。

最后，如果你调用 swap，请确保在你的函数中包含一个 using declaration 使 std::swap 可见，然后在调用 swap 时不使用任何 namespace 限定条件。

唯一没有解决的问题就是我的警告——绝不要让 swap 的成员版本抛出异常。这是因为 swap 的非常重要的应用之一是为类（以及类模板）提供强大的异常安全（exception-safety）保证。Item 29 将提供所有的细节，但是这项技术基于 swap 的成员版本绝不会抛出异常的假设。这一强制约束仅仅应用在成员版本上！它不能够应用在非成员版本上，因?ª swap 的缺省版本基于拷贝构造和拷贝赋值，而在通常情况下，这两个函数都允许抛出异常。如果你写了一个 swap 的自定义版本，那么，典型情况下你是为了提供一个更有效率的交换值的方法，你也要保证这个方法不会抛出?斐！W魑　桓鲆话愎燮颍　饬街? swap 的特型将紧密地结合在一起，因为高效的交换几乎总是基于内建类型（诸如在 pimpl idiom 之下的指针）的操作，而对内建类型的操作绝不会抛出异常。

Things to Remember

- 如果 std::swap 对于你的类型来说是低效的，请提供一个 swap 成员函数。并确保你的 swap 不会抛出异常。
- 如果你提供一个成员 swap，请同时提供一个调用成员 swap 的非成员 swap 。对于类（非模板），还要特化 std::swap。
- 调用 swap 时，请为 std::swap 使用一个 using declaration，然后在调用 swap 时不使用任何 namespace 限定条件。
- 为用户定义类型完全地特化 std 模板没有什么问题，但是绝不要试图往 std 中加入任何全新的东西。

窗体底端

Chapter 5.?实现

在极大程度上，为你的类（包括类模板）和函数（包括函数模板）提供正确的定义是战斗的关键性部?帧R坏├愕玫秸　返慕崩　　嘤Φ氖迪趾芄蟪潭壬暇褪侵苯亓说钡摹5　侨匀挥幸恍┬⑭馐孪钚枰　?心。过早地定义变量会对性能产生拖累。过度使用强制转换会导致缓慢的，难以维护的，被微妙的 bug
困扰的代码。返回一个类内部构件的句柄会破坏封装并将空悬句柄留给客户。疏忽了对异常产生的影?斓目悸腔岬贾伦试吹男孤┗褪　其峁沟钠苹怠9　帜诹　　?inlining
）会导致代码膨胀。过度的耦合会导致令人无法接受的漫长的建构时间。

这一切问题都可以避免，本章就是说明应该如何去做。

Item 26: 只要有可能就推迟变量定义

只要你定义了一个带有构造函数和析构函数的类型的变量，当控制流程到达变量定义的时候会使你担?汗乖斐杀荆　北淞坷肪　饕糜虻氖焙蚣崾鼓愕8何齧钩杀尽H绻　形抻帽淞吭斐烧庖怀杀荆　憔鸵?尽你所能去避免它。

你可能认为你从来不会定义无用的变量，但是也许你应该再想一想。考虑下面这个函数，只要 password 的长度满足要求，它就返回一个 password 的加密版本。如果 password 太短，函数就会抛出一个定义在标准 C++ 库中的 `logic_error` 类型的异常（参见 Item 54）：

```
// this function defines the variable "encrypted" too soon
std::string encryptPassword(const std::string& password)
{
?using namespace std;

?string encrypted;

?if (password.length() < MinimumPasswordLength) {
    ?throw logic_error("Password is too short");
?}
?...                        ?// do whatever is necessary to place an
                               // encrypted version of password in
encrypted
?return encrypted;
}
```

对象 `encrypted` 在这个函数中并不是完全无用，但是如果抛出了一个异常，它就是无用的。换句话说，即使 `encryptPassword` 抛出一个异常，你也要为构造和析构 `encrypted` 付出代价。因此得出以下结论：你最好将 `encrypted` 的定义推迟到你确信你真的需要它的时候：

```
// this function postpones encrypted's definition until it's truly
necessary
std::string encryptPassword(const std::string& password)
{
?using namespace std;

?if (password.length() < MinimumPasswordLength) {
     throw logic_error("Password is too short");
?}

?string encrypted;

?...                      ?// do whatever is necessary to place an
```

```
                                    // encrypted version of password in encrypted
?return encrypted;
}
```

这一代码仍然没有达到它本可以达到的那样紧凑，因为定义 encrypted
的时候没有任何初始化参数。这就意味着很多情况下将使用它的缺省构造函数，对于一个对象你首先?
Ω米龅木褪歉　　恍╤担　饩　？梢酝ü　持道赐瓶伞?Item 4 解释了为什么缺省构造（
default-constructing
）一个对象然后赋值给它比用你真正需要它持有的值初始化它更低效。那个分析也适用于此。例如，?
偕? encryptPassword 的核心部分是在这个函数中完成的：

```
void encrypt(std::string& s);              // encrypts s in place
```

那么，encryptPassword 就可以这样实现，即使它还不是最好的方法：

```
// this function postpones encrypted's definition until
// it's necessary, but it's still needlessly inefficient
std::string encryptPassword(const std::string& password)
{
?...                                       // check length as above

?std::string encrypted;                  ?// default-construct encrypted
?encrypted = password;                     // assign to encrypted

?encrypt(encrypted);
?return encrypted;
}
```

一个更可取得方法是用 password 初始化 encrypted
，从而跳过毫无意义并可能很昂贵的缺省构造：

```
// finally, the best way to define and initialize encrypted
std::string encryptPassword(const std::string& password)
{
?...                                         // check length

?std::string encrypted(password);          ?// define and initialize
                                            ?// via copy constructor

?encrypt(encrypted);
?return encrypted;
}
```

这个建议就是本 Item 的标题中的“只要有可能（as long as possible）”
的真正含义。你不仅应该推迟一个变量的定义直到你不得不用它之前的最后一刻，而且应该试图推迟?
宙丁逯钡侥愕玫搅怂　某跚蓟　问　Mü　庑　淖龇ǎ　憧梢员范夤乖旌臀龋刮抻枚韵螅　　一箍梢?
避免不必要的缺省构造。更进一步，通过在它们的含义已经非常明确的上下文中初始化它们，有助于?
员淞康淖饕梦牡祷　？

“但是对于循环会如何？”
你可能会有这样的疑问。如果一个变量仅仅在一个循环内使用，是循环外面定义它并在每次循环迭代?
备持蹈　　靡恍　　故窃谘　纺诓慷丁逭飧霰淞扛　靡恍├兀恳簿褪撬担　旅黼饬礁龃笈碌慕峁怪?
哪个更好一些?

```
// Approach A: define outside loop    // Approach B: define inside loop

Widget w;
```

```
for (int i = 0; i < n; ++i){
?w = some value dependent on i;
);
?...
}
```

```
for (int i = 0; i < n; ++i) {
 Widget w(some value dependent on i
 ?...
?}
```

这里我将一个类型 string 的对象换成了一个类型 Widget 的对象，以避免对这个对象的构造、析构或赋值操作的成本的任何已有的预见。

对于 Widget 的操作而言，就是下面这两个方法的成本：

- 方法 A：1 个构造函数 + 1 个析构函数 + n 个赋值。
- 方法 B：n 个构造函数 + n 个析构函数。

对于那些赋值的成本低于一个构造函数/析构函数对的成本的类，方法 A 通常更高效。特别是在 n 变得很大的情况下。否则，方法 B 可能更好一些。此外，方法 A 与方法 B 相比，使得名字 w 在一个较大的区域（包含循环的那个区域）内均可见，这可能会破坏程序的易理解性和可维护性。因?说贸鲆韵陆崧郑撼　悄闵沸乓韵铝降悖海?1）赋值比构造函数/析构函数对成本更低，而且（2）你正在涉及你的代码中的性能敏感的部分，否则，你应该默认使用方法 B。

Things to Remember

- 只要有可能就推迟变量定义。这样可以增加程序的清晰度并提高程序的性能。

窗体底端

Item 27: 将强制转型减到最少

C++
的规则设计为保证不会发生类型错误。在理论上，如果你的程序想顺利地通过编译，你就不应该试图对任何对?
笞鏊魏尾话踩　幕蛏抟庖宓牟僮鳖U馑且桓龇浅S屑壑档谋V讠　悴挥Ω们蝴椎胤牌　　?

不幸的是，强制转型破坏了类型系统。它会引起各种各样的麻烦，其中一些容易被察觉，另一些则格外地微妙?
H绻　愦? C，Java，或 C# 转到 C++，请一定注意，因为强制转型在那些语言中比在 C++
中更有必要，危险也更少。但是 C++ 不是 C，也不是 Java，也不是 C#
。在这一语言中，强制转型是一个你必须全神贯注才可以靠近的特性。

我们就从回顾强制转型的语法开始，因为对于同样的强制转型通常有三种不同的写法。C 风格（C-style
）强制转型如下：

```
(T) expression                    ?// cast expression to be of type T
```

函数风格（Function-style）强制转型使用这样的语法：

```
T(expression)                      // cast expression to be of type T
```

这两种形式之间没有本质上的不同，它纯粹就是一个把括号放在哪的问题。我把这两种形式称为旧风格（
old-style）的强制转型。

C++ 同时提供了四种新的强制转型形式（通常称为新风格的或 C++ 风格的强制转型）：

```
const_cast<T>(expression)
dynamic_cast<T>(expression)
reinterpret_cast<T>(expression)
static_cast<T>(expression)
```

每一种适用于特定的目的：

- const_cast 一般用于强制消除对象的常量性。它是唯一能做到这一点的 C++
  风格的强制转型。
- dynamic_cast 主要用于执行"安全的向下转型（safe downcasting）"
  ，也就是说，要确定一个对象是否是一个继承体系中的一个特定类型。它是唯一不能用旧风格语法
  执行的强制转型。也是唯一可能有重大运行时代价的强制转型。（过一会儿我再提供细节。）
- reinterpret_cast 是特意用于底层的强制转型，导致实现依赖（
  implementation-dependent
  ）（就是说，不可移植）的结果，例如，将一个指针转型为一个整数。这样的强制转型在底层代码
  以外应该极为罕见。在本书中我只用了一次，而且还仅仅是在讨论你应该如何为裸内存（raw
  memory）写一个调谐分配者（debugging allocator）的时候（参见 Item 50）。
- static_cast 可以被用于强制隐型转换（例如，non-const 对象转型为 const 对象（就像
  Item 3 中的），int 转型为 double
  ，等等）。它还可以用于很多这样的转换的反向转换（例如，void*
  指针转型为有类型指针，基类指针转型为派生类指针），但是它不能将一个 const 对象转型为
  non-const 对象。（只有 const_cast 能做到。）

旧风格的强制转型依然合法，但是新的形式更可取。首先，在代码中它们更容易识别（无论是人还是像 grep
这样的工具都是如此），这样就简化了在代码中寻找类型系统被破坏的地方的过程。第二，更精确地指定每一?
銮恐谱　偷哪康模　沟帽嘁肫髡锒鲜褂么砦蟪晌　赡堋@　纾　绻　闶酝际褂靡桓? const_cast
以外的新风格强制转型来消除常量性，你的代码将无法编译。

当我要调用一个 explicit
构造函数用来传递一个对象给一个函数的时候，大概就是我仅有的使用旧风格的强制转换的时候。例如：

```
class Widget {
public:
```

```
?explicit Widget(int size);
?...
};

void doSomeWork(const Widget& w);

doSomeWork(Widget(15));                    ?// create Widget from int
                                             // with function-style cast

doSomeWork(static_cast<Widget>(15));         // create Widget from int
                                             // with C++-style cast
```

由于某种原因，有条不紊的对象创建感觉上不像一个强制转型，所以在这个强制转型中我多半会用函数风格的?
恐谱　痛　? static_cast。反过来说，在你写出那些导致 core dump
的代码时，你通常都感觉你有合理的理由，所以你最好忽略你的感觉并始终都使用新风格的强制转型。

很多程序员认为强制转型除了告诉编译器将一种类型看作另一种之外什么都没做，但这是错误的。任何种类的?
嘈妥　唬丐蘼凼峭ü　恐谱　偷南允降幕故潜嘁肫魈砑拥囊　降模┑蓟岬贾略诵惺钡目芍葱写　舜@　纾　谖飧龅　肫　现校?

```
int x, y;
...
double d = static_cast<double>(x)/y;            // divide x by y, but use
                                                // floating point division
```

int x 到 double 的强制转型理所当然要生成代码，因为在大多数系统架构中，一个 int 的底层表示与 double 的不同。这可能还不怎么令人吃惊，但是下面这个例子可能会让你稍微开一下眼：

```
class Base { ... };

class Derived: public Base { ... };

Derived d;

Base *pb = &d;                              // implicitly convert Derived*?→ Base*
```

这里我们只是创建了一个指向派生类对象的基类指针，但是有时候，这两个指针的值并不相同。在当前情况下?
嵩谠诵惺痹? Derived* 指针上应用一个偏移量以得到正确的 Base* 指针值。

这后一个例子表明一个单一的对象（例如，一个类型为 Derived
的对象）可能会有不止一个地址（例如，它的被一个 Base* 指针指向的地址和它的被一个 Derived*
指针指向的地址）。这在 C 中就不会发生，也不会在 Java 中发生，也不会在 C# 中发生，它仅在 C++
中发生。实际上，如果使用了多继承，则一定会发生，但是在单继承下也会发生。与其它事情合在一起，就意?
蹲拍阌Ω米苁潜范舛? C++
如何摆放事物做出假设，你当然也不应该基于这样的假设执行强制转型。例如，将一个对象的地址强制转型为 char* 指针，然后对其使用指针运算，这几乎总是会导致未定义行为。

但是请注意我说一个偏移量是"有时"
被需要。对象摆放的方法和他们的地址的计算方法在不同的编译器之间有所变化。这就意味着仅仅因为你的"
我知道事物是如何摆放的"
而使得强制转型能工作在一个平台上，并不意味着它们也能在其它平台工作。这个世界被通过痛苦的道路学得?
馓蹙　桐目闪　某绦蛟彼　渎　?

关于强制转型的一件有趣的事是很容易写出看起来对（在其它语言中也许是对的）实际上错的东西。例如，许?
嘤τ每蚣埽?application framework
）要求在派生类中实现虚成员函数时要首先调用它们的基类对应物。假设我们有一个 Window 基类和一个 SpecialWindow 派生类，它们都定义了虚函数 onResize。进一步假设 SpecialWindow 的 onResize

被期望首先调用 Window 的 onResize。这就是实现这个的一种方法，它看起来正确实际并不正确：

```
class Window {                                  ?// base class
public:
?virtual void onResize() { ... }                 // base onResize impl
?...
};

class SpecialWindow: public Window {            ?// derived class
public:
?virtual void onResize() {                       // derived onResize impl;
  ?static_cast<Window>(*this).onResize();     ?// cast *this to Window,
                                                ?// then call its onResize;
                                                ?// this doesn't work!

  ?...                                           // do SpecialWindow-
?}                                               // specific stuff

?...

};
```

我突出了代码中的强制转型。（这是一个新风格的强制转型，但是使用旧风格的强制转型也于事无补。）正像?囡 谕 模 虢? *this 强制转型为一个 Window。因此调用 onResize 的结果就是调用 Window::onResize。你也许并不期待它没有调用当前对象的那个函数！作为替代，强制转型创建了一个 *this 的基类部分的新的，临时的拷贝，然后调用这个拷贝的 onResize！上面的代码没有调用当前对象的 Window::onResize，然后再对这个对象执行 SpecialWindow 特有的动作——它在对当前对象执行 SpecialWindow 特有的动作之前，调用了当前对象的基类部分的一份拷贝的 Window::onResize。如果 Window::onResize 改变了当前对象（可能性并不小，因为 onResize 是一个 non-const 成员函数），当前对象并不会改变。作为替代，那个对象的一份拷贝被改变。如果 SpecialWindow::onResize 改变了当前对象，无论如何，当前对象将被改变，导致的境况是那些代码使当前对象进入一种病态，没有做基?嗟谋涓 醋隽伺缮 嗟谋涓 ？

解决方法就是消除强制转型，用你真正想表达的来代替它。你不应该哄骗编译器将 *this 当作一个基类对象来处理，你应该调用当前对象的 onResize 的基类版本。就是这样：

```
class SpecialWindow: public Window {
public:
?virtual void onResize() {
  ?Window::onResize();                    ?// call Window::onResize
  ?...                                    ?// on *this
?}
?...

};
```

这个例子也表明如果你发现自己要做强制转型，这就是你可能做错了某事的一个信号。在你想用 dynamic_cast 时尤其如此。

- 翻译：Effective C++, 3rd Edition, Item 27: 将强制转型减到最少（下）

（点击此处，接上篇）

在探究 dynamic_cast 的设计意图之前，值得留意的是很多 dynamic_cast 的实现都相当慢。例如，至少有一种通用的实现部分地基于对类名字进行字符串比较。如果你在一个?挥谒牟闵畹牡ゼ坛刑逑抵械亩韵笕现葱? dynamic_cast，在这样一个实现下的每一个 dynamic_cast 都要付出相当于四次调用 strcmp 来比较类名字的成本。对于一个更深的或使用了多继承的继承体系，付出的代价会更加昂贵。一些实?钟谜庵址椒ü ぷ魇怯性　虻模ㄋ　遣坏貌徽庋　鲆灾Ｃ侄　　唇樱　＞」苋缡要　　嗽谄毡橐庖迳暇枨恐谱　屯猓　谛阅范艋械拇　胫校　阌Ω锰毛鹌　？dynamic_casts。

对 dynamic_cast 的需要通常发生在这种情况下：你要在一个你确信为派生类的对象上执行派生类的操作，但是你只能?ü　桓龌　嗟闹刚牖蛞　美床倏卣飧韶韵蟆Ｓ辛礁鲆话愕姆椒ń梢员范庶飧鑫侍狻?

第一个，使用存储着直接指向派生类对象的指针（通常是智能指针——参见 Item 13 ）的容器，从而消除通过基类接口操控这个对象的需要。例如，如果在我们的 Window/SpecialWindow 继承体系中，只有 SpecialWindows 支持 blinking，对于这样的做法：

```
class Window { ... };

class SpecialWindow: public Window {
public:
?void blink();
?...
};
typedef                                          ?// see Item 13 for
info
?std::vector<std::tr1::shared_ptr<Window> > VPW;?// on tr1::shared_ptr

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin();         // undesirable code:
     iter != winPtrs.end();                      ?// uses dynamic_cast
     ++iter) {
?if (SpecialWindow *psw = dynamic_cast<SpecialWindow*>(iter->get()))
     psw->blink();
}
```

设法用如下方法代替：

```
typedef std::vector<std::tr1::shared_ptr<SpecialWindow> > VPSW;

VPSW winPtrs;

...

for (VPSW::iterator iter = winPtrs.begin();      ?// better code: uses
     iter != winPtrs.end();                      ?// no dynamic_cast
     ++iter)
?(*iter)->blink();
```

当然，这个方法不允许你在同一个容器中存储所有可能的 Window

的派生类的指针。为了与不同的窗口类型一起工作，你可能需要多个类型安全（type-safe）的容器。

一个候选方法可以让你通过一个基类的接口操控所有可能的 Window 派生类，就是在基类中提供一个让你做你想做的事情的虚函数。例如，尽管只有 SpecialWindows 能 blink，在基类中声明这个函数，并提供一个什么都不做的缺省实现或许是有意义的：

```
class Window {
public:
?virtual void blink() {}                          // default impl is no-op;
?...                                              // see Item 34 for why
};                                               ?// a default impl may be
                                                 ?// a bad idea

class SpecialWindow: public Window {
public:
?virtual void blink() { ... };                    // in this class, blink
?...                                              // does something
};

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;                                     ?// container holds
                                                 ?// (ptrs to) all possible
...                                                // Window types

for (VPW::iterator iter = winPtrs.begin();
     iter != winPtrs.end();
     ++iter)                                     ?// note lack of
?(*iter)->blink();                                // dynamic_cast
```

无论哪种方法——使用类型安全的容器或在继承体系中上移虚函数——都不是到处适用的，但在很多情况下，它们提供了 dynamic_casting 之外另一个可行的候选方法。当它们可用时，你应该加以利用。

你应该绝对避免的一件东西就是包含了极联 dynamic_casts 的设计，也就是说，看起来类似这样的任何东西：

```
class Window { ... };

...                                          // derived classes are defined
here

typedef std::vector<std::tr1::shared_ptr<Window> > VPW;

VPW winPtrs;

...

for (VPW::iterator iter = winPtrs.begin(); iter != winPtrs.end();
++iter)
{
?if (SpecialWindow1 *psw1 =
       dynamic_cast<SpecialWindow1*>(iter->get())) { ... }

?else if (SpecialWindow2 *psw2 =
```

```
          ?dynamic_cast<SpecialWindow2*>(iter->get()) { ... }

?else if (SpecialWindow3 *psw3 =
          ?dynamic_cast<SpecialWindow3*>(iter->get()) { ... }

?...
}
```

这样的 C++ 会生成的代码又大又慢，而且很脆弱，因为每次 Window
类继承体系发生变化，所有这样的代码都要必须被检查，以确认是否需要更新。（例如，如果增加了?
桓鲂碌呐缮　啵　谏厦娴募　　谢蛐砭托枰　尤胍桓鲂碌奶登　种Ａ#┘雌鹄蠢嗨普庚　拇　胪Ω米?
是用基于虚函数的调用的某种东西来替换。

好的 C++ 极少使用强制转型，但在通常情况下完全去除也不实际。例如，第 118 页从 int 到
double
的强制转型，就是对强制转型的合理运用，虽然它并不是绝对必要。（那些代码应该被重写，声明一?
鲂碌睦嘈臀? double 的变量，并用 x
的值进行初始化。）就像大多数可疑的结构成分，强制转型应该被尽可能地隔离，典型情况是隐藏在?
　　诓浚　煤　　慕涌诒；さ饕谜唉独肱诓康奈刍嗟墓ぷ鳖?

Things to Remember

- 避免强制转型的随时应用，特别是在性能敏感的代码中应用 dynamic_casts
  ，如果一个设计需要强制转型，设法开发一个没有强制转型的侯选方案。
- 如果必须要强制转型，设法将它隐藏在一个函数中。客户可以用调用那个函数来代替在他?
  亲约旱拇　胫屑尤肭恐谱　汀?
- 尽量用 C++
  风格的强制转型替换旧风格的强制转型。它们更容易被注意到，而且他们做的事情也更加?
  魅贰?

窗体底端

由于程序的原因，本文件未被完整保存。

窗体顶端

- Item 28: 避免返回对象内部构件的“°句柄”±

假设你正在一个包含矩形的应用程序上工作。每一个矩形都可以用它的左上角和右下角表示出来。为?私 桓?Rectangle 对象保持在较小状态，你可能决定那些点的定义的域不应该包含在 Rectangle 本身之中，更合适的做法是放在一个由 Rectangle 指向的辅助的结构体中：

```
class Point {                       ?// class for representing points
public:
?Point(int x, int y);
?...

?void setX(int newVal);
?void setY(int newVal);
?...
};

struct RectData {                    ?// Point data for a Rectangle
?Point ulhc;                        ?// ulhc = " upper left-hand corner"
?Point lrhc;                        ?// lrhc = " lower right-hand corner"
};

class Rectangle {
?...

private:
?std::tr1::shared_ptr<RectData> pData;        ?// see Item 13 for info
on
};                                                  // tr1::shared_ptr
```

由于 Rectangle 的客户需要有能力操控 Rectangle 的区域，因此类提供了 upperLeft 和 lowerRight 函数。可是，Point 是一个用户定义类型，所以，留心 Item 20 关于在典型情况下，以传引用的方式传递用户定义类型比传值的方式更加高效的观点，这些函数返回?虻撞?Point 对象的引用：

```
class Rectangle {
public:
?...
?Point& upperLeft() const { return pData->ulhc; }
?Point& lowerRight() const { return pData->lrhc; }
?...
};
```

这个设计可以编译，但它是错误的。实际上，它是自相矛盾的。一方面，upperLeft 和 lowerRight 是被声明为 const 的成员函数，因为它们被设计成仅仅给客户提供一个获得 Rectangle 的点的方法，而不允许客户改变这个 Rectangle（参见 Item 3 ）。另一方面，两个函数都返回引向私有的内部数据的引用——调用者可以利用这些引用修改内部数据！例如：

```
Point coord1(0, 0);
Point coord2(100, 100);

const Rectangle rec(coord1, coord2);     // rec is a const rectangle
from
                                         // (0, 0) to (100, 100)

rec.upperLeft().setX(50);               ?// now rec goes from
                                         // (50, 0) to (100, 100)!
```

请注意这里，upperLeft 的调用者是怎样利用返回的 rec 的内部 Point 数据成员的引用来改变这个成员的。但是 rec 却被期望为 const！

这直接引出两条经验。第一，一个数据成员被封装，但是具有最高可访问级别的函数还是能够返回引?蛩 囊 谩T诂鼻扒榭鲋拢 淙?ulhc 和 lrhc 被声明为 private
，它们还是被有效地公开了，因为 public 函数 upperLeft 和 lowerRight
返回了引向它们的引用。第二，如果一个 const
成员函数返回一个引用，引向一个与某个对象有关并存储在这个对象本身之外的数据，这个函数的调?谜呔涂梢愿谋渭歉鏊 苋尸庶 嵌 莆怀A啃缘木室扌裕ú渭?Item 3）的一个副作用）。

我们前面做的每件事都涉及到成员函数返回的引用，但是，如果它们返回指针或者迭代器，因为同样?脑 蛞不崇婪谕 奈侍狻R 茫 刚耄 偷 鞠际蔷浔 ╤andle）（持有其它对象的方法），而返回一个对象内部构件的句柄总是面临危及对象封装安全的风险。就像我们看到的，它同时还能导致 const 成员函数改变了一个对象的状态。

我们通常认为一个对象的“°内部构件”±
就是它的数据成员，但是不能被常规地公开访问的成员函数（也就是说，它是 protected 或 private
的）也是对象内部构件的一部分。同样地，不要返回它们的句柄也很重要。这就意味着你绝不应该有?桓龊稍焙 祷匾桓鲋赶蛲涤薪闲〉目煞梦始侗鸬某稍焙 闹刚舜H绻 阙度 隽耍 目煞梦始?别就会与那个拥有较大的可访问级别的函数相同，因为客户能够得到指向这个拥有较小的可访问级别?暮 闹刚耄 缓缶涂梢酝ü 飧鲋刚氲饕谜飧龌 ?

无论如何，返回指向成员函数的指针的函数是难得一见的，所以让我们把注意力返回到 Rectangle 类和它的 upperLeft 和 lowerRight
成员函数。我们在这些函数中挑出来的问题都只需简单地将 const
用于它们的返回类型就可以排除：

```
class Rectangle {
public:
?...
?const Point& upperLeft() const { return pData->ulhc; }
?const Point& lowerRight() const { return pData->lrhc; }
?...
};
```

通过这个修改的设计，客户可以读取定义一个矩形的 Points，但他们不能写它们。这就意味着将 upperLeft 和 upperRight 声明为 const
不再是一句空话，因为他们不再允许调用者改变对象的状态。至于封装的问题，我们总是故意让客户?吹阶龊梢桓?Rectangle 的 Point
s，所以这是封装的一个故意的放松之处。更重要的，它是一个有限的放松：只有读访问是被这些函数允许的，写访问依然被禁止。

虽然如此，upperLeft 和 lowerRight
仍然返回一个对象内部构件的句柄，而这有可能造成其它方面的问题。特别是，这会导致空悬句柄：?
味瞬辉俅婪诂亩韵蟮墓辜 木浔 U庵室 丫亩韵蟮淖钇胀ū睦丛淳褪呛 祷 刂怠@ 纾 悸且桓龌 祷卦谝桓鼍匾未疤逯械?GUI 对象的 bounding box：

```
class GUIObject { ... };

const Rectangle                                  // returns a rectangle by
?boundingBox(const GUIObject& obj);       ?// value; see Item 3 for why
                                           ?// return type is const
```

现在，考虑客户可能会这样使用这个函数：

```
GUIObject *pgo;                                  // make pgo point to
...                                              // some GUIObject

const Point *pUpperLeft =                        // get a ptr to the upper
?&(boundingBox(*pgo).upperLeft());              // left point of its
                                                ?// bounding box
```

对 boundingBox 的调用会返回一个新建的临时的 Rectangle
对象。这个对象没有名字，所以我们就称它为 temp。于是 upperLeft 就在 temp
上被调用，这个调用返回一个引向 temp 的一个内部构件的引用，特别是，它是由 Points
构成的。随后 pUpperLeft 指向这个 Point

对象。到此为止，一切正常，但是我们无法继续了，因为在这个语句的末尾，boundingBox 的返回值—— temp ——被销毁了，这将间接导致 temp 的 Points 的析构。接下来，剩下 pUpperLeft 指向一个已经不再存在的对象；pUpperLeft 空悬在创建它的语句的末尾！

这就是为什么任何返回一个对象的内部构件的句柄的函数都是危险的。它与那个句柄是指针，引用，？故堑　　髏皇裁垂叵怠K　胧欠袷芳?cosnt
的限制没什么关系。它与那个成员函数返回的句柄本身是否是 const
没什么关系。全部的问题在于一个句柄被返回了，因为一旦这样做了，你就面临着这个句柄比它引用?亩韵蟪　な俚姆缦铡?

这并不意味着你永远不应该让一个成员函数返回一个句柄。有时你必须如此。例如，operator[] 允许你从 string 和 vector 中取出单独的元素，而这些 operator[]s
就是通过返回引向容器中的数据的引用来工作的（参见 Itme 3）——
当容器本身被销毁，数据也将销毁。尽管如此，这样的函数属于特例，而不是惯例。

Things to Remember

- 避免返回对象内部构件的句柄（引用，指针，或迭代器）。这样会提高封装性，帮助 const 成员函数产生 cosnt 效果，并将空悬句柄产生的可能性降到最低。

窗体底端

Item 29: 争取异常安全（exception-safe）的代码

异常安全（Exception safety）有点像怀孕（pregnancy）
但是，请把这个想法先控制一会儿。我们还不能真正地议论生育（reproduction
），直到我们排除万难渡过求爱时期（courtship）。（此段作者使用的 3 个词均有双关含义，pregnancy
也可理解为富有意义，reproduction 也可理解为再现，再生，courtship
也可理解为争取，谋求。为了与后面的译文对应，故按照现在的译法。——译者注）

假设我们有一个类，代表带有背景图像的 GUI
菜单。这个类被设计成在多线程环境中使用，所以它有一个用于并行控制（concurrency control
）的互斥体（mutex）：

```
class PrettyMenu {
public:
?...
?void changeBackground(std::istream& imgSrc);           // change background
?...                                                   ?// image

private:

?Mutex mutex;                   ?// mutex for this object

?Image *bgImage;                  // current background image
?int imageChanges;                // # of times image has been changed
};
```

考虑这个 PrettyMenu 的 changeBackground 函数的可能的实现：

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
?lock(&mutex);                     ?// acquire mutex (as in Item 14)

?delete bgImage;                   ?// get rid of old background
?++imageChanges;                   ?// update image change count
?bgImage = new Image(imgSrc);       // install new background

?unlock(&mutex);                   ?// release mutex
}
```

从异常安全的观点看，这个函数烂到了极点。异常安全有两条要求，而这里全都没有满足。

当一个异常被抛出，异常安全的函数应该：

- 没有资源泄露。上面的代码没有通过这个测试，因为如果 "new Image(imgSrc)"
  表达式产生一个异常，对 unlock 的调用就永远不会执行，而那个互斥体也将被永远挂起。
- 不允许数据结构恶化。如果 "new Image(imgSrc)" 抛出异常，bgImage
  被遗留下来指向一个被删除对象。另外，尽管并没有将一张新的图像设置到位，imageChanges
  也已经被增加。（在另一方面，旧的图像被明确地删除，所以我料想你会争辩说图像已经被"改变"
  了。）

规避资源泄露问题比较容易，因为 Item 13 解释了如何使用对象管理资源，而 Item 14 又引进了 Lock
类作为一种时尚的确保互斥体被释放的方法：

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
?Lock ml(&mutex);                  // from Item 14: acquire mutex and
```

```
                                        // ensure its later release
?delete bgImage;
?++imageChanges;
?bgImage = new Image(imgSrc);
}
```

关于像 Lock 这样的资源管理类的最好的事情之一是它们通常会使函数变短。看到对 unlock 的调用不再需要了吗？作为一个一般的规则，更少的代码就是更好的代码。因为在改变的时候这样可以较少误? 肫缤静13.仙俨　　蠼狻?

随着资源泄露被我们甩在身后，我们可以把我们的注意力集中到数据结构恶化。在这里我们有一个选择，但是? 谖颐悄苎≡裰　埃　颐潜甑胂让娑远丁逦颐堑难≡竦氖跤铼?

异常安全函数提供下述三种保证之一：

- 函数提供基本保证（**the basic guarantee**）
  ，允诺如果一个异常被抛出，程序中剩下的每一件东西都处于合法状态。没有对象或数据结构被破坏，而且所有的对象都处于内部调和状态（所有的类不变量都被满足）。然而，程序的精确状态可能是不可预期的。例如，我们可以重写 changeBackground，以致于如果一个异常被抛出，
  PrettyMenu
  对象可以继续保留原来的背景图像，或者它可以持有某些缺省的背景图像，但是客户无法预知到底是哪一个。（为了查明这一点，他们大概必须调用某个可以告诉他们当前背景图像是什么的成员函数。）
- 函数提供强力保证（**the strong guarantee**）
  ，允诺如果一个异常被抛出，程序的状态不会发生变化。调用这样的函数在感觉上是极其微弱的，如果它们成功了，它们就完全成功，如果它们失败了，程序的状态就像它们从没有被调用过一样。

与提供强力保证的函数一起工作比与只提供基本保证的函数一起工作更加容易，因为调用提供强力保证的函数? 螅　鲇辛街挚赡芳某绦蜃刺　合裨て谝谎　晒ｘ葱辛撕　　蛘呒绦　3趾　　坏饕檬钡笔钡淖刺　Ｓ胫　? 比，如果调用只提供基本保证的函数引发了异常，程序可能存在于任何合法的状态。

- 函数提供不抛出保证（**the nothrow guarantee**）
  ，允诺决不抛出异常，因为它们只做它们答应要做的。所有对内建类型（例如，ints
  ，指针，等等）的操作都是不抛出（nothrow
  ）的（也就是说，提供不抛出保证）。这是异常安全代码中必不可少的基础构件。

假定一个带有空的异常规格（exception specification ）的函数是不抛出的似乎是合理的，但这不一定正确的。例如，考虑这个函数：

```
int doSomething() throw();          ?// note empty exception spec.
```

这并不是说 doSomething 永远不会抛出异常；而是说如果 doSomething 抛出一个异常，它就是一个严重的错误，应该调用 unexpected 函数[1]。实际上，doSomething 可能根本不提供任何异常保证。一个函数的声明（如果有的话，也包括它的异常规格（exception specification ））不能告诉你一个函数是否正确，是否可移植，或是否高效，而且，即便有，它也不能告诉你它会提供哪一? 忠斐 0 踩　Ｖぁ Ｋ　姓庑┫甑远加珊　　氖迪志韶ǎ　　皇撬　纳　骼芫韶ū摹?

[1] 关于 unexpected 函数的资料，可以求助于你中意的搜索引擎或包罗万象的 C++ 课本。（你或许有幸搜到 set_unexpected，这个函数用于指定 unexpected 函数。）

异常安全函数必须提供上述三种保证中的一种。如果它没有提供，它就不是异常安全的。于是，选择就在于决? 阈吹拿恳桓龌　　烤挂　峁┣闹直Ｖぁ 3　且　　硪帕粝吕吹姆且斐 0 踩　拇　耄ū? Item 稍后我们要讨论这个问题），只有当你的最高明的需求分析团队为你的应用程序识别出的一项需求就是泄漏资? 匆约霸诵杏诒黄苹档氖　萁峁怪　鲜保　惶峁┭斐 0 踩　Ｖげ拍艹晌　桓鲅∠睢?

作为一个一般性的规则，你应该提供实际可达到的最强力的保证。从异常安全的观点看，不抛出的函数（nothrow functions）是极好的，但是在 C++ 的 C 部分之外部不调用可能抛出异常的函数简直就是寸步难行。使用动态分配内存的任何东西（例如，所有的 STL 容器）如果不能找到足够的内存来满足一个请求（参见 Item 49），在典型情况下，它就会抛出一个 bad_alloc 异常。只要你能做到就提供不抛出保证，但是对于大多数函数，选择是在基本的保证和强力的保证之间的。

在 changeBackground 的情况下，提供差不多的强力保证并不困难。首先，我们将 PrettyMenu 的 bgImage 数据成员的类型从一个内建的 Image* 指针改变为 Item 13 中描述的智能资源管理指针中的一种。坦白地讲，在预防资源泄漏的基本原则上，这完全是一个好主意。它帮? 颐翘峁┣看蟮囊斐 0 踩 Vさ氖率到 徊郊忧苛? Item 13 的论点——使用对象（诸如智能指针）管理资源是良好设计的基础。在下面的代码中，我展示了 tr1::shared_ptr 的使用，因为当进行通常的拷贝时它的更符合直觉的行为使得它比 auto_ptr 更可取。

第二，我们重新排列 changeBackground 中的语句，以致于直到图像发生变化，才增加 imageChanges。这是一个很好的策略——直到某件事情真正发生了，再改变一个对象的状态来表示某事已经发生。

这就是修改之后的代码：

```
class PrettyMenu {
?...
?std::tr1::shared_ptr<Image> bgImage;
?...
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
?Lock ml(&mutex);

?bgImage.reset(new Image(imgSrc));?// replace bgImage's internal
                                    // pointer with the result of the
                                    // "new Image" expression
?++imageChanges;
}
```

注意这里不再需要手动删除旧的图像，因为在智能指针内部已经被处理了。此外，只有当新的图像被成功创建? 松境 形 呕岱⑸ 8 既返厮担 挥械? tr1::shared_ptr::reset 函数的参数（"new Image(imgSrc)" 的结果）被成功创建了，这个函数才会被调用。只有在对 reset 的调用的内部才会使用 delete，所以如果这个函数从来不曾进入，delete 就从来不曾使用。同样请注意一个管理资源（动态分配的 Image）的对象（tr1::shared_ptr）的使用再次缩短了 changeBackground 的长度。

正如我所说的，这两处改动差不多有能力使 changeBackground 提供强力异常安全保证。美中不足的是什么呢？参数 imgSrc。如果 Image 的构造函数抛出一个异常，输入流（input stream）的读标记（read marker）可能已经被移动，而这样的移动就成为对程序的其它部分来说可见的一个状态的变化。直到 changeBackground 着手解决这个问题之前，它只能提供基本异常安全保证。

窗体底端

由于程序的原因，本文件未被完整保存。

- 翻译：Effective C++, 3rd Edition, Item 29:
  争取异常安全（exception-safe）的代码（下）

  （点击此处，接上篇）

  无论如何，让我们把它放在一边，并且依然假装 changeBackground
  可以提供强力保证。（我相信你至少能用一种方法做到这一点，或许可以通过将它的参数从一个
  istream
  改变到包含图像数据的文件的文件名。）有一种通常的设计策略可以有代表性地产生强力保证，而且?
  煜に　欠浅 1 圔　摹U飧霾呗员怀莆?″copy and
  swap″。它的原理很简单。先做出一个你要改变的对象的拷贝，然后在这个拷贝上做出全部所需的改变
  。如果改变过程中的某些操作抛出了异常，最初的对象保持不变。在所有的改变完全成功之后，将被?
  谋涞亩韵蠛妥畛醢蕴亩韵笤谝桓霾换峭壮鲆斐 5 牟僮髦薪　薪换弧?

  这通常通过下面的方法实现：将每一个对象中的全部数据从“°真正的”±
  对象中放入到一个单独的实现对象中，然后将一个指向实现对象的指针交给真正对象。这通常被称为
  ″pimpl idiom″，Item 31 描述了它的一些细节。对于 PrettyMenu 来说，它一般就像这样：

```
struct PMImpl {                                  // PMImpl = "PrettyMenu
?std::tr1::shared_ptr<Image> bgImage;       ?// Impl."; see below for
?int imageChanges;                            // why it's a struct
};

class PrettyMenu {
?...

private:
?Mutex mutex;
?std::tr1::shared_ptr<PMImpl> pImpl;
};

void PrettyMenu::changeBackground(std::istream& imgSrc)
{
?using std::swap;                                ?// see Item 25

?Lock ml(&mutex);                                ?// acquire the mutex

?std::tr1::shared_ptr<PMImpl>                    ?// copy obj. data
  ?pNew(new PMImpl(*pImpl));

?pNew->bgImage.reset(new Image(imgSrc));       // modify the copy
?++pNew->imageChanges;

?swap(pImpl, pNew);                              ?// swap the new
                                                  ?// data into place

}                                                  // release the mutex
```

  在这个例子中，我选择将 PMImpl 做成一个结构体，而不是类，因为通过让 pImpl 是 private
  就可以确保 PrettyMenu 数据的封装。将 PMImpl
  做成一个类虽然有些不那么方便，却没有增加什么好处。（这也会使有面向对象洁癖者走投无路。）?
  绻　阁敢猓?PMImpl 可以嵌套在 PrettyMenu
  内部，像这样的打包问题与我们这里所关心的写异常安全的代码的问题没有什么关系。

  copy-and-swap
  策略是一种全面改变或丝毫不变一个对象的状态的极好的方法，但是，在通常情况下，它不能保证全?
  亢　　际乔苛σ斐 0 踩　摹N　伺　透　颍　悸且桓?changeBackground 的抽象化身——
  someFunc，它使用了 copy-and-swap，但是它包含了对另外两个函数（f1 和 f2）的调用：

```
void someFunc()
{
?...                                          // make copy of local state
```

```
?f1();
?f2();
?...                                        // swap modified state into
place
}
```

很明显，如果 f1 或 f2 低于强力异常安全，someFunc 就很难成为强力异常安全的。例如，假设 f1 仅提供基本保证。为了让 someFunc 提供强力保证，它必须写代码在调用 f1 之前测定整个程序的状态，并捕捉来自 f1 的所有异常，然后恢复到最初的状态。

即使 f1 和 f2 都是强力异常安全的，事情也好不到哪去。如果 f1 运行完成，程序的状态已经发生了毫无疑问的变化，所以如果随后 f2 抛出一个异常，即使 f2 没有改变任何东西，程序的状态也已经和调用 someFunc 时不同。

问题在于副作用。只要函数仅对局部状态起作用（例如，someFunc 仅仅影响调用它的那个对象的状态），它提供强力保证就相对容易。当函数的副作用影响了非局部数?荩　突崂　训枚唷@　纾　绻　饕?f1 的副作用是改变数据库，让 someFunc 成为强力异常安全就非常困难。一般情况下，没有办法撤销已经提交的数据库变化，其他数据库客户?赡艽丫　醇　耸　菸獾男伦刺　?

类似这样的问题会阻止你为函数提供强力保证，即使你希望去做。另一个问题是效率。copy-and-swap 的要点是这样一个想法：改变一个对象的数据的拷贝，然后在一个不会抛出异常的操作中将被改变的?　莺驮　际　萁　薪换弧Ｕ饩托枰　龊雒恳桓鲆　谋涞亩韵蟮目奖矗　钰赡芑崾玫侥悴荒芑蛋磺樵付?用的时间和空间。强力保证是非常值得的，当它可用时你应该提供它，除非在它不能 100% 可用的时候。

当它不可用时，你就必须提供基本保证。在实践中，你可能会发现你能为某些函数提供强力保证，但?切　屎透丛佣鹊某杀臼沟盟　岩灾Ｃ执罅康钠渌　　Ｎ蘼酆问保　灰　阕鞒龋　桓鎏峁┣苛ＰＶさ?合理的成果，就没有人会因为你仅仅提供了基本保证而站在批评你的立场上。对于很多函数来说，基?颈Ｖな且桓鳖耆　侠淼难═瘪?

如果你写了一个根本没有提供异常安全保证的函数，事情就不同了，因为在这一点上有罪推定是合情?侠淼模　钡侥阒っ髯约菏乔灏椎摹Ｄ阌Ω眯闯鲆斐０踩　拇　舜３　悄隰茏龊鲇兴捣　Φ拇鸨纭Ｇ朐?次考虑 someFunc 的实现，它调用了函数 f1 和 f2。假设 f2 根本没有提供异常安全保证，甚至没有基本保证。这就意味着如果 f2 发生一个异常，程序可能会在 f2 内部泄漏资源。这也意味着 f2 可能会恶化数据结构，例如，已排序数组可能不再排序，一个正在从一个数据结构传送到另一个数据?峁谷サ亩韵罂赡芏　Ｂ　鹊取Ｃ挥腥魏伟旆ń梢匀?someFunc 能弥补这些问题。如果 someFunc 调用的函数不提供异常安全保证，someFunc 本身就不能提供任何保证。

请允许我回到怀孕。一个女性或者怀孕或者没有。局部怀孕是绝不可能的。与此相似，一个软件或者?且斐０踩　幕蛘卟皇恰Ｃ挥邢褚桓鼍植恳斐０踩　南低痴庑　亩　鳖Ｒ桓鲋低臣词怪挥幸桓龌　　皇?异常安全的，那么系统作为一个整体就不是异常安全的，因为调用那个函数可能发生泄漏资源和恶化?　其峁埂２恍业氖牵　芏?Ｃ++ 的遗留代码在写的时候没有留意异常安全，所以现在的很多系统都不是异常安全的。它们混合了用非?斐０踩　┤xception-unsafe）的方式书写的代码。

没有理由让事情的这种状态永远持续下去。当书写新的代码或改变现存代码时，要仔细考虑如何使它?斐０踩　Ｒ允褂枚韵蠊芾碜试纯　肌＃ɑ故遣渭?Item 13。）这样可以防止资源泄漏。接下来，决定三种异常安全保证中的哪一种是你实际上能够为你写的每?桓龌　　峁┑ 淖钋康谋Ｖぃ　挥械蹦悴坏饕靡帕舸　刖捅鹞扪═竦氖焙颍　拍劳　阌诿挥斜Ｖぁ＜仁?为你的函数的客户也是为了将来的维护人员，文档化你的决定。一个函数的异常安全保证是它的接口?目杉　糠郑　　阅阌Ω锰匾庋═袼　　拖衲闾匾庋═褚桓龌　　涌诘钠渌　矫妗?

四十年前，到处都是 goto 的代码被尊为最佳实践。现在我们为书写结构化控制流程而奋斗。二十年前，全局可访问数据被尊为?罢咽导　Ｏ衷谖颐俏　庾笆　获　芏罚　　暌郧埃　春　　辈槐乜悸且斐５挠跋戮蛔鹞　罢咽导　Ｏ?在我们为写异常安全的代码而奋斗。

时光在流逝。我们生活着。我们学习着。

Things to Remember

- 即使当异常被抛出时，异常安全的函数不会泄露资源，也不允许数据结构被恶化。这样的函数提供基本的，强力的，或者不抛出保证。
- 强力保证经常可以通过 copy-and-swap 被实现，但是强力保证并非对所有函数都可用。
- 一个函数通常能提供的保证不会强于他所调用的函数中最弱的保证。

窗体底端

Item 30: 理解 inline 化的介入和排除

inline 函数——
多么棒的主意啊！它们看起来像函数，它们产生的效果也像函数，它们在各方面都比宏好得太多太多（参见 Item 2），而你却可以在调用它们时不招致函数调用的成本。你还有什么更多的要求呢？

实际上你得到的可能比你想的更多，因为避免函数调用的成本只是故事的一部分。在典型情况下，编译器的优? 俏 艘欢瘟 拿挥泻 饕玫拇 肮杓频模 缘蹦? inline
化一个函数，你可能就使得编译器能够对函数体实行上下文相关的特殊优化。大多数编译器都不会对 "outlined" 函数调用实行这样的优化。

然而，在编程中，就像在生活中，没有免费午餐，而 inline 函数也不例外。一个 inline
函数背后的思想是用函数本体代替每一处对这个函数的调用，而且不必拿着统计表中的 Ph.D.
就可以看出这样可能会增加你的目标代码的大小。在有限内存的机器上，过分热衷于 inline
化会使得程序对于可用空间来说过于庞大。即使使用了虚拟内存，inline
引起的代码膨胀也会导致附加的分页调度，减少指令缓存命中率，以及随之而来的性能损失。

在另一方面，如果一个 inline
函数本体很短，为函数本体生成的代码可能比为一个函数调用生成的代码还要小。如果是这种情况，inline
化这个函数可以实际上导致更小的目标代码和更高的指令缓存命中率！

记住，inline
是向编译器发出的一个请求，而不是一个命令。这个请求能够以显式的或隐式的方式提出。隐式的方法就是在? 桓隼喽丁宓哪诓慷丁逡桓龌 ?

```
class Person {
public:
?...
?int age() const { return theAge; }  ?// an implicit inline request: age is
?...                                   // defined in a class definition

private:
?int theAge;
};
```

这样的函数通常是成员函数，但是 Item 46
解释了友元函数也能被定义在类的内部，如果它们在那里，它们也被隐式地声明为 inline。

显式地声明一个 inline 函数的方法是在它的声明之前加上 `inline` 关键字。例如，以下就是标准 `max`
模板（来自 `<algorithm>`）经常用到的的实现方法：

```
template<typename T>                                  // an explicit inline
inline const T& std::max(const T& a, const T& b)      // request: std::max is
{ return a < b ? b : a; }                           ?// preceded by "inline"
```

`max` 是一个模板的事实引出一个观察结论：inline
函数和模板一般都是定义在头文件中的。这就使得一些程序员得出结论断定函数模板必须是 inline
。这个结论是非法的而且有潜在的危害，所以它值得我们考察一下。

inline 函数一般必须在头文件内，因为大多数构建环境在编译期间进行 inline
化。为了用被调用函数的函数本体替换一个函数调用，编译器必须知道函数看起来像什么样子。（有一些构建? 肪晨梢栽诹 悠诩浣 ? inline 化，还有少数几个——比如，基于 .NET Common Language Infrastructure (CLI) 的控制环境——居然能在运行时 inline 化。然而，这些环境都是例外，并非规则。inline 化在大多数 C++ 程序中是一个编译时行为。）

模板一般在头文件内，因为编译器需要知道一个模板看起来像什么以便用到它时对它进行实例化。（同样，也? 皇侨 咳绱恕 R 恍 ┗菇α肪晨梢栽诹 悠诩浣 心 0 迨道 H 欢 嘁肫谴道 毡椤＃?

模板实例化与 inline 化无关。如果你写了一个模板，而且你认为所有从这个模板实例化出来的函数都应该是

inline 的，那么就声明这个模板为 inline，这就是上面的 `std::max` 的实现被做的事情。但是如果你为没有理由要 inline 化的函数写了一个模板，就要避免声明这个模板为 inline（无论显式的还是隐式的）。inline 化是有成本的，而且你不希望在毫无预见的情况下遭遇它们。我们已经说到 inline 化是如何引起代码膨胀的（这对于模板作者来说是极为重要的一个考虑事项——参见 Item 44），但是，还有其它的成本，过一会儿我们再讨论。

在做这件事之前，我们先来完成对这个结论的考察：inline 是一个编译器可能忽略的请求。大多数编译器拒绝它们认为太复杂的 inline 函数（例如，那些包含循环或者递归的），而且，除了最细碎的以外的全部虚拟函数的调用都不会被 inline 化。不应该对这后一个结论感到惊讶。虚拟意味着“等待，直到运行时才能断定哪一个函数被调用”，而 inline 意味着“执行之前，用被调用函数取代调用的地方”。如果编译器不知道哪一个函数将被调用，你很难责备它们拒绝 inline 化这个函数本体。

所有这些加在一起，得出：一个被指定的 inline 函数是否能真的被 inline 化，取决于你所使用的构建环境——主要是编译器。幸运的是，大多数编译器都有一个诊断层次，在它们不能 inline 化一个你提出的函数时，会导致一个警告（参见 Item 53）。

窗体底端

由于程序的原因，本文件未被完整保存。

翻译：Effective C++, 3rd Edition, Item 30: 理解 inline 化的介入和排除（下）

（点击此处，接上篇）

有时候，即使当编译器完全心甘情愿地 inline 化一个函数，他们还是会为这个 inline 函数生成函数本体。例如，如果你的程序要持有一个 inline 函数的地址，编译器必须为它生成一个 outlined 函数本体。他们怎么能生成一个指向根本不存在的函数的指针呢？再加上，编译器一般不会对通过函数指针的?饕媒 ?inline 化，这就意味着，对一个 inline 函数的调用可能被也可能不被 inline 化，依赖于这个调用是如何做成的：

```
inline void f() {...}    ?// assume compilers are willing to inline calls to f

void (*pf)() = f;        ?// pf points to f

...

f();                     ?// this call will be inlined, because it's a "normal"
call
pf();                      // this call probably won't be, because it's through
                         ?// a function pointer
```

甚至在你从来没有使用函数指针的时候，未 inline 化的 inline 函数的幽灵也会时不时地拜访你，因为程序员并不必然是函数指针的唯一需求者。有时候编译器会生成构造函? 臀龋购 ?out-of-line 拷贝，以便它们能得到指向这些函数的指针，在对数组中的对象进行构造和析构时使用。

事实上，构造函数和析构函数对于 inline 化来说经常是一个比你在不经意的检查中所能显示出来的更加糟糕的候选者。例如，考虑下面这个类 Derived 的构造函数：

```
class Base {
public:
?..

private:
   std::string bm1, bm2;               // base members 1 and 2
};

class Derived: public Base {
public:
?Derived() {}                         // Derived's ctor is empty - or is it?
?...

private:
?std::string dm1, dm2, dm3;           // derived members 1-3
};
```

这个构造函数看上去像一个 inline 化的极好的候选者，因为它不包含代码。但是视觉会被欺骗。

C++ 为对象被创建和被销毁时所发生的事情做出了各种保证。例如，当你使用 new 时，你的动态的被创建对象会被它们的构造函数自动初始化，而当你使用 delete 。则相应的析构函数会被调用。当你创建一个对象时，这个对象的每一个基类和每一个数据成员都会自动构造? 币桓韶韵蟊幌 偶保 蚍⑸ 赜谖龋沟姆聪蚬 獭H绻 谝桓韶韵蠊乖贫诩溆幸桓鲆斐 1 慌壮觯 飧韶韵?已经完成构造的任何部分都被自动销毁。所有这些情节，C++ 只说什么必须发生，但没有说如何发生。那是编译器的实现者的事，但显然这些事情不会自己发生。在你的程?蛑斜斜胗幸恍┐ 胧拐庑┼路⑸ 庑┐ ?——由编译器写出的代码和在编译期间插入你的程序的代码——必须位于某处。有时它们最终就位于构造函数和析构函数中，所以我们可以设想实现为上面那个声称为空的 Derived 的构造函数生成的代码就相当于下面这样：

```
Derived::Derived()                      // conceptual implementation of
{                                      ?// "empty" Derived ctor

 Base::Base();                          // initialize Base part
```

```
  try { dm1.std::string::string(); }     ?// try to construct dm1
  catch (...) {                            // if it throws,
    Base::~Base();                       ?// destroy base class part and
    throw;                               ?// propagate the exception
  }

  try { dm2.std::string::string(); }     ?// try to construct dm2
  catch(...) {                           ?// if it throws,
    dm1.std::string::~string();              // destroy dm1,
    Base::~Base();                       ?// destroy base class part, and
    throw;                               ?// propagate the exception
  }

  try { dm3.std::string::string(); }     ?// construct dm3
  catch(...) {                           ?// if it throws,
    dm2.std::string::~string();              // destroy dm2,
    dm1.std::string::~string();              // destroy dm1,
    Base::~Base();                       ?// destroy base class part, and
    throw;                               ?// propagate the exception
  }
}
```

这些代码并不代表真正的编译器所生成的，因为真正的编译器会用更复杂的方法处理异常。尽管如此，它还是?既返胤从沉?Derived 的“°空”±构造函数必须提供的行为。不论一个编译器的异常多么复杂，Derived 的构造函数至少必须调用它的数据成员和基类的构造函数，而这些调用（它们自己也可能是 inline 的）会影响它对于 inline 化的吸引力。

同样的原因也适用于 Base 的构造函数，所以如果它是 inline 的，插入它的全部代码也要插入 Derived 的构造函数（通过 Derived 的构造函数对 Base 的构造函数的调用）。而且如果 string 的构造函数碰巧也是 inline 的，Derived 的构造函数中将增加五个那个函数代码的拷贝，分别对应于 Derived 对象中的五个 strings（两个继承的加上三个它自己声明的）。也许在现在，为什么说是否 inline 化 Derived 的构造函数不是一个不经大脑的决定就很清楚了。类似的考虑也适用于 Derived 的析构函数，用同样的或者不同的方法，必须保证所有被 Derived 的构造函数初始化的对象被完全销毁。

库设计者必须评估声明函数为 inline 的影响，因为为库中的客户可见的 inline 函数提供二进制升级版本是不可能的。换句话说，如果 f 是一个库中的一个 inline 函数，库的客户将函数 f 的本体编译到他们的应用程序中。如果一个库的实现者后来决定修改 f，所有使用了 f 的客户都必须重新编译。这常常会令人厌烦。在另一方面，如果 f 是一个非 inline 函数，对 f 的改变只需要客户重新连接。这与重新编译相比显然减轻了很大的负担，而且，如果库中包含的函数是动态链?拥模　饪褪且恢侄杂谟没IO此低耆　该鞒姆椒ā?

为了程序开发的目标，在头脑中牢记这些需要考虑的事项是很重要的，但是从编码期间的实用观点来看，占有?　涞匚坏氖率凳牵捍蠖嗍　魇云骰嵘?inline 函数发生冲突。这不应该是什么重大的发现。你怎么能在一个不在那里的函数中设置断点呢？虽然一些构建环?成璽ㄖC?inline 函数的调试，多数环境还是简单地为调试构建取消了 inline 化。

这就导出了一个用于决定哪些函数应该被声明为 inline，哪些不应该的合乎逻辑的策略。最初，不要 inline 任何东西，或者至少要将你的 inline 化的范围限制在那些必须 inline 的（参见 Item 46）和那些实在微不足道的（就像第 135 页上的 Person::age）函数上。通过慎重地使用 inline，你可以使调试器的使用变得容易，但是你也将 inline 化放在了它本来应该在的地位：作为一种手动的优化。不要忘记由经验确定的 80-20 规则，它宣称一个典型的程序用 80% 的时间执行 20% 的代码。这是一个重要的规则，因为它提醒你作为一个软件开发者的目标是识别出能全面提升你的程序性能的 20% 的代码。你可以 inline 或者用其他方式无限期地调节你的函数，但除非你将精力集中在正确的函数上，否则就是白白浪费精力。

Things to Remember

- 将大部分 inline 限制在小的，调用频繁的函数上。这使得程序调试和二进制升级更加容易，最小化潜在的代码膨胀，?⑲畚蠡　峋叱绦蛩俣鹊募嘎省?
- 不要仅仅因为函数模板出现在头文件中，就将它声明为 inline。

窗体底端

Item 31: 最小化文件之间的编译依赖

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

你进入到你的程序中，并对一个类的实现进行了细微的改变。提醒你一下，不是类的接口，只是实现，仅仅是 private 的东西。然后你重建（rebuild ）这个程序，预计这个任务应该只花费几秒钟。毕竟只有一个类被改变。你在 Build 上点击或者键入 make （或者其它等价行为），接着你被惊呆了，继而被郁闷，就像你突然意识到整个世界都被重新编译和连接！当?度　氖虑榉⑸　氖焙颍　悴惶盅威　穑?

问题在于 C++ 没有做好从实现中剥离接口的工作。一个类定义不仅指定了一个类的接口而且有相当数量的实现细节。例如：

```
class Person {
public:
?Person(const std::string& name, const Date& birthday,
         const Address& addr);
?std::string name() const;
?std::string birthDate() const;
?std::string address() const;
?...

private:
    ?std::string theName;       ?// implementation detail
    ?Date theBirthDate;         ?// implementation detail
    ?Address theAddress;          // implementation detail
};
```

在这里，如果不访问 Person 的实现使用到的类，也就是 string，Date 和 Address 的定义，类 Person 就无法编译。这样的定义一般通过 #include 指令提供，所以在定义 Person 类的文件中，你很可能会找到类似这样的东西：

```
#include <string>
#include "date.h"
#include "address.h"
```

不幸的是，这样就建立了定义 Person 的文件和这些头文件之间的编译依赖关系。如果这些头文件中的一些发生了变化，或者这些头文件所依赖的文?　⑸　吮浠　　? Person 类的文件和使用了 Person 的文件一样必须重新编译，这样的层叠编译依赖关系为项目带来数不清的麻烦。

你也许想知道 C++ 为什么坚持要将一个类的实现细节放在类定义中。例如，你为什么不能这样定义 Person ，单独指定这个类的实现细节呢？

```
namespace std {
     class string;              // forward declaration (an incorrect
}                             ?// one - see below)

class Date;                   ?// forward declaration
class Address;                  // forward declaration
```

```
class Person {
public:
    ?Person(const std::string& name, const Date& birthday,
                const Address& addr);
    ?std::string name() const;
    ?std::string birthDate() const;
    ?std::string address() const;
  ?...
};
```

如果这样可行，只有在类的接口发生变化时，Person 的客户才必须重新编译。

这个主意有两个问题。第一个，string 不是一个类，它是一个 typedef (for basic_string<char>)。造成的结果就是，string 的前向声明（forward declaration）是不正确的。正确的前向声明要复杂得多，因为它包括另外的模板。然而，这还不是要紧的，因为你不应该?宰攀侄　鞅曜伎獾牟考　W魑　娲　苯邮褂檬实钡? #includeS并让它去做。标准头文件不太可能成为编译的瓶颈，特别是在你的构建环境允许你利用预编译头文件时。如果?馓霰曜纪肺募　娴某晌　桓鑫侍狻D阅残硒枰　谋溉愕慕涌谏杓疲　范馑褂玫贾虏皇芑队　? #includeS的标准库部件。

第二个（而且更重要的）难点是前向声明的每一件东西必须让编译器在编译期间知道它的对象的大小。考虑：

```
int main()
{
 int x;              ?// define an int

 Person p( params );    // define a Person
   ...
}
```

当编译器看到 x 的定义，它们知道它们必须为保存一个 int 分配足够的空间（一般是在栈上）。这没什么问题，每一个编译器都知道一个 int 有多大。当编译器看到 p 的定义，它们知道它们必须为一个 Person 分配足够的空间，但是它们怎么推测出一个 Person 对象有多大呢？它们得到这个信息的唯一方法是参考这个类的定义，但是如果一个省略了实现细节的类定义是?戏ū模　嘁肫髡趺粗　酪　峙涠啻蟮目占溉兀?

这个问题在诸如 Smalltalk 和 Java 这样的语言中就不会发生，因为，在这些语言中，当一个类被定义，编译器仅仅为一个指向一个对象的指针分?渥愎坏目占洹R簿褪撬担　　谴　砩厦娴拇　刖拖裾庑┐　胧钦庋　吹模?

```
int main()
{
?int x;               // define an int

?Person *p;           // define a pointer to a Person
?...
}
```

当然，这是合法的 C++，所以你也可以自己来玩这种"将类的实现隐藏在一个指针后面"的游戏。对 Person 做这件事的一种方法就是将它分开到两个类中，一个仅仅提供一个接口，另一个实现这个接口。如果那个实现?嗝　? PersonImpl，Person 就可以如此定义：

```
#include <string>                   ?// standard library components
                                      // shouldn't be forward-declared

#include <memory>                   ?// for tr1::shared_ptr; see below
```

```
class PersonImpl;                       ?// forward decl of Person impl. class
class Date;                             ?// forward decls of classes used in

class Address;                            // Person interface
class Person {
public:
  erson(const std::string& name, const Date& birthday,
      ?const Address& addr);
  td::string name() const;
  td::string birthDate() const;
  td::string address() const;
?..

private:                                        // ptr to implementation;
?std::tr1::shared_ptr<PersonImpl> pImpl;?// see Item 13 for info on
};                                              // std::tr1::shared_ptr
```

这样，主类（Person）除了一个指向它的实现类（PersonImpl）的指针（这里是一个 tr1::shared_ptr ——参见 Item 13）之外不包含其它数据成员。这样一个设计经常被说成是使用了 *pimpl* 惯用法（指向实现的指针 "pointer to implementation"）。在这样的类中，那个指针的名字经常是 pImpl，就像上面那个。

用这样的设计，使 Person 的客户脱离 dates，addresses 和 persons 的细节。这些类的实现可以随心所欲地改变，但 Person 的客户却不必重新编译。另外，因为他们看不到 Person 的实现细节，客户就不太可能写出以某种方式依赖那些细节的代码。这就是接口和实现的真正分离。

这个分离的关键就是用对声明的依赖替代对定义的依赖。这就是最小化编译依赖的精髓：只要能实现，就让你?耐肺募　懒⑲宰悖　绻　荒埽　鸵览灯渌　募　械纳　鳎　　皇嵌丁濉Ｆ渌　恳患　露即诱飧黾虻ヂ纳杓撇呗?产生。所以：

窗体底端

- 翻译：Effective C++, 3rd Edition, Item 31: 最小化文件之间的编译依赖（下）

（点击此处，接上篇）

- 当对象的引用和指针可以做到时就避免使用对象。
  仅需一个类型的声明，你就可以定义到这个类型的引用或指针。而定义一个类型的对象必?
  胍　燮谖飧隼嘈偷亩丁濉?
- 只要你能做到，就用对类声明的依赖替代对类定义的依赖。
  注意你声明一个使用一个类的函数时绝对不需要有这个类的定义，即使这个函数通过传值?
  绞酱　菏蚍祷卣飧隼啵?

```
class Date;                          ?// class declaration

Date today();                        ?// fine - no definition
void clearAppointments(Date d);      ?// of Date is needed
```

当然，传值通常不是一个好主意（参见 Item 20
），但是如果你发现你自己因为某种原因而使用它，依然不能为引入不必要的编译依赖辩解。

不声明 Date 就可以声明 today 和 clearAppointments
的能力可能会令你感到惊奇，但是它其实并不像看上去那么不同寻常。如果有人调用这些函数，则 Date
的定义必须在调用之前被看到。为什么费心去声明没有人调用的函数，你想知道吗？很简单。并不是?
挥腥说褰盟　牵　　遣 7.敲扛鏊硕家　褰盟　恰Ｈ绻　阌幸桓霭　　芏嗪　　鞣目猓　恳桓隹突Φ?
要调用每一个函数是不太可能的。通过将提供类定义的责任从你的声明函数的头文件转移到客户的包?
　　褰玫奈募　　憔拖　　丝突Φ运　遣 2.徽娴男枰　睦嘈偷囊览怠?

- 为声明和定义分别提供头文件。
  为了便于坚持上面的指导方针，头文件需要成对出现：一个用于声明，另一个用于定义。?
  比唬　庑┤募　匾氡 3 忠恢隆Ｈ绻　桓鏊　彭谝桓龅胤奖桓谋淞耍　　匾胸诹酱Ｘ急桓谋?
  。得出的结果是：库的客户应该总是 #include
  一个声明文件，而不是自己前向声明某些东西，而库的作者应该提供两个头文件。例如，?
  胍　　? today 和 clearAppointments 的 Date
  的客户不应该像前面展示的那样手动前向声明 Date。更合适的是，它应该 #include
  适当的用于声明的头文件：

```
#include "datefwd.h"            ?// header file declaring (but not
                                ?// defining) class Date

Date today();                     // as before
void clearAppointments(Date d);
```

仅有声明的头文件的名字 "datefwd.h" 基于来自标准 C++ 库（参见 Item 54）的头文件
<iosfwd>。<iosfwd> 包含 iostream
组件的声明，而它们相应的定义在几个不同的头文件中，包括 <sstream>，<streambuf>，
<fstream> 和 <iostream>。

<iosfwd> 在其它方面也有启发意义，而且它解释了本 Item
的建议对于模板和非模板一样有效。尽管 Item 30
解释了在很多构建环境中，模板定义的典型特征是位于头文件中，但有些环境允许模板定义在非头文?
　校　　晕　 0 逄峁┰桓毫鲇猩　鞣耐肺募　廊皇怯幸庖宓摹?<iosfwd> 就是一个这样的头文件。

C++ 还提供了 export 关键字允许将模板声明从模板定义中分离出来。不幸的是，支持 export
的编译器非常少，而与 export 打交道的实际经验就更少了。结果是，现在就说 export 在高效
C++ 编程中扮演什么角色还为时尚早。

像 Person 这样的使用 pimpl 惯用法的类经常被称为 Handle
类。为了避免你对这样的类实际上做什么事的好奇心，一种方法是将所有对他们的函数调用都转送给?
嘤Φ氖迪掷啵　　褂檬迪掷喃醋稣爨　墓ぷ鳖@　纾　饩褪橇礁? Person
的成员函数可以被如何实现的例子：

```
#include "Person.h"    ?  ?// we're implementing the Person class,
                            // so we must #include its class definition

#include "PersonImpl.h"    ?// we must also #include PersonImpl's class
                            // definition, otherwise we couldn't call
                            // its member functions; note that
                            // PersonImpl has exactly the same
                            // member functions as Person - their
                            // interfaces are identical

Person::Person(const std::string& name, const Date& birthday,
               const Address& addr)
: pImpl(new PersonImpl(name, birthday, addr))
{}

std::string Person::name() const
{
?return pImpl->name();
}
```

注意 Person 的成员函数是如何调用 PersonImpl 的成员函数的（通过使用 new ——参见 Item 16），以及 Person::name 是如何调用 PersonImpl::name 的。这很重要。使 Person 成为一个 Handle 类不需要改变 Person 要做的事情，仅仅是改变了它做事的方法。

另一个不同于 Handle 类的候选方法是使 Person 成为一个被叫做 Interface 类的特殊种类的抽象基类。这样一个类的作用是为派生类指定一个接口（参见 Item 34）。结果，它的典型特征是没有数据成员，没有构造函数，有一个虚析构函数（参见 Item 7）和一组指定接口的纯虚函数。

Interface 类类似 Java 和 .NET 中的 Interfaces，但是 C++ 并不会为 Interface 类强加那些 Java 和 .NET 为 Interfaces 强加的种种约束。例如，Java 和 .NET 都不允许 Interfaces 中有数据成员和函数实现，但是 C++ 不禁止这些事情。C++ 的巨大弹性是有用处的。就像 Item 36 解释的，在一个继承体系的所有类中非虚拟函数的实现应该相同，因此将这样的函数实现为声明它们? Ã Interface 类的一部分就是有意义的。

一个 Person 的 Interface 类可能就像这样：

```
class Person {
public:
?virtual ~Person();

?virtual std::string name() const = 0;
?virtual std::string birthDate() const = 0;
?virtual std::string address() const = 0;
?...
};
```

这个类的客户必须针对 Person 的指针或引用编程，因为实例化包含纯虚函数的类是不可能的。（然而，实例化从 Person 派生的类是可能的——参见后面。）和 Handle 类的客户一样，除非 Interface 类的接口发生变化，否则 Interface 类的客户不需要重新编译。

一个 Interface 类的客户必须有办法创建新的对象。他们一般通过调用一个为"可以真正实例化的派生类"扮演构造函数的角色的函数做到这一点的。这样的函数一般称为 factory 函数（参见 Item 13）或虚拟构造函数（*virtual constructors*）。他们返回指向动态分配的支持

Interface 类的接口的对象的指针（智能指针更合适——参见 Item 18）。这样的函数在 Interface 类内部一般声明为 static：

```
class Person {
public:
?..

 static std::tr1::shared_ptr<Person>  ?// return a tr1::shared_ptr to a
new
   create(const std::string& name,    ?// Person initialized with the
        ?const Date& birthday,         // given params; see Item 18 for
        ?const Address& addr);         // why a tr1::shared_ptr is
returned
?..
};
```

客户就像这样使用它们：

```
std::string name;
Date dateOfBirth;
Address address;
...

// create an object supporting the Person interface
std::tr1::shared_ptr<Person> pp(Person::create(name, dateOfBirth,
address));

...

std::cout << pp->name()                  // use the object via the
        ?<< " was born on "           ?// Person interface
        ?<< pp->birthDate()
        ?<< " and now lives at "
        ?<< pp->address();
...                                      // the object is automatically
                                        ?// deleted when pp goes out of
                                        ?// scope - see Item 13
```

当然，在某些地点，必须定义支持 Interface
类的接口的具体类并调用真正的构造函数。这所有的一切发生的场合，在那个文件中所包含虚拟构造?
　氖迪种　蟮牡胤健@　纾?Interface 类 Person
可以有一个提供了它继承到的虚函数的实现的具体的派生类 RealPerson：

```
class RealPerson: public Person {
public:
?RealPerson(const std::string& name, const Date& birthday,
            const Address& addr)
?: theName(name), theBirthDate(birthday), theAddress(addr)
?{}

?virtual ~RealPerson() {}

?std::string name() const;      ?// implementations of these
?std::string birthDate() const;   // functions are not shown, but
?std::string address() const;     // they are easy to imagine
```

```
private:
?std::string theName;
?Date theBirthDate;
?Address theAddress;
};
```

对这个特定的 RealPerson，写 Person::create 确实没什么价值：

```
std::tr1::shared_ptr<Person> Person::create(const std::string& name,
                                            ?const Date& birthday,
                                            ?const Address& addr)
{
?return std::tr1::shared_ptr<Person>(new RealPerson(name,
birthday,addr));
}
```

Person::create
的一个更现实的实现会创建不同派生类型的对象，依赖于诸如，其他函数的参数值，从文件或数据库?脸龅氖 苣 肪潮淞康鹊取?

RealPerson 示范了两个最通用的实现一个 Interface 类机制之一：从 Interface 类（Person）继承它的接口规格，然后实现接口中的函数。实现一个 Interface 类的第二个方法包含多继承（multiple inheritance），在 Item 40 中探讨这个话题。

Handle 类和 Interface
类从实现中分离出接口，因此减少了文件之间的编译依赖。如果你是一个喜好挖苦的人，我知道你正?谖倚『抛痔逍闯傻南拗啤?"所有这些把戏会骗走我什么呢？"
你小声嘀咕着。答案是计算机科学中非常平常的：它会消耗一些运行时的速度，再加上每个对象的一?┑钊獾哪诖妗?

在 Handle
类的情况下，成员函数必须通过实现的指针得到对象的数据。这就在每次访问中增加了一个间接层。?夷惚匾胸诖娲⑼恳桓龆韵笏 璧哪诖媪恐性黾诱庖皇迪值闹刚氲拇笮 W詈螅 庖皇迪值闹刚氡匾?被初始化（在 Handle
类的构造函数中）为指向一个动态分配的实现的对象，所以你要承受动态内存分配（以及随后的释放?┤ 逃械某杀竞驮麻? bad_alloc (out-of-memory) 异常的可能性。

对于 Interface
类，每一个函数调用都是虚拟的，所以你每调用一次函数就要支付一个间接跳转的成本（参见 Item 7）。还有，从 Interface 派生的对象必须包含一个 virtual table 指针（还是参见 Item 7）。这个指针可能增加存储一个对象所需的内存的量，依赖于这个 Interface
类是否是这个对象的虚函数的唯一来源。

最后，无论 Handle 类还是 Interface 类都不能在 inline 函数的外面大量使用。Item 30 解释了为什么函数本体一般必须在头文件中才能做到 inline，但是 Handle 类和 Interface
类一般都设计成隐藏类似函数本体这样的实现细节。

然而，因为它们所涉及到的成本而简单地放弃 Handle 类和 Interface
类会成为一个严重的错误。虚拟函数也是一样，但你还是不能放弃它们，你能吗？（如果能，你看错?榱恕#┬魑 娲 悸且砸恢指慕 姆绞绞褂谜庑┘际酢T诓 9. 讨校 褂? Handle 类和 Interface 类来最小化实现发生变化时对客户的影响。当能看出在速度和/
或大小上的不同足以证明增加类之间的耦合是值得的时候，可以用具体类取代 Handle 类和 Interface 类供产品使用。

Things to Remember

- 最小化编译依赖后面的一般想法是用对声明的依赖取代对定义的依赖。基于此想法的两个?椒ㄊ? Handle 类和 Interface 类。
- 库头文件应该以完整并且只有声明的形式存在。无论是否包含模板都适用于这一点。

Item 35: 考虑可选的 virtual functions（虚拟函数）的替代方法

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

现在你工作在一个视频游戏上，你在游戏中为角色设计了一个 hierarchy（继承体系）。你的游戏中有着变化多端的恶劣环境，角色被伤害或者其它的健康状态降低的情况并不罕见。?蛇四憔韶为峁┾桓? member function（成员函数）healthValue，它返回一个象征角色健康状况如何的整数。因为不同的角色计算健康值的方法可能不同，将 healthValue 声明为 virtual（虚拟）似乎是显而易见的设计选择：

```
class GameCharacter {
public:
?virtual int healthValue() const;       ?// return character's health rating;
?...                                      // derived classes may redefine this
};
```

healthValue 没有被声明为 pure virtual（纯虚）的事实暗示这里有一个计算健康值的缺省算法（参见 Item 34）。

这确实是一个显而易见的设计选择，而在某种意义上，这是它的缺点。因为这样的设计过于显而易见，你可能?换岫运　钠渌　裳》椒ù　枳愎坏墓刈ｉＮ　税镏　阃牙? object-oriented design（面向对象设计）的习惯性道路，我们来考虑一些处理这个问题的其它方法。

The Template Method Pattern via the Non-Virtual Interface Idiom（经由非虚拟接口惯用法实现的模板方法模式）

我们以一个主张 virtual functions（虚拟函数）应该几乎总是为 private（私有的）的有趣观点开始。这一观点的拥护者提出：一个较好的设计应该保留作为 public member function（公有成员函数）的 healthValue，但应将它改为 non-virtual（非虚拟的）并让它调用一个 private virtual function（私有虚拟函数）来做真正的工作，也就是说，doHealthValue：

```
class GameCharacter {
public:
?int healthValue() const                // derived classes do not redefine
?{                                      // this - see Item 36

  ?...                                  // do "before" stuff - see below

  ?int retVal = doHealthValue();        // do the real work

  ?...                                  // do "after" stuff - see below

  ?return retVal;
?}
?...

private:
?virtual int doHealthValue() const      // derived classes may redefine this
?{
  ?...                                  // default algorithm for calculating
```

```
?}                                          // character's health
};
```

在这个代码（以及本 Item 的其它代码）中，我在类定义中展示 member functions（成员函数）的本体。就像 Item 30 中所解释的，这会将它们隐式声明为 `inline`（内联）。我用这种方法展示代码仅仅是这样更易于看到它在做些什么。我所描述的设计与是否 inline 化无关，所以不必深究 member functions（成员函数）定义在类的内部有什么意味深长的含义。根本没有。

这个基本的设计——让客户通过 public non-virtual member functions（公有非虚拟成员函数）调用 private virtual functions（私有虚拟函数）——被称为 *non-virtual interface (NVI) idiom*（非虚拟接口惯用法）。这是一个更通用的被称为 Template Method（一个模式，很不幸，与 C++ templates（模板）无关）的 design pattern（设计模式）的特殊形式。我将那个 non-virtual function（非虚拟函数）（例如，`healthValue`）称为 virtual function's *wrapper*（虚拟函数的外壳）。

NVI idiom（惯用法）的一个优势通过 "do 'before' stuff" 和 "do 'after' stuff" 两个注释在代码中标示出来。这些注释标出的代码片断在做真正的工作的 virtual function（虚拟函数）之前或之后调用。这就意味着那个 wrapper（外壳）可以确保在 virtual function（虚拟函数）被调用前，特定的背景环境被设置，而在调用结束之后，这些背景环境被清理。例如，"before" stuff 可以包括锁闭一个 mutex（互斥体），生成一条日志条目，校验类变量和函数的 preconditions（前提条件）是否被满足，等等。"after" stuff 可以包括解锁一个 mutex（互斥体），校验函数的 postconditions（结束条件），类不变量的恢复，等等。如果你让客户直接调用 virtual functions（虚拟函数），确实没有好的方法能够做到这些。

涉及 derived classes（派生类）重定义 private virtual functions（私有虚拟函数）（这些重定义函数它们不能调用！）的 NVI idiom 可能会搅乱你的头脑。这里没有设计上的矛盾。重定义一个 virtual function（虚拟函数）指定如何做某些事。调用一个 virtual function（虚拟函数）指定什么时候去做。互相之间没有关系。NVI idiom 允许 derived classes（派生类）重定义一个 virtual function（虚拟函数），这样就给了它们控制功能如何实现的能力，但是 base class（基类）保留了决定函数何时被调用的权利。乍一看很奇怪，但是 C++ 规定 derived classes（派生类）可以重定义 private inherited virtual functions（私有继承来的虚拟函数）是非常明智的。

在 NVI idiom 之下，virtual functions（虚拟函数）成为 private（私有的）并不是绝对必需的。在一些 class hierarchies（类继承体系）中，一个 virtual function（虚拟函数）的 derived class（派生类）实现被期望调用其 base class（基类）的对应物（例如，第 120 页的例子），而为了这样的调用能够合法，虚拟必须成为 protected（保护的），而非 private（私有的）。有时一个 virtual function（虚拟函数）甚至必须是 public（公有的）（例如，polymorphic base classes（多态基类）中的 destructors（析构函数）——参见 Item 7），但这样一来 NVI idiom 就不能被真正应用。

The Strategy Pattern via Function Pointers（经由函数指针实现的策略模式）

NVI idiom 是 public virtual functions（公有虚拟函数）的有趣的可选替代物，但从设计的观点来看，它比装点门也多不了多少东西。毕竟，我们还? 窃谟? virtual functions（虚拟函数）来计算每一个角色的健康值。一个更引人注目的设计主张认为计算一个角色的健康值不依赖于角? 睦嘈?——这样的计算根本不需要成为角色的一部分。例如，我们可能需要为每一个角色的 constructor（构造函数）传递一个指向健康值计算函数的指针，而我们可以调用这个函数进行实际的计算：

```
class GameCharacter;                                // forward declaration

// function for the default health calculation algorithm
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;
};
```

这个方法是另一个通用 design pattern（设计模式）—— Strategy 的简单应用，相对于基于 GameCharacter hierarchy（继承体系）中的 virtual functions （虚拟函数）的方法，它提供了某些更引人注目的机动性：

- 相同角色类型的不同实例可以有不同的健康值计算函数。例如：

```
class EvilBadGuy: public GameCharacter {
public:
?explicit EvilBadGuy(HealthCalcFunc hcf = defaultHealthCalc)
?: GameCharacter(hcf)
?{ ... }

?...

};
int loseHealthQuickly(const GameCharacter&);  ?// health calculation
int loseHealthSlowly(const GameCharacter&);     // funcs with different
                                              ?// behavior

EvilBadGuy ebg1(loseHealthQuickly);             // same-type charac-
EvilBadGuy ebg2(loseHealthSlowly);            ?// ters with different
                                              ?// health-related
                                              ?// behavior
```

- 对于一个指定的角色健康值的计算函数可以在运行时改变。例如，GameCharacter 可以提供一个 member function（成员函数）setHealthCalculator ，它被允许代替当前的健康值计算函数。

在另一方面，健康值计算函数不再是 GameCharacter hierarchy（继承体系）的一个 member function （成员函数）的事实，意味着它不再拥有访问它所计算的那个对象内部构件的特权。例如，defaultHealthCalc 不能访问 EvilBadGuy 的 non-public （非公有）构件。如果一个角色的健康值计算能够完全基于通过角色的 public interface （公有接口）可以得到的信息，这就没什么问题，但是，如果准确的健康值计算需要 non-public （非公有）信息，就会有问题。实际上，在任何一个你要用 class（类）外部的等价机能（例如，经由一个 non-member non-friend function（非成员非友元函数）或经由另一个 class（类）的 non-friend member function（非友元成员函数））代替 class（类）内部的机能（例如，经由一个 member function （成员函数））的时候，它都是一个潜在的问题。这个问题将持续影响本 Item 的剩余部分，因为所有我们要考虑的其它设计选择都包括 GameCharacter hierarchy （继承体系）的外部函数的使用。

作为一个通用规则，解决对 on-member functions（非成员函数）对类的 non-public （非公有）构件的访问的需要”的唯一方法就是削弱类的 encapsulation（封装性）。例如，class （类）可以将 non-member functions（非成员函数）声明为 friends（友元），或者，它可以提供对“ 在其它情况下它更希望保持隐藏的本身的实现部分”的 public accessor functions （公有访问者函数）。使用一个 function pointer（函数指针）代替一个 virtual function （虚拟函数）的优势（例如，具有逐对象健康值计算函数的能力和在运行时改变这样的函数的能力）是否能抵?

赡芳慕档? GameCharacter 的 encapsulation
（封装性）的需要是你必须在设计时就做出决定的重要部分。

窗体底端

翻译：Effective C++, 3rd Edition, Item 35: 考虑可选的 virtual functions（虚拟函数）的替代方法（下）

（点击此处，接上篇）

The Strategy Pattern via `tr1::function`（经由 `tr1::function` 实现的策略模式）

一旦你习惯了 templates（模板）和 implicit interfaces（隐式接口）（参见 Item 41）的应用，function-pointer-based（基于函数指针）的方法看上去就有些死板了。健康值的计算为什么必须是一个 function（函数），而不能是某种简单的行为类似 function（函数）的东西（例如，一个 function object（函数对象））？如果它必须是一个 function（函数），为什么不能是一个 member function（成员函数）？为什么它必须返回一个 int，而不是某种能够转型为 int 的类型？

如果我们用一个 tr1::function 类型的对象代替一个 function pointer（函数指针）（诸如 healthFunc），这些约束就会消失。就像 Item 54 中的解释，这样的对象可以持有 *any callable entity*（任何可调用实体）（例如，function pointer（函数指针），function object（函数对象），或 member function pointer（成员函数指针）），这些实体的标志性特征就是兼容于它所期待的东西。我们马上就会看到这样的设计，这?问褂昧? tr1::function：

```
class GameCharacter;                                  // as before
int defaultHealthCalc(const GameCharacter& gc);     ?// as before

class GameCharacter {
public:
   // HealthCalcFunc is any callable entity that can be called with
   // anything compatible with a GameCharacter and that returns anything
   // compatible with an int; see below for details
   typedef std::tr1::function<int (const GameCharacter&)> HealthCalcFunc;
   explicit GameCharacter(HealthCalcFunc hcf = defaultHealthCalc)
   : healthFunc(hcf)
   {}

   int healthValue() const
   { return healthFunc(*this);   }

   ...

private:
?HealthCalcFunc healthFunc;
};
```

就像你看到的，HealthCalcFunc 是一个 tr1::function instantiation（实例化）的 typedef。这意味着它的行为类似一个普通的 function pointer（函数指针）类型。我们近距离看看 HealthCalcFunc 究竟是一个什么东西的 typedef：

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

这里我突出了这个 tr1::function instantiation（实例化）的 arget signature（目标识别特征）”。这个 target signature（目标识别特征）是“取得一个引向 const GameCharacter 的 reference（引用），并返回一个 int 的函数”。这个 tr1::function 类型的（例如，HealthCalcFunc 类型的）对象可以持有兼容于这个 target signature（目标识别特征）的 any callable entity（任何可调用实体）。兼容意味着这个实体的参数能够隐式地转型为一个 const GameCharacter&，而它的返回类型能够隐式地转型为一个 int。

与我们看到的最近一个设计（在那里 GameCharacter
持有一个指向一个函数的指针）相比，这个设计几乎相同。仅有的区别是目前的 GameCharacter 持有一个 tr1::function 对象——指向一个函数的 *generalized*（泛型化）指针。除了达到 lients（客户）在指定健康值计算函数时有更大的灵活性”
的效果之外，这个变化是如此之小，以至于我宁愿对它视而不见：

```
short calcHealth(const GameCharacter&);         ?// health calculation
                                                  // function; note
                                                  // non-int return type

struct HealthCalculator {                        ?// class for health
?int operator()(const GameCharacter&) const       // calculation function
?{ ... }                                        ?// objects
};

class GameLevel {
public:
?float health(const GameCharacter&) const;     ?// health calculation
?...                                           ?// mem function; note
};                                                // non-int return type


class EvilBadGuy: public GameCharacter {          // as before
?...
};
class EyeCandyCharacter:   public GameCharacter {?// another character
?...                                             ?// type; assume same
};                                                  // constructor as
                                                    // EvilBadGuy


EvilBadGuy ebg1(calcHealth);                        // character using a
                                                    // health calculation
                                                    // function


EyeCandyCharacter ecc1(HealthCalculator());       ?// character using a
                                                    // health calculation
                                                    // function object


GameLevel currentLevel;
...
EvilBadGuy ebg2(                                    // character using a
?std::tr1::bind(&GameLevel::health,                // health calculation
        ?currentLevel,                            ?// member function;
        ?_1)                                      ?// see below for details
);
```

就个人感觉而言：我发现 tr1::function
能让你做的事情是如此让人惊喜，它令我浑身兴奋异常。如果你没有感到兴奋，那可能是因为你正目不转睛地?⑲? ebg2 的定义并对 tr1::bind 的调用会发生什么迷惑不解。请耐心地听我解释。

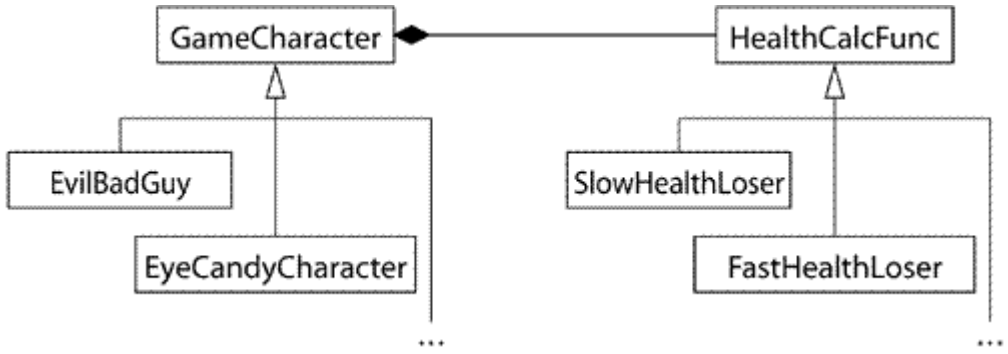
比方说我们要计算 ebg2 的健康等级，应该使用 GameLevel class（类）中的 health member function（成员函数）。现在，GameLevel::health 是一个被声明为取得一个参数（一个引向 GameCharacter 的引用）的函数，但是它实际上取得了两个参数，因为它同时得到一个隐式的 GameLevel 参数——指向 this。然而，GameCharacters 的健康值计算函数只取得单一的参数：将被计算健康值的

GameCharacter。如果我们要使用 GameLevel::health 计算 ebg2 的健康值，我们必须以某种方式"改造"它，以使它适应只取得唯一的参数（一个 GameCharacter），而不是两个（一个 GameCharacter 和一个 GameLevel）。在本例中，我们总是要使用 currentLevel 作为 GameLevel 对象来计算 ebg2 的健康值，所以每次调用 GameLevel::health 计算 ebg2 的健康值时，我们就要 "bind"（凝固）currentLevel 来作为 GameLevel 的对象来使用。这就是 tr1::bind 的调用所做的事情：它指定 ebg2 的健康值计算函数应该总是使用 currentLevel 作为 GameLevel 对象。

我们跳过一大堆的细节，诸如为什么 "_1" 意味着"当为了 ebg2 调用 GameLevel::health 时使用 currentLevel 作为 GameLevel 对象"
。这样的细节并没有什么启发性，而且它们将转移我所关注的基本点：在计算一个角色的健康值时，通过使用 tr1::function 代替一个 function pointer（函数指针），我们将允许客户使用 *any compatible callable entity*（任何兼容的可调用实体）。很酷是不是？

The "Classic" Strategy Pattern（"经典的"策略模式）

如果你比 C++ 更加深入地进入 design patterns（设计模式），一个 Strategy 的更加习以为常的做法是将 health-calculation function（健康值计算函数）做成一个独立的 health-calculation hierarchy（健康值计算继承体系）的 virtual member function（虚拟成员函数）。做成的 hierarchy（继承体系）设计看起来就像这样：



如果你不熟悉 UML 记法，这不过是在表示当把 EvilBadGuy 和 EyeCandyCharacter 作为 derived classes（派生类）时，GameCharacter 是这个 inheritance hierarchy（继承体系）的根；HealthCalcFunc 是另一个带有 derived classes（派生类）SlowHealthLoser 和 FastHealthLoser 的 inheritance hierarchy（继承体系）的根；而每一个 GameCharacter 类型的对象包含一个指向"从 HealthCalcFunc 派生的对象"的指针。

这就是相应的框架代码：

```
class GameCharacter;                            ?// 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;
};
```

这个方法的吸引力在于对于熟悉"标准的 trategy pattern（策略模式）实现的人可以很快地识别出来，再加上它提供了通过在 HealthCalcFunc hierarchy（继承体系）中增加一个 derived class（派生类）而微调已存在的健康值计算算法的可能性。

Summary（概要）

这个 Item 的基本建议是当你为尝试解决的问题寻求一个设计时，你应该考虑可选的 virtual functions（虚拟函数）的替代方法。以下是对我们考察过的可选方法的一个简略的回顾：

- 使用 **non-virtual interface idiom** (NVI idiom)（非虚拟接口惯用法），这是用 public non-virtual member functions（公有非虚拟成员函数）包装可访问权限较小的 virtual functions（虚拟函数）的 Template Method design pattern（模板方法模式）的一种形式。
- 用 **function pointer data members**（函数指针数据成员）代替 virtual functions（虚拟函数），一种 Strategy design pattern（策略模式）的显而易见的形式。
- 用 **`tr1::function` data members**（数据成员）代替 virtual functions（虚拟函数），这样就允许使用兼容于你所需要的东西的 any callable entity（任何可调用实体）。这也是 Strategy design pattern（策略模式）的一种形式。
- 用 **virtual functions in another hierarchy**（另外一个继承体系中的虚拟函数）代替 virtual functions in one hierarchy（单独一个继承体系中的虚拟函数）。这是 Strategy design pattern（策略模式）的习以为常的实现。

这不是一个可选的 virtual functions（虚拟函数）的替代设计的详尽无遗的列表，但是它足以使你确信这些是可选的方法。此外，它们之间互为比?系挠帕佑Ω檬鼓憧悸撬 鞘备 魅贰?

为了避免陷入 object-oriented design（面向对象设计）的习惯性道路，时不时地给车轮一些有益的颠簸。有很多其它的道路。值得花一些时间去考?撬 恰?

Things to Remember

- 可选的 virtual functions（虚拟函数）的替代方法包括 NVI 惯用法和 Strategy design pattern（策略模式）的各种变化形式。NVI 惯用法本身是 Template Method design pattern（模板方法模式）的一个实例。
- 将一个机能从一个 member function（成员函数）中移到 class（类）之外的某个函数中的一个危害是 non-member function（非成员函数）没有访问类的 non-public members（非公有成员）的途径。
- `tr1::function` 对象的行为类似 generalized function pointers（泛型化的函数指针）。这样的对象支持所有兼容于一个给定的目标特征的 callable entities（可调用实体）。

窗体底端

由于程序的原因，本文件未被完整保存。

Item 36: 绝不要重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

假设我告诉你 class（类）D 从 class（类）B publicly derived（公有继承），而且在 class（类）B 中定义了一个 public member function（公有成员函数）mf。mf
的参数和返回值类型是无关紧要的，所以我们就假设它们都是 void。换句话说，我的意思是：

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

甚至不必知道关于 B，D，或 mf 的任何事情，给定一个类型为 D 的 object（对象）x，

```
D x;                                ?// x is an object of type D
```

对此你或许非常吃惊，

```
B *pB = &x;                          // get pointer to x
pB->mf();                            // call mf through pointer
```

的行为不同于以下代码：

```
D *pD = &x;                          // get pointer to x
pD->mf();                            // call mf through pointer
```

因为在两种情况中，你都调用了 object（对象）x 中的 member function（成员函数）mf
。因为两种情况中都是同样的 function（函数）和同样的
object（对象），它们的行为应该有相同的方式，对吗？

是的，应该。但是也可能不，特别地，如果 mf 是 non-virtual（非虚拟）而 D 定义了它自己的版本的 mf
：

```
class D: public B {
public:
?void mf();                        ?// hides B::mf; see Item33

?...

};

pB->mf();                            // calls B::mf

pD->mf();                            // calls D::mf
```

这种行为两面性的原因是像 B::mf 和 D::mf 这样的 non-virtual functions（非虚拟函数）是
statically bound（静态绑定）的（参见 Item 37）。这就意味着因为 pB 被声明为 pointer-to-B
类型，所以，即使就像本例中的做法，让 pB 指向一个从 B 继承的类的对象，通过 pB 调用的
non-virtual functions（非虚拟函数）也总是定义在 class B 中的那一个。

在另一方面，virtual functions（虚拟函数）是 dynamically bound（动态绑定）的（再次参见 Item 37
），所以它们不会发生这个问题。如果 mf 是一个 virtual function（虚拟函数），无论通过 pB 还是 pD
调用 mf 都将导致 D::mf 的调用，因为 pB 和 pD 都实际地指向一个 type（类型）D 的
object（对象）。

如果你在编写 class D 而且你重定义了一个你从 class B 继承到的 non-virtual function（非虚拟函数）mf，D 的 objects（对象）将很可能表现出不协调的行为。特别是，当 mf 被调用时，任何给定的 D object（对象）的行为既可能像 B 也可能像 D，而且决定因素与 object（对象）本身无关，但是和指向它的 pointer（指针）的声明类型有关。references（引用）也会像 pointers（指针）一样表现出莫名其妙的行为。

但这仅仅是一个从实用出发的论据。我知道，你真正需要的是不能重定义 inherited non-virtual functions（通过继承得到的非虚拟函数）的理论上的理由。我很愿意效劳。

Item 32 解释了 public inheritance（公有继承）意味着 is-a，Item 34 记述了为什么在一个 class（类）中声明一个 non-virtual function（非虚拟函数）是为这个 class（类）设定一个 invariant over specialization（超越特殊化的不变量），如果你将这些经验应用于 classes（类）B 和 D 以及 non-virtual member function（非虚拟函数）B::mf，那么：

每一件适用于 B objects（对象）的事情也适用于 D objects（对象），因为每一个 D objects 都 is-a（是一个）D objects（对象）；

从 B 继承的 classes（类）必须同时继承 mf 的 interface（接口）和 implementation（实现），因为 mf 在 B 中是 non-virtual（非虚拟）的。

现在，如果 D 重定义 mf，你的设计中就有了一处矛盾。如果 D 真的需要实现不同于 B 的 mf，而且如果每一个 B objects（对象）——无论如何特殊——都必须使用 B 对 mf 的实现，那么每一个 D 都 is-a（是一个）B 就完全不成立。在那种情况下，D 就不应该从 B publicly inherit（公有继承）。另一方面，如果 D 真的必须从 B publicly inherit（公有继承），而且如果 D 真的需要实现不同于 B 的 mf，那么 mf 反映了一个 B 的 invariant over specialization（超越特殊化的不变量）就不会成立。在那种情况下，mf 应该是 virtual（虚拟）的。最后，如果每一个 D 真的都 is-a（是一个）B，而且如果 mf 真的相当于一个 B 的 invariant over specialization（超越特殊化的不变量），那么 D 就不会真的需要重定义 mf，而且想都不能想。

不管使用那一条规则，必须做出某些让步，而且无条件地禁止重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）。

如果阅读这个 Item 给你 dé à?vu（似曾相识）的感觉，那可能是因为你已经阅读了 Item 7，那个 Item 解释了为什么 polymorphic base classes（多态基类）中的 destructors（析构函数）应该是 virtual（虚拟）的。如果你违反了那个 guideline（指导方针）（例如，如果你在一个 polymorphic base class（多态基类）中声明一个 non-virtual destructor（非虚拟析构函数）），你也同时违反了这里这个 guideline（指导方针），因为 derived classes（派生类）总是要重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）：base class（基类）的 destructor（析构函数）。甚至对于没有声明 destructor（析构函数）的 derived classes（派生类）这也是成立的，因为，就像 Item 5 的解释，destructor（析构函数）是一个“° 如果你没有定义你自己的，编译器就会为你生成一个”±的 member functions（成员函数）。其实，Item 7 只相当于本 Item 的一个特殊情况，尽管它重要到足以把它提出来独立成篇。

Things to Remember

- 绝不要重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）。

窗体底端

- # Item 37: 绝不要重定义一个函数的 inherited default parameter value（通过继承得到的缺省参数值）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

我们直接着手简化这个话题。只有两种函数能被你 inherit（继承）：virtual（虚拟的）和 non-virtual（非虚拟的）。然而，重定义一个 inherited non-virtual function（通过继承得到的非虚拟函数）永远都是一个错误（参见 Item 36），所以我们可以安全地将我们的讨论限制在你继承了一个 *virtual* function with a default parameter value（带有一个缺省参数值的虚拟函数）的情形。

在这种情况下，本 Item 的理由就变得非常地直截了当：virtual functions（虚拟函数）是 dynamically bound（动态绑定），而 default parameter values（缺省参数值）是 statically bound（静态绑定）。

那又怎样呢？你说 static（静态）和 dynamic binding（动态绑定）之间的区别早已塞入你负担过重的头脑？（不要忘了，static binding（静态绑定）也以 *early binding*（前期绑定）闻名，而 dynamic binding（动态绑定）也以 *late binding*（后期绑定）闻名。）那么，我们就再来回顾一下。

一个 object（对象）的 *static type*（静态类型）就是你在程序文本中声明给它的 type（类型）。考虑这个 class hierarchy（类继承体系）：

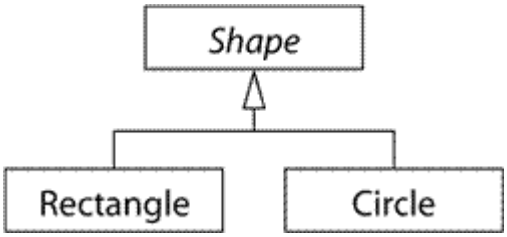
```
// a class for geometric shapes
class Shape {
public:
?enum ShapeColor { Red, Green, Blue };

?// all shapes must offer a function to draw themselves
?virtual void draw(ShapeColor color = Red) const = 0;
?...
};

class Rectangle: public Shape {
public:
?// notice the different default parameter value — bad!
?virtual void draw(ShapeColor color = Green) const;
?...
};

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

直观地看，它看起来就像这个样子：



现在考虑这些 pointers（指针）：

```
Shape *ps;                          // static type = Shape*
Shape *pc = new Circle;           ?// static type = Shape*
Shape *pr = new Rectangle;          // static type = Shape*
```

在本例中，ps，pc 和 pr 全被声明为 pointer-to-Shape 类型，所以它们全都以此作为它们的 static type（静态类型）。注意这就使得它们真正指向的东西完全没有区别——无论如何，它们的 static type（静态类型）都是 Shape*。

一个 object（对象）的 *dynamic type*（动态类型）取决于它当前引用的 object（对象）的 type（类型）。也就是说，它的 dynamic type（动态类型）表明它有怎样的行为。在上面的例子中，pc 的 dynamic type（动态类型）是 Circle*，而 pr 的 dynamic type（动态类型）是 Rectangle*。至于 ps，它没有一个实际的 dynamic type（动态类型），因为它（还）不能引用任何 object（对象）。

dynamic types（动态类型），就像它的名字所暗示的，能在程序运行中变化，特别是通过 assignments（赋值）：

```
ps = pc;                            // ps's dynamic type is
                                    // now Circle*

ps = pr;                            // ps's dynamic type is
                                    // now Rectangle*
```

virtual functions（虚拟函数）是 *dynamically bound*（动态绑定），意味着被调用的特定函数取决于被用来调用它的那个 object（对象）的 dynamic type（动态类型）：

```
pc->draw(Shape::Red);               // calls Circle::draw(Shape::Red)

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

我知道，这全是老生常谈；你的确已经理解了 virtual functions（虚拟函数）。但是，当你考虑 virtual functions with default parameter values（带有缺省参数值的虚拟函数）时，就全乱了套，因为，如上所述，virtual functions（虚拟函数）是 dynamically bound（动态绑定），但 default parameters（缺省参数）是 statically bound（静态绑定）。这就意味着你最终调用了一个定义在 *derived class*（派生类）中的 virtual function（虚拟函数）却使用了一个来自 *base class*（基类）的 default parameter value（缺省参数值）。

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

在此情况下，pr 的 dynamic type（动态类型）是 Rectangle*，所以正像你所希望的，Rectangle 的 virtual function（虚拟函数）被调用。在 Rectangle::draw 中，default parameter value（缺省参数值）是 Green。然而，因为 pr 的 static type（静态类型）是 Shape*，这个函数调用的 default parameter value（缺省参数值）是从 Shape class

中取得的，而不是 Rectangle class！导致的结果就是一个调用由"奇怪的和几乎完全出乎意料的 Shape 和 Rectangle 两个 classes（类）中的 draw 声明的混合物"所组成。

ps，pc，和 pr 是 pointers（指针）的事实与这个问题并无因果关系，如果它们是 references（引用），问题依然会存在。唯一重要的事情是 draw 是一个 virtual function（虚拟函数），而它的一个 default parameter values（缺省参数值）在一个 derived class（派生类）中被重定义。

为什么 C++ 要坚持按照这种不正常的方式动作？答案是为了运行时效率。如果 default parameter values（缺省参数值）是 dynamically bound（动态绑定），compilers（编译器）就必须提供一种方法在运行时确定 virtual functions（虚拟函数）的 parameters（参数）的 default value(s)（缺省值），这比目前在编译期确定它们的机制更慢而且更复杂。最终的决定偏向了速度和实现的简? フ庖槐撸　　斐傻慕峁　褪悄阆衷诓梢韵硎茂呸ｒ诵械睦秩い　　牵　绻　阃　橇粜谋? Item 的建议，就会陷入困惑。

这样就很彻底而且完美了，但是看看如果你试图遵循本规则为 base（基类）和 derived classes（派生类）的用户提供同样的 default parameter values（缺省参数值）时会发生什么：

```
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;
?...
};
```

噢，code duplication（代码重复）。code duplication（代码重复）带来 dependencies（依赖关系）：如果 Shape 中的 default parameter values（缺省参数值）发生变化，所有重复了它的 derived classes（派生类）必须同时变化。否则它们就陷入重定义一个 inherited default parameter value（通过继承得到的缺省参数值）。怎么办呢？

当你要一个 virtual function（虚拟函数）按照你希望的方式运行有困难的时候，考虑可选的替代设计是很明智的，而且 Item 35 给出了多个 virtual function（虚拟函数）的替代方法。替代方法之一是 *non-virtual interface idiom* (NVI idiom)（非虚拟接口惯用法）：用 base class（基类）中的 public non-virtual function（公有非虚拟函数）调用 derived classes（派生类）可能重定义的 private virtual function（私有虚拟函数）。这里，我们用 non-virtual function（非虚拟函数）指定 default parameter（缺省参数），同时使用 virtual function（虚拟函数）做实际的工作：

```
class Shape {
public:
?enum ShapeColor { Red, Green, Blue };

?void draw(ShapeColor color = Red) const           // now non-virtual
?{
  ?doDraw(color);                                 ?// calls a virtual
?}

?...
```

```
private:
?virtual void doDraw(ShapeColor color) const = 0;?// the actual work is
};                                                  ?// done in this func

class Rectangle: public Shape {
public:

?...

private:
?virtual void doDraw(ShapeColor color) const;       // note lack of a
?...                                               ?// default param val.
};
```

因为 non-virtual functions（非虚拟函数）绝不应该被 derived classes（派生类） overridden（覆盖）（参见 Item 36），这个设计使得 `draw` 的 `color` parameter（参数）的 default value（缺省值）应该永远是 `Red` 变得明确。

Things to Remember

- 绝不要重定义一个 inherited default parameter value（通过继承得到的缺省参数值），因为 default parameter value（缺省参数值）是 statically bound（静态绑定），而 virtual functions ——应该是你可以 overriding（覆盖）的仅有的函数——是 dynamically bound（动态绑定）。

窗体底端

Item 38: 通过 composition（复合）模拟 "has-a"（有一个）或 "is-implemented-in-terms-of"（是根据...-...-实现的）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

composition（复合）是在 objects of one type（一个类型的对象）包含 objects of another type（另一个类型的对象）时，types（类型）之间的关系。例如：

```
class Address { ... };              // where someone lives

class PhoneNumber { ... };

class Person {
public:
?...

private:
?std::string name;                    // composed object
?Address address;                   ?// ditto
?PhoneNumber voiceNumber;           ?// ditto
?PhoneNumber faxNumber;             ?// ditto
};
```

此例之中，Person objects（对象）由 string，Address，和 PhoneNumber objects（对象）组成。在程序员中，术语 composition（复合）有很多同义词。它也可以称为 layering，containment，aggregation，和 embedding。

Item 32 解释了 public inheritance（公有继承）意味着 "is-a"（是一个）。composition（复合）也有一个含意。实际上，他有两个含意。composition（复合）既意味着 "has-a"（有一个），又意味着 "is-implemented-in-terms-of"（是根据...-...-实现的）。这是因为你要在你的软件中处理两个不同的 domains（领域）。你程序中的一些 objects（对象）对应你所模拟的世界里的东西，例如，people（人），vehicles（交通工具），video frames（视频画面）等等。这样的 objects（对象）是 application domain（应用领域）的部分。另外的 objects（对象）纯粹是 implementation artifacts（实现的产物），例如，buffers（缓冲区），mutexes（互斥体），search trees（搜索树）等等。这些各类 objects（对象）定义应你的软件的 implementation domain（实现领域）。当 composition（复合）发生在 application domain（应用领域）的 objects（对象）之间，它表达一个 has-a（有一个）的关系，当它发生在 implementation domain（实现领域），它表达一个 is-implemented-in-terms-of（是根据...-...-实现的）的关系

上面的 Person class（类）示范了 has-a（有一个）的关系。一个 Person object（对象）has a（有一个）名字，一个地址，以及语音和传真电话号码。你不能说一个人 is a（是一个）名字或一个人 is an（是一个）地址。你可以说一个人 has a（有一个）名字和 has an（有一个）地址。大多数人对此区别不难理解，所以混淆 is-a（是一个）和 has-a（有一个）之间的角色的情况非常少见。

is-a（是一个）和 is-implemented-in-terms-of（是根据...-...-实现的）之间的区别稍微有些棘手。例如，假设你需要一个类的模板来表现相当小的 objects（对象）的 sets，也就是说，排除重复的集合。因为 reuse（复用）是一件受人欢迎的事情，你的第一个直觉就是使用标准库中的 set template（模板）。当你能使用已经被写好的东西时，为什么还要写一个新的 template（模板）呢?

不幸的是，set 的典型实现导致每个元素三个指针的开销。这是因为 sets 通常被作为 balanced search trees（平衡搜索树）来实现，这允许它们保证 logarithmic-time（对数时间）的 lookups（查找），insertions（插入）和 erasures（删除）。当速度比空间更重要的时候，这是一个合理的设计，但是当空间比速度更重要时，对你的程序来说就有问题了。因而，对你来说，标准库的 set 为你提供了不合理的交易。看起来你终究还是要写你自己的 template（模板）。

reuse（复用）依然是一件受人欢迎的事情。作为 data structure（数据结构）的专家，你知道实现 sets 的诸多选择，其中一种是使用 linked lists（线性链表）。你也知道标准的 C++ 库中有一个 list template（模板），所以你决定（复）用它。

具体地说，你决定让你的新的 Set template（模板）从 list 继承。也就是说，Set<T> 将从 list<T> 继承。毕竟，在你的实现中，一个 Set object（对象）实际上就是一个 list object（对象）。于是，你就像这样声明你的 Set template（模板）：

```
template<typename T>                          // the wrong way to use list for
Set
class Set: public std::list<T> { ... };
```

在这里，看起来每件事情都很好。但实际上有一个很大的错误。就像 Item 32 中的解释，如果 D is-a（是一个）B，对于 B 成立的每一件事情对 D 也成立。然而，一个 list object（对象）可以包含重复，所以如果值 3051 被插入一个 list<int> 两次，那个 list 将包含 3051 的两个拷贝。与此对照，一个 Set 不可以包含重复，所以如果值 3051 被插入一个 Set<int> 两次，那个 set 只包含该值的一个拷贝。因此一个 Set is-a（是一个）list 是不正确的，因为对 list objects（对象）成立的某些事情对 Set objects（对象）不成立。

因为这两个 classes（类）之间的关系不是 is-a（是一个），public inheritance（公有继承）不是模拟这个关系的正确方法。正确的方法是认识到一个 Set object（对象）可以 be?implemented in terms of a list object（是根据一个 list 对象实现的）：

```
template<class T>                     // the right way to use list for Set
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;                    // representation for Set data
};
```

Set 的 member functions（成员函数）可以极大程度地依赖 list 和标准库的其它部分已经提供的机能，所以只要你熟悉了用 STL 编程的基本方法，实现就非常简单了：

```
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 =                 // see Item 42 for info on
  ?std::find(rep.begin(), rep.end(), item);           // "typename" here
?if (it != rep.end()) rep.erase(it);
}

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

这些函数足够简单，使它们成为 inlining（内联化）的合理候选者，可是我知道在坚定 inlining（内联化）的决心之前，你可能需要回顾一下 Item 30 中的讨论。

一个有说服力的观点是，根据 Item 18 的关于将 interfaces（接口）设计得易于正确使用，而难以错误使用的论述，如果要遵循 STL container（容器）的惯例，Set 的 interface（接口）应该更多，但是在这里遵循那些惯例就需要在 Set 中填充大量 stuff（材料），这将使得它和 list 之间的关系变得暧昧不清。因为这个关系是本 Item 的重点，我们用教学的清晰性替换了 STL 的兼容性。除此之外，Set 的 interface（接口）的幼稚不应该遮掩关于 Set 的无可争辩的正确：它和 list 之间的关系。这个关系不是 is-a（是一个）（虽然最初看上去可能很像），而是 is-implemented-in-terms-of（是根据...-...-实现的）。

Things to Remember

- composition（复合）与 public inheritance（公有继承）的意义完全不同。
- 在 application domain（应用领域）中，composition（复合）意味着 has-a（有一个）。在 implementation domain（实现领域）中意味着 is-implemented-in-terms-of（是根据...-...-实现的）。

窗体底端

Item 39: 谨慎使用 private inheritance（私有继承）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

Item 32 论述了 C++ 将 public inheritance（公有继承）视为一个 is-a 关系。当给定一个 hierarchy（继承体系），其中有一个 class Student 从一个 class Person 公有继承，当为了成功调用一个函数而必需时，就要将 Students 隐式转型为 Persons，它通过向编译器展示来做到这一点。用 private inheritance（私有继承）代替 public inheritance（公有继承）把这个例子的一部分重做一下是值得的：

```
class Person { ... };
class Student: private Person { ... };      // inheritance is now private

void eat(const Person& p);                  // anyone can eat

void study(const Student& s);             ?// only students study

Person p;                                 ?// p is a Person
Student s;                                  // s is a Student

eat(p);                                   ?// fine, p is a Person

eat(s);                                   ?// error! a Student isn't a Person
```

很明显，private inheritance（私有继承）不意味着 is-a。那么它意味着什么呢？

“喂！”你说：“在我们得到它的含义之前，我们先看看它的行为。private inheritance（私有继承）有怎样的行为呢？”好吧，支配 private inheritance（私有继承）的第一个规则你只能从动作中看到：与 public inheritance（公有继承）对照，如果 classes（类）之间的 inheritance relationship（继承关系）是 private（私有）的，编译器通常不会将一个 derived class object（派生类对象）（诸如 Student）转型为一个 base class object（基类对象）（诸如 Person）。这就是为什么为 object（对象）s 调用 eat 会失败。第二个规则是从一个 private base class（私有基类）继承的 members（成员）会成为 derived class（派生类）的 private members（私有成员），即使它们在 base class（基类）中是 protected（保护）的或 public（公有）的。

行为不过如此。这就给我们带来了含义。private inheritance（私有继承）意味着 is-implemented-in-terms-of（是根据　实现的）。如果你使 class（类）D 从 class（类）B 私有继承，你这样做是因为你对于利用在 class（类）B 中才可用的某些特性感兴趣，而不是因为在 types（类型）B 和 types（类型）D 的 objects（对象）之间有什么概念上的关系。同样地，private inheritance（私有继承）纯粹是一种实现技术。（这也就是为什么你从一个 private base class（私有基类）继承的每一件东西都在你的 class（类）中变成 private（私有）的原因：它全部都是实现的细节。）利用 Item 34 中提出的条款，private inheritance（私有继承）意味着只有 implementation（实现）应该被继承；interface（接口）应该被忽略。如果 D 从 B 私有继承，它就意味着 D objects are implemented in terms of B objects（D 对象是根据 B 对象实现的），没有更多了。private inheritance（私有继承）在 software *design*（软件设计）期间没有任何意义，只在 software *implementation*（软件实现）期间才有。

private inheritance（私有继承）意味着 is-implemented-in-terms-of（是根据实现的）的事实有一点混乱，因为 Item 38 指出 composition

（复合）也有同样的含义。你怎么预先在它们之间做出选择呢？答案很简单：只要你能就用 composition（复合），只有在绝对必要的时候才用 private inheritance（私有继承）。什么时候是绝对必要呢？主要是当 protected members（保护成员）和/或 virtual functions（虚拟函数）掺和进来的时候，另外还有一种与空间相关的极端情况会使天平向 private inheritance（私有继承）倾斜。我们稍后再来操心这种极端情况。毕竟，它只是一种极端情况。

假设我们工作在一个包含 Widgets 的应用程序上，而且我们认为我们需要更好地理解 Widgets 是怎样被使用的。例如，我们不仅要知道 Widget member functions（成员函数）被调用的频度，还要知道 call ratios（调用率）随着时间的流逝如何变化。带有清晰的执行阶段的程序在不同的执行阶段可以有不同的行为侧重。？ 纾 桓霰嘁肫髡诮镶毫锥味院 氖褂糜胗呕 痛 肽 山锥尉陀泻艿蟮牟煌 ？

我们决定修改 Widget class 以持续跟踪每一个 member function（成员函数）被调用了多少次。在运行时，我们可以周期性地检查这一信息，与每一个 Widget 的这个值相伴的可能还有我们觉得有用的其它数据。为了进行这项工作，我们需要设立某种类型的 timer（计时器），以便在到达收集用法统计的时间时我们可以知道。

尽可能复用已有代码，而不是写新的代码，我在我的工具包中翻箱倒柜，而且满意地找到下面这个 class（类）：

```
class Timer {
public:
?explicit Timer(int tickFrequency);
   virtual void onTick() const;        ?// automatically called for each tick
?...
};
```

这正是我们要找的：一个我们能够根据我们的需要设定 tick 频率的 Timer object，而在每次 tick 时，它调用一个 virtual function（虚拟函数）。我们可以重定义这个 virtual function（虚拟函数）以便让它检查 Widget 所在的当前状态。很完美！

为了给 Widget 重定义 Timer 中的一个 virtual function（虚拟函数），Widget 必须从 Timer 继承。但是 public inheritance（公有继承）在这种情况下不合适。Widget is-a（是一个）Timer 不成立。Widget 的客户不应该能够在一个 Widget 上调用 onTick，因为在概念上那不是的 Widget 的 interface（接口）的一部分。允许这样的函数调用将使客户更容易误用 Widget 的 interface（接口），这是一个对 Item 18 的关于"使接口易于正确使用，而难以错误使用"的建议的明显违背。public inheritance（公有继承）在这里不是正确的选项。

窗体底端

由于程序的原因，本文件未被完整保存。

- # Item 40: 谨慎使用 multiple inheritance（多继承）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

触及 multiple inheritance (MI)（多继承）的时候，C++ 社区就会鲜明地分裂为两个基本的阵营。一个阵营认为如果 single inheritance (SI)（单继承）是有好处的，multiple inheritance（多继承）一定更有好处。另一个阵营认为 single inheritance（单继承）有好处，但是多继承引起的麻烦使它得不偿失。在这个 Item 中，我们的主要目的是理解在 MI 问题上的这两种看法。

首要的事情之一是要承认当将 MI 引入设计领域时，就有可能从多于一个的 base class（基类）中继承相同的名字（例如，函数，typedef，等等）。这就为歧义性提供了新的时机。例如：

```
class BorrowableItem {              // something a library lets you
borrow
public:
?void checkOut();                   // check the item out from the library

?...
};

class ElectronicGadget {
private:
?bool checkOut() const;             // perform self-test, return whether

?...                               ?// test succeeds
};

class MP3Player:                     // note MI here
?public BorrowableItem,             // (some libraries loan MP3 players)
?public ElectronicGadget
{ ... };                             // class definition is unimportant

MP3Player mp;

mp.checkOut();                       // ambiguous! which checkOut?
```

注意这个例子，即使两个函数中只有一个是可访问的，对 checkOut 的调用也是有歧义的。（checkOut 在 BorrowableItem 中是 public（公有）的，但在 ElectronicGadget 中是 private（私有）的。）这与 C++ 解析 overloaded functions（重载函数）调用的规则是一致的：在看到一个函数的是否可访问之前，C++ 首先确定与调用匹配最好的那个函数。只有在确定了 best-match function（最佳匹配函数）之后，才检查可访问性。这目前的情况下，两个 checkOuts 具有相同的匹配程度，所以就不存在最佳匹配。因此永远也不会检查到 ElectronicGadget::checkOut 的可访问性。

为了消除歧义性，你必须指定哪一个 base class（基类）的函数被调用：

```
mp.BorrowableItem::checkOut();              ?// ah, that checkOut...
```

当然，你也可以尝试显式调用 ElectronicGadget::checkOut，但这样做会有一个 "you're trying to call a private member function"（你试图调用一个私有成员函数）错误代替歧义性错误。

multiple inheritance（多继承）仅仅意味着从多于一个的 base class（基类）继承，但是在还有 higher-level base classes（更高层次基类）的 hierarchies（继承体系）中出现 MI 也并不罕见。这会导致有时被称为 "deadly MI diamond"（致命的多继承菱形）的后果。

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

你拥有一个“°在一个 base class（基类）和一个 derived class（派生类）之间有多于一条路径的 inheritance hierarchy（继承体系）”±（就像上面在 File 和 IOFile 之间，有通过 InputFile 和 OutputFile 的两条路径）的任何时候，你都必须面对是否需要为每一条路径复制 base class（基类）中的 data members（数据成员）的问题。例如，假设 File class 有一个 data members（数据成员）fileName。IOFile 中应该有这个 field（字段）的多少个拷贝呢？一方面，它从它的每一个 base classes（基类）继承一个拷贝，这就暗示 IOFile 应该有两个 fileName data members（数据成员）。另一方面，简单的逻辑告诉我们一个 IOFile object（对象）应该仅有一个 file name（文件名），所以通过它的两个 base classes（基类）继承来的 fileName field（字段）不应该被复制。

C++ 在这个争议上没有自己的立场。它恰当地支持两种选项，虽然它的缺省方式是执行复制。如果那不是? 阆胍 模 惚甄肴谜飧?class（类）带有一个 virtual base class（虚拟基类）的数据（也就是 File）。为了做到这一点，你要让从它直接继承的所有的 classes（类）使用 virtual inheritance（虚拟继承）：

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

标准 C++ 库包含一个和此类似的 MI hierarchy（继承体系），只是那个 classes（类）是 class templates（类模板），名字是 basic_ios，basic_istream，basic_ostream 和 basic_iostream，而不是 File，InputFile，OutputFile 和 IOFile。

从正确行为的观点看，public inheritance（公有继承）应该总是 virtual（虚拟）的。如果这是唯一的观点，规则就变得简单了：你使用 public inheritance（公有继承）的任何时候，都使用 virtual public inheritance（虚拟公有继承）。唉，正确性不是唯一的视角。避免 inherited fields（继承来的字段）复制需要在编译器的一部分做一些 behind-the-scenes legerdemain（幕后的戏法），而结果是从使用 virtual inheritance（虚拟继承）的 classes（类）创建的 objects（对象）通常比不使用 virtual inheritance（虚拟继承）的要大。访问 virtual base classes（虚拟基类）中的 data members（数据成员）也比那些 non-virtual base classes（非虚拟基类）中的要慢。编译器与编译器之间有一些细节不同，但基本的要点很清楚：virtual inheritance costs（虚拟继承要付出成本）。

它也有一些其它方面的成本。支配 initialization of virtual base classes（虚拟基类初始化）的规则比 non-virtual bases（非虚拟基类）的更加复杂而且更不直观。初始化一个 virtual base（虚拟基）的职责由 hierarchy（继承体系）中 most derived class （层次最低的派生类）承担。这个规则中包括的含义：（1） 从需要 initialization（初始化）的 virtual bases（虚拟基）派生的 classes（类）必须知道它们的 virtual bases（虚拟基），无论它距离那个 bases（基）有多远；（2） 当一个新的 derived class（派生类）被加入继承体系时，它必须为它的 virtual bases（虚拟基）（包括直接的和间接的）承担 initialization responsibilities（初始化职责）。

我对于 virtual base classes（虚拟基类）（也就是 virtual inheritance（虚拟继承））的建议很简单。首先，除非必需，否则不要使用 virtual bases（虚拟基）。缺省情况下，使用 non-virtual inheritance（非虚拟继承）。第二，如果你必须使用 virtual base classes（虚拟基类），试着避免在其中放置数据。这样你就不必在意它的

initialization（初始化）（以及它的 turns
out（清空），assignment（赋值））规则中的一些怪癖。值得一提的是 Java 和 .NET 中的
Interfaces（接口）不允许包含任何数据，它们在很多方面可以和 C++ 中的 virtual base
classes（虚拟基类）相比照。

现在我们使用下面的 C++ Interface class（接口类）（参见 Item 31）来为
persons（人）建模：

```
class IPerson {
public:
?virtual ~IPerson();

?virtual std::string name() const = 0;
?virtual std::string birthDate() const = 0;
};
```

IPerson 的客户只能使用 IPerson 的 pointers（指针）和
references（引用）进行编程，因为 abstract
classes（抽象类）不能被实例化。为了创建能被当作 IPerson objects（对象）使用的
objects（对象），IPerson 的客户使用 factory functions（工厂函数）（再次参见 Item 31
）instantiate（实例化）从 IPerson 派生的 concrete classes（具体类）：

```
// factory function to create a Person object from a unique database ID;
// see Item 18 for why the return type isn't a raw pointer
std::tr1::shared_ptr<IPerson> makePerson(DatabaseID personIdentifier);

// function to get a database ID from the user
DatabaseID askUserForDatabaseID();


DatabaseID id(askUserForDatabaseID());
std::tr1::shared_ptr<IPerson> pp(makePerson(id));  ?// create an object
                                                     // supporting the
                                                     // IPerson
interface

...                                                ?// manipulate *pp
via
                                                     // IPerson's member
                                                     // functions
```

但是 makePerson 怎样创建它返回的 pointers（指针）所指向的
objects（对象）呢？显然，必须有一些 makePerson 可以实例化的从 IPerson 派生的
concrete class（具体类）。

假设这个 class（类）叫做 CPerson。作为一个 concrete class（具体类），CPerson
必须提供它从 IPerson 继承来的 pure virtual functions（纯虚拟函数）的
implementations（实现）。它可以从头开始写，但利用包含大多数或全部必需品的现有组件更好一?
@ 纾 偕枰桓隼鲜降?database-specific class（老式的数据库专用类）PersonInfo
提供了 CPerson 所需要的基本要素：

```
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;    ?// see
?virtual const char * valueDelimClose() const;     // below
```

```
?...
};
```

你可以看出这是一个老式的 class（类），因为 member functions（成员函数）返回 const char*s 而不是 string objects（对象）。尽管如此，如果鞋子合适，为什么不穿呢？这个 class（类）的 member functions（成员函数）的名字暗示结果很可能会非常合适。

你突然发现 PersonInfo 是设计用来帮助以不同的格式打印 database fields（数据库字段）的，每一个字段的值的开始和结尾通过指定的字符串定界。缺省情况下，字段?悼 已徒嵛捕ń绶 欠嚼ê牛 宰侄沃?"Ring-tailed Lemur" 很可能被安排成这种格式：

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

根据方括号并非满足 PersonInfo 的全体客户的期望的事实，virtual functions（虚拟函数）valueDelimOpen 和 valueDelimClose 允许 derived classes（派生类）指定它们自己的开始和结尾定界字符串。PersonInfo 的 member functions（成员函数）的 implementations（实现）调用这些 virtual functions（虚拟函数）在它们返回的值上加上适当的定界符。作为一个例子使用 PersonInfo::theName，代码如下：

```
const char * PersonInfo::valueDelimOpen() const
{
?return "[";                          // default opening delimiter
}

const char * PersonInfo::valueDelimClose() const
{
?return "]";                          // default closing delimiter
}

const char * PersonInfo::theName() const
{
?// reserve buffer for return value; because this is
?// static, it's automatically initialized to all zeros
?static char value[Max_Formatted_Field_Value_Length];

?// write opening delimiter
?std::strcpy(value, valueDelimOpen());

?append to the string in value this object's   name field (being careful
?to avoid buffer overruns!)

?// write closing delimiter
?std::strcat(value, valueDelimClose());

?return value;
}
```

有人可能会质疑 PersonInfo::theName 的陈旧的设计（特别是一个 fixed-size static buffer（固定大小静态缓冲区）的使用，这样的东西发生 overrun（越界）和 threading（线程）问题是比较普遍的——参见 Item 21 ），但是请把这样的问题放到一边而注意这里：theName 调用 valueDelimOpen 生成它要返回的 string（字符串）的开始定界符，然后它生成名字值本身，然后它调用 valueDelimClose。

因为 valueDelimOpen 和 valueDelimClose 是 virtual functions（虚拟函数），theName 返回的结果不仅依赖于 PersonInfo，也依赖于从 PersonInfo 派生的 classes（类）。

对于 CPerson 的实现者，这是好消息，因为当细读 IPerson documentation（文档）中的 fine print（晦涩的条文）时，你发现 name 和 birthDate 需要返回未经修饰的值，也就是，不允许有定界符。换句话说，如果一个人的名字叫 Homer，对那个人的 name 函数的一次调用应该返回 "Homer"，而不是 "[Homer]"。

CPerson 和 PersonInfo 之间的关系是 PersonInfo 碰巧有一些函数使得 CPerson 更容易实现。这就是全部。因而它们的关系就是 is-implemented-in-terms-of，而我们知道有两种方法可以表现这一点：经由 composition（复合）（参见 Item 38）和经由 private inheritance（私有继承）（参见 Item 39）。Item 39 指出 composition（复合）是通常的首选方法，但如果 virtual functions（虚拟函数）要被重定义，inheritance（继承）就是必不可少的。在当前情况下，CPerson 需要重定义 valueDelimOpen 和 valueDelimClose，所以简单的 composition（复合）做不到。最直截了当的解决方案是让 CPerson 从 PersonInfo privately inherit（私有继承），虽然 Item 39 说过只要多做一点工作，则 CPerson 也能用 composition（复合）和 inheritance（继承）的组合有效地重定义 PersonInfo 的 virtuals（虚拟函数）。这里，我们用 private inheritance（私有继承）。

但是 CPerson 还必须实现 IPerson interface（接口），而这被称为 public inheritance（公有继承）。这就引出一个 multiple inheritance（多继承）的合理应用：组合 public inheritance of an interface（一个接口的公有继承）和 private inheritance of an implementation（一个实现的私有继承）：

```
class IPerson {                              ?// this class specifies the
public:                                      ?// interface to be implemented
?virtual ~IPerson();

?virtual std::string name() const = 0;
?virtual std::string birthDate() const = 0;
};

class DatabaseID { ... };                    ?// used below; details are
                                               // unimportant

class PersonInfo {                             // this class has functions
public:                                      ?// useful in implementing
?explicit PersonInfo(DatabaseID pid);         // the IPerson interface
?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 {      // note use of
MI
public:
?explicit CPerson(  ?DatabaseID pid): PersonInfo(pid) {}
?virtual std::string name() const                     ?// implementations
?{ return PersonInfo::theName(); }                      // of the
required
                                                       ?// IPerson member
?virtual std::string birthDate() const                  // functions
?{ return PersonInfo::theBirthDate(); }
private:                                                ?// redefinitions
of
?const char * valueDelimOpen() const { return ""; }  ?// inherited
virtual
?const char * valueDelimClose() const { return ""; }    // delimiter
};                                                      ?// functions
```

在 UML 中，这个设计看起来像这样：

这个例子证明 MI 既是有用的，也是可理解的。

时至今日，multiple inheritance（多继承）不过是 object-oriented toolbox（面向对象工具箱）里的又一种工具而已，典型情况下，它的使用和理解更加复杂，所以如果你得到一个或多或少等同于一个 MI 设计的 SI 设计，则 SI 设计总是更加可取。如果你能拿出来的仅有的设计包含 MI，你应该更加用心地考虑一下——总会有一些方法使得 SI 也能做到。但同时，MI 有时是最清晰的，最易于维护的，最合理的完成工作的方法。在这种情况下，毫不畏惧地使用它。只?且　繁＝魅鞣厥褂盟　？

Things to Remember

- multiple inheritance（多继承）比 single inheritance（单继承）更复杂。它能导致新的歧义问题和对 virtual inheritance（虚拟继承）的需要。
- virtual inheritance（虚拟继承）增加了 size（大小）和 speed（速度）成本，以及 initialization（初始化）和 assignment（赋值）的复杂度。当 virtual base classes（虚拟基类）没有数据时它是最适用的。
- multiple inheritance（多继承）有合理的用途。一种方案涉及组合从一个 Interface class（接口类）的 public inheritance（公有继承）和从一个有助于实现的 class（类）的 private inheritance（私有继承）。

窗体底端

Item 42: 理解 typename 的两个含义

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

问题：在下面的 template declarations（模板声明）中 class 和 typename 有什么不同?

```
template<class T> class Widget;                    // uses "class"

template<typename T> class Widget;              ?// uses "typename"
```

答案：没什么不同。在声明一个 template type parameter（模板类型参数）的时候，class 和 typename 意味着完全相同的东西。一些程序员更喜欢在所有的时间都用 class，因为它更容易输入。其他人（包括我本人）更喜欢 typename，因为它暗示着这个参数不必要是一个 class type（类类型）。少数开发者在任何类型都被允许的时候使用 typename，而把 class 保留给仅接受 user-defined types（用户定义类型）的场合。但是从 C++ 的观点看，class 和 typename 在声明一个 template parameter（模板参数）时意味着完全相同的东西。

然而，C++ 并不总是把 class 和 typename 视为等同的东西。有时你必须使用 typename。为了理解这一点，我们不得不讨论你会在一个 template（模板）中涉及到的两种名字。

假设我们有一个函数的模板，它能取得一个 STL-compatible container（STL 兼容容器）中持有的能赋值给 ints 的对象。进一步假设这个函数只是简单地打印它的第二个元素的值。它是一个用糊涂的方法实现的糊涂的函数?
揖拖裎蚁旅嫘吹模 踔敛荒鼙嘁耄 乔虢 庑┤孪确旁谝槐?——有一种方法能发现我的愚蠢：

```
template<typename C>                              ?// print 2nd element in
void print2nd(const C& container)                   // container;
{                                                   // this is not valid C++!
?if (container.size() >= 2) {
     C::const_iterator iter(container.begin()); // get iterator to 1st element
     ++iter;                                    ?// move iter to 2nd element
     int value = *iter;                             // copy that element to an int
     std::cout << value;                        ?// print the int
?}
}
```

我突出了这个函数中的两个 local variables（局部变量），iter 和 value。iter 的类型是 C::const_iterator，一个依赖于 template parameter（模板参数）C 的类型。一个 template（模板）中的依赖于一个 template parameter（模板参数）的名字被称为 *dependent names*（依赖名字）。当一个 dependent names（依赖名字）嵌套在一个 class（类）的内部时，我称它为 *nested dependent name*（嵌套依赖名字）。C::const_iterator 是一个 nested dependent name（嵌套依赖名字）。实际上，它是一个 *nested dependent type name*（嵌套依赖类型名），也就是说，一个涉及到一个 type（类型）的 nested dependent name（嵌套依赖名字）。

print2nd 中的另一个 local variable（局部变量）value 具有 int 类型。int 是一个不依赖于任何 template parameter（模板参数）的名字。这样的名字以 *non-dependent names*（非依赖名字）闻名。（我想不通为什么他们不称它为 independent names（无依赖名字）。如果，像我一样，你发现术语 "non-dependent" 是一个令人厌恶的东西，你就和我产生了共鸣，但是 "non-dependent"

就是这类名字的术语，所以，像我一样，转转眼睛放弃你的自我主张。）

nested dependent name（嵌套依赖名字）会导致解析困难。例如，假设我们更加愚蠢地以这种方法开始 print2nd：

```
template<typename C>
void print2nd(const C& container)
{
?C::const_iterator * x;
?...
}
```

这看上去好像是我们将 x 声明为一个指向 C::const_iterator 的 local variable（局部变量）。但是它看上去如此仅仅是因为我们知道 C::const_iterator 是一个 type（类型）。但是如果 C::const_iterator 不是一个 type（类型）呢？如果 C 有一个 static data member（静态数据成员）碰巧就叫做 const_iterator 呢？再如果 x 碰巧是一个 global variable（全局变量）的名字呢？在这种情况下，上面的代码就不是声明一个 local variable（局部变量），而是成为 C::const_iterator 乘以 x！当然，这听起来有些愚蠢，但它是可能的，而编写 C++ 解析器的人必须考虑所有可能的输入，甚至是愚蠢的。

直到 C 成为已知之前，没有任何办法知道 C::const_iterator 到底是不是一个 type（类型），而当 template（模板）print2nd 被解析的时候，C 还不是已知的。C++ 有一条规则解决这个歧义：如果解析器在一个 template（模板）中遇到一个 nested dependent name（嵌套依赖名字），它假定那个名字不是一个 type（类型），除非你用其它方式告诉它。缺省情况下，nested dependent name（嵌套依赖名字）不是 types（类型）。（对于这条规则有一个例外，我待会儿告诉你。）

记住这个，再看看 print2nd 的开头：

```
template<typename C>
void print2nd(const C& container)
{
?if (container.size() >= 2) {
     C::const_iterator iter(container.begin());   // this name is assumed to
     ...                                          ?// not be a type
```

这为什么不是合法的 C++ 现在应该很清楚了。iter 的 declaration（声明）仅仅在 C::const_iterator 是一个 type（类型）时才有意义，但是我们没有告诉 C++ 它是，而 C++ 就假定它不是。要想转变这个形势，我们必须告诉 C++ C::const_iterator 是一个 type（类型）。我们将 typename 放在紧挨着它的前面来做到这一点：

```
template<typename C>                              // this is valid C++
void print2nd(const C& container)
{
?if (container.size() >= 2) {
  ?typename C::const_iterator iter(container.begin());
  ?...
?}
}
```

通用的规则很简单：在你涉及到一个在 template（模板）中的 nested dependent type name（嵌套依赖类型名）的任何时候，你必须把单词 typename 放在紧挨着它的前面。（重申一下，我待会儿要描述一个例外。）

typename 应该仅仅被用于标识 nested dependent type name（嵌套依赖类型名）；其它名字不应该用它。例如，这是一个取得一个 container（容器）和这个 container

（容器）中的一个 iterator（迭代器）的 function template（函数模板）：

```
template<typename C>                    // typename allowed (as is "class")
void f(const C& container,              // typename not allowed
       typename C::iterator iter);      // typename required
```

C 不是一个 nested dependent type name（嵌套依赖类型名）（它不是嵌套在依赖于一个 template parameter（模板参数）的什么东西内部的），所以在声明 container 时它不必被 typename 前置，但是 C::iterator 是一个 nested dependent type name（嵌套依赖类型名），所以它必需被 typename 前置。

"typename must precede nested dependent type names"（"typename 必须前置于嵌套依赖类型名"）规则的例外是 typename 不必前置于在一个 list of base classes（基类列表）中的或者在一个 member initialization list（成员初始化列表）中作为一个 base classes identifier（基类标识符）的 nested dependent type name（嵌套依赖类型名）。例如：

```
template<typename T>
class Derived: public Base<T>::Nested { // base class list: typename not
public:                                  // allowed
?explicit Derived(int x)
?: Base<T>::Nested(x)                 ?// base class identifier in mem
?{                                       // init. list: typename not allowed

  ?typename Base<T>::Nested temp;     ?// use of nested dependent type
  ?...                                   // name not in a base class list or
?}                                       // as a base class identifier in a
?...                                     // mem. init. list: typename required
};
```

这样的矛盾很令人讨厌，但是一旦你在经历中获得一点经验，你几乎不会在意它。

让我们来看最后一个 typename
的例子，因为它在你看到的真实代码中具有代表性。假设我们在写一个取得一个 iterator（迭代器）的 function template（函数模板），而且我们要做一个 iterator（迭代器）指向的 object（对象）的局部拷贝 temp，我们可以这样做：

```
template<typename IterT>
void workWithIterator(IterT iter)
{
?typename std::iterator_traits<IterT>::value_type temp(*iter);
?...
}
```

不要让 std::iterator_traits<IterT>::value_type 吓倒你。那仅仅是一个 standard traits class（标准特性类）（参见 Item 47）的使用，用 C++ 的说法就是 "the type of thing pointed to by objects of type IterT"（"被类型为 IterT 的对象所指向的东西的类型"）。这个语句声明了一个与 IterT objects 所指向的东西类型相同的 local variable（局部变量）(temp)，而且用 iter 所指向的 object（对象）对 temp 进行了初始化。如果 IterT 是 vector<int>::iterator，temp 就是 int 类型。如果 IterT 是 list<string>::iterator，temp 就是 string 类型。因为 std::iterator_traits<IterT>::value_type 是一个 nested dependent type name（嵌套依赖类型名）（value_type 嵌套在 iterator_traits<IterT> 内部，而且 IterT 是一个 template parameter（模板参数）），我们必须让它被 typename 前置。

如果你觉得读 std::iterator_traits<IterT>::value_type
令人讨厌，就想象那个与它相同的东西来代表它。如果你像大多数程序员，对多次输入它感到恐惧，那么你就? 枰　唇T桓? typedef。对于像 value_type 这样的 traits member names（特性成员名）（再次参见

Item 47 关于 traits 的资料），一个通用的惯例是 typedef name 与 traits member name 相同，所以这样的一个 local typedef 通常定义成这样：

```
template<typename IterT>
void workWithIterator(IterT iter)
{
?typedef typename std::iterator_traits<IterT>::value_type value_type;

?value_type temp(*iter);
?...
}
```

很多程序员最初发现 "typedef typename" 并列不太和谐，但它是涉及 nested dependent type names（嵌套依赖类型名）规则的一个合理的附带结果。你会相当快地习惯它。你毕竟有着强大的动机。你输入 typename std::iterator_traits<IterT>::value_type 需要多少时间？

作为结束语，我应该提及编译器与编译器之间对围绕 typename 的规则的执行情况的不同。一些编译器接受必需 typename 时它却缺失的代码；一些编译器接受不许 typename 时它却存在的代码；还有少数的（通常是老旧的）会拒绝 typename 出现在它必需出现的地方。这就意味着 typename 和 nested dependent type names（嵌套依赖类型名）的交互作用会导致一些轻微的可移植性问题。

Things to Remember

- 在声明 template parameters（模板参数）时，class 和 typename 是可互换的。
- 用 typename 去标识 nested dependent type names（嵌套依赖类型名），在 base class lists（基类列表）中或在一个 member initialization list（成员初始化列表）中作为一个 base class identifier（基类标识符）时除外。

窗体底端

Item 43: 了解如何访问 templatized base classes（模板化基类）中的名字

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

假设我们要写一个应用程序，它可以把消息传送到几个不同的公司去。消息既可以以加密方式也可以以明文（?患用埽¬ 姆绞酱　汀H绻　颐怯凶愎坏男畔⑯诒嘁肫诩淙范　母鲅　13.　(7)透　母齰　荆　颐蔷涂梢杂靡桓? template-based（模板基）来解决问题：

```
class CompanyA {
public:
?...
?void sendCleartext(const std::string& msg);
?void sendEncrypted(const std::string& msg);
?...
};

class CompanyB {
public:
?...
?void sendCleartext(const std::string& msg);
?void sendEncrypted(const std::string& msg);
?...
};
...                                     // classes for other companies

class MsgInfo { ... };                 ?// class for holding information
                                       ?// used to create a message
template<typename Company>
class MsgSender {
public:
?...                                    // ctors, dtor, etc.

?void sendClear(const MsgInfo& info)
?{
  ?std::string msg;
  ?create msg from info;

  ?Company c;
  ?c.sendCleartext(msg);
?}

?void sendSecret(const MsgInfo& info)   // similar to sendClear, except
?{ ... }                               ?// calls c.sendEncrypted
};
```

这个能够很好地工作，但是假设我们有时需要在每次发送消息的时候把一些信息记录到日志中。通过一个 derived class（派生类）可以很简单地增加这个功能，下面这个似乎是一个合理的方法：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
```

```
public:
?...                                    ?// ctors, dtor, etc.
?void sendClearMsg(const MsgInfo& info)
?{
  ?write "before sending" info to the log;

  ?sendClear(info);                      // call base class function;
                                          // this code will not compile!
  ?write "after sending" info to the log;
?}
?...

};
```

注意 derived class（派生类）中的 message-sending function（消息发送函数）的名字 (sendClearMsg) 与它的 base class（基类）中的那个（在那里，它被称为 sendClear）不同。这是一个好的设计，因为它避开了 hiding inherited names（隐藏继承来的名字）的问题（参见 Item 33）和重定义一个 inherited non-virtual function（继承来的非虚拟函数）的与生俱来的问题（参见 Item 36）。但是上面的代码不能通过编译，至少在符合标准的编译器上不能。这样的编译器会抱怨 sendClear 不存在。我们可以看见 sendClear 就在 base class（基类）中，但编译器不会到那里去寻找它。我们有必要理解这是为什么。

问题在于当编译器遇到 class template（类模板）LoggingMsgSender 的 definition（定义）时，它们不知道它从哪个 class（类）继承。当然，它是 MsgSender<Company>，但是 Company 是一个 template parameter（模板参数），这个直到更迟一些才能被确定（当 LoggingMsgSender 被实例化的时候）。不知道 Company 是什么，就没有办法知道 class（类）MsgSender<Company> 是什么样子的。特别是，没有办法知道它是否有一个 sendClear function（函数）。

为了使问题具体化，假设我们有一个要求加密通讯的 class（类）CompanyZ：

```
class CompanyZ {                          // this class offers no
public:                                  ?// sendCleartext function
?...
?void sendEncrypted(const std::string& msg);
?...
};
```

一般的 MsgSender template（模板）不适用于 CompanyZ，因为那个模板提供一个 sendClear function（函数）对于 CompanyZ objects（对象）没有意义。为了纠正这个问题，我们可以创建一个 MsgSender 针对 CompanyZ 的特化版本：

```
template<>                                // a total specialization of
class MsgSender<CompanyZ> {              ?// MsgSender; the same as the
public:                                  ?// general template, except
?...                                    ?// sendCleartext is omitted
?void sendSecret(const MsgInfo& info)
?{ ... }
};
```

注意这个 class definition（类定义）开始处的 "template <>" 语法。它表示这既不是一个 template（模板），也不是一个 standalone class（独立类）。正确的说法是，它是一个用于 template argument（模板参数）为 CompanyZ 时的 MsgSender template（模板）的 specialized version（特化版本）。这以 *total template specialization*（完全模板特化）闻名：template（模板）MsgSender 针对类型 CompanyZ 被特化，而且这个 specialization（特化）是 *total*（完全）的——只要 ype parameter（类型参数）被定义成了 CompanyZ，就没有剩下能被改变的其它 template's parameters（模板参数）。

窗体底端

由于程序的原因，本文件未被完整保存。

窗体顶端

翻译：Effective C++, 3rd Edition, Item 43: 了解如何访问 templatized base classes（模板化基类）中的名字（下）

（点击此处，接上篇）

已知 MsgSender 针对 CompanyZ 被特化，再次考虑 derived class（派生类）LoggingMsgSender：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
?...
?void sendClearMsg(const MsgInfo& info)
?{
  ?write "before sending" info to the log;

  ?sendClear(info);                          ?// if Company == CompanyZ,
                                              ?// this function doesn't exist!
  ?write "after sending" info to the log;
?}
?...

};
```

就像注释中写的，当 base class（基类）是 MsgSender<CompanyZ> 时，这里的代码是无意义的，因为那个类没有提供 sendClear function（函数）。这就是为什么 C++ 拒绝这个调用：它认识到 base class templates（基类模板）可能被特化，而这样的 specializations（特化）不一定提供和 general template（通用模板）相同的 interface（接口）。结果，它通常会拒绝在 templatized base classes（模板化基类）中寻找 inherited names（继承来的名字）。在某种意义上，当我们从 Object-oriented C++ 跨越到 Template C++（参见 Item 1）时，inheritance（继承）会停止工作。

为了重新启动它，我们必须以某种方式使 C++ 的 "don't look in templatized base classes"（不在模板基类中寻找）行为失效。有三种方法可以做到这一点。首先，你可以在被调用的 base class functions（基类函数）前面加上 "this->"：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:

?...

?void sendClearMsg(const MsgInfo& info)
?{
  ?write "before sending" info to the log;

  ?this->sendClear(info);                ?// okay, assumes that
                                          ?// sendClear will be inherited
  ?write "after sending" info to the log;
?}

?...

};
```

第二，你可以使用一个 using declaration，如果你已经读过 Item 33，这应该是你很熟悉的一种解决方案。那个 Item 解释了 using declarations 如何将被隐藏的 base class names（基类名字）引入到一个 derived class（派生类）范围中。因此我们可以这样写 sendClearMsg：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
```

```
?using MsgSender<Company>::sendClear;   // tell compilers to assume
?...                                  ?// that sendClear is in the
                                         // base class
?void sendClearMsg(const MsgInfo& info)
?{
  ?...
  ?sendClear(info);                    ?// okay, assumes that
  ?...                                ??// sendClear will be inherited
?}

?...
};
```

（虽然 using declaration 在这里和 Item 33 中都可以工作，但要解决的问题是不同的。这里的情形不是 base class names（基类名字）被 derived class names（派生类名字）隐藏，而是如果我们不告诉编译器去做，它们就不会搜索 base class 范围。）

最后一个让你的代码通过编译的办法是显式指定被调用的函数是在 base class（基类）中的：

```
template<typename Company>
class LoggingMsgSender: public MsgSender<Company> {
public:
?...
?void sendClearMsg(const MsgInfo& info)
?{
  ?...
  ?MsgSender<Company>::sendClear(info);    ?// okay, assumes that
  ?...                                       // sendClear will be
?}                                           // inherited

?...
};
```

通常这是一个解决这个问题的最不合人心的方法，因为如果被调用函数是 virtual（虚拟）的，显式限定会关闭 virtual binding（虚拟绑定）行为。

从名字可见性的观点来看，这里每一个方法都做了同样的事情：它向编译器保证任何后继的 base class template（基类模板）的 specializations（特化）都将支持 general template（通用模板）提供的 interface（接口）。所有的编译器在解析一个像 LoggingMsgSender 这样的 derived class template（派生类模板）时，这样一种保证都是必要的，但是如果保证被证实不成立，真相将在后继的编译过程中暴露。例如，如果后面的源代码中包含这些，

```
LoggingMsgSender<CompanyZ> zMsgSender;

MsgInfo msgData;

...                                      ?// put info in msgData

zMsgSender.sendClearMsg(msgData);        ?// error! won't compile
```

对 sendClearMsg 的调用将不能编译，因为在此刻，编译器知道 base class（基类）是 template specialization（模板特化）MsgSender<CompanyZ>，它们也知道那个 class（类）没有提供 sendClearMsg 试图调用的 sendClear function（函数）。

从根本上说，问题就是编译器是早些（当 derived class template definitions（派生类模板定义）被解析的时候）诊断对 base class members（基类成员）的非法引用，还是晚些时候（当那些 templates（模板）被特定的 template arguments（模板参数）实例化的时候）再进行。C++ 的方针是宁愿早诊断，而这就是为什么当那些 classes（类）被从 templates（模板）实例化的时候，它假装不知道 base classes（基类）的内容。

Things to Remember

- 在 derived class templates（派生类模板）中，可以经由 "this->" 前缀，经由 using declarations，或经由一个 explicit base class qualification（显式基类限定）引用 base class templates（基类模板）中的名字。

窗体底端

- # Item 45: 用 member function templates（成员函数模板） 接受 "all compatible types"（“所有兼容类型”）

作者：Scott Meyers

译者：fatalerror99 (iTePub's Nirvana)

发布：http://blog.csdn.net/fatalerror99/

smart pointers（智能指针）是行为很像指针但是增加了指针没有提供的功能的 objects。例如，Item 13 阐述了标准 `auto_ptr` 和 `tr1::shared_ptr` 是怎样被应用于在恰当的时间自动删除的 heap-based resources（基于堆的资源）的。STL containers 内的 iterators（迭代器）几乎始终是 smart pointers（智能指针）；你绝对不能指望用 "++" 将一个 built-in pointer（内建指针）从一个 linked list（线性链表）的一个节点移动到下一个，但是 `list::iterators` 可以做到。

real pointers（真正的指针）做得很好的一件事是支持 implicit conversions（隐式转换）。derived class pointers（派生类指针）隐式转换到 base class pointers（基类指针），pointers to non-`const` objects（指向非常量对象的指针）转换到 pointers to `const` objects（指向常量对象的指针），等等。例如，考虑在一个 three-level hierarchy（三层继承体系）中能发生的一些转换：

```
class Top { ... };
class Middle: public Top { ... };
class Bottom: public Middle { ... };
Top *pt1 = new Middle;                    // convert Middle* => Top*
Top *pt2 = new Bottom;                    // convert Bottom* => Top*
const Top *pct2 = pt1;                    // convert Top* => const Top*
```

在 user-defined smart pointer classes（用户定义智能指针类）中模仿这些转换是需要技巧的。我们要让下面的代码能够编译：

```
template<typename T>
class SmartPtr {
public:                                 // smart pointers are typically
?explicit SmartPtr(T *realPtr);   ?// initialized by built-in pointers
?...
};

SmartPtr<Top> pt1 =                       // convert SmartPtr<Middle> =>
SmartPtr<Middle>(new Middle);      ?//  martPtr<Top>

SmartPtr<Top> pt2 =                       // convert SmartPtr<Bottom> =>
SmartPtr<Bottom>(new Bottom);     ??/  martPtr<Top>

SmartPtr<const Top> pct2 = pt1;           // convert SmartPtr<Top> =>
                                   ?//  martPtr<const Top>
```

在同一个 template（模板）的不同 instantiations（实例化）之间没有 inherent relationship（继承关系），所以编译器认为 `SmartPtr<Middle>` 和 `SmartPtr<Top>` 是完全不同的 classes，并不比（比方说）`vector<float>` 和 `Widget` 的关系更近。为了得到我们想要的在 `SmartPtr` classes 之间的转换，我们必须显式地为它们编程。

在上面的 smart pointer（智能指针）的示例代码中，每一个语句创建一个新的 smart pointer

object（智能指针对象），所以现在我们就集中于我们如何写 smart pointer constructors（智能指针的构造函数），让它以我们想要的方式运转。一个关键的事实是我们无法写出我们需要的? ？constructors（构造函数）。在上面的 hierarchy（继承体系）中，我们能从一个 SmartPtr<Middle> 或一个 SmartPtr<Bottom> 构造出一个 SmartPtr<Top>，但是如果将来这个 hierarchy（继承体系）被扩充，SmartPtr<Top> objects 还必须能从其它 smart pointer types（智能指针类型）构造出来。例如，如果我们后来加入

```
class BelowBottom: public Bottom { ... };
```

我们就需要支持从 SmartPtr<BelowBottom> objects 到 SmartPtr<Top> objects 的创建，而且我们当然不希望为了做到这一点而必须改变 SmartPtr template。

大体上，我们需要的 constructors（构造函数）的数量是无限的。因为一个 template（模板）能被实例化而产生无数个函数，所以好像我们不需要为 SmartPtr 提供一个 constructor *function*（构造函数函数），我们需要一个 constructor *template*（构造函数模板）。这样的 templates（模板）是 *member function templates*（成员函数模板）（常常被恰如其分地称为 *member templates*（成员模板））——生成一个 class 的 member functions（成员函数）的 templates（模板）的范例：

```
template<typename T>
class SmartPtr {
public:
?template<typename U>                          // member template
?SmartPtr(const SmartPtr<U>& other);       ?// for a "generalized
?...                                       ?// copy constructor"
};
```

这就是说对于每一种类型 T 和每一种类型 U，都能从一个 SmartPtr<U> 创建出一个 SmartPtr<T>，因为 SmartPtr<T> 有一个取得一个 SmartPtr<U> 参数的 constructor（构造函数）。像这样的 constructor（构造函数）——从一个类型是同一个 template（模板）的不同实例化的 object 创建另一个 object 的 constructor（构造函数）（例如，从一个 SmartPtr<U> 创建一个 SmartPtr<T>）——有时被称为 *generalized copy constructors*（泛型化拷贝构造函数）。

上面的 generalized copy constructor（泛型化拷贝构造函数）没有被声明为 explicit（显式）的。这是故意为之的。built-in pointer types（内建指针类型）之间的类型转换（例如，从派生类指针到基类指针）是隐式的和不需要 cast（强制转型）的，所以让 smart pointers（智能指针）模仿这一行为是合理的。在 templatized constructor（模板化构造函数）中省略 explicit 正好做到这一点。

作为声明，SmartPtr 的 generalized copy constructor（泛型化拷贝构造函数）提供的东西比我们想要的还多。是的，我们需要能够从一个 SmartPtr<Bottom> 创建一个 SmartPtr<Top>，但是我们不需要能够从一个 SmartPtr<Top> 创建一个 SmartPtr<Bottom>，这就像颠倒 public inheritance（公有继承）的含义（参见 Item 32）。我们也不需要能够从一个 SmartPtr<double> 创建一个 SmartPtr<int>，因为这和从 int* 到 double* 的 implicit conversion（隐式转换）是不相称的。我们必须设法过滤从这个 member template（成员模板）生成的 member functions（成员函数）的群体。

假如 SmartPtr 跟随 auto_ptr 和 tr1::shared_ptr 的脚步，提供一个返回被这个 smart pointer（智能指针）持有的 built-in pointer（内建指针）的拷贝的 get member function（get 成员函数）（参见 Item 15），我们可以用 constructor template（构造函数模板）的实现将转换限定在我们想要的范围：

```
template<typename T>
class SmartPtr {
public:
```

```
?template<typename U>
?SmartPtr(const SmartPtr<U>& other)          // initialize this held ptr
?: heldPtr(other.get()) { ... }              // with other's held ptr

?T* get() const { return heldPtr; }
?...

private:                                       // built-in pointer held
?T *heldPtr;                                 ?// by the SmartPtr
};
```

我们通过 member initialization list（成员初始化列表），用 SmartPtr<U> 持有的类型为 U* 的指针初始化 SmartPtr<T> 的类型为 T* 的 data member（数据成员）。这只有在"存在一个从一个 U* 指针到一个 T* 指针的 implicit conversion（隐式转换）"的条件下才能编译，而这正是我们想要的。最终的效果就是 SmartPtr<T> 现在有一个 generalized copy constructor（泛型化拷贝构造函数），它只有在传入一个 compatible type（兼容类型）的参数时才能编译。

member function templates（成员函数模板）的用途并不限于 constructors（构造函数）。它们的另一个常见的任务是用于支持 assignment（赋值）。例如，TR1 的 shared_ptr（再次参见 Item 13）支持从所有兼容的 built-in pointers（内建指针），tr1::shared_ptrs，auto_ptrs 和 tr1::weak_ptrs（参见 Item 54）构造，以及从除 tr1::weak_ptrs 以外所有这些赋值。这里是从 TR1 规范中摘录出来的一段关于 tr1::shared_ptr 的内容，包括它在声明 template parameters（模板参数）时使用 class 而不是 typename 的偏好。（就像 Item 42 中阐述的，在这里的上下文环境中，它们的含义严格一致。）

```
template<class T> class shared_ptr {
public:
?template<class Y>                                  // construct from
  ?explicit shared_ptr(Y * p);                      // any compatible
?template<class Y>                                  // built-in
pointer,
  ?shared_ptr(shared_ptr<Y> const& r);              // shared_ptr,
?template<class Y>                                  // weak_ptr, or
  ?explicit shared_ptr(weak_ptr<Y> const& r);      ?// auto_ptr
?template<class Y>
  ?explicit shared_ptr(auto_ptr<Y>& r);
?template<class Y>                                  // assign from
  ?shared_ptr& operator=(shared_ptr<Y> const& r);  ?// any compatible
?template<class Y>                                  // shared_ptr or
  ?shared_ptr& operator=(auto_ptr<Y>& r);          ?// auto_ptr
?...
};
```

除了 generalized copy constructor（泛型化拷贝构造函数），所有这些 constructors（构造函数）都是 explicit（显式）的。这就意味着从 shared_ptr 的一种类型到另一种的 implicit conversion（隐式转换）是被允许的，但是从一个 built-in pointer（内建指针）或其 smart pointer type（智能指针类型）的 *implicit* conversion（隐式转换）是不被许可的。（*explicit* conversion（显式转换）——例如，经由一个 cast（强制转型）——还是可以的。）同样引起注意的是 auto_ptrs 被传送给 tr1::shared_ptr 的 constructors（构造函数）和 assignment operators（赋值操作符）的方式没有被声明为 const，于此对照的是 tr1::shared_ptrs 和 tr1::weak_ptrs 的被传送的方式。这是 auto_ptrs 被复制时需要独一无二的被改变的事实的一个必然结果（参见 Item 13）。

member function templates（成员函数模板）是一个极好的东西，但是它们没有改变这个语言的基本规则。Item 5

阐述的编译器可以产生的四个 member functions（成员函数）其中两个是 copy constructor（拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）。tr1::shared_ptr 声明了一个 generalized copy constructor（泛型化拷贝构造函数），而且很明显，当类型 T 和 Y 相同时，generalized copy constructor（泛型化拷贝构造函数）就能被实例化而成为 "normal" copy constructor（"常规"拷贝构造函数）。那么，当一个 tr1::shared_ptr object 从另一个相同类型的 tr1::shared_ptr object 构造时，编译器是为 tr1::shared_ptr 生成一个 copy constructor（拷贝构造函数），还是实例化 generalized copy constructor template（泛型化拷贝构造函数模板）？

就像我说过的，member templates（成员模板）不改变语言规则，而且规则规定如果一个 copy constructor（拷贝构造函数）是必需的而你没有声明，将为你自动生成一个。在一个 class 中声明一个 generalized copy constructor（泛型化拷贝构造函数）（一个 member template（成员模板））不会阻止编译器生成它们自己的 copy constructor（拷贝构造函数）（非模板的），所以如果你要全面支配 copy construction（拷贝构造），你必须既声明一个 generalized copy constructor（泛型化拷贝构造函数）又声明一个 "normal" copy constructor（"常规"拷贝构造函数）。这同样适用于 assignment（赋值）。这是从 tr1::shared_ptr 的定义中摘录的一段，可以作为例子：

```
template<class T> class shared_ptr {
public:
?shared_ptr(shared_ptr const& r);                  // copy constructor

?template<class Y>                                 ?// generalized
  ?shared_ptr(shared_ptr<Y> const& r);             ?// copy constructor

?shared_ptr& operator=(shared_ptr const& r);       ?// copy assignment

?template<class Y>                                 ?// generalized
  ?shared_ptr& operator=(shared_ptr<Y> const& r); // copy assignment
?...
};
```

Things to Remember

- 使用 member function templates（成员函数模板）生成接受所有兼容类型的函数。
- 如果你为 generalized copy construction（泛型化拷贝构造）或 generalized assignment（泛型化赋值）声明了 member templates（成员模板），你依然需要声明 normal copy constructor（常规拷贝构造函数）和 copy assignment operator（拷贝赋值运算符）。

窗体底端