第一章 从C转向C++

对每个人来说，习惯C++需要一些时间，对于已经熟悉C的程序员来说，这个过程尤其令人苦恼。因为C是C++的子集，所有的C的技术都可以继续使用，但很多用起来又不太合适。例如，C++程序员会认为指针的指针看起来很古怪，他们会问：为什么不用指针的引用来代替呢？

C是一种简单的语言。它真正提供的只有有宏、指针、结构、数组和函数。不管什么问题，C都靠宏、指针、结构、数组和函数来解决。而C++不是这样。宏、指针、结构、数组和函数当然还存在，此外还有私有和保护型成员、函数重载、缺省参数、构造和析构函数、自定义操作符、内联函数、引用、友元、模板、异常、名字空间，等等。用C++比用C具有更宽广的空间，因为设计时有更多的选择可以考虑。

在面对这么多的选择时，许多C程序员墨守成规，坚持他们的老习惯。一般来说，这也不是什么很大的罪过。但某些C的习惯有悖于C++的精神本质，他们都在下面的条款进行了阐述。

条款 1：尽量用const和inline而不用#define

这个条款最好称为：“尽量用编译器而不用预处理”，因为#define经常被认为好象不是语言本身的一部分。这是问题之一。再看下面的语句：
#define ASPECT_RATIO 1.653

编译器会永远也看不到ASPECT_RATIO这个符号名，因为在源码进入编译器之前，它会被预处理程序去掉，于是ASPECT_RATIO不会加入到符号列表中。如果涉及到这个常量的代码在编译时报错，就会很令人费解，因为报错信息指的是 1.653，而不是ASPECT_RATIO。如果ASPECT_RATIO不是在你自己写的头文件中定义的，你就会奇怪 1.653 是从哪里来的，甚至会花时间跟踪下去。这个问题也会出现在符号调试器中，因为同样地，你所写的符号名不会出现在符号列表中。
解决这个问题的方案很简单：不用预处理宏，定义一个常量：
const double ASPECT_RATIO = 1.653;

这种方法很有效。但有两个特殊情况要注意。
首先，定义指针常量时会有点不同。因为常量定义一般是放在头文件中（许多源文件会包含它），除了指针所指的类型要定义成const外，重要的是指针也经常要定义成const。例如，要在头文件中定义一个基于char*的字符串常量，你要写两次const：
const char * const authorName = "Scott Meyers";

关于const的含义和用法，特别是和指针相关联的问题，参见条款 21。

另外，定义某个类(class)的常量一般也很方便，只有一点点不同。要把常量限制在类中，首先要使它成为类的成员；为了保证常量最多只有一份拷贝，还要把它定义为静态成员：

```
class GamePlayer {
private:
        static const int NUM_TURNS = 5; // constant eclaration
        int scores[NUM_TURNS];                  // use of constant
        ...
};
```

还有一点，正如你看到的，上面的语句是NUM_TURNS的声明，而不是定义，所以你还必须在类的实现代码文件中定义类的静态成员：
const int GamePlayer::NUM_TURNS; // mandatory definition;
                // goes in class impl.file

你不必过于担心这种小事。如果你忘了定义，链接器会提醒你。

旧一点的编译器会不接受这种语法，因为它认为类的静态成员在声明时定义初始值是非法的；而且，类内只允许初始化整数类型(如：int, bool, char 等)，还只能是常量。
在上面的语法不能使用的情况下，可以在定义时赋初值：

```
class EngineeringConstants { // this goes in the class
private:                     // header file
        static const double FUDGE_FACTOR;
        ...
};
        // this goes in the class implementation file
        const double EngineeringConstants::FUDGE_FACTOR = 1.35;
```

大多数情况下你只要做这么多。唯一例外的是当你的类在编译时需要用到这个类的常量的情况，例如上面GamePlayer::scores数组的声明(编译过程中编译器一定要知道数组的大小)。所以，为了弥补那些(不正确地)禁止类内进行整型类常量初始化的编译器的不足，可以采用称之为“借用enum”的方法来解决。这种技术很好地利用了当需要int类型时可以使用枚举类型的原则，所以GamePlayer也可以象这样来定义：

```
class GamePlayer {
private:
        enum { NUM_TURNS = 5 }    // "the enum hack" — makes
        // NUM_TURNS a symbolic name
        // for 5
        int scores[NUM_TURNS];// fine
};
```

除非你正在用老的编译器(即写于 1995 年之前)，你不必借用enum。当然，知道有这种方法还是值得的，因为这种可以追溯到很久以前的时代的代码可是不常见的哟。

回到预处理的话题上来。另一个普遍的#define指令的用法是用它来实现那些看起来象函数而又不会导致函数调用的宏。典型的例子是计算两个对象的最大值：

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

这个语句有很多缺陷，光想想都让人头疼，甚至比在高峰时间到高速公路去开车还让人痛苦。无论什么时候你写了象这样的宏，你必须记住在写宏体时对每个参数都要加上括号；否则，别人调用你的宏时如果用了表达式就会造成很大的麻烦。但是即使你象这样做了，还会有象下面这样奇怪的事发生：

```
int a = 5, b = 0;
max(++a, b);// a 的值增加了 2 次
max(++a, b+10); // a 的值只增加了 1 次
```

这种情况下，max内部发生些什么取决于它比较的是什么值！
幸运的是你不必再忍受这样愚笨的语句了。你可以用普通函数实现宏的效率，再加上可预计的行为和类型安全，这就是内联函数(见条款 33)：

```
inline int max(int a, int b) { return a > b ? a : b; }
```

不过这和上面的宏不大一样，因为这个版本的max只能处理int类型。但模板可以很轻巧地解决这个问题：

```
template<class T>
inline const T& max(const T& a, const T& b)
{ return a > b ? a : b; }
```

这个模板产生了一整套函数，每个函数拿两个可以转换成同种类型的对象进行比较然后返回较大的(常量)对象的引用。因为不知道T的类型，返回时传递引用可以提高效率(见条款 22)。

顺便说一句，在你打算用模板写象max这样有用的通用函数时，先检查一下标准库(见条款 49)，看看他们是不是已经存在。比如说上面说的max，你会惊喜地发现你可以后人乘凉：max是C++标准库的一部分。

有了const和inline，你对预处理的需要减少了，但也不能完全没有它。抛弃#include的日子还很远，#ifdef/#ifndef在控制编译的过程中还扮演重要角色。预处理还不能退休，但你一定要计划给它经常放长假。

条款 2：尽量用<iostream>而不用<stdio.h>

是的，scanf和printf很轻巧，很高效，你也早就知道怎么用它们，这我承认。但尽管他们很有用，事实上scanf和printf及其系列还可以做些改进。尤其是，他们不是类型安全的，而且没有扩展性。因为类型安全和扩展性是C++的基石，所以你也要服从这一点。另外，scanf/printf系列函数把要读写的变量和控制读写格式的信息分开来，就象古老的FORTRAN那样。是该向五十年代说诀别的时候了！

不必惊奇，scanf/printf的这些弱点正是操作符>>和<<的强项:

```
int i;
Rational r;// r 是个有理数

...

cin >> i >> r;
cout << i << r;
```

上面的代码要通过编译，>>和<<必须是可以处理Rational类型对象的重载函数(可能要通过隐式类型转换)。如果没有实现这样的函数，就会出错(处理int不用这样做，因为它是标准用法)。另外，编译器自己可以根据不同的变量类型选择操作符的不同形式，所以不必劳你去指定第一个要读写的对象是int而第二个是Rational。

另外，在传递读和写的对象时采用的语法形式相同，所以不必象scanf那样死记一些规定，比如如果没有得到指针，必须加上地址符，而如果已经得到了指针，又要确定不要加上地址符。这些完全可以交给C++编译器去做。编译器没别的什么事好做的，而你却不一样。最后要注意的是，象int这样的固定类型和象Rational这样的自定义类型在读写时方式是一样的。而你用sacnf和printf试试看！

你所写的表示有理数的类的代码可能象下面这样：

```
class Rational {
public:
        Rational(int numerator = 0, int denominator = 1);

        ...

private:
        int n, d;// 分子，分母

        friend ostream& operator<<(ostream& s, const Rational& );
};
```

```
ostream& operator<<(ostream& s, const Rational& r)
{
        s<< r.n << '/' << r.d;
        return s;
}
```

上面的代码涉及到operator<<的一些微妙(但很重要)的用法，这在本书其他地方详细讨论。例如：上面的operator<<不是成员函数(条款 19解释了为什么)，而且，传递给operator<<的不是Rational对象，而是定义为const的对象的引用(参见条款 22)。operator>>的声明和实现也类似。

尽管我不大愿意承认，可有些情况下回到那些经过证明而且正确的老路上去还是很有意义的。第一，有些iostream的操作实现起来比相应的C stream效率要低，所以不同的选择会给你的程序有可能(虽然不一定，参见条款M16)带来很大的不同。但请牢记，这不是对所有的iostream而言，只是一些特殊的实现；参见条款M23。第二，在标准化的过程中，iostream库在底层做了很多修改(参见条款 49)，所以对那些要求最大可移植性的应用程序来说，会发现不同的厂商遵循标准的程度也不同。第三，iostream库的类有构造函数而<stdio.h>里的函数没有，在某些涉及到静态对象初始化顺序的时候，如果可以确认不会带来隐患，用标准C库会更简单实用。

iostream 库的类和函数所提供的类型安全和可扩展性的价值远远超过你当初的想象，所以不要仅仅因为你用惯了<stdio.h>而舍弃它。毕竟，转换到 iostream 后，你也不会忘掉<stdio.h>。

顺便说一句，本条款的标题没有打印错；我确实说的是<iostream>而非<iostream.h>。从技术上说，其实没有<iostream.h>这样的东西——标准化委员会在简化非C标准头文件时用<iostream>取代了它。他们这样做的原因在条款 49进行了解释。还必须知道的是，如果编译器同时支持 <iostream>和<iostream.h>，那头文件名的使用会很微妙。例如，如果使用了#include <iostream>, 得到的是置于名字空间std(见条款 28)下的iostream库的元素；如果使用#include <iostream.h>，得到的是置于全局空间的同样的元素。在全局空间获取元素会导致名字冲突，而设计名字空间的初衷正是用来避免这种名字冲突的发生。还有，打字时<iostream>比<iostream.h>少两个字，这也是很多人用它的原因。:)

条款 3：尽量用 new 和 delete 而不用 malloc 和 free

malloc 和 free(及其变体)会产生问题的原因在于它们太简单：他们不知道构造函数和析构函数。

假设用两种方法给一个包含 10 个 string 对象的数组分配空间，一个用 malloc，另一个用 new：

```
string *stringarray1 =
static_cast<string*>(malloc(10 * sizeof(string)));

string *stringarray2 = new string[10];
```

其结果是，stringarray1 确实指向的是可以容纳 10 个 string 对象的足够空间，但内存里并没有创建这些对象。而且，如果你不从这种晦涩的语法怪圈(详见条款 m4 和 m8 的描述)里跳出来的话，你没有办法来初始化数组里的对象。换句话说，stringarray1 其实一点用也没有。相反，stringarray2 指向的是一个包含 10 个完全构造好的 string 对象的数组,每个对象可以在任何读取 string 的操作里安全使用。

假设你想了个怪招对 stringarray1 数组里的对象进行了初始化，那么在你后面的程序里你一定会这么做：

```
free(stringarray1);
delete [] stringarray2;// 参见条款 5：这里为什么要加上个"[]"
```

调用 free 将会释放 stringarray1 指向的内存，但内存里的 string 对象不会调用析构函数。如果 string 对象象一般情况那样，自己已经分配了内存，那这些内存将会全部丢失。相反，当对 stringarray2 调用 delete 时，数组里的每个对象都会在内存释放前调用析构函数。

既然 new 和 delete 可以这么有效地与构造函数和析构函数交互，选用它们是显然的。

把 new 和 delete 与 malloc 和 free 混在一起用也是个坏想法。对一个用 new 获取来的指针调用 free，或者对一个用 malloc 获取来的指针调用 delete，其后果是不可预测的。大家都知道“不可预测”的意思：它可能在开发阶段工作良好，在测试阶段工作良好，但也可能会最后在你最重要的客户的脸上爆炸。

new/delete 和 malloc/free 的不兼容性常常会导致一些严重的复杂性问题。举个例子，<string.h>里通常有个 strdup 函数，它得到一个 char*字符串然后返回其拷贝：

```
char * strdup(const char *ps);// 返回 ps 所指的拷贝
```

在有些地方，c 和 c++用的是同一个 strdup 版本，所以函数内部是用 malloc 分配内存。这样的话，一些不知情的 c++程序员会在调用 strdup 后忽视了必须对 strdup 返回的指针进行 free 操作。为了防止这一情况，有些地方会专门为 c++重写 strdup，并在函数内部调用了

new，这就要求其调用者记得最后用 delete。你可以想象，这会导致多么严重的移植性问题，因为代码中 strdup 以不同的形式在不同的地方之间颠来倒去。

c++程序员和 c 程序员一样对代码重用十分感兴趣。大家都知道，有大量基于 malloc 和 free 写成的代码构成的 c 库都非常值得重用。在利用这些库时，最好是你不用负责去 free 掉由库自己 malloc 的内存，并且/或者，你不用去 malloc 库自己会 free 掉的内存，这样就太好了。其实，在 c++程序里使用 malloc 和 free 没有错，只要保证用 malloc 得到的指针用 free，或者用 new 得到的指针最后用 delete 来操作就可以了。千万别马虎地把 new 和 free 或 malloc 和 delete 混起来用，那只会自找麻烦。

既然 malloc 和 free 对构造函数和析构函数一无所知，把 malloc/free 和 new/delete 混起来用又象嘈杂拥挤的晚会那样难以控制，那么，你最好就什么时候都一心一意地使用 new 和 delete 吧。

条款 4：尽量使用 c++风格的注释

旧的 c 注释语法在 c++里还可以用，c++新发明的行尾注释语法也有其过人之处。例如下面这种情形：

```
if ( a > b ) {
        // int temp = a;        // swap a and b
        // a = b;
        // b = temp;
}
```

假设你出于某种原因要注释掉这个代码块。从软件工程的角度看，写这段代码的程序员也做得很好，他最初的代码里也写了一个注释，以解释代码在做什么。用 c++形式的句法来注释掉这个程序块时，嵌在里面的最初的注释不受影响，但如果选择 c 风格的注释就会发生严重的错误：

```
if ( a > b ) {
        /*      int temp = a;   /* swap a and b */
                a = b;
                b = temp;
        */
}
```

请注意嵌在代码块里的注释是怎么无意间使本来想注释掉整个代码块的注释提前结束的。

c 风格的注释当然还有它存在的价值。例如，它们在 c 和 c++编译器都要处理的头文件中是无法替代的。尽管如此，只要有可能，你最好尽量用 c++风格的注释。

值得指出的是，有些老的专门为 c 写的预处理程序不知道处理 c++风格的注释，所以象下面这种情形时，事情就不会象预想的那样：

```
#define light_speedp 3e8       // m/sec (in a vacuum)
```

对于不熟悉c++的预处理程序来说，行尾的注释竟然成为了宏的一部分！当然，正象条款 1所说的那样，你无论如何也不会用预处理来定义常量的。

第二章 内存管理

c++中涉及到的内存的管理问题可以归结为两方面：正确地得到它和有效地使用它。好的程序员会理解这两个问题为什么要以这样的顺序列出。因为执行得再快、体积再小的程序如果它不按你所想象地那样去执行，那也一点用处都没有。“正确地得到”的意思是正确地调用内存分配和释放程序；而“有效地使用”是指写特定版本的内存分配和释放程序。这里，“正确地得到”显得更重要一些。

然而说到正确性，c++其实从 c 继承了一个很严重的头疼病，那就是内存泄露隐患。虚拟内存是个很好的发明，但虚拟内存也是有限的，并不是每个人都可以最先抢到它。

在 c 中，只要用 malloc 分配的内存没有用 free 返回，就会产生内存泄露。在 c++中，肇事者的名字换成了 new 和 delete，但情况基本上是一样的。当然，因为有了析构函数的出现，情况稍有改善，因为析构函数为所有将被摧毁的对象提供了一个方便的调用 delete 的场所。但这同时又带来了更多的烦恼，因为 new 和 delete 是隐式地调用构造函数和析构函数的。而且，因为可以在类内和类外自定义 new 和 delete 操作符，这又带来了复杂性，增加了出错的机会。下面的条款(还有条款 m8)将告诉你如何避免产生那些普遍发生的问题。

条款 5：对应的 new 和 delete 要采用相同的形式

下面的语句有什么错？

```
string *stringarray = new string[100];

...

delete stringarray;
```

一切好象都井然有序——一个 new 对应着一个 delete——然而却隐藏着很大的错误：程序的运行情况将是不可预测的。至少，stringarray 指向的 100 个 string 对象中的 99 个不会被正确地摧毁，因为他们的析构函数永远不会被调用。

用new的时候会发生两件事。首先，内存被分配(通过operator new 函数，详见条款 7-10 和条款m8)，然后，为被分配的内存调用一个或多个构造函数。用delete的时候，也有两件事发生：首先，为将被释放的内存调用一个或多个析构函数，然后，释放内存(通过operator delete 函数，详见条款 8和m8)。对于 delete来说会有这样一个重要的问题：内存中有多少个对象要被删除？答案决定了将有多少个析构函数会被调用。

这个问题简单来说就是：要被删除的指针指向的是单个对象呢，还是对象数组？这只有你来告诉 delete。如果你在用 delete 时没用括号，delete 就会认为指向的是单个对象，否则，它就会认为指向的是一个数组：

```
string *stringptr1 = new string;
string *stringptr2 = new string[100];

...

delete stringptr1;//  删除一个对象
delete [] stringptr2;//  删除对象数组
```

如果你在 stringptr1 前加了"[]"会怎样呢？答案是：那将是不可预测的；如果你没在 stringptr2 前没加上"[]"又会怎样呢？答案也是：不可预测。而且对于象 int 这样的固定类型来说，结果也是不可预测的，即使这样的类型没有析构函数。所以，解决这类问题的规则很简单：如果你调用 new 时用了[]，调用 delete 时也要用[]。如果调用 new 时没有用[]，那调用 delete 时也不要用[]。

在写一个包含指针数据成员，并且提供多个构造函数的类时，牢记这一规则尤其重要。因为这样的话，你就必须在所有初始化指针成员的构造函数里采用相同的new的形式。否则，析构函数里将采用什么形式的delete呢？关于这一话题的进一步阐述，参见条款 11。

这个规则对喜欢用 typedef 的人来说也很重要，因为写 typedef 的程序员必须告诉别人，用 new 创建了一个 typedef 定义的类型的对象后，该用什么形式的 delete 来删除。举例如下：

```
typedef string addresslines[4];          //一个人的地址，共 4 行，每行一个 string
                                         //因为 addresslines 是个数组，使用 new:
string *pal = new addresslines;          // 注意"new addresslines"返回 string*，和
                                         // "new string[4]"返回的一样
```

delete 时必须以数组形式与之对应：

```
delete pal;// 错误!
delete [] pal;// 正确
```

为了避免混乱，最好杜绝对数组类型用typedefs。这其实很容易，因为标准c++库(见条款49)包含有stirng和vector模板，使用他们将会使对数组的需求减少到几乎零。举例来说，addresslines可以定义为一个字符串(string)的向量(vector)，即addresslines可定义为vector<string>类型。

条款 6：析构函数里对指针成员调用 delete

大多数情况下，执行动态内存分配的的类都在构造函数里用 new 分配内存，然后在析构函数里用 delete 释放内存。最初写这个类的时候当然不难做，你会记得最后对在所有构造函数里分配了内存的所有成员使用 delete。

然而，这个类经过维护、升级后，情况就会变得困难了，因为对类的代码进行修改的程序员不一定就是最早写这个类的人。而增加一个指针成员意味着几乎都要进行下面的工作：

·在每个构造函数里对指针进行初始化。对于一些构造函数，如果没有内存要分配给指针的话，指针要被初始化为 0(即空指针)。

·删除现有的内存，通过赋值操作符分配给指针新的内存。

·在析构函数里删除指针。

如果在构造函数里忘了初始化某个指针，或者在赋值操作的过程中忘了处理它，问题会出现得很快，很明显，所以在实践中这两个问题不会那么折磨你。但是，如果在析构函数里没有删除指针，它不会表现出很明显的外部症状。相反，它可能只是表现为一点微小的内存泄露，并且不断增长，最后吞噬了你的地址空间，导致程序夭折。因为这种情况经常不那么引人注意，所以每增加一个指针成员到类里时一定要记清楚。

另外，删除空指针是安全的(因为它什么也没做)。所以，在写构造函数，赋值操作符，或其他成员函数时，类的每个指针成员要么指向有效的内存，要么就指向空，那在你的析构函数里你就可以只用简单地 delete 掉他们，而不用担心他们是不是被 new 过。

当然对本条款的使用也不要绝对。例如，你当然不会用 delete 去删除一个没有用 new 来初始化的指针，而且，就象用智能指针对象时不用劳你去删除一样，你也永远不会去删除一个传递给你的指针。换句话说，除非类成员最初用了 new,否则是不用在析构函数里用 delete 的。

说到智能指针，这里介绍一种避免必须删除指针成员的方法，即把这些成员用智能指针对象来代替，比如 c++标准库里的 auto_ptr。想知道它是如何工作的，看看条款 m9 和 m10。

条款 7：预先准备好内存不够的情况

operator new 在无法完成内存分配请求时会抛出异常(以前的做法一般是返回 0，一些旧一点的编译器还这么做。你愿意的话也可以把你的编译器设置成这样。关于这个话题我将推迟到本条款的结尾处讨论)。大家都知道，处理内存不够所产生的异常真可以算得上是个道德上的行为，但实际做起来又会象刀架在脖子上那样痛苦。所以，你有时会不去管它，也许一直没去管它。但你心里一定还是深深地隐藏着一种罪恶感：万一 new 真的产生了异常怎么办？

你会很自然地想到处理这种情况的一种方法，即回到以前的老路上去，使用预处理。例如，c 的一种常用的做法是，定义一个类型无关的宏来分配内存并检查分配是否成功。对于 c++来说，这个宏看起来可能象这样：

```
#define new(ptr, type)                          \
try { (ptr) = new type; }           \
catch (std::bad_alloc&) { assert(0); }
```

（“慢！std::bad_alloc是做什么的？”你会问。bad_alloc是operator new不能满足内存分配请求时抛出的异常类型，std是bad_alloc所在的名字空间(见条款 28)的名称。“好！”你会继续问，“assert又有什么用？”如果你看看标准c头文件<assert.h>(或与它相等价的用到了名字空间的版本<cassert>，见条款 49)，就会发现assert是个宏。这个宏检查传给它的表达式是否非零，如果不是非零值，就会发出一条出错信息并调用abort。assert只是在没定义标准宏ndebug的时候，即在调试状态下才这么做。在产品发布状态下，即定义了ndebug的时候，assert什么也不做，相当于一条空语句。所以你只能在调试时才能检查断言(assertion)）。

new 宏不但有着上面所说的通病，即用 assert 去检查可能发生在已发布程序里的状态(然而任何时候都可能发生内存不够的情况)，同时，它还在 c++里有另外一个缺陷：它没有考虑到 new 有各种各样的使用方式。例如，想创建类型 t 对象，一般有三种常见的语法形式，你必须对每种形式可能产生的异常都要进行处理：

```
new t;
new t(constructor arguments);
new t[size];
```

这里对问题大大进行了简化，因为有人还会自定义(重载)operator new，所以程序里会包含任意个使用 new 的语法形式。

那么，怎么办？如果想用一个很简单的出错处理方法，可以这么做：当内存分配请求不能满足时，调用你预先指定的一个出错处理函数。这个方法基于一个常规，即当operator new不能满足请求时，会在抛出异常之前调用客户指定的一个出错处理函数——一般称为new-handler函数。(operator new实际工作起来要复杂一些，详见条款 8)

指定出错处理函数时要用到 set_new_handler 函数，它在头文件<new>里大致是象下面这样定义的：

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler p) throw();
```

可以看到，new_handler 是一个自定义的函数指针类型，它指向一个没有输入参数也没有返回值的函数。set_new_handler 则是一个输入并返回 new_handler 类型的函数。

set_new_handler 的输入参数是 operator new 分配内存失败时要调用的出错处理函数的指针，返回值是 set_new_handler 没调用之前就已经在起作用的旧的出错处理函数的指针。

可以象下面这样使用 set_new_handler：

```
// function to call if operator new can't allocate enough memory
void nomorememory()
{
        cerr << "unable to satisfy request for memory\n";
        abort();
}


int main()
{
        set_new_handler(nomorememory);
        int *pbigdataarray = new int[100000000];

        ...

}
```

假如 operator new 不能为 100,000,000 个整数分配空间，nomorememory 将会被调用，程序发出一条出错信息后终止。这就比简单地让系统内核产生错误信息来结束程序要好。(顺便考虑一下，假如 cerr 在写错误信息的过程中要动态分配内存，那将会发生什么...)

operator new不能满足内存分配请求时，new-handler函数不只调用一次，而是不断重复，直至找到足够的内存。实现重复调用的代码在条款 8里可以看到，这里我用描述性的的语言来说明：一个设计得好的new-handler函数必须实现下面功能中的一种。

·产生更多的可用内存。这将使operator new下一次分配内存的尝试有可能获得成功。实施这一策略的一个方法是：在程序启动时分配一个大的内存块，然后在第一次调用new-handler时释放。释放时伴随着一些对用户的警告信息，如内存数量太少，下次请求可能会失败，除非又有更多的可用空间。

·安装另一个不同的new-handler函数。如果当前的new-handler函数不能产生更多的可用内存，可能它会知道另一个new-handler函数可以提供更多的资源。这样的话，当前的new-handler可以安装另一个new-handler来取代它(通过调用set_new_handler)。下一次operator new调用new-handler时，会使用最近安装的那个。(这一策略的另一个变通办法是让new-handler可以改变它自己的运行行为，那么下次调用时，它将做不同的事。方法是使new-handler可以修改那些影响它自身行为的静态或全局数据。)

·卸除new-handler。也就是传递空指针给set_new_handler。没有安装new-handler，operator new分配内存不成功时就会抛出一个标准的std::bad_alloc类型的异常。

·抛出std::bad_alloc或从std::bad_alloc继承的其他类型的异常。这样的异常不会被operator new捕捉，所以它们会被送到最初进行内存请求的地方。(抛出别的不同类型的异常会违反operator new异常规范。规范中的缺省行为是调用abort，所以new-handler要抛出一个异常时，一定要确信它是从std::bad_alloc继承来的。想更多地了解异常规范，参见条款m14。)

·没有返回。典型做法是调用abort或exit。abort/exit可以在标准c库中找到(还有标准c++库，参见条款 49)。

上面的选择给了你实现 new-handler 函数极大的灵活性。

处理内存分配失败的情况时采取什么方法，取决于要分配的对象的类：

```
class x {
public:
        static void
        outofmemory();

        ...

};

class y {
public:
        static void outofmemory();

        ...

};

x* p1 = new x;      // 若分配成功，调用 x::outofmemory
y* p2 = new y;      // 若分配不成功，调用 y::outofmemory
```

c++不支持专门针对于类的 new-handler 函数，而且也不需要。你可以自己来实现它，只要在每个类中提供自己版本的 set_new_handler 和 operator new。类的 set_new_handler 可以为类指定 new-handler(就象标准的 set_new_handler 指定全局 new-handler 一样)。类的 operator new 则保证为类的对象分配内存时用类的 new-handler 取代全局 new-handler。

假设处理类 x 内存分配失败的情况。因为 operator new 对类型 x 的对象分配内存失败时，每次都必须调用出错处理函数，所以要在类里声明一个 new_handler 类型的静态成员。那么类 x 看起来会象这样：

```
class x {
public:
```

```
        static new_handler set_new_handler(new_handler p);
        static void * operator new(size_t size);

private:
        static new_handler currenthandler;
};
```

类的静态成员必须在类外定义。因为想借用静态对象的缺省初始化值 0，所以定义 x::currenthandler 时没有去初始化。

```
new_handler x::currenthandler;          //缺省设置 currenthandler 为 0(即 null)
```

类 x 中的 set_new_handler 函数会保存传给它的任何指针，并返回在调用它之前所保存的任何指针。这正是标准版本的 set_new_handler 所做的：

```
new_handler x::set_new_handler(new_handler p)
{
        new_handler oldhandler = currenthandler;
        currenthandler = p;
        return oldhandler;
}
```

最后看看 x 的 operator new 所做的：

1. 调用标准 set_new_handler 函数，输入参数为 x 的出错处理函数。这使得 x 的 new-handler 函数成为全局 new-handler 函数。注意下面的代码中，用了"::"符号显式地引用 std 空间(标准 set_new_handler 函数就存在于 std 空间)。

2. 调用全局 operator new 分配内存。如果第一次分配失败，全局 operator new 会调用 x 的 new-handler，因为它刚刚(见 1.)被安装成为全局 new-handler。如果全局 operator new 最终未能分配到内存，它抛出 std::bad_alloc 异常，x 的 operator new 会捕捉到它。x 的 operator new 然后恢复最初被取代的全局 new-handler 函数，最后以抛出异常返回。

3. 假设全局 operator new 为类型 x 的对象分配内存成功，, x 的 operator new 会再次调用标准 set_new_handler 来恢复最初的全局出错处理函数。最后返回分配成功的内存的指针。c++是这么做的：

```
void * x::operator new(size_t size)
{
        new_handler globalhandler =             //  安装 x 的 new_handler

                std::set_new_handler(currenthandler);

void *memory;

try {       //  尝试分配内存
```

```
        memory = ::operator new(size);
}


catch (std::bad_alloc&) {       // 恢复旧的 new_handler
        std::set_new_handler(globalhandler);
        throw;      // 抛出异常
}
        std::set_new_handler(globalhandler);   // 恢复旧的 new_handler
        return memory;
}
```

如果你对上面重复调用 std::set_new_handler 看不顺眼，可以参见条款 m9 来除去它们。

使用类 x 的内存分配处理功能时大致如下：

```
void nomorememory();// x 的对象分配内存失败时调用的 new_handler 函数的声明

x::set_new_handler(nomorememory);
                        // 把 nomorememory 设置为 x 的
                        // new-handling 函数

x *px1 = new x;
                        // 如内存分配失败，
                        // 调用 nomorememory
string *ps = new string;
                        // 如内存分配失败，调用全局 new-handling 函数

x::set_new_handler(0);
                        // 设 x 的 new-handling 函数为空

x *px2 = new x;
                        // 如内存分配失败，立即抛出异常
                        // (类 x 没有 new-handling 函数)
```

你会注意到，处理以上类似情况，如果不考虑类的话，实现代码是一样的，这就很自然地想到在别的地方也能重用它们。正如条款 41所说明的，继承和模板可以用来设计可重用代码。在这里，我们把两种方法结合起来使用，从而满足了你的要求。

你只要创建一个“混合风格”(mixin-style)的基类，这种基类允许子类继承它某一特定的功能——这里指的是建立一个类的 new-handler 的功能。之所以设计一个基类，是为了让所有的子类可以继承 set_new_handler 和 operator new 功能，而设计模板是为了使每个子类有不同的 currenthandler 数据成员。这听起来很复杂，不过你会看到代码其实很熟悉。区别只不过是它现在可以被任何类重用了。

```
template<class t>   // 提供类 set_new_handler 支持的
class newhandlersupport {     // 混合风格”的基类
public:
        static new_handler set_new_handler(new_handler p);
        static void * operator new(size_t size);

private:
        static new_handler currenthandler;
};

template<class t>
new_handler newhandlersupport<t>::set_new_handler(new_handler p)
{
        new_handler oldhandler = currenthandler;
        currenthandler = p;
        return oldhandler;
}

template<class t>
void * newhandlersupport<t>::operator new(size_t size)
{
        new_handler globalhandler =
                std::set_new_handler(currenthandler);
        void *memory;
        try {
                memory = ::operator new(size);
        }
        catch (std::bad_alloc&) {
                std::set_new_handler(globalhandler);
                throw;
        }

        std::set_new_handler(globalhandler);
        return memory;
}
// this sets each currenthandler to 0

template<class t>
new_handler newhandlersupport<t>::currenthandler;
```

有了这个模板类，对类 x 加上 set_new_handler 功能就很简单了：只要让 x 从 newhandlersupport<x>继承：

```
// note inheritance from mixin base class template. (see
// my article on counting objects for information on why
```

```
// private inheritance might be preferable here.)
class x: public newhandlersupport<x> {

...                     // as before, but no declarations for
};                      // set_new_handler or operator new
```

使用 x 的时候依然不用理会它幕后在做些什么；老代码依然工作。这很好！那些你常不去理会的东西往往是最可信赖的。

使用set_new_handler是处理内存不够情况下一种方便，简单的方法。这比把每个new都包装在try模块里当然好多了。而且，newhandlersupport这样的模板使得向任何类增加一个特定的new-handler变得更简单。“混合风格”的继承不可避免地将话题引入到多继承上去，在转到这个话题前，你一定要先阅读条款 43。

1993 年前，c++一直要求在内存分配失败时operator new要返回 0，现在则是要求operator new抛出std::bad_alloc异常。很多c++程序是在编译器开始支持新规范前写的。c++标准委员会不想放弃那些已有的遵循返回 0 规范的代码，所以他们提供了另外形式的operator new(以及operator new[]——见条款 8)以继续提供返回 0 功能。这些形式被称为“无抛出”，因为他们没用过一个throw，而是在使用new的入口点采用了nothrow对象：

```
class widget { ... };

widget *pw1 = new widget;//  分配失败抛出 std::bad_alloc if

if (pw1 == 0) ...       //  这个检查一定失败
        widget *pw2 = new (nothrow) widget;    //  若分配失败返回 0

if (pw2 == 0) ...       //  这个检查可能会成功
```

不管是用“正规”(即抛出异常)形式的 new 还是“无抛出”形式的 new，重要的是你必须为内存分配失败做好准备。最简单的方法是使用 set_new_handler，因为它对两种形式都有用。

条款 8: 写 operator new 和 operator delete 时要遵循常规

自己重写 operator new 时(条款 10 解释了为什么有时要重写它)，很重要的一点是函数提供的行为要和系统缺省的 operator new 一致。实际做起来也就是：要有正确的返回值；可用内存不够时要调用出错处理函数(见条款 7)；处理好 0 字节内存请求的情况。此外，还要避免不小心隐藏了标准形式的 new，不过这是条款 9 的话题。

有关返回值的部分很简单。如果内存分配请求成功，就返回指向内存的指针；如果失败，则遵循条款 7 的规定抛出一个 std::bad_alloc 类型的异常。

但事情也不是那么简单。因为 operator new 实际上会不只一次地尝试着去分配内存，它要在每次失败后调用出错处理函数，还期望出错处理函数能想办法释放别处的内存。只有在指向出错处理函数的指针为空的情况下，operator new 才抛出异常。

另外，c++标准要求，即使在请求分配 0 字节内存时，operator new 也要返回一个合法指针。(实际上，这个听起来怪怪的要求确实给 c++语言其它地方带来了简便)

这样，非类成员形式的 operator new 的伪代码看起来会象下面这样：

```
void * operator new(size_t size)        // operator new 还可能有其它参数
{

  if (size == 0) {                      // 处理 0 字节请求时，
    size = 1;                           // 把它当作 1 个字节请求来处理
  }
  while (1) {
    分配 size 字节内存;

    if (分配成功)
      return (指向内存的指针);

    // 分配不成功，找出当前出错处理函数
    new_handler globalhandler = set_new_handler(0);
    set_new_handler(globalhandler);

    if (globalhandler) (*globalhandler)();
    else throw std::bad_alloc();
  }
}
```

处理零字节请求的技巧在于把它作为请求一个字节来处理。这看起来也很怪，但简单，合法，有效。而且，你又会多久遇到一次零字节请求的情况呢？

你又会奇怪上面的伪代码中为什么把出错处理函数置为 0 后又立即恢复。这是因为没有办法可以直接得到出错处理函数的指针，所以必须通过调用 set_new_handler 来找到。办法很笨但也有效。

条款 7 提到 operator new 内部包含一个无限循环，上面的代码清楚地说明了这一点——while (1)将导致无限循环。跳出循环的唯一办法是内存分配成功或出错处理函数完成了条款 7 所描述的事件中的一种：得到了更多的可用内存；安装了一个新的 new-handler(出错处理函数)；卸除了 new-handler；抛出了一个 std::bad_alloc 或其派生类型的异常；或者返回失败。现在明白了为什么 new-handler 必须做这些工作中的一件。如果不做，operator new 里面的循环就不会结束。

很多人没有认识到的一点是 operator new 经常会被子类继承。这会导致某些复杂性。上面的伪代码中，函数会去分配 size 字节的内存(除非 size 为 0)。size 很重要，因为它是传递给函数的参数。但是大多数针对类所写的 operator new(包括条款 10 中的那种)都是只为特定的类设计的，不是为所有的类，也不是为它所有的子类设计的。这意味着，对于一个类 x 的 operator new 来说，函数内部的行为在涉及到对象的大小时，都是精确的 sizeof(x)：不会大也不会小。但由于存在继承，基类中的 operator new 可能会被调用去为一个子类对象分配内存：

```
class base {
public:
  static void * operator new(size_t size);
  ...
};

class derived: public base       // derived 类没有声明 operator new
{ ... };                         //

derived *p = new derived;        //  调用 base::operator new
```

如果 base 类的 operator new 不想费功夫专门去处理这种情况——这种情况出现的可能性不大——那最简单的办法是把这个“错误”数量的内存分配请求转给标准 operator new 来处理，象下面这样：

```
void * base::operator new(size_t size)
{
  if (size != sizeof(base))            //  如果数量“错误”，让标准 operator new
    return ::operator new(size);       //  去处理这个请求
                                       //

  ...                                  //  否则处理这个请求
}
```

“停!”我听见你在叫，“你忘了检查一种虽然不合理但是有可能出现的一种情况——size 有可能为零!”是的，我没检查，但拜托下次再叫出声的时候不要这么文绉绉的。:)但实际上检查还是做了，只不过融合到 size != sizeof(base)语句中了。c++标准很怪异，其中之一就是规定所以独立的(freestanding)类的大小都是非零值。所以 sizeof(base)永远不可能是零(即使

base 类没有成员)，如果 size 为零，请求会转到::operator new，由它来以一种合理的方式对请求进行处理。(有趣的是，如果 base 不是独立的类，sizeof(base)有可能是零，详细说明参见"my article on counting objects")。

如果想控制基于类的数组的内存分配，必须实现 operator new 的数组形式——operator new[](这个函数常被称为“数组 new”，因为想不出"operator new[]")该怎么发音)。写 operator new[]时，要记住你面对的是“原始”内存，不能对数组里还不存在的对象进行任何操作。实际上，你甚至还不知道数组里有多少个对象，因为不知道每个对象有多大。基类的 operator new[]会通过继承的方式被用来为子类对象的数组分配内存，而子类对象往往比基类要大。所以，不能想当然认为 base::operator new[]里的每个对象的大小都是 sizeof(base)，也就是说，数组里对象的数量不一定就是(请求字节数)/sizeof(base)。关于 operator new[]的详细介绍参见条款 m8。

重写operator new(和operator new[])时所有要遵循的常规就这些。对于operator delete(以及它的伙伴operator delete[])，情况更简单。所要记住的只是，c++保证删除空指针永远是安全的，所以你要充分地应用这一保证。下面是非类成员形式的operator delete的伪代码：

```
void operator delete(void *rawmemory)
{
  if (rawmemory == 0) return;     file://如/果指针为空，返回
                              //

  释放 rawmemory 指向的内存;

  return;
}
```

这个函数的类成员版本也简单，只是还必须检查被删除的对象的大小。假设类的 operator new 将“错误”大小的分配请求转给::operator new，那么也必须将“错误”大小的删除请求转给::operator delete：

```
class base {                        // 和前面一样，只是这里声明了
public:                            // operator delete
  static void * operator new(size_t size);
  static void operator delete(void *rawmemory, size_t size);
  ...
};

void base::operator delete(void *rawmemory, size_t size)
{
  if (rawmemory == 0) return;       // 检查空指针

  if (size != sizeof(base)) {       // 如果 size"错误"，
    ::operator delete(rawmemory);   // 让标准 operator 来处理请求
    return;
  }
```

```
  释放指向 rawmemory 的内存;

  return;
}
```

可见，有关 operator new 和 operator delete(以及他们的数组形式)的规定不是那么麻烦，重要的是必须遵守它。只要内存分配程序支持 new-handler 函数并正确地处理了零内存请求，就差不多了；如果内存释放程序又处理了空指针，那就没其他什么要做的了。至于在类成员版本的函数里增加继承支持，那将很快就可以完成。

条款 9: 避免隐藏标准形式的 new

因为内部范围声明的名称会隐藏掉外部范围的相同的名称，所以对于分别在类的内部和全局声明的两个相同名字的函数 f 来说，类的成员函数会隐藏掉全局函数：

```
void f();                       // 全局函数

class x {
public:
  void f();                     // 成员函数
};

x x;

f();                            // 调用 f

x.f();                          // 调用 x::f
```

这不会令人惊讶，也不会导致混淆，因为调用全局函数和成员函数时总是采用不同的语法形式。然而如果你在类里增加了一个带多个参数的 operator new 函数，结果就有可能令人大吃一惊。

```
class x {
public:
  void f();

  // operator new 的参数指定一个
  // new-hander(new 的出错处理)函数
  static void * operator new(size_t size, new_handler p);
};

void specialerrorhandler();         // 定义在别的地方

x *px1 =
  new (specialerrorhandler) x;      // 调用 x::operator new

x *px2 = new x;                     // 错误!
```

在类里定义了一个称为“operator new”的函数后，会不经意地阻止了对标准 new 的访问。条款 50 解释了为什么会这样，这里我们更关心的是如何想个办法避免这个问题。

一个办法是在类里写一个支持标准 new 调用方式的 operator new，它和标准 new 做同样的事。这可以用一个高效的内联函数来封装实现。

```
class x {
public:
  void f();

  static void * operator new(size_t size, new_handler p);

  static void * operator new(size_t size)
  { return ::operator new(size); }
};

x *px1 =
  new (specialerrorhandler) x;      // 调用 x::operator
                                    // new(size_t, new_handler)

x* px2 = new x;                     // 调用 x::operator
                                    // new(size_t)
```

另一种方法是为每一个增加到 operator new 的参数提供缺省值(见条款 24)：

```
class x {
public:
  void f();

  static
    void * operator new(size_t size,              // p 缺省值为 0
                        new_handler p = 0);       //
};

x *px1 = new (specialerrorhandler) x;             // 正确

x* px2 = new x;                                   // 也正确
```

无论哪种方法，如果以后想对“标准”形式的 new 定制新的功能，只需要重写这个函数。调用者重新编译链接后就可以使用新功能了。

条款 10: 如果写了 operator new 就要同时写 operator delete

让我们回过头去看看这样一个基本问题：为什么有必要写自己的 operator new 和 operator delete？

答案通常是：为了效率。缺省的 operator new 和 operator delete 具有非常好的通用性，它的这种灵活性也使得在某些特定的场合下，可以进一步改善它的性能。尤其在那些需要动态分配大量的但很小的对象的应用程序里，情况更是如此。

例如有这样一个表示飞机的类：类 airplane 只包含一个指针，它指向的是飞机对象的实际描述(此技术在条款 34 进行说明)：

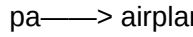
```
class airplanerep { ... };      // 表示一个飞机对象
                                //
class airplane {
public:
  ...
private:
  airplanerep *rep;             // 指向实际描述
};
```

一个 airplane 对象并不大，它只包含一个指针（正如条款 14 和 m24 所说明的，如果 airplane 类声明了虚函数，会隐式包含第二个指针）。但当调用 operator new 来分配一个 airplane 对象时，得到的内存可能要比存储这个指针（或一对指针）所需要的要多。之所以会产生这种看起来很奇怪的行为，在于 operator new 和 operator delete 之间需要互相传递信息。

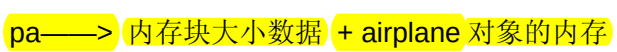
因为缺省版本的 operator new 是一种通用型的内存分配器，它必须可以分配任意大小的内存块。同样，operator delete 也要可以释放任意大小的内存块。operator delete 想弄清它要释放的内存有多大，就必须知道当初 operator new 分配的内存有多大。有一种常用的方法可以让 operator new 来告诉 operator delete 当初分配的内存大小是多少，就是在它所返回的内存里预先附带一些额外信息，用来指明被分配的内存块的大小。也就是说，当你写了下面的语句，

```
airplane *pa = new airplane;
```

你不会得到一块看起来象这样的内存块：



而是得到象这样的内存块：



对于象 airplane 这样很小的对象来说，这些额外的数据信息会使得动态分配对象时所需要的的内存的大小翻番（特别是类里没有虚拟函数的时候）。

如果软件运行在一个内存很宝贵的环境中，就承受不起这种奢侈的内存分配方案了。为 airplane 类专门写一个 operator new，就可以利用每个 airplane 的大小都相等的特点，不必在每个分配的内存块上加上附带信息了。

具体来说，有这样一个方法来实现你的自定义的 operator new：先让缺省 operator new 分配一些大块的原始内存，每块的大小都足以容纳很多个 airplane 对象。airplane 对象的内存块就取自这些大的内存块。当前没被使用的内存块被组织成链表——称为自由链表——以备未来 airplane 使用。听起来好象每个对象都要承担一个 next 域的开销（用于支持链表），但不会：rep 域的空间也被用来存储 next 指针（因为只是作为 airplane 对象来使用的内存块才需要 rep 指针；同样，只有没作为 airplane 对象使用的内存块才需要 next 指针），这可以用 union 来实现。

具体实现时，就要修改 airplane 的定义，从而支持自定义的内存管理。可以这么做：

```
class airplane {          // 修改后的类 — 支持自定义的内存管理
public:                   //

  static void * operator new(size_t size);

  ...

private:
  union {
    airplanerep *rep;      // 用于被使用的对象
    airplane *next;        // 用于没被使用的（在自由链表中）对象
  };

  // 类的常量，指定一个大的内存块中放多少个
  // airplane 对象，在后面初始化
  static const int block_size;

  static airplane *headoffreelist;

};
```

上面的代码增加了的几个声明：一个 operator new 函数，一个联合（使得 rep 和 next 域占用同样的空间），一个常量（指定大内存块的大小），一个静态指针（跟踪自由链表的表头）。表头指针声明为静态成员很重要，因为整个类只有一个自由链表，而不是每个 airplane 对象都有。

下面该写 operator new 函数了：

```
void * airplane::operator new(size_t size)
{
  // 把“错误”大小的请求转给::operator new()处理;
  // 详见条款 8
```

```
  if (size != sizeof(airplane))
    return ::operator new(size);

  airplane *p =           // p 指向自由链表的表头
    headoffreelist;       //

  // p 若合法，则将表头移动到它的下一个元素
  //
  if (p)
    headoffreelist = p->next;

  else {
    // 自由链表为空，则分配一个大的内存块，
    // 可以容纳 block_size 个 airplane 对象
    airplane *newblock =
      static_cast<airplane*>(::operator new(block_size *
                                    sizeof(airplane)));

    // 将每个小内存块链接起来形成一个新的自由链表
    // 跳过第 0 个元素，因为它要被返回给 operator new 的调用者
    //
    for (int i = 1; i < block_size-1; ++i)
      newblock[i].next = &newblock[i+1];

    // 用空指针结束链表
    newblock[block_size-1].next = 0;

    // p 设为表的头部，headoffreelist 指向的
    // 内存块紧跟其后
    p = newblock;
    headoffreelist = &newblock[1];
  }

  return p;
}
```

如果你读了条款 8，就会知道在 operator new 不能满足内存分配请求时，会执行一系列与 new-handler 函数和例外有关的例行性动作。上面的代码没有这些步骤，这是因为 operator new 管理的内存都是从::operator new 分配来的。这意味着只有::operator new 失败时，operator new 才会失败。而如果::operator new 失败，它会去执行 new-handler 的动作（可能最后以抛出异常结束），所以不需要 airplane 的 operator new 也去处理。换句话说，其实 new-handler 的动作都还在，你只是没看见，它隐藏在::operator new 里。

有了 operator new，下面要做的就是给出 airplane 的静态数据成员的定义：

```
airplane *airplane::headoffreelist;

const int airplane::block_size = 512;
```

没必要显式地将 headoffreelist 设置为空指针，因为静态成员的初始值都被缺省设为 0。block_size 决定了要从::operator new 获得多大的内存块。

这个版本的 operator new 将会工作得非常好。它为 airplane 对象分配的内存要比缺省 operator new 更少，而且运行得更快，可能会快 2 次方的等级。这没什么奇怪的，通用型的缺省 operator new 必须应付各种大小的内存请求，还要处理内部外部的碎片；而你的 operator new 只用操作链表中的一对指针。抛弃灵活性往往可以很容易地换来速度。

下面我们将讨论 operator delete。还记得 operator delete 吗？本条款就是关于 operator delete 的讨论。但直到现在为止，airplane 类只声明了 operator new，还没声明 operator delete。想想如果写了下面的代码会发生什么:

```
airplane *pa = new airplane;        // 调用
                                    // airplane::operator new
...

delete pa;                          // 调用 ::operator delete
```

读这段代码时，如果你竖起耳朵，会听到飞机撞毁燃烧的声音，还有程序员的哭泣。问题出在 operator new（在 airplane 里定义的那个）返回了一个不带头信息的内存的指针，而 operator delete（缺省的那个）却假设传给它的内存包含头信息。这就是悲剧产生的原因。

这个例子说明了一个普遍原则：operator new 和 operator delete 必须同时写，这样才不会出现不同的假设。如果写了一个自己的内存分配程序，就要同时写一个释放程序。（关于为什么要遵循这条规定的另一个理由，参见 article on counting objects 一文的 the sidebar on placement 章节）

因而，继续设计 airplane 类如下：

```
class airplane {        // 和前面的一样，只不过增加了一个
public:                 // operator delete 的声明
  ...

  static void operator delete(void *deadobject,
                              size_t size);

};

// 传给 operator delete 的是一个内存块，如果
// 其大小正确，就加到自由内存块链表的最前面
//
```

```
void airplane::operator delete(void *deadobject,
                               size_t size)
{
  if (deadobject == 0) return;          // 见条款 8

  if (size != sizeof(airplane))     {   // 见条款 8
    ::operator delete(deadobject);
    return;
  }

  airplane *carcass =
    static_cast<airplane*>(deadobject);

  carcass->next = headoffreelist;
  headoffreelist = carcass;
}
```

因为前面在 operator new 里将“错误”大小的请求转给了全局 operator new（见条款 8），那么这里同样要将“错误”大小的对象交给全局 operator delete 来处理。如果不这样，就会重现你前面费尽心思想避免的那种问题——new 和 delete 句法上的不匹配。

有趣的是，如果要删除的对象是从一个没有虚析构函数的类继承而来的，那传给 operator delete 的 size_t 值有可能不正确。这就是必须保证基类必须要有虚析构函数的原因，此外条款 14 还列出了第二个、理由更充足的原因。这里只要简单地记住，基类如果遗漏了虚拟构函数，operator delete 就有可能工作不正确。

所有一切都很好，但从你皱起的眉头我可以知道你一定在担心内存泄露。有着大量开发经验的你不会没注意到，airplane 的 operator new 调用::operator new 得到了大块内存，但 airplane 的 operator delete 却没有释放它们。内存泄露！内存泄露！我分明听见了警钟在你脑海里回响。

但请仔细听我回答，这里没有内存泄露！

引起内存泄露的原因在于内存分配后指向内存的指针丢失了。如果没有垃圾处理或其他语言之外的机制，这些内存就不会被收回。但上面的设计没有内存泄露，因为它决不会出现内存指针丢失的情况。每个大内存块首先被分成 airplane 大小的小块，然后这些小块被放在自由链表上。当客户调用 airplane::operator new 时，小块被自由链表移除，客户得到指向小块的指针。当客户调用 operator delete 时，小块被放回到自由链表上。采用这种设计，所有的内存块要不被 airplane 对象使用（这种情况下，是由客户来负责避免内存泄露），要不就在自由链表上（这种情况下内存块有指针）。所以说这里没有内存泄露。

然而确实，::operator new 返回的内存块是从来没有被 airplane::operator delete 释放，这个内存块有个名字，叫内存池。但内存泄漏和内存池有一个重要的不同之处。内存泄漏会无限地增长，即使客户循规蹈矩；而内存池的大小决不会超过客户请求内存的最大值。

修改 airplane 的内存管理程序使得::operator new 返回的内存块在不被使用时自动释放并不难，但这里不会这么做，这有两个原因：

第一个原因和你自定义内存管理的初衷有关。你有很多理由去自定义内存管理，最基本的一条是你确认缺省的 operator new 和 operator delete 使用了太多的内存或（并且）运行很慢。和采用内存池策略相比，跟踪和释放那些大内存块所写的每一个额外的字节和每一条额外的语句都会导致软件运行更慢，用的内存更多。在设计性能要求很高的库或程序时，如果你预计内存池的大小会在一个合理的范围之内，那采用内存池的方法再好不过了。

第二个原因和处理一些不合理的程序行为有关。假设 airplane 的内存管理程序被修改了，airplane 的 operator delete 可以释放任何没有对象存在的大块的内存。那看下面的程序：

```
int main()
{
  airplane *pa = new airplane;     // 第一次分配: 得到大块内存，
                                   // 生成自由链表，等

  delete pa;                       // 内存块空;
                                   // 释放它

  pa = new airplane;               // 再次得到大块内存,
                                   // 生成自由链表，等

  delete pa;                       // 内存块再次空，
                                   // 释放

  ...                              // 你有了想法...

  return 0;
}
```

这个糟糕的小程序会比用缺省的 operator new 和 operator delete 写的程序运行得还慢，占用还要多的内存，更不要和用内存池写的程序比了。

当然有办法处理这种不合理的情况，但考虑的特殊情况越多，就越有可能要重新实现内存管理函数，而最后你又会得到什么呢？内存池不能解决所有的内存管理问题，在很多情况下是很适合的。

实际开发中，你会经常要给许多不同的类实现基于内存池的功能。你会想，“一定有什么办法把这种固定大小内存的分配器封装起来，从而可以方便地使用”。是的，有办法。虽然我在这个条款已经唠叨这么长时间了，但还是要简单介绍一下，具体实现留给读者做练习。

下面简单给出了一个 pool 类的最小接口（见条款 18），pool 类的每个对象是某类对象（其大小在 pool 的构造函数里指定）的内存分配器。

```
class pool {
public:
  pool(size_t n);                    // 为大小为 n 的对象创建
                                     // 一个分配器


  void * alloc(size_t n)   ;         // 为一个对象分配足够内存
                                     // 遵循条款 8 的 operator new 常规

  void free(   void *p, size_t n);   // 将 p 所指的内存返回到内存池；
                                     // 遵循条款 8 的 operator delete 常规

  ~pool();                           // 释放内存池中全部内存

};
```

这个类支持 pool 对象的创建，执行分配和释放操作，以及被摧毁。pool 对象被摧毁时，会释放它分配的所有内存。这就是说，现在有办法避免 airplane 的函数里所表现的内存泄漏似的行为了。然而这也意味着，如果 pool 的析构函数调用太快（使用内存池的对象没有全部被摧毁），一些对象就会发现它正在使用的内存猛然间没了。这造成的结果通常是不可预测的。

有了这个 pool 类，即使 java 程序员也可以不费吹灰之力地在 airplane 类里增加自己的内存管理功能：

```
class airplane {
public:

  ...                                // 普通 airplane 功能

  static void * operator new(size_t size);
  static void operator delete(void *p, size_t size);

private:
  airplanerep *rep;                  // 指向实际描述的指针
  static pool mempool;               // airplanes 的内存池

};

inline void * airplane::operator new(size_t size)
{ return mempool.alloc(size); }

inline void airplane::operator delete(void *p,
                                      size_t size)
{ mempool.free(p, size); }
```

```
// 为 airplane 对象创建一个内存池，
// 在类的实现文件里实现
pool airplane::mempool(sizeof(airplane));
```

这个设计比前面的要清楚、干净得多，因为 airplane 类不再和非 airplane 的代码混在一起。union，自由链表头指针，定义原始内存块大小的常量都不见了，它们都隐藏在它们应该呆的地方——pool 类里。让写 pool 的程序员去操心内存管理的细节吧，你的工作只是让 airplane 类正常工作。

现在应该明白了，自定义的内存管理程序可以很好地改善程序的性能，而且它们可以封装在象 pool 这样的类里。但请不要忘记主要的一点，operator new 和 operator delete 需要同时工作，那么你写了 operator new，就也一定要写 operator delete。

第三章 构造函数，析构函数和赋值操作符

几乎所有的类都有一个或多个构造函数，一个析构函数和一个赋值操作符。这没什么奇怪的，因为它们提供的都是一些最基本的功能。构造函数控制对象生成时的基本操作，并保证对象被初始化；析构函数摧毁一个对象并保证它被彻底清除；赋值操作符则给对象一个新的值。在这些函数上出错就会给整个类带来无尽的负面影响，所以一定要保证其正确性。本章我将指导如何用这些函数来搭建一个结构良好的类的主干。

条款 11: 为需要动态分配内存的类声明一个拷贝构造函数和一个赋值操作符

看下面一个表示 string 对象的类：

```
// 一个很简单的 string 类
class string {
public:
  string(const char *value);
  ~string();

  ...                    // 没有拷贝构造函数和 operator=

private:
  char *data;
};

string::string(const char *value)
{
  if (value) {
    data = new char[strlen(value) + 1];
    strcpy(data, value);
  }
  else {
    data = new char[1];
    *data = '\0';
  }
}

inline string::~string() { delete [] data; }
```

请注意这个类里没有声明赋值操作符和拷贝构造函数。这会带来一些不良后果。

如果这样定义两个对象：

```
string a("hello");
string b("world");
```

其结果就会如下所示：

```
a:  data——> "hello\0"
b:  data——> "world\0"
```

对象 a 的内部是一个指向包含字符串"hello"的内存的指针，对象 b 的内部是一个指向包含字符串"world"的内存的指针。如果进行下面的赋值：

```
b = a;
```

因为没有自定义的 operator=可以调用，c++会生成并调用一个缺省的 operator=操作符（见条款 45）。这个缺省的赋值操作符会执行从 a 的成员到 b 的成员的逐个成员的赋值操作，对指针(a.data 和 b.data) 来说就是逐位拷贝。赋值的结果如下所示：

```
a:   data --------> "hello\0"
           /
b:   data --/       "world\0"
```

这种情况下至少有两个问题。第一，b 曾指向的内存永远不会被删除，因而会永远丢失。这是产生内存泄漏的典型例子。第二，现在 a 和 b 包含的指针指向同一个字符串，那么只要其中一个离开了它的生存空间，其析构函数就会删除掉另一个指针还指向的那块内存。

```
string a("hello");                  // 定义并构造 a

{                                   // 开一个新的生存空间
  string b("world");                // 定义并构造 b

  ...

  b = a;                            // 执行 operator=,
                                    // 丢失 b 的内存

}                                   // 离开生存空间, 调用
                                    // b 的析构函数

string c = a;                       // c.data 的值不能确定!
                                    // a.data 已被删除
```

例子中最后一个语句调用了拷贝构造函数，因为它也没有在类中定义，c++以与处理赋值操作符一样的方式生成一个拷贝构造函数并执行相同的动作：对对象里的指针进行逐位拷贝。这会导致同样的问题，但不用担心内存泄漏，因为被初始化的对象还不能指向任何的内存。比如上面代码中的情形，当 c.data 用 a.data 的值来初始化时没有内存泄漏，因为 c.data 没指向任何地方。不过，假如 c 被 a 初始化后，c.data 和 a.data 指向同一个地方，那这个地方会被删除两次：一次在 c 被摧毁时，另一次在 a 被摧毁时。

拷贝构造函数的情况和赋值操作符还有点不同。在传值调用的时候，它会产生问题。当然正如条款 22 所说明的，一般很少对对象进行传值调用，但还是看看下面的例子：

```
void donothing(string localstring) {}

string s = "the truth is out there";
```

donothing(s);

一切好象都很正常。但因为被传递的 localstring 是一个值，它必须从 s 通过（缺省）拷贝构造函数进行初始化。于是 localstring 拥有了一个 s 内的指针的拷贝。当 donothing 结束运行时，localstring 离开了其生存空间，调用析构函数。其结果也将是：s 包含一个指向 localstring 早已删除的内存的指针。

顺便指出，用 delete 去删除一个已经被删除的指针，其结果是不可预测的。所以即使 s 永远也没被使用，当它离开其生存空间时也会带来问题。

解决这类指针混乱问题的方案在于，只要类里有指针时，就要写自己版本的拷贝构造函数和赋值操作符函数。在这些函数里，你可以拷贝那些被指向的数据结构，从而使每个对象都有自己的拷贝；或者你可以采用某种引用计数机制（见条款 m29）去跟踪当前有多少个对象指向某个数据结构。引用计数的方法更复杂，而且它要求构造函数和析构函数内部做更多的工作，但在某些（虽然不是所有）程序里，它会大量节省内存并切实提高速度。

对于有些类，当实现拷贝构造函数和赋值操作符非常麻烦的时候，特别是可以确信程序中不会做拷贝和赋值操作的时候，去实现它们就会相对来说有点得不偿失。前面提到的那个遗漏了拷贝构造函数和赋值操作符的例子固然是一个糟糕的设计，那当现实中去实现它们又不切实际的情况下，该怎么办呢？很简单，照本条款的建议去做：可以只声明这些函数（声明为 private 成员）而不去定义（实现）它们。这就防止了会有人去调用它们，也防止了编译器去生成它们。关于这个俏皮的小技巧的细节，参见条款 27。

关于本条款中所用到的那个 string 类，还要注意一件事。构造函数体内，在两个调用 new 的地方都小心地用了[]，尽管有一个地方实际只需要单个对象。正如条款 5 所说，在配套使用 new 和 delete 时一定要采用相同的形式，所以这里也这么做了。一定要经常注意，当且仅当相应的 new 用了[]的时候，delete 才要用[]。

条款 12: 尽量使用初始化而不要在构造函数里赋值

看这样一个模板，它生成的类使得一个名字和一个 t 类型的对象的指针关联起来。

```
template<class t>
class namedptr {
public:
  namedptr(const string& initname, t *initptr);
  ...

private:
  string name;
  t *ptr;
};
```

（因为有指针成员的对象在进行拷贝和赋值操作时可能会引起指针混乱（见条款 11），namedptr 也必须实现这些函数（见条款 2））

在写 namedptr 构造函数时，必须将参数值传给相应的数据成员。有两种方法来实现。第一种方法是使用成员初始化列表：

```
template<class t>
namedptr<t>::namedptr(const string& initname, t *initptr   )
: name(initname), ptr(initptr)
{}
```

第二种方法是在构造函数体内赋值：

```
template<class t>
namedptr<t>::namedptr(const string& initname, t *initptr)
{
  name = initname;
  ptr = initptr;
}
```

两种方法有重大的不同。

从纯实际应用的角度来看，有些情况下必须用初始化。特别是 const 和引用数据成员只能用初始化，不能被赋值。所以，如果想让 namedptr<t>对象不能改变它的名字或指针成员，就必须遵循条款 21 的建议声明成员为 const：

```
template<class t>
class namedptr {
public:
```

```
  namedptr(const string& initname, t *initptr);
  ...

private:
  const string name;
  t * const ptr;
};
```

这个类的定义要求使用一个成员初始化列表，因为 const 成员只能被初始化，不能被赋值。

如果 namedptr<t>对象包含一个现有名字的引用，情况会非常不同。但还是要在构造函数的初始化列表里对引用进行初始化。还可以对名字同时声明 const 和引用，这样就生成了一个其名字成员在类外可以被修改而在内部是只读的对象。

```
template<class t>
class namedptr {
public:
  namedptr(const string& initname, t *initptr);
  ...

private:
  const string& name;              // 必须通过成员初始化列表
                                   // 进行初始化

  t * const ptr;                   // 必须通过成员初始化列表
                                   // 进行初始化
};
```

然而前面最初的类模板不包含 const 和引用成员。即使这样，用成员初始化列表还是比在构造函数里赋值要好。这次的原因在于效率。当使用成员初始化列表时，只有一个 string 成员函数被调用。而在构造函数里赋值时，将有两个被调用。为了理解为什么，请看在声明 namedptr<t>对象时都发生了些什么。

对象的创建分两步：
1. 数据成员初始化。（参见条款 13）
2. 执行被调用构造函数体内的动作。

（对有基类的对象来说，基类的成员初始化和构造函数体的执行发生在派生类的成员初始化和构造函数体的执行之前）

对 namedptr 类来说，这意味着 string 对象 name 的构造函数总是在程序执行到 namedptr 的构造函数体之前就已经被调用了。问题只在于：string 的哪个构造函数会被调用？

这取决于 namedptr 类的成员初始化列表。如果没有为 name 指定初始化参数，string 的缺省构造函数会被调用。当在 namedptr 的构造函数里对 name 执行赋值时，会对 name 调用

operator=函数。这样总共有两次对 string 的成员函数的调用：一次是缺省构造函数，另一次是赋值。

相反，如果用一个成员初始化列表来指定 name 必须用 initname 来初始化，name 就会通过拷贝构造函数以仅一个函数调用的代价被初始化。

即使是一个很简单的 string 类型，不必要的函数调用也会造成很高的代价。随着类越来越大，越来越复杂，它们的构造函数也越来越大而复杂，那么对象创建的代价也越来越高。养成尽可能使用成员初始化列表的习惯，不但可以满足 const 和引用成员初始化的要求，还可以大大减少低效地初始化数据成员的机会。

换句话说，通过成员初始化列表来进行初始化总是合法的，效率也决不低于在构造函数体内赋值，它只会更高效。另外，它简化了对类的维护（见条款 m32），因为如果一个数据成员以后被修改成了必须使用成员初始化列表的某种数据类型，那么，什么也不用变。

但有一种情况下，对类的数据成员用赋值比用初始化更合理。这就是当有大量的固定类型的数据成员要在每个构造函数里以相同的方式初始化的时候。例如，这里有个类可以用来说明这种情形：

```
class manydatambrs {
public:
  // 缺省构造函数
  manydatambrs();

  // 拷贝构造函数
  manydatambrs(const manydatambrs& x);

private:
  int a, b, c, d, e, f, g, h;
  double i, j, k, l, m;
};
```

假如想把所有的 int 初始化为 1 而所有的 double 初始化为 0，那么用成员初始化列表就要这样写：

```
manydatambrs::manydatambrs()
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }

manydatambrs::manydatambrs(const manydatambrs& x)
: a(1), b(1), c(1), d(1), e(1), f(1), g(1), h(1), i(0),
  j(0), k(0), l(0), m(0)
{ ... }
```

这不仅仅是一项讨厌而枯燥的工作，而且从短期来说它很容易出错，从长期来说很难维护。

然而你可以利用固定数据类型的（非 const, 非引用）对象其初始化和赋值没有操作上的不同的特点，安全地将成员初始化列表用一个对普通的初始化函数的调用来代替。

```
class manydatambrs {
public:
  // 缺省构造函数
  manydatambrs();

  // 拷贝构造函数
  manydatambrs(const manydatambrs& x);

private:
  int a, b, c, d, e, f, g, h;
  double i, j, k, l, m;

  void init();        // 用于初始化数据成员

};

void manydatambrs::init()
{
  a = b = c = d = e = f = g = h = 1;
  i = j = k = l = m = 0;
}

manydatambrs::manydatambrs()
{
  init();

  ...

}

manydatambrs::manydatambrs(const manydatambrs& x)
{
  init();

  ...

}
```

因为初始化函数只是类的一个实现细节，所以当然要把它声明为 private 成员。

请注意 static 类成员永远也不会在类的构造函数初始化。静态成员在程序运行的过程中只被初始化一次，所以每当类的对象创建时都去“初始化”它们没有任何意义。至少这会影响效率：既然是“初始化”，那为什么要去做多次？而且，静态类成员的初始化和非静态类成员有很大的不同，这专门有一个条款 m47 来说明。

条款 13: 初始化列表中成员列出的顺序和它们在类中声明的顺序相同

顽固的 pascal 和 ada 程序员会经常想念那种可以任意设定数组下标上下限的功能，即，数组下标的范围可以设为 10 到 20，不一定要是 0 到 10。资深的 c 程序员会坚持一定要从 0 开始计数，但想个办法来满足那些还在用 begin/end 的人的这个要求也很容易，这只需要定义一个自己的 array 类模板：

```
template<class t>
class array {
public:
  array(int lowbound, int highbound);
  ...

private:
  vector<t> data;              // 数组数据存储在 vector 对象中
                               // 关于 vector 模板参见条款 49

  size_t size;                 // 数组中元素的数量

  int lbound, hbound;          // 下限，上限
};

template<class t>
array<t>::array(int lowbound, int highbound)
: size(highbound - lowbound + 1),
  lbound(lowbound), hbound(highbound),
  data(size)
{}
```

构造函数会对参数进行合法性检查，以保证 highbound 至少要大于等于 lowbound，但这里有个很糟糕的错误：即使数组的上下限值合法，也绝对没人会知道 data 里会有多少个元素。

“这怎么可能？”我听见你在叫。“我小心地初始化了 size 后才把它传给 vector 的构造函数！”但不幸的是，你没有——你只是想这样做，但没遵守游戏规则：类成员是按照它们在类里被声明的顺序进行初始化的，和它们在成员初始化列表中列出的顺序没一点关系。用上面的 array 模板生成的类里，data 总会被首先初始化，然后是 size, lbound 和 hbound。

看起来似乎有悖常理，但这么做是有理由的。看下面这种情况：

```
class wacko {
public:
```

```
  wacko(const char *s): s1(s), s2(0) {}
  wacko(const wacko& rhs): s2(rhs.s1), s1(0) {}

private:
  string s1, s2;
};

wacko w1 = "hello world!";
wacko w2 = w1;
```

如果成员按它们在初始化列表上出现的顺序被初始化，那 w1 和 w2 中的数据成员被创建的顺序就会不同。我们知道，对一个对象的所有成员来说，它们的析构函数被调用的顺序总是和它们在构造函数里被创建的顺序相反。那么，如果允许上面的情况（即，成员按它们在初始化列表上出现的顺序被初始化）发生，编译器就要为每一个对象跟踪其成员初始化的顺序，以保证它们的析构函数以正确的顺序被调用。这会带来昂贵的开销。所以，为了避免这一开销，同一种类型的所有对象在创建（构造）和摧毁（析构）过程中对成员的处理顺序都是相同的,而不管成员在初始化列表中的顺序如何。

实际上，如果你深究一下的话，会发现只是非静态数据成员的初始化遵守以上规则。静态数据成员的行为有点象全局和名字空间对象，所以只会被初始化一次（详见条款 47）。另外，基类数据成员总是在派生类数据成员之前被初始化，所以使用继承时，要把基类的初始化列在成员初始化列表的最前面。（如果使用多继承，基类被初始化的顺序和它们被派生类继承的顺序一致，它们在成员初始化列表中的顺序会被忽略。使用多继承有很多地方要考虑。条款 43 关于多继承应考虑哪些方面的问题提出了很多建议。）

基本的一条是：如果想弄清楚对象被初始化时到底是怎么做的，请确信你的初始化列表中成员列出的顺序和成员在类内声明的顺序一致。

条款 14: 确定基类有虚析构函数

有时，一个类想跟踪它有多少个对象存在。一个简单的方法是创建一个静态类成员来统计对象的个数。这个成员被初始化为 0，在构造函数里加 1，析构函数里减 1。（条款 m26 里说明了如何把这种方法封装起来以便很容易地添加到任何类中，“my article on counting objects”提供了对这个技术的另外一些改进）

设想在一个军事应用程序里，有一个表示敌人目标的类：

```
class enemytarget {
public:
  enemytarget() { ++numtargets; }
  enemytarget(const enemytarget&) { ++numtargets; }
  ~enemytarget() { --numtargets; }

  static size_t numberoftargets()
  { return numtargets; }

  virtual bool destroy();       // 摧毁 enemytarget 对象后
                                // 返回成功

private:
  static size_t numtargets;     // 对象计数器
};

// 类的静态成员要在类外定义;
// 缺省初始化为 0
size_t enemytarget::numtargets;
```

这个类不会为你赢得一份政府防御合同，它离国防部的要求相差太远了，但它足以满足我们这儿说明问题的需要。

敌人的坦克是一种特殊的敌人目标，所以会很自然地想到将它抽象为一个以公有继承方式从 enemytarget 派生出来的类（参见条款 35 及 m33）。因为不但要关心敌人目标的总数，也要关心敌人坦克的总数，所以和基类一样，在派生类里也采用了上面提到的同样的技巧：

```
class enemytank: public enemytarget {
public:
  enemytank() { ++numtanks; }

  enemytank(const enemytank& rhs)
  : enemytarget(rhs)
  { ++numtanks; }

  ~enemytank() { --numtanks; }
```

```
  static size_t numberoftanks()
  { return numtanks; }

  virtual bool destroy();

private:
  static size_t numtanks;        // 坦克对象计数器
};
```

（写完以上两个类的代码后，你就更能够理解条款 m26 对这个问题的通用解决方案了。）

最后，假设程序的其他某处用 new 动态创建了一个 enemytank 对象，然后用 delete 删除掉：

```
enemytarget *targetptr = new enemytank;

...

delete targetptr;
```

到此为止所做的一切好象都很正常：两个类在析构函数里都对构造函数所做的操作进行了清除；应用程序也显然没有错误，用 new 生成的对象在最后也用 delete 删除了。然而这里却有很大的问题。程序的行为是不可预测的——无法知道将会发生什么。

c++语言标准关于这个问题的阐述非常清楚：当通过基类的指针去删除派生类的对象，而基类又没有虚析构函数时，结果将是不可确定的。这意味着编译器生成的代码将会做任何它喜欢的事：重新格式化你的硬盘，给你的老板发电子邮件，把你的程序源代码传真给你的对手，无论什么事都可能发生。（实际运行时经常发生的是，派生类的析构函数永远不会被调用。在本例中，这意味着当 targetptr 删除时，enemytank 的数量值不会改变，那么，敌人坦克的数量就是错的，这对需要高度依赖精确信息的部队来说，会造成什么后果？）

为了避免这个问题，只需要使 enemytarget 的析构函数为 virtual。声明析构函数为虚就会带来你所希望的运行良好的行为：对象内存释放时，enemytank 和 enemytarget 的析构函数都会被调用。

和绝大部分基类一样，现在 enemytarget 类包含一个虚函数。虚函数的目的是让派生类去定制自己的行为（见条款 36），所以几乎所有的基类都包含虚函数。

如果某个类不包含虚函数，那一般是表示它将不作为一个基类来使用。当一个类不准备作为基类使用时，使析构函数为虚一般是个坏主意。请看下面的例子，这个例子基于 arm(“the annotated c++ reference manual”)一书的一个专题讨论。

```
// 一个表示 2d 点的类
class point {
public:
  point(short int xcoord, short int ycoord);
  ~point();
```

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

如果一个 short int 占 16 位，一个 point 对象将刚好适合放进一个 32 位的寄存器中。另外，一个 point 对象可以作为一个 32 位的数据传给用 c 或 fortran 等其他语言写的函数中。但如果 point 的析构函数为虚，情况就会改变。

实现虚函数需要对象附带一些额外信息，以使对象在运行时可以确定该调用哪个虚函数。对大多数编译器来说，这个额外信息的具体形式是一个称为 vptr（虚函数表指针）的指针。vptr 指向的是一个称为 vtbl（虚函数表）的函数指针数组。每个有虚函数的类都附带有一个 vtbl。当对一个对象的某个虚函数进行请求调用时，实际被调用的函数是根据指向 vtbl 的 vptr 在 vtbl 里找到相应的函数指针来确定的。

虚函数实现的细节不重要（当然，如果你感兴趣，可以阅读条款 m24），重要的是，如果 point 类包含一个虚函数，它的对象的体积将不知不觉地翻番，从 2 个 16 位的 short 变成了 2 个 16 位的 short 加上一个 32 位的 vptr！point 对象再也不能放到一个 32 位寄存器中去了。而且，c++中的 point 对象看起来再也不具有和其他语言如 c 中声明的那样相同的结构了，因为这些语言里没有 vptr。所以，用其他语言写的函数来传递 point 也不再可能了，除非专门去为它们设计 vptr，而这本身是实现的细节，会导致代码无法移植。

所以基本的一条是，无故的声明虚析构函数和永远不去声明一样是错误的。实际上，很多人这样总结：当且仅当类里包含至少一个虚函数的时候才去声明虚析构函数。

这是一个很好的准则，大多数情况都适用。但不幸的是，当类里没有虚函数的时候，也会带来非虚析构函数问题。 例如，条款 13 里有个实现用户自定义数组下标上下限的类模板。假设你（不顾条款 m33 的建议）决定写一个派生类模板来表示某种可以命名的数组(即每个数组有一个名字)。

```
template<class t>              // 基类模板
class array {                  // (来自条款 13)
public:
  array(int lowbound, int highbound);
  ~array();

private:
  vector<t> data;
  size_t size;
  int lbound, hbound;
};

template<class t>
class namedarray: public array<t> {
public:
```

```
  namedarray(int lowbound, int highbound, const string& name);
  ...

private:
  string arrayname;
};
```

如果在应用程序的某个地方你将指向 namedarray 类型的指针转换成了 array 类型的指针，然后用 delete 来删除 array 指针，那你就会立即掉进“不确定行为”的陷阱中。

```
namedarray<int> *pna =
  new namedarray<int>(10, 20, "impending doom");

array<int> *pa;

...


pa = pna;               // namedarray<int>* -> array<int>*

...

delete pa;              // 不确定！实际中，pa->arrayname
                    // 会造成泄漏，因为*pa 的 namedarray
                    // 永远不会被删除
```

现实中，这种情形出现得比你想象的要频繁。让一个现有的类做些什么事，然后从它派生一个类做和它相同的事，再加上一些特殊的功能，这在现实中不是不常见。namedarray 没有重定义 array 的任何行为——它继承了 array 的所有功能而没有进行任何修改——它只是增加了一些额外的功能。但非虚析构函数的问题依然存在（还有其他问题，参见 m33）

最后，值得指出的是，在某些类里声明纯虚析构函数很方便。纯虚函数将产生抽象类——不能实例化的类（即不能创建此类型的对象）。有些时候，你想使一个类成为抽象类，但刚好又没有任何纯虚函数。怎么办？因为抽象类是准备被用做基类的，基类必须要有一个虚析构函数，纯虚函数会产生抽象类，所以方法很简单：在想要成为抽象类的类里声明一个纯虚析构函数。

这里是一个例子：

```
class awov {              // awov = "abstract w/o
                      // virtuals"
public:
  virtual ~awov() = 0;      // 声明一个纯虚析构函数

};
```

这个类有一个纯虚函数，所以它是抽象的，而且它有一个虚析构函数，所以不会产生析构函数问题。但这里还有一件事：必须提供纯虚析构函数的定义：

```
awov::~awov() {}          // 纯虚析构函数的定义
```

这个定义是必需的，因为虚析构函数工作的方式是：最底层的派生类的析构函数最先被调用，然后各个基类的析构函数被调用。这就是说，即使是抽象类，编译器也要产生对~awov 的调用，所以要保证为它提供函数体。如果不这么做，链接器就会检测出来，最后还是得回去把它添上。

可以在函数里做任何事，但正如上面的例子一样，什么事都不做也不是不常见。如果是这种情况，那很自然地会想到将析构函数声明为内联函数，从而避免对一个空函数的调用所产生的开销。这是一个很好的方法，但有一件事要清楚。

因为析构函数为虚，它的地址必须进入到类的 vtbl（见条款 m24）。但内联函数不是作为独立的函数存在的（这就是“内联”的意思），所以必须用特殊的方法得到它们的地址。条款 33 对此做了全面的介绍，其基本点是：如果声明虚析构函数为 inline，将会避免调用它们时产生的开销，但编译器还是必然会在什么地方产生一个此函数的拷贝。

条款 15: 让 operator=返回*this 的引用

c++的设计者 bjarne stroustrup 下了很大的功夫想使用户自定义类型尽可能地和固定类型的工作方式相似。这就是为什么你可以重载运算符，写类型转换函数（见条款 m5），控制赋值和拷贝构造函数，等等。他做了这么多努力，那你最少也该继续做下去。

让我们看看赋值。用固定类型的情况下，赋值操作可以象下面这样链起来：

```
int w, x, y, z;
w = x = y = z = 0;
```

所以，你也应该可以将用户自定义类型的赋值操作链起来：

```
string w, x, y, z;      // string 是由标准 c++库
                                    // “自定义”的类型
                                    // (参见条款 49)


w = x = y = z = "hello";
```

因为赋值运算符的结合性天生就是由右向左，所以上面的赋值可以解析为：

```
w = (x = (y = (z = "hello")));
```

很值得把它写成一个完全等价的函数形式。除非是个 lisp 程序员，否则下面的例子会很令人感到高兴，因为它定义了一个中缀运算符：

```
w.operator=(x.operator=(y.operator=(z.operator=("hello"))));
```

这个格式在此很具有说明性，因为它强调了 w.operator=, x.operator=和 y.operator=的参数是前一个 operator=调用的返回值。所以 operator=的返回值必须可以作为一个输入参数被函数自己接受。在一个类 c 中，缺省版本的 operator=函数具有如下形式（见条款 45）：

```
c& c::operator=(const c&);
```

一般情况下几乎总要遵循 operator=输入和返回的都是类对象的引用的原则，然而有时候需要重载 operator=使它能够接受不同类型的参数。例如，标准 string 类型提供了两个不同版本的赋值运算符：

```
string&    // 将一个 string
operator=(const string& rhs); // 赋给一个 string

string&    // 将一个 char*
operator=(const char *rhs);    // 赋给一个 string
```

请注意，即使在重载时，返回类型也是类的对象的引用。

c++程序员经常犯的一个错误是让 operator=返回 void，这好象没什么不合理的，但它妨碍了连续（链式）赋值操作，所以不要这样做。

另一个常犯的错误是让 operator=返回一个 const 对象的引用，象下面这样：

```
class widget {
public:
...
const widget& operator=(const widget& rhs);
...
};
```

这样做通常是为了防止程序中做象下面这样愚蠢的操作：

```
widget w1, w2, w3;
...
(w1 = w2) = w3;                    // w2 赋给 w1, 然后 w3 赋给其结果
                                   //(给 operator=一个 const 返回值
                                   // 就使这个语句不能通过编译)
```

这可能是很愚蠢，但固定类型这么做并不愚蠢:

```
int i1, i2, i3;
...
(i1 = i2) = i3;          // 合法! i2 赋给 i1
                                   // 然后 i3 赋给 i1!
```

这样的做法实际中很少看到，但它对 int 来说是可以的，对我和我的类来说也可以。那它对你和你的类也应该可以。为什么要无缘无故地和固定类型的常规做法不兼容呢？

采用缺省形式定义的赋值运算符里，对象返回值有两个很明显的候选者：赋值语句左边的对象（被 this 指针指向的对象）和赋值语句右边的对象（参数表中被命名的对象）。哪一个是正确的呢？

例如，对 string 类（假设你想在这个类中写赋值运算符，参见条款 11 中的解释）来说有两种可能：

```
string& string::operator=(const string& rhs)
{
...
return *this;          // 返回左边的对象
}
string& string::operator=(const string& rhs)
{
...
```

```
return rhs;             // 返回右边的对象
}
```

对你来说，这好象是拿六个一和十二的一半来比较一样为难。实际上他们有很大的不同。

首先，返回 rhs 的那个版本不会通过编译，因为 rhs 是一个 const string 的引用，而 operator= 要返回的是一个 string 的引用。当要返回一个非 const 的引用而对象自身是 const 时，编译器会给你带来无尽的痛苦。看起来这个问题很容易解决——只用象这样重新声明 operator=：

```
string& string::operator=(string& rhs) { ... }
```

这次又轮到用到它的应用程序不能通过编译了！再看看最初那个连续赋值语句的后面部分：

```
x = "hello";            // 和 x.op = ("hello"); 相同
```

因为赋值语句的右边参数不是正确的类型——它是一个字符数组，不是一个 string——编译器就要产生一个临时的 string 对象（通过 stirng 构造函数——参见条款 m19）使得函数继续运行。就是说，编译器必须产生大致象下面这样的代码：

```
const string temp("hello");     // 产生临时 string
x = temp;               // 临时 string 传给 operator=
```

编译器一般会产生这样的临时值（除非显式地定义了所需要的构造函数——见条款 19），但注意临时值是一个 const。这很重要，因为它可以防止传递到函数内的临时值被修改。否则，程序员就会很奇怪地发现，只有编译器产生的临时值可以修改而他们在函数调用时实际传进去的参数却不行。（关于这一点是有事实根据的，早期版本的 c++允许这类的临时值可以被产生，传递，修改，结果很多程序员感到很奇怪）

现在我们就可以知道如果 string 的 operator=声明传递一个非 const 的 stirng 参数，应用程序就不能通过编译的原因了：对于没有声明相应参数为 const 的函数来说，传递一个 const 对象是非法的。这是一个关于 const 的很简单的规定。

所以，结论是，这种情况下你将别无选择：当定义自己的赋值运算符时，必须返回赋值运算符左边参数的引用，*this。如果不这样做，就会导致不能连续赋值，或导致调用时的隐式类型转换不能进行，或两种情况同时发生。

条款 16: 在 operator=中对所有数据成员赋值

条款 45 说明了如果没写赋值运算符的话，编译器就会为你生成一个，条款 11 则说明了为什么你会经常不喜欢编译器为你生成的这个赋值运算符，所以你会想能否有个两全其美的办法，让编译器生成一个缺省的赋值运算符，然后可以有选择地重写不喜欢的部分。这是不可能的！只要想对赋值过程的某一个部分进行控制，就必须负责做赋值过程中所有的事。

实际编程中，这意味着写赋值运算符时，必须对对象的每一个数据成员赋值：

```
template<class t>          // 名字和指针相关联的类的模板
class namedptr {           // （源自条款 12）
public:
  namedptr(const string& initname, t *initptr);
  namedptr& operator=(const namedptr& rhs);

private:
  string name;
  t *ptr;
};

template<class t>
namedptr<t>& namedptr<t>::operator=(const namedptr<t>& rhs)
{
  if (this == &rhs)
    return *this;              // 见条款 17

  // assign to all data members
  name = rhs.name;             // 给 name 赋值

  *ptr = *rhs.ptr;             // 对于 ptr，赋的值是指针所指的值，
                               // 不是指针本身

  return *this;                // 见条款 15
}
```

初写这个类时当然很容易记住上面的原则，但同样重要的是，当类里增加新的数据成员时，也要记住更新赋值运算符函数。例如，打算升级 namedptr 模板使得名字改变时附带一个时间标记，那就要增加一个新的数据成员，同时需要更新构造函数和赋值运算符。但现实中，因为忙于升级类的具体功能和增加新的成员函数等，这一点往往很容易被忘记。

当涉及到继承时，情况就会更有趣，因为派生类的赋值运算符也必须处理它的基类成员的赋值！看看下面：

```
class base {
public:
  base(int initialvalue = 0): x(initialvalue) {}

private:
  int x;
};

class derived: public base {
public:
  derived(int initialvalue)
  : base(initialvalue), y(initialvalue) {}

  derived& operator=(const derived& rhs);

private:
  int y;
};
```

逻辑上说，derived 的赋值运算符应该象这样：

```
// erroneous assignment operator
derived& derived::operator=(const derived& rhs)
{
  if (this == &rhs) return *this;    // 见条款 17

  y = rhs.y;                         // 给 derived 仅有的
                                     // 数据成员赋值

  return *this;                      // 见条款 15
}
```

不幸的是，它是错误的，因为 derived 对象的 base 部分的数据成员 x 在赋值运算符中未受影响。例如，考虑下面的代码段：

```
void assignmenttester()
{
  derived d1(0);                     // d1.x = 0, d1.y = 0
  derived d2(1);                     // d2.x = 1, d2.y = 1

  d1 = d2;         // d1.x = 0, d1.y = 1!
}
```

请注意 d1 的 base 部分没有被赋值操作改变。

解决这个问题最显然的办法是在 derived::operator=中对 x 赋值。但这不合法，因为 x 是 base 的私有成员。所以必须在 derived 的赋值运算符里显式地对 derived 的 base 部分赋值。

也就是这么做：

```
// 正确的赋值运算符
derived& derived::operator=(const derived& rhs)
{
  if (this == &rhs) return *this;

  base::operator=(rhs);    // 调用 this->base::operator=
  y = rhs.y;

  return *this;
}
```

这里只是显式地调用了 base::operator=，这个调用和一般情况下的在成员函数中调用另外的成员函数一样，以*this 作为它的隐式左值。base::operator=将针对*this 的 base 部分执行它所有该做的工作——正如你所想得到的那种效果。

但如果基类赋值运算符是编译器生成的，有些编译器会拒绝这种对于基类赋值运算符的调用（见条款 45）。为了适应这种编译器，必须这样实现 derived::operator=：

```
derived& derived::operator=(const derived& rhs)
{
  if (this == &rhs) return *this;

  static_cast<base&>(*this) = rhs;      // 对*this 的 base 部分
                                        // 调用 operator=
  y = rhs.y;

  return *this;
}
```

这段怪异的代码将*this 强制转换为 base 的引用，然后对其转换结果赋值。这里只是对 derived 对象的 base 部分赋值。还要注意的重要一点是，转换的是 base 对象的引用，而不是 base 对象本身。如果将*this 强制转换为 base 对象，就要导致调用 base 的拷贝构造函数，创建出来的新对象（见条款 m19）就成为了赋值的目标，而*this 保持不变。这不是所想要的结果。

不管采用哪一种方法，在给 derived 对象的 base 部分赋值后，紧接着是 derived 本身的赋值，即对 derived 的所有数据成员赋值。

另一个经常发生的和继承有关的类似问题是在实现派生类的拷贝构造函数时。看看下面这个构造函数，其代码和上面刚讨论的类似：

```
class base {
public:
  base(int initialvalue = 0): x(initialvalue) {}
  base(const base& rhs): x(rhs.x) {}

private:
  int x;
};

class derived: public base {
public:
  derived(int initialvalue)
  : base(initialvalue), y(initialvalue) {}

  derived(const derived& rhs)      // 错误的拷贝
  : y(rhs.y) {}                    // 构造函数

private:
  int y;
};
```

类 derived 展现了一个在所有 c++环境下都会产生的 bug：当 derived 的拷贝创建时，没有拷贝其基类部分。当然，这个 derived 对象的 base 部分还是创建了，但它是用 base 的缺省构造函数创建的，成员 x 被初始化为 0（缺省构造函数的缺省参数值），而没有顾及被拷贝的对象的 x 值是多少！

为避免这个问题，derived 的拷贝构造函数必须保证调用的是 base 的拷贝构造函数而不是 base 的缺省构造函数。这很容易做，只要在 derived 的拷贝构造函数的成员初始化列表里对 base 指定一个初始化值：

```
class derived: public base {
public:
  derived(const derived& rhs): base(rhs), y(rhs.y) {}

  ...

};
```

现在，当用一个已有的同类型的对象来拷贝创建一个 derived 对象时，它的 base 部分也将被拷贝了。

条款 17: 在 operator=中检查给自己赋值的情况

做类似下面的事时，就会发生自己给自己赋值的情况：

```
class x { ... };

x a;

a = a;                    // a 赋值给自己
```

这种事做起来好象很无聊，但它完全是合法的，所以看到程序员这样做不要感到丝毫的怀疑。更重要的是，给自己赋值的情况还可以以下面这种看起来更隐蔽的形式出现：

```
a = b;
```

如果 b 是 a 的另一个名字（例如，已被初始化为 a 的引用），那这也是对自己赋值，虽然表面上看起来不象。这是别名的一个例子：同一个对象有两个以上的名字。在本条款的最后将会看到，别名可以以大量任意形式的伪装出现，所以在写函数时一定要时时考虑到它。

在赋值运算符中要特别注意可能出现别名的情况，其理由基于两点。其中之一是效率。如果可以在赋值运算符函数体的首部检测到是给自己赋值，就可以立即返回，从而可以节省大量的工作，否则必须去实现整个赋值操作。例如，条款 16 指出，一个正确的派生类的赋值运算符必须调用它的每个基类的的赋值运算符，所以在派生类中省略赋值运算符函数体的操作将会避免大量对其他函数的调用。另一个更重要的原因是保证正确性。一个赋值运算符必须首先释放掉一个对象的资源（去掉旧值），然后根据新值分配新的资源。在自己给自己赋值的情况下，释放旧的资源将是灾难性的，因为在分配新的资源时会需要旧的资源。

看看下面 string 对象的赋值，赋值运算符没有对给自己赋值的情况进行检查：

```
class string {
public:
  string(const char *value);     //  函数定义参见条款 11
                                 //

  ~string();                     //  函数定义参见条款 11
                                 //
  ...

  string& operator=(const string& rhs);

private:
  char *data;
};
```

```
// 忽略了给自己赋值的情况
// 的赋值运算符
string& string::operator=(const string& rhs)
{
  delete [] data;     // delete old memory

  // 分配新内存，将 rhs 的值拷贝给它
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);

  return *this;       // see item 15
}
```

看看下面这种情况将会发生什么：

```
string a = "hello";

a = a;                // same as a.operator=(a)
```

赋值运算符内部，*this 和 rhs 好象是不同的对象，但在现在这种情况下它们却恰巧是同一个对象的不同名字。可以这样来表示这种情况：

```
*this   data ------------> "hello\0"
                /
               /
rhs     data -----
```

赋值运算符做的第一件事是用 delete 删除 data，其结果将如下所示：

```
*this   data ------------> ???
                /
               /
rhs     data -----
```

现在，当赋值运算符对 rhs.data 调用 strlen 时，结果将无法确定。这是因为 data 被删除的时候 rhs.data 也被删除了，data，this->data 和 rhs.data 其实都是同一个指针！从这一点看，情况只会越变越糟糕。现在可以知道，解决问题的方案是对可能发生的自己给自己赋值的情况先进行检查，如果有这种情况就立即返回。不幸的是，这种检查说起来容易做起来难，因为你必须定义两个对象怎么样才算是“相同”的。

你面临的这个问题学术上称为 object identity，它在面向对象领域是个很有名的论题。本书不是讲述 object identity 的地方，但有必要提到两个解决这个问题的基本方法。

一个方法是，如果两个对象具有相同的值，就说它们是相同的（具有相同的身份）。例如，两个 string 对象如果都表示的是相同顺序的字符序列，它们就是相同的：

```
string a = "hello";
string b = "world";
string c = "hello";
```

a 和 c 具有相同值，所以它们被认为是完全相同的；b 和它们都不同。如果把这个定义用到 string 类中，赋值运算符看起来就象这样：

```
string& string::operator=(const string& rhs)
{
  if (strcmp(data, rhs.data) == 0) return *this;

  ...

}
```

值相等通常由 operator==来检测，所以对于一个用值相等来检测对象身份的类 c 来说，它的赋值运算符的一般形式是：

```
c& c::operator=(const c& rhs)
{
  // 检查对自己赋值的情况
  if (*this == rhs)          // 假设 operator=存在
    return *this;

  ...

}
```

注意这个函数比较的是对象（通过 operator=），而不是指针。用值相等来确定对象身份和两个对象是否占用相同的内存没有关系；有关系的只是它们所表示的值。另一个确定对象身份是否相同的方法是用内存地址。采用这个定义，两个对象当且仅当它们具有相同的地址时才是相同的。这个定义在 c++程序中运用更广泛，可能是因为它很容易实现而且计算很快，而采用值相等的定义则不一定总具有这两个优点。采用地址相等的定义，一个普通的赋值运算符看起来象这样：

```
c& c::operator=(const c& rhs)
{
  // 检查对自己赋值的情况
  if (this == &rhs) return *this;

  ...

}
```

它对很多程序都适用。

如果需要一个更复杂的机制来确定两个对象是否相同，这就要靠程序员自己来实现。最普通的方法是实现一个返回某种对象标识符的成员函数：

```
class c {
public:
  objectid identity() const;     // 参见条款 36

  ...

};
```

对于两个对象指针 a 和 b，当且仅当 a->identity() == b->identity()的时候，它们所指的对象是完全相同的。当然，必须自己来实现 objectids 的 operator==。别名和 object identity 的问题不仅仅局限在 operator=里。在任何一个用到的函数里都可能会遇到。在用到引用和指针的场合，任何两个兼容类型的对象名称都可能指的是同一个对象。下面列出的是别名出现的其它情形：

```
class base {
  void mf1(base& rb);          // rb 和*this 可能相同
  ...

};

void f1(base& rb1,base& rb2);  // rb1 和 rb2 可能相同
                               //

class derived: public base {
  void mf2(base& rb);          // rb 和*this 可能相同
                               //
  ...

};

int f2(derived& rd, base& rb); // rd 和 rb 可能相同
                               //
```

这些例子刚好都用的是引用，指针也一样。

可以看到，别名可以以各种形式出现，所以决不要忘记它或期望自己永远不会碰到它。也许你不会碰到，但我们大多数会碰到。而很明显的一条是，处理它会达到事半功倍的效果。所以任何时候写一个函数，只要别名有可能出现，就必须在写代码时进行处理。

第四章 类和函数：设计与声明

在程序中声明一个新类将导致产生一种新的类型：类的设计就是类型设计。可能你对类型设计没有太多经验，因为大多数语言没有为你提供实践的机会。在 c++中，这却是很基本的特性，不是因为你想去做才可以这么做，而是因为每次你声明一个类的时候实际上就在做，无论你想不想做。

设计一个好的类很具有挑战性，因为设计好的类型很具有挑战性。好的类型具有自然的语法，直观的语义和高效的实现。在 c++中，一个糟糕的类的定义是无法实现这些目标的。即使一个类的成员函数的性能也是由这些成员函数的声明和定义决定的。

那么，怎么着手设计高效的类呢？首先，必须清楚你面临的问题。实际上，设计每个类时都会遇到下面的问题，它的答案将影响到你的设计。

·对象将如何被创建和摧毁？它将极大地影响构造函数和析构函数的设计，以及自定义的 operator new, operator new[], operator delete, 和 operator delete[]。（条款 m8 描述了这些术语的区别）

·对象初始化和对象赋值有什么不同？答案决定了构造函数和赋值运算符的行为以及它们之间的区别。

·通过值来传递新类型的对象意味着什么？记住，拷贝函数负责对此做出回答。

·新类型的合法值有什么限制？这些限制决定了成员函数（特别是构造函数和赋值运算符）内部的错误检查的种类。它可能还影响到函数抛出的例外的种类以及函数的例外规范（参见条款 m14），如果你使用它们的话。

·新类型符合继承关系吗？如果是从已有的类继承而来，那么新类的设计就要受限于这些类，特别是受限于被继承的类是虚拟的还是非虚拟的。如果新类允许被别的类继承，这将影响到函数是否要声明为虚拟的。

·允许哪种类型转换？如果允许类型 a 的对象隐式转换为类型 b 的对象，就要在类 a 中写一个类型转换函数，或者，在类 b 中写一个可以用单个参数来调用的非 explicit 构造函数。如果只允许显式转换，就要写函数来执行转换功能，但不用把它们写成类型转换运算符和或单参数的非 explicit 构造函数。（条款 m5 讨论了用户自定义转换函数的优点和缺点）

·什么运算符和函数对新类型有意义？答案决定了将要在类接口中声明什么函数。

·哪些运算符和函数要被明确地禁止？它们需要被声明为 private。

·谁有权访问新类型的成员？这个问题有助于决定哪些成员是公有的，哪些是保护的，哪些私有的。它还有助于确定哪些类和/或函数必须是友元，以及将一个类嵌套到另一个类中是否有意义。

·新类型的通用性如何？也许你实际上不是在定义一个新的类型，而是在定义一整套的类型。如果是这样，就不要定义一个新类，而要定义一个新的类模板。

这些都是很难回答的问题，所以 c++中定义一个高效的类远不是那么简单。但如果做好了，c++中用户自定义的类所产生的类型就会和固定类型几乎没什么区别，如果能达到这样的效果，其价值也就体现出来了。

上面每一个问题如果要详细讨论都可以单独组成一本书。所以后面条款中所介绍的准则决不会面面俱到。但是，它们强调了在设计中一些很重要的注意事项，提醒一些常犯的错误，对设计者常碰到的一些问题提供了解决方案。很多建议对非成员函数和成员函数都适用，所以本章节我也考虑了全局函数和名字空间中的函数的设计和声明。

条款 18: 争取使类的接口完整并且最小

类的用户接口是指使用这个类的程序员所能访问得到的接口。典型的接口里只有函数存在，因为在用户接口里放上数据成员会有很多缺点（见条款 20）。

哪些函数该放在类的接口里呢？有时这个问题会使你发疯，因为有两个截然不同的目标要你去完成。一方面，设计出来的类要易于理解，易于使用，易于实现。这意味着函数的数量要尽可能地少，每一个函数都完成各自不同的任务。另一方面，类的功能要强大，要方便使用，这意味着要不时增加函数以提供对各种通用功能的支持。你会怎样决定哪些函数该放进类里，哪些不放呢？

试试这个建议：类接口的目标是完整且最小。

一个完整的接口是指那种允许用户做他们想做的任何合理的事情的接口。也就是说，对用户想完成的任何合理的任务，都有一个合理的方法去实现，即使这个方法对用户来说没有所想象的那样方便。相反，一个最小的接口，是指那种函数尽可能少、每两个函数都没有重叠功能的接口。如果能提供一个完整、最小的接口，用户就可以做任何他们想做的事，但类的接口不必再那样复杂。

追求接口的完整看起来很自然，但为什么要使接口最小呢？为什么不让用户做任何他们想做的事，增加更多的函数，使大家都高兴呢？

撇开处世原则方面的因素不谈——牵就你的用户真的正确吗？——充斥着大量函数的类的接口从技术上来说有很多缺点。第一，接口中函数越多，以后的潜在用户就越难理解。他们越难理解，就越不愿意去学该怎么用。一个有 10 个函数的类好象对大多数人来说都易于使用，但一个有 100 个函数的类对许多程序员来说都难以驾驭。在扩展类的功能使之尽可能地吸引用户的时候，注意不要去打击用户学习使用它们的积极性。

大的接口还会带来混淆。假设在一个人工智能程序里建立一个支持识别功能的类。其中一个成员函数叫 think（想），后来有些人想把函数名叫做 ponder（深思），另外还一些人喜欢叫 ruminate（沉思）。为了满足所有人的需要，你提供了三个函数，虽然他们做同样的事。那么想想，以后某个使用这个类的用户会怎么想呢？这个用户会面对三个不同的函数，每个函数好象都是做相同的事。真的吗？难道这三个函数有什么微妙的不同，效率上，通用性上，或可靠性上？如果没有不同，为什么会有三个函数？这样的话，这个用户不但不感激你提供的灵活性，还会纳闷你究竟在想（或者深思，或者沉思）些什么？

大的类接口的第二个缺点是难以维护（见条款 m32）。含有大量函数的类比含有少量函数的类更难维护和升级，更难以避免重复代码（以及重复的 bug），而且难以保持接口的一致性。同时，也难以建立文档。

最后，长的类定义会导致长的头文件。因为程序在每次编译时都要读头文件（见条款 34），类的定义太长会导致项目开发过程中浪费大量的编译时间。

概括起来就是说，无端地在接口里增加函数不是没有代价的，所以在增加一个新函数时要仔细考虑：它所带来的方便性（只有在接口完整的前提下才应该考虑增加一个新函数以提供方便性）是否超过它所带来的额外代价，如复杂性，可读性，可维护性和编译时间等。

但太过吝啬也没必要。在最小的接口上增加一些函数有时是合理的。如果一个通用的功能用成员函数实现起来会更高效，这将是把它增加到接口中的好理由。（但，有时不会，参见条款 m16）如果增加一个成员函数使得类易于使用，或者可以防止用户错误，也都是把它加入到接口中的有力依据。

看一个具体的例子：一个类模板，实现了用户自定义下标上下限的数组功能，另外提供上下限检查选项。模板的开头部分如下所示：

```
template<class t>
class array {
public:
  enum boundscheckingstatus {no_check_bounds = 0,
                             check_bounds = 1};

  array(int lowbound, int highbound,
      boundscheckingstatus check = no_check_bounds);

  array(const array& rhs);

  ~array();

  array& operator=(const array& rhs);

private:
  int lbound, hbound;          // 下限，上限

  vector<t> data;              // 数组内容; 关于 vector，
                               // 请参见条款 49

  boundscheckingstatus checkingbounds;
};
```

目前为止声明的成员函数是基本上不用想（或深思，沉思）就该声明的。一个允许用户确定每个数组上下限的构造函数，一个拷贝构造函数，一个赋值运算符和一个析构函数。析构函数被声明为非虚拟的，意味着这个类将不作为基类使用（见条款 14）。

对于赋值运算符的声明，第一眼看上去会觉得目的不那么明确。毕竟，c++中固定类型的数组是不允许赋值的，所以好象也应该不允许 array 对象赋值（参见条款 27）。但另一方面，数组似的 vector 模板（存在于标准库——参见条款 49）允许 vector 对象间赋值。在本例中，决定遵循 vector 的规定，正如下面将会看到的，这个决定将影响到类的接口的其他部分。

老的 c 程序员看到这个接口会被吓退：怎么竟然不支持固定大小的数组声明？很容易增加一个构造函数来实现啊：

```
array(int size,
      boundscheckingstatus check = no_check_bounds);
```

但这就不能成为最小接口了，因为带上下限参数的那个构造函数可以完成同样的事。尽管如此，出于某些目的去迎合那些老程序员们的需要也可能是明智的，特别是出于和基本语言（c 语言）一致的考虑。

还需要哪些函数？对于一个完整的接口来说当然还需要对数组的索引：

```
// 返回可以读/写的元素
t& operator[](int index);

// 返回只读元素
const t& operator[](int index) const;
```

通过两次声明同一个函数，一次带 const 一次没有 const，就提供了对 const 和非 const array 对象的支持。返回值不同很重要，条款 21 对此进行了说明。

现在，array 模板支持构造函数，析构函数，传值，赋值，索引，你可能想到这已经是一个完整的接口了。但再看清楚一些。假如一个用户想遍历一个整数数组，打印其中的每一个元素，如下所示：

```
array<int> a(10, 20);      // 下标上下限为：10 到 20

...

for (int i = a 的下标下限; i <= a 的下标上限; ++i)
  cout << "a[" << i << "] = " << a[i] << '\n';
```

用户怎么得到 a 的下标上下限呢？答案取决于 array 对象的赋值操作做了些什么，即在 array::operator=里做了什么。特别是，如果赋值操作可以改变 array 对象的上下限，就必须提供一个返回当前上下限值的成员函数，因为用户无法总能在程序的某个地方推出上下限值是多少。比如上面的例子，a 是在被定义后、用于循环前的时间段里被赋值的，用户在循环语句中就无法知道 a 当前的上下限值。

如果 array 对象的上下限值在赋值时不能改变，那它在 a 被定义时就固定下来了，用户就可能有办法（虽然很麻烦）对其进行跟踪。这种情况下，提供一个函数返回当前上下限值是很方便，但接口就不能做到最小。

继续前面的赋值操作可以改变对象上下限的假设，上下限函数可以这样声明：

```
int lowbound() const;
int highbound() const;
```

因为这两个函数不对它们所在的对象进行任何修改操作，而且为遵循“能用 const 就尽量用 const”的原则（见条款 21），它们被声明为 const 成员函数。有了这两个函数，循环语句可以象下面这样写：

```
for (int i = a.lowbound(); i <= a.highbound(); ++i)
  cout << "a[" << i << "] = " << a[i] << '\n';
```

当然，要使这样一个操作类型 t 的对象数组的循环语句工作，还要为类型 t 的对象定义一个 operator<<函数。（说得不太准确。应该是，必须有一个类型 t 的 operator<<，或，t 可以隐式转换（见条款 m5）成的其它类型的 operator<<）

一些人会争论，array 类应该提供一个函数以返回 array 对象里元素的数量。元素的数量可以简单地得到：highbound()-lowbound()+1，所以这个函数不是那么真的必要。但考虑到很多人经常忘了"+1"，增加这个函数也不是坏主意。

还有一些其他函数可以加到类里，包括那些输入输出方面的操作，还有各种关系运算符（例如，<, >, ==, 等）。但这些函数都不是最小接口的一部分，因为它们都可以通过包含 operator[] 调用的循环来实现。

说到象 operator<<, operator>>这样的函数以及关系运算符，条款 19 解释了为什么它们经常用非成员的友元函数而不用成员函数来实现。另外，不要忘记友元函数在所有实际应用中都是类的接口的一部分。这意味着友元函数影响着类的接口的完整性和最小性。

条款 19: 分清成员函数，非成员函数和友元函数

成员函数和非成员函数最大的区别在于成员函数可以是虚拟的而非成员函数不行。所以，如果有个函数必须进行动态绑定（见条款 38），就要采用虚拟函数，而虚拟函数必定是某个类的成员函数。关于这一点就这么简单。如果函数不必是虚拟的，情况就稍微复杂一点。

看下面表示有理数的一个类：

```
class rational {
public:
  rational(int numerator = 0, int denominator = 1);
  int numerator() const;
  int denominator() const;

private:
  ...
};
```

这是一个没有一点用处的类。（用条款 18 的术语来说，接口的确最小，但远不够完整。）所以，要对它增加加，减，乘等算术操作支持，但是，该用成员函数还是非成员函数，或者，非成员的友元函数来实现呢？

当拿不定主意的时候，用面向对象的方法来考虑！有理数的乘法是和 rational 类相联系的，所以，写一个成员函数把这个操作包到类中。

```
class rational {
public:

  ...

  const rational operator*(const rational& rhs) const;
};
```

（如果你不明白为什么这个函数以这种方式声明——返回一个 const 值而取一个 const 的引用作为它的参数——参考条款 21-23。）

现在可以很容易地对有理数进行乘法操作：

```
rational oneeighth(1, 8);
rational onehalf(1, 2);

rational result = onehalf * oneeighth;    // 运行良好

result = result * oneeighth;              // 运行良好
```

但不要满足，还要支持混合类型操作，比如，rational 要能和 int 相乘。但当写下下面的代码时，只有一半工作：

```
result = onehalf * 2;      // 运行良好

result = 2 * onehalf;      // 出错!
```

这是一个不好的苗头。记得吗？乘法要满足交换律。

如果用下面的等价函数形式重写上面的两个例子，问题的原因就很明显了：

```
result = onehalf.operator*(2);      // 运行良好

result = 2.operator*(onehalf);      // 出错!
```

对象 onehalf 是一个包含 operator*函数的类的实例，所以编译器调用了那个函数。而整数 2 没有相应的类，所以没有 operator*成员函数。编译器还会去搜索一个可以象下面这样调用的非成员的 operator*函数（即，在某个可见的名字空间里的 operator*函数或全局的 operator*函数）：

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

但没有这样一个参数为 int 和 rational 的非成员 operator*函数，所以搜索失败。

再看看那个成功的调用。它的第二参数是整数 2，然而 rational::operator*期望的参数却是 rational 对象。怎么回事？为什么 2 在一个地方可以工作而另一个地方不行？

秘密在于隐式类型转换。编译器知道传的值是 int 而函数需要的是 rational，但它也同时知道调用 rational 的构造函数将 int 转换成一个合适的 rational，所以才有上面成功的调用（见条款 m19）。换句话说，编译器处理这个调用时的情形类似下面这样：

```
const rational temp(2);      // 从 2 产生一个临时
                             // rational 对象

result = onehalf * temp;     // 同 onehalf.operator*(temp);
```

当然，只有所涉及的构造函数没有声明为 explicit 的情况下才会这样，因为 explicit 构造函数不能用于隐式转换，这正是 explicit 的含义。如果 rational 象下面这样定义：

```
class rational {
public:
  explicit rational(int numerator = 0,     // 此构造函数为
                    int denominator = 1);  // explicit
  ...

  const rational operator*(const rational& rhs) const;
```

```
  ...

};
```

那么，下面的语句都不能通过编译：

```
result = onehalf * 2;          // 错误!
result = 2 * onehalf;          // 错误!
```

这不会为混合运算提供支持，但至少两条语句的行为一致了。

然而，我们刚才研究的这个类是要设计成可以允许固定类型到 rational 的隐式转换的——这就是为什么 rational 的构造函数没有声明为 explicit 的原因。这样，编译器将执行必要的隐式转换使上面 result 的第一个赋值语句通过编译。实际上，如果需要的话，编译器会对每个函数的每个参数执行这种隐式类型转换。但它只对函数参数表中列出的参数进行转换，决不会对成员函数所在的对象（即，成员函数中的*this 指针所对应的对象）进行转换。这就是为什么这个语句可以工作：

```
result = onehalf.operator*(2);      // converts int -> rational
```

而这个语句不行：

```
result = 2.operator*(onehalf);      // 不会转换
                                    // int -> rational
```

第一种情形操作的是列在函数声明中的一个参数，而第二种情形不是。

尽管如此，你可能还是想支持混合型的算术操作，而实现的方法现在应该清楚了：使 operator*成为一个非成员函数，从而允许编译器对所有的参数执行隐式类型转换：

```
class rational {

  ...                           // contains no operator*

};

// 在全局或某一名字空间声明，
// 参见条款 m20 了解为什么要这么做
const rational operator*(const rational& lhs,
                         const rational& rhs)
{
  return rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

```
rational onefourth(1, 4);
rational result;

result = onefourth * 2;          // 工作良好
result = 2 * onefourth;          // 万岁，它也工作了!
```

这当然是一个完美的结局，但还有一个担心：operator*应该成为 rational 类的友元吗？

这种情况下，答案是不必要。因为 operator*可以完全通过类的公有（public）接口来实现。上面的代码就是这么做的。只要能避免使用友元函数就要避免，因为，和现实生活中差不多，友元（朋友）带来的麻烦往往比它（他/她）对你的帮助多。

然而，很多情况下，不是成员的函数从概念上说也可能是类接口的一部分，它们需要访问类的非公有成员的情况也不少。

让我们回头再来看看本书那个主要的例子，string 类。如果想重载 operator>>和 operator<<来读写 string 对象，你会很快发现它们不能是成员函数。如果是成员函数的话，调用它们时就必须把 string 对象放在它们的左边：

```
// 一个不正确地将 operator>>和
// operator<<作为成员函数的类
class string {
public:
  string(const char *value);

  ...

  istream& operator>>(istream& input);
  ostream& operator<<(ostream& output);

private:
  char *data;
};

string s;

s >> cin;                // 合法，但
                         // 有违常规

s << cout;               // 同上
```

这会把别人弄糊涂。所以这些函数不能是成员函数。注意这种情况和前面的不同。这里的目标是自然的调用语法，前面关心的是隐式类型转换。

所以，如果来设计这些函数，就象这样：

```
istream& operator>>(istream& input, string& string)
{
  delete [] string.data;

  read from input into some memory, and make string.data
  point to it

  return input;
}

ostream& operator<<(ostream& output,
                    const string& string)
{
  return output << string.data;
}
```

注意上面两个函数都要访问 string 类的 data 成员，而这个成员是私有（private）的。但我们已经知道，这个函数一定要是非成员函数。这样，就别无选择了：需要访问非公有成员的非成员函数只能是类的友元函数。

本条款得出的结论如下。假设 f 是想正确声明的函数，c 是和它相关的类：

·虚函数必须是成员函数。如果 f 必须是虚函数，就让它成为 c 的成员函数。

·operator>>和 operator<<决不能是成员函数。如果 f 是 operator>>或 operator<<，让 f 成为非成员函数。如果 f 还需要访问 c 的非公有成员，让 f 成为 c 的友元函数。

·只有非成员函数对最左边的参数进行类型转换。如果 f 需要对最左边的参数进行类型转换，让 f 成为非成员函数。如果 f 还需要访问 c 的非公有成员，让 f 成为 c 的友元函数。

·其它情况下都声明为成员函数。如果以上情况都不是，让 f 成为 c 的成员函数。

条款 20: 避免 public 接口出现数据成员

首先，从“一致性”的角度来看这个问题。如果 public 接口里都是函数，用户每次访问类的成员时就用不着抓脑袋去想：是该用括号还是不该用括号呢？——用括号就是了！因为每个成员都是函数。一生中，这可以避免你多少次抓脑袋啊！

你不买“一致性”的帐？那你总得承认采用函数可以更精确地控制数据成员的访问权这一事实吧？如果使数据成员为 public，每个人都可以对它读写；如果用函数来获取或设定它的值，就可以实现禁止访问、只读访问和读写访问等多种控制。甚至，如果你愿意，还可以实现只写访问：

```
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;            // 禁止访问这个 int

  int readonly;            // 可以只读这个 int

  int readwrite;           // 可以读/写这个 int

  int writeonly;           // 可以只写这个 int
};
```

还没说服你？那只得搬出这门重型大炮：功能分离（functional abstraction）。如果用函数来实现对数据成员的访问，以后就有可能用一段计算来取代这个数据成员，而使用这个类的用户却一无所知。

例如，假设写一个用自动化仪器检测汽车行驶速度的应用程序。每辆车行驶过来时，计算出的速度值添加到一个集中了当前所有的汽车速度数据的集合里：

```
class speeddatacollection {
public:
  void addvalue(int speed);       // 添加新速度值

  double averagesofar() const;    // 返回平均速度
};
```

现在考虑怎么实现成员函数 averagesofar（另见条款 m18）。一种方法是用类的一个数据成员来保存当前收集到的所有速度数据的运行平均值。只要 averagesofar 被调用，就返回这个数据成员的值。另一个不同的方法则是在 averagesofar 每次被调用时才通过检查集合中的所有的数据值计算出结果。（关于这两个方法的更全面的讨论参见条款 m17 和 m18。）

第一种方法——保持一个运行值——使得每个 speeddatacollection 对象更大，因为必须为保存运行值的数据成员分配空间。但 averagesofar 实现起来很高效：它可以是一个仅用返回数据成员值的内联函数（见条款 33）。相反，每次调用时都要计算平均值的方案则使得 averagesofar 运行更慢，但每个 speeddatacollection 对象会更小。

谁能说哪个方法更好？在内存很紧张的机器里，或在不是频繁需要平均值的应用程序里，每次计算平均值是个好方案。在频繁需要平均值的应用程序里，速度是最根本的，内存不是主要问题，保持一个运行值的方法更可取。重要之处在于，用成员函数来访问平均值，就可以使用任何一种方法，它具有极大价值的灵活性，这是那个在 public 接口里包含平均值数据成员的方案所不具有的。

所以，结论是，在 public 接口里放上数据成员无异于自找麻烦，所以要把数据成员安全地隐藏在与功能分离的高墙后。如果现在就开始这么做，那我们就可以无需任何代价地换来一致性和精确的访问控制。

条款 21: 尽可能使用 const

使用 const 的好处在于它允许指定一种语意上的约束——某种对象不能被修改——编译器具体来实施这种约束。通过 const，你可以通知编译器和其他程序员某个值要保持不变。只要是这种情况，你就要明确地使用 const ，因为这样做就可以借助编译器的帮助确保这种约束不被破坏。

const 关键字实在是神通广大。在类的外面，它可以用于全局或名字空间常量（见条款 1 和 47），以及静态对象（某一文件或程序块范围内的局部对象）。在类的内部，它可以用于静态和非静态成员（见条款 12）。

对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或二者同时指定为 const，还有，两者都不指定为 const：

```
char *p              = "hello";         //  非 const 指针,
                                        //  非 const 数据

const char *p        = "hello";         //  非 const 指针,
                                        // const 数据

char * const p       = "hello";         // const 指针,
                                        //  非 const 数据

const char * const p = "hello";         // const 指针,
                                        // const 数据
```

语法并非看起来那么变化多端。一般来说，你可以在头脑里画一条垂直线穿过指针声明中的星号（*）位置，如果 const 出现在线的左边，指针指向的数据为常量；如果 const 出现在线的右边，指针本身为常量；如果 const 在线的两边都出现，二者都是常量。

在指针所指为常量的情况下，有些程序员喜欢把 const 放在类型名之前，有些程序员则喜欢把 const 放在类型名之后、星号之前。所以，下面的函数取的是同种参数类型：

```
class widget { ... };

void f1(const widget *pw);      // f1 取的是指向
                                // widget 常量对象的指针

void f2(widget const *pw);      //  同 f2
```

因为两种表示形式在实际代码中都存在，所以要使自己对这两种形式都习惯。

const 的一些强大的功能基于它在函数声明中的应用。在一个函数声明中，const 可以指的是函数的返回值，或某个参数；对于成员函数，还可以指的是整个函数。

让函数返回一个常量值经常可以在不降低安全性和效率的情况下减少用户出错的几率。实际上正如条款 29 所说明的，对返回值使用 const 有可能提高一个函数的安全性和效率，否则还会出问题。

例如，看这个在条款 19 中介绍的有理数的 operator*函数的声明：

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

很多程序员第一眼看到它会纳闷：为什么 operator*的返回结果是一个 const 对象？因为如果不是这样，用户就可以做下面这样的坏事：

```
rational a, b, c;

...

(a * b) = c;      // 对 a*b 的结果赋值
```

我不知道为什么有些程序员会想到对两个数的运算结果直接赋值，但我却知道：如果 a，b 和 c 是固定类型，这样做显然是不合法的。一个好的用户自定义类型的特征是，它会避免那种没道理的与固定类型不兼容的行为。对我来说，对两个数的运算结果赋值是非常没道理的。声明 operator*的返回值为 const 可以防止这种情况，所以这样做才是正确的。

关于 const 参数没什么特别之处要强调——它们的运作和局部 const 对象一样。（但，见条款 m19，const 参数会导致一个临时对象的产生）然而，如果成员函数为 const，那就是另一回事了。

const 成员函数的目的当然是为了指明哪个成员函数可以在 const 对象上被调用。但很多人忽视了这样一个事实：仅在 const 方面有不同的成员函数可以重载。这是 c++的一个重要特性。再次看这个 string 类：

```
class string {
public:

  ...

  // 用于非 const 对象的 operator[]
  char& operator[](int position)
  { return data[position]; }

  // 用于 const 对象的 operator[]
  const char& operator[](int position) const
  { return data[position]; }
```

```
private:
  char *data;
};

string s1 = "hello";
cout << s1[0];              // 调用非 const
                            // string::operator[]
const string s2 = "world";
cout << s2[0];              // 调用 const
                            // string::operator[]
```

通过重载 operator[]并给不同版本不同的返回值，就可以对 const 和非 const string 进行不同的处理：

```
string s = "hello";          // 非 const string 对象

cout << s[0];                // 正确——读一个
                             // 非 const string

s[0] = 'x';                  // 正确——写一个
                             // 非 const string

const string cs = "world";   // const string 对象

cout << cs[0];               // 正确——读一个
                             // const string

cs[0] = 'x';                 // 错误!——写一个
                             // const string
```

另外注意，这里的错误只和调用 operator[]的返回值有关；operator[]调用本身没问题。 错误产生的原因在于企图对一个 const char&赋值，因为被赋值的对象是 const 版本的 operator[]函数的返回值。

还要注意，非 const operator[]的返回类型必须是一个 char 的引用——char 本身则不行。如果 operator[]真的返回了一个简单的 char，如下所示的语句就不会通过编译：

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

因为，修改一个“返回值为固定类型”的函数的返回值绝对是不合法的。即使合法，由于 c++“通过值（而不是引用）来返回对象”（见条款 22）的内部机制的原因，s.data[0]的一个拷贝会被修改，而不是 s.data[0]自己，这就不是你所想要的结果了。

让我们停下来看一个基本原理。一个成员函数为 const 的确切含义是什么？有两种主要的看法：数据意义上的 const（bitwise constness）和概念意义上的 const（conceptual constness）。

bitwise constness 的坚持者认为，当且仅当成员函数不修改对象的任何数据成员（静态数据成员除外）时，即不修改对象中任何一个比特(bit)时，这个成员函数才是 const 的。bitwise constness 最大的好处是可以很容易地检测到违反 bitwise constness 规定的事件：编译器只用去寻找有无对数据成员的赋值就可以了。实际上，bitwise constness 正是 c++对 const 问题的定义，const 成员函数不被允许修改它所在对象的任何一个数据成员。

不幸的是，很多不遵守 bitwise constness 定义的成员函数也可以通过 bitwise 测试。特别是，一个“修改了指针所指向的数据”的成员函数，其行为显然违反了 bitwise constness 定义，但如果对象中仅包含这个指针，这个函数也是 bitwise const 的，编译时会通过。这就和我们的直觉有差异：

```
class string {
public:
  // 构造函数，使 data 指向一个
  // value 所指向的数据的拷贝
  string(const char *value);

  ...

  operator char *() const { return data;}

private:
  char *data;
};

const string s = "hello";      // 声明常量对象

char *nasty = s;              // 调用  operator char*() const

*nasty = 'm';                 // 修改 s.data[0]

cout << s;                    // 输出"mello"
```

显然，在用一个值创建一个常量对象并调用对象的 const 成员函数时一定有什么错误，对象的值竟然可以修改！（关于这个例子更详细的讨论参见条款 29）

这就导致 conceptual constness 观点的引入。此观点的坚持者认为，一个 const 成员函数可以修改它所在对象的一些数据（bits），但只有在用户不会发觉的情况下。例如，假设 string 类想保存对象每次被请求时数据的长度：

```
class string {
public:
  // 构造函数，使 data 指向一个
  // value 所指向的数据的拷贝
  string(const char *value): lengthisvalid(false) { ... }
```

```
  ...

  size_t length() const;

private:
  char *data;

  size_t datalength;          // 最后计算出的
                              // string 的长度

  bool lengthisvalid;         // 长度当前
                              // 是否合法
};

size_t string::length() const
{
  if (!lengthisvalid) {
    datalength = strlen(data); // 错误!
    lengthisvalid = true;      // 错误!
  }

  return datalength;
}
```

这个 length 的实现显然不符合“bitwise const”的定义——datalength 和 lengthisvalid 都可以修改——但对 const string 对象来说，似乎它一定要是合法的才行。但编译器也不同意， 它们坚持“bitwise constness”，怎么办？

解决方案很简单：利用 c++标准组织针对这类情况专门提供的有关 const 问题的另一个可选方案。此方案使用了关键字 mutable，当对非静态数据成员运用 mutable 时，这些成员的“bitwise constness”限制就被解除：

```
class string {
public:

  ...    // same as above

private:
  char *data;

  mutable size_t datalength;      // 这些数据成员现在
                                  // 为 mutable；他们可以在
  mutable bool lengthisvalid;     // 任何地方被修改，即使
                                  // 在 const 成员函数里
};
```

```
size_t string::length() const
{
  if (!lengthisvalid) {
    datalength = strlen(data);    // 现在合法
    lengthisvalid = true;         // 同样合法
  }

  return datalength;
}
```

mutable 在处理“bitwise-constness 限制”问题时是一个很好的方案，但它被加入到 c++标准中的时间不长，所以有的编译器可能还不支持它。如果是这样，就不得不倒退到 c++黑暗的旧时代去，在那儿，生活很简陋，const 有时可能会被抛弃。

类 c 的一个成员函数中，this 指针就好象经过如下的声明：

```
c * const this;              // 非 const 成员函数中

const c * const this;        // const 成员函数中
```

这种情况下（即编译器不支持 mutable 的情况下），如果想使那个有问题的 string::length 版本对 const 和非 const 对象都合法，就只有把 this 的类型从 const c * const 改成 c * const。不能直接这么做，但可以通过初始化一个局部变量指针，使之指向 this 所指的同一个对象来间接实现。然后，就可以通过这个局部指针来访问你想修改的成员：

```
size_t string::length() const
{
  // 定义一个不指向 const 对象的
  // 局部版本的 this 指针
  string * const localthis =
    const_cast<string * const>(this);

  if (!lengthisvalid) {
    localthis->datalength = strlen(data);
    localthis->lengthisvalid = true;
  }

  return datalength;
}
```

做的不是很漂亮。但为了完成想要的功能也就只有这么做。

当然，如果不能保证这个方法一定可行，就不要这么做：比如，一些老的“消除 const”的方法就不行。特别是，如果 this 所指的对象真的是 const，即，在定义时被声明为 const，那

么，“消除 const”就会导致不可确定的后果。所以，如果想在成员函数中通过转换消除 const，就最好先确信你要转换的对象最初没有被定义为 const。

还有一种情况下，通过类型转换消除 const 会既有用又安全。这就是：将一个 const 对象传递到一个取非 const 参数的函数中，同时你又知道参数不会在函数内部被修改的情况时。第二个条件很重要，因为对一个只会被读的对象（不会被写）消除 const 永远是安全的，即使那个对象最初曾被定义为 const。

例如，已经知道有些库不正确地声明了象下面这样的 strlen 函数：

```
size_t strlen(char *s);
```

strlen 当然不会去修改 s 所指的数据——至少我一辈子没看见过。但因为有了这个声明，对一个 const char *类型的指针调用这个函数时就会不合法。为解决这个问题，可以在给 strlen 传参数时安全地把这个指针的 const 强制转换掉：

```
const char *klingongreeting = "nuqneh"; // "nuqneh"即"hello"
                                        //
size_t length =
  strlen(const_cast<char*>(klingongreeting));
```

但不要滥用这个方法。只有在被调用的函数（比如本例中的 strlen）不会修改它的参数所指的数据时，才能保证它可以正常工作。

条款 22: 尽量用“传引用”而不用“传值”

c 语言中，什么都是通过传值来实现的，c++继承了这一传统并将它作为默认方式。除非明确指定，函数的形参总是通过“实参的拷贝”来初始化的，函数的调用者得到的也是函数返回值的拷贝。

正如我在本书的导言中所指出的，“通过值来传递一个对象”的具体含义是由这个对象的类的拷贝构造函数定义的。这使得传值成为一种非常昂贵的操作。例如，看下面这个（只是假想的）类的结构：

```
class person {
public:
  person();                    // 为简化，省略参数
                               //
  ~person();

  ...

private:
  string name, address;
};

class student: public person {
public:
  student();                   // 为简化，省略参数
                               //
  ~student();

  ...

private:
  string schoolname, schooladdress;
};
```

现在定义一个简单的函数 returnstudent，它取一个 student 参数（通过值）然后立即返回它（也通过值）。定义完后，调用这个函数：

```
student returnstudent(student s) { return s; }

student plato;                 // plato（柏拉图）在
                               // socrates（苏格拉底）门下学习

returnstudent(plato);          // 调用 returnstudent
```

这个看起来无关痛痒的函数调用过程，其内部究竟发生了些什么呢？

简单地说就是：首先，调用了 student 的拷贝构造函数用以将 s 初始化为 plato；然后再次调用 student 的拷贝构造函数用以将函数返回值对象初始化为 s；接着，s 的析构函数被调用；最后，returnstudent 返回值对象的析构函数被调用。所以，这个什么也没做的函数的成本是两个 student 的拷贝构造函数加上两个 student 析构函数。

但没完，还有！student 对象中有两个 string 对象，所以每次构造一个 student 对象时必须也要构造两个 string 对象。student 对象还是从 person 对象继承而来的，所以每次构造一个 student 对象时也必须构造一个 person 对象。一个 person 对象内部有另外两个 string 对象，所以每个 person 的构造也必然伴随另两个 string 的构造。所以，通过值来传递一个 student 对象最终导致调用了一个 student 拷贝构造函数，一个 person 拷贝构造函数，四个 string 拷贝构造函数。当 student 对象被摧毁时，每个构造函数对应一个析构函数的调用。所以，通过值来传递一个 student 对象的最终开销是六个构造函数和六个析构函数。因为 returnstudent 函数使用了两次传值（一次对参数，一次对返回值），这个函数总共调用了十二个构造函数和十二个析构函数！

在 c++编译器的设计者眼里，这是最糟糕的情况。编译器可以用来消除一些对拷贝构造函数的调用（c++标准——见条款 50——描述了具体在哪些条件下编译器可以执行这类的优化工作，条款 m20 给出了例子）。一些编译器也这样做了。但在不是所有编译器都普遍这么做的情况下，一定要对通过值来传递对象所造成的开销有所警惕。

为避免这种潜在的昂贵的开销，就不要通过值来传递对象，而要通过引用：

```
const student& returnstudent(const student& s)
{ return s; }
```

这会非常高效：没有构造函数或析构函数被调用，因为没有新的对象被创建。

通过引用来传递参数还有另外一个优点：它避免了所谓的“切割问题（slicing problem）”。当一个派生类的对象作为基类对象被传递时，它（派生类对象）的作为派生类所具有的行为特性会被“切割”掉，从而变成了一个简单的基类对象。这往往不是你所想要的。例如，假设设计这么一套实现图形窗口系统的类：

```
class window {
public:
  string name() const;            // 返回窗口名
  virtual void display() const;   // 绘制窗口内容
};

class windowwithscrollbars: public window {
public:
  virtual void display() const;
};
```

每个 window 对象都有一个名字，可以通过 name 函数得到；每个窗口都可以被显示，着可以通过调用 display 函数实现。display 声明为 virtual 意味着一个简单的 window 基类对象被

显示的方式往往和价格昂贵的 windowwithscrollbars 对象被显示的方式不同（见条款 36，37，m33）。

现在假设写一个函数来打印窗口的名字然后显示这个窗口。下面是一个用错误的方法写出来的函数：

```
// 一个受“切割问题”困扰的函数
void printnameanddisplay(window w)
{
  cout << w.name();
  w.display();
}
```

想象当用一个 windowwithscrollbars 对象来调用这个函数时将发生什么：

```
windowwithscrollbars wwsb;

printnameanddisplay(wwsb);
```

参数 w 将会作为一个 windows 对象而被创建（它是通过值来传递的，记得吗？），所有 wwsb 所具有的作为 windowwithscrollbars 对象的行为特性都被“切割”掉了。printnameanddisplay 内部，w 的行为就象是一个类 window 的对象（因为它本身就是一个 window 的对象），而不管当初传到函数的对象类型是什么。尤其是，printnameanddisplay 内部对 display 的调用总是 window::display，而不是 windowwithscrollbars::display。

解决切割问题的方法是通过引用来传递 w：

```
// 一个不受“切割问题”困扰的函数
void printnameanddisplay(const window& w)
{
  cout << w.name();
  w.display();
}
```

现在 w 的行为就和传到函数的真实类型一致了。为了强调 w 虽然通过引用传递但在函数内部不能修改，就要采纳条款 21 的建议将它声明为 const。

传递引用是个很好的做法，但它会导致自身的复杂性，最大的一个问题就是别名问题，这在条款 17 进行了讨论。另外，更重要的是，有时不能用引用来传递对象，参见条款 23。最后要说的是，引用几乎都是通过指针来实现的，所以通过引用传递对象实际上是传递指针。因此，如果是一个很小的对象——例如 int——传值实际上会比传引用更高效。

条款 23: 必须返回一个对象时不要试图返回一个引用

据说爱因斯坦曾提出过这样的建议：尽可能地让事情简单，但不要过于简单。在 c++语言中相似的说法应该是：尽可能地使程序高效，但不要过于高效。

一旦程序员抓住了“传值”在效率上的把柄（参见条款 22），他们会变得十分极端，恨不得挖出每一个隐藏在程序中的传值操作。岂不知，在他们不懈地追求纯粹的“传引用”的过程中，他们会不可避免地犯另一个严重的错误：传递一个并不存在的对象的引用。这就不是好事了。

看一个表示有理数的类，其中包含一个友元函数，用于两个有理数相乘：

```
class rational {
public:
  rational(int numerator = 0, int denominator = 1);

  ...

private:
  int n, d;             // 分子和分母

friend
  const rational                    // 参见条款 21：为什么
    operator*(const rational& lhs,    // 返回值是 const
              const rational& rhs)
};

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

很明显，这个版本的 operator*是通过传值返回对象结果，如果不去考虑对象构造和析构时的开销，你就是在逃避作为一个程序员的责任。另外一件很明显的事实是，除非确实有必要，否则谁都不愿意承担这样一个临时对象的开销。那么，问题就归结于：确实有必要吗？

答案是，如果能返回一个引用，当然就没有必要。但请记住，引用只是一个名字，一个其它某个已经存在的对象的名字。无论何时看到一个引用的声明，就要立即问自己：它的另一个名字是什么呢？因为它必然还有另外一个什么名字（见条款 m1）。拿 operator*来说，如果函数要返回一个引用，那它返回的必须是其它某个已经存在的 rational 对象的引用，这个对象包含了两个对象相乘的结果。

但，期望在调用 operator*之前有这样一个对象存在是没道理的。也就是说，如果有下面的代码：

```
rational a(1, 2);              // a = 1/2
rational b(3, 5);              // b = 3/5
rational c = a * b;             // c 为 3/10
```

期望已经存在一个值为 3/10 的有理数是不现实的。如果 operator* 一定要返回这样一个数的引用，就必须自己创建这个数的对象。

一个函数只能有两种方法创建一个新对象：在堆栈里或在堆上。在堆栈里创建对象时伴随着一个局部变量的定义，采用这种方法，就要这样写 operator*：

```
// 写此函数的第一个错误方法
inline const rational& operator*(const rational& lhs,
                                 const rational& rhs)
{
  rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

这个方法应该被否决，因为我们的目标是避免构造函数被调用，但 result 必须要象其它对象一样被构造。另外，这个函数还有另外一个更严重的问题，它返回的是一个局部对象的引用，关于这个错误，条款 31 进行了深入的讨论。

那么，在堆上创建一个对象然后返回它的引用呢？基于堆的对象是通过使用 new 产生的，所以应该这样写 operator*：

```
// 写此函数的第二个错误方法
inline const rational& operator*(const rational& lhs,
                                 const rational& rhs)
{
  rational *result =
    new rational(lhs.n * rhs.n, lhs.d * rhs.d);
  return *result;
}
```

首先，你还是得负担构造函数调用的开销，因为 new 分配的内存是通过调用一个适当的构造函数来初始化的（见条款 5 和 m8）。另外，还有一个问题：谁将负责用 delete 来删除掉 new 生成的对象呢？

实际上，这绝对是一个内存泄漏。即使可以说服 operator*的调用者去取函数返回值地址，然后用 delete 去删除它（绝对不可能——条款 31 展示了这样的代码会是什么样的），但一些复杂的表达式会产生没有名字的临时值，程序员是不可能得到的。例如：

```
rational w, x, y, z;

w = x * y * z;
```

两个对 operator*的调用都产生了没有名字的临时值，程序员无法看到，因而无法删除。（再次参见条款 31）

也许，你会想你比一般的熊——或一般的程序员——要聪明；也许，你注意到在堆栈和堆上创建对象的方法避免不了对构造函数的调用；也许，你想起了我们最初的目标是为了避免这种对构造函数的调用；也许，你有个办法可以只用一个构造函数来搞掂一切；也许，你的眼前出现了这样一段代码：operator*返回一个“在函数内部定义的静态 rational 对象”的引用：

```
// 写此函数的第三个错误方法
inline const rational& operator*(const rational& lhs,
                                 const rational& rhs)
{
  static rational result;      // 将要作为引用返回的
                               // 静态对象

  lhs 和 rhs 相乘，结果放进 result；

  return result;
}
```

这个方法看起来好象有戏，虽然在实际实现上面的伪代码时你会发现，不调用一个 rational 构造函数是不可能给出 result 的正确值的，而避免这样的调用正是我们要谈论的主题。就算你实现了上面的伪代码，但，你再聪明也不能最终挽救这个不幸的设计。

想知道为什么，看看下面这段写得很合理的用户代码：

```
bool operator==(const rational& lhs,      // rationals 的 operator==
                const rational& rhs);     //

rational a, b, c, d;

...

if ((a * b) == (c * d)) {

  处理相等的情况;

} else {

  处理不相等的情况;

}
```

看出来了吗？((a*b) == (c*d)) 会永远为 true，不管 a，b，c 和 d 是什么值！

用等价的函数形式重写上面的相等判断语句就很容易明白发生这一可恶行为的原因了：

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

注意当 operator==被调用时，总有两个 operator*刚被调用，每个调用返回 operator*内部的静态 rational 对象的引用。于是，上面的语句实际上是请求 operator==对“operator*内部的静态 rational 对象的值”和“operator*内部的静态 rational 对象的值”进行比较，这样的比较不相等才怪呢！

幸运的话，我以上的说明应该足以说服你：想“在象 operator*这样的函数里返回一个引用”实际上是在浪费时间。但我没幼稚到会相信幸运总会光临自己。一些人——你们知道这些人是指谁——此刻会在想，“唔，上面那个方法，如果一个静态变量不够用，也许可以用一个静态数组……”

请就此打住！我们难道还没受够吗？

我不能让自己写一段示例代码来太高这个设计，因为即使只抱有上面这种想法都足以令人感到羞愧。首先，你必须选择一个 n，指定数组的大小。如果 n 太小，就会没地方储存函数返回值，这和我们前面否定的那个“采用单个静态变量的设计”相比没有什么改进。如果 n 太大，就会降低程序的性能，因为函数第一次被调用时数组中每个对象都要被创建。这会带来 n 个构造函数和 n 个析构函数的开销，即使这个函数只被调用一次。如果说"optimization"（最优化）是指提高软件的性能的过程， 那么现在这种做法简直可以称为"pessimization"（最差化）。最后，想想怎么把需要的值放到数组的对象中以及需要多大的开销？在对象间传值的最直接的方法是通过赋值，但赋值的开销又有多大呢？一般来说，它相当于调用一个析构函数（摧毁旧值）再加上调用一个构造函数（拷贝新值）。但我们现在的目标正是为了避免构造和析构的开销啊！面对现实吧：这个方法也绝对不能选用。

所以，写一个必须返回一个新对象的函数的正确方法就是让这个函数返回一个新对象。对于 rational 的 operator*来说，这意味着要不就是下面的代码（就是最初看到的那段代码），要不就是本质上和它等价的代码：

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

的确，这会导致“operator*的返回值构造和析构时带来的开销”，但归根结底它只是用小的代价换来正确的程序运行行为而已。况且，你所担心的开销还有可能永远不会出现：和所有程序设计语言一样，c++允许编译器的设计者采用一些优化措施来提高所生成的代码的性能，所以，在有些场合，operator*的返回值会被安全地除去（见条款 m20）。当编译器采用了这种优化时（当前大部分编译器这么做），程序和以前一样继续工作，只不过是运行速度比你预计的要快而已。

以上讨论可以归结为：当需要在返回引用和返回对象间做决定时，你的职责是选择可以完成正确功能的那个。至于怎么让这个选择所产生的代价尽可能的小，那是编译器的生产商去想的事。

条款 24: 在函数重载和设定参数缺省值间慎重选择

会对函数重载和设定参数缺省值产生混淆的原因在于，它们都允许一个函数以多种方式被调用：

```
void f();   // f 被重载
void f(int x);
f();        //  调用 f()
f(10);      //  调用 f(int)
void g(int x = 0);     // g  有一个
                       //  缺省参数值
g();        //  调用 g(0)
g(10);      //  调用 g(10)
```

那么，什么时候该用哪种方法呢？

答案取决于另外两个问题。第一，确实有那么一个值可以作为缺省吗？第二，要用到多少种算法？一般来说，如果可以选择一个合适的缺省值并且只是用到一种算法，就使用缺省参数（参见条款 38）。否则，就使用函数重载。

下面是一个最多可以计算五个 int 的最大值的函数。这个函数使用了——深呼一口气，看清楚啦——std::numeric_limits<int>::min()，作为缺省参数值。等会儿再进一步介绍这个值，这里先给出函数的代码：

```
int max(int a,
        int b = std::numeric_limits<int>::min(),
        int c = std::numeric_limits<int>::min(),
        int d = std::numeric_limits<int>::min(),
        int e = std::numeric_limits<int>::min())
{
 int temp = a > b ? a : b;
 temp = temp > c ? temp : c;
 temp = temp > d ? temp : d;
 return temp > e ? temp : e;
}
```

现在可以放松了。std::numeric_limits<int>::min()是 c++标准库用一种特有的新方法所表示的一个在 c 里已经定义了的东西，即 c 在<limits.h>中定义的 int_min 宏所表示的那个东西

——处理你的 c++原代码的编译器所产生的 int 的最小可能值。是的，它的句法背离了 c 所具有的简洁，但在那些冒号以及其它奇怪的句法背后，是有道理可循的。

假设想写一个函数模板，其参数为固定数字类型，模板产生的函数可以打印用“实例化类型”表示的最小值。这个模板可以这么写：

```
template<class t>
void printminimumvalue()
{
  cout <<  表示为 t 类型的最小值;
}
```

如果只是借助<limits.h>和<float.h>来写这个函数会觉得很困难，因为不知道 t 是什么，所以不知道该打印 int_min 还是 dbl_min，或其它什么类型的值。

为避开这些困难，标准 c++库（见条款 49）在头文件<limits>

中定义了一个类模板 numeric_limits，这个类模板本身也定义了一些静态成员函数。每个函数返回的是“实例化这个模板的类型”的信息。也就是说，numeric_limits<int>中的函数返回的信息是关于类型 int 的，numeric_limits<double>

中的函数返回的信息是关于类型 double 的。numeric_limits 中有一个函数叫 min，min 返回可表示为“实例化类型”的最小值，所以 numeric_limits<int>::min()返回的是代表整数类型的最小值。

有了 numeric_limits（和标准库中其它东西一样，numeric_limits 存在于名字空间 std 中；numeric_limits 本身在头文件<limits>中），写 printminimumvalue 就可以象下面这样容易：

```
template<class t>
void printminimumvalue()
{
  cout << std::numeric_limits<t>::min();
}
```

采用基于 numeric_limits 的方法来表示“类型相关常量”看起来开销很大，其实不然。因为原代码的冗长的语句不会反映到生成的目标代码中。实际上，对 numeric_limits 的调用根本就

不产生任何指令。想知道怎么回事，看看下面，这是 numeric_limits<int>::min 的一个很简单的实现：

```
#include
namespace std {
  inline int numeric_limits<int>::min() throw ()
  { return INT_MIN; }
}
```

因为此函数声明为 inline，对它的调用会被函数体代替（见条款 33）。它只是个 INT_MIN，也就是说，它本身仅仅是个简单的“实现时定义的常量”的#define。所以即使本条款开头的那个 max 函数看起来好象对每个缺省参数进行了函数调用，其实只不过是用了另一种聪明的方法来表示一个类型相关常量而已（本例中常量值为 INT_MIN）。象这样一些高效巧妙的应用在 c++标准库里俯拾皆是，这可以参考条款 49。

回到 max 函数上来：最关键的一点是，不管函数的调用者提供几个参数，max 计算时采用的是相同（效率很低）的算法。在函数内部任何地方都不用在意哪些参数是“真”的，哪些是缺省值；而且，所选用的缺省值不可能影响到所采用的算法计算的正确性。这就是使用缺省参数值的方案可行的原因。

对很多函数来说，会找不到合适的缺省值。例如，假设想写一个函数来计算最多可达 5 个 int 的平均值。这里就不能用缺省参数，因为函数的结果取决于传入的参数的个数：如果传入 3 个值，就要将总数除以 3；如果传入 5 个值，就要将总数除以 5。另外，假如用户没有提供某个参数时，没有一个“神奇的数字”可以作为缺省值，因为所有可能的 int 都可以是有效参数。这种情况下就别无选择：必须重载函数：

```
double avg(int a);
double avg(int a, int b);
double avg(int a, int b, int c);
double avg(int a, int b, int c, int d);
double avg(int a, int b, int c, int d, int e);
```

另一种必须使用重载函数的情况是：想完成一项特殊的任务，但算法取决于给定的输入值。这种情况对于构造函数很常见：“缺省”构造函数是凭空（没有输入）构造一个对象，而拷贝构造函数是根据一个已存在的对象构造一个对象：

```
// 一个表示自然数的类
class natural {
public:
  natural(int initvalue);
  natural(const natural& rhs);

private:
  unsigned int value;

  void init(int initvalue);
  void error(const string& msg);
};

inline
void natural::init(int initvalue) { value = initvalue; }
natural::natural(int initvalue)
{
  if (initvalue > 0) init(initvalue);
  else error("illegal initial value");
}

inline natural::natural(const natural& x)
{ init(x.value); }
```

输入为 int 的构造函数必须执行错误检查，而拷贝构造函数不需要，所以需要两个不同的函数来实现，这就是重载。还请注意，两个函数都必须对新对象赋一个初值。这会导致在两个构造函数里出现重复代码，所以要写一个“包含有两个构造函数公共代码”的私有成员函数 init 来解决这个问题。这个方法——在重载函数中调用一个“为重载函数完成某些功能”的公共的底层函数——很值得牢记，因为它经常有用（见条款 12）。

条款 25: 避免对指针和数字类型重载

快速抢答：什么是“零”？

更明确地说，下面的代码会发生什么？

```
void f(int x);
void f(string *ps);

f(0);                                   // 调用 f(int)还是 f(string*)?
```

答案是，0 是一个 int——准确地说，一个字面上的整数常量——所以，“总是”f(int)被调用。这就是问题所在：因为不是所有的人总是希望它这样执行。这是 c++世界中特有的一种情况：当人们认为某个调用应该具有多义性时，编译器却不这么干。

如果能想办法用符号名（比如，null 表示 null 指针）来解决这类问题就好了，但实现起来比想象的要难得多。

```
void * const null = 0;              // 可能的 null 定义

f(0);                                   // 还是调用 f(int)
f(static_cast< string*>(null));  // 调用 f(string*)
f(static_cast< string*> (0));    // 调用 f(string*)
```

不过细想一下，用 null 来表示一个 void*常量的方法还是比最初要好一点，因为如果能保证只是用 null 来表示 null 指针的话，是可以避免歧义的：

```
f(0);                                   // 调用 f(int)
f(null);                  // 错误! — 类型不匹配
f(static_cast< string*>(null));  // 正确, 调用 f(string*)
```

至少现在已经把一个运行时的错误（对 0 调用了“错误的”f 函数）转移成了一个编译时的错误（传递一个 void*给 string*参数）。情况稍微有点改善（见条款 46），但需要进行类型转换还是令人讨厌。

如果想可耻地退回去求助于欲处理，你会发现它也解决不了问题，因为最明显的办法不外乎：

```
#define null 0
```

或

```
#define null ((void*) 0)
```

第一种办法只不过是字面上的 0，本质上还是一个整数常量（如果你记得的话，还是最初的问题）；第二种方法则又把你拉回到“传 void*指针给某种类型的指针”的麻烦中。

如果对类型转换的规则有研究，你就会知道，c++会认为“从 long int 0 到 null 指针的转换”和“从 long int 到 int 的转换”一样，没什么不妥的。所以可以利用这一点，将多义性引入到上面那个你可能认为有“int/指针”问题的地方：

```
#define null 0l                    // null 现在是一个 long int

void f(int x);
void f(string *p);

f(null);                   // 错误!——歧义
```

然而，当想重载 long int 和指针时，它又不起作用了：

```
#define null 0l

void f(long int x);    // 这个 f 现在的参数为 long
void f(string *p);

f(null);                   // 正确，调用 f(long int)
```

实际编程中，这比把 null 定义为 int 可能要安全，但它无非只是在转移问题，而不是消除问题。

这个问题可以消除，但需要使用 c++语言最新增加的一个特性：成员函数模板（往往简称为成员模板）。顾名思义，成员函数模板是在类的内部为类生成成员函数的模板。拿上面关于 null 的讨论来说，我们需要一个“对每一个 t 类型，运作起来都象 static_cast<t*>(0)表达式”的对象。即，使 null 成为一个“包含一个隐式类型转换运算符”的类的对象，这个类型转换运算符可以适用于每种可能的指针类型。这就需要很多转换运算符，但它们可以求助于 c++从成员模板生成：

```
// 一个可以产生 null 指针对象的类的第一步设计
class nullclass {
public:
  template<class T>                                    // 为所有类型的 t
    operator T*() const { return 0; }      // 产生 operator t*；
};                                                     // 每个函数返回一个
                                                       // null 指针
                                                       //

const nullclass null;                      // null 是类型 nullclass
                                                       // 的一个对象

void f(int x);                                         // 和以前一样

void f(string *p);                         // 同上
```

```
f(null);                                    // 将 null 转换为 string*,
                                                    // 然后调用 f(string*)
```

这是一个很好的初步设计，但还可以从几方面进行改进。第一，我们实际上只需要一个 nullclass 对象，所以给这个类一个名字没必要；我们只需要定义一个匿名类并使 null 成为这种类型。第二，既然我们是想让 null 可以转换为任何类型的指针，那就也要能够处理成员指针。这就需要定义第二个成员模板，它的作用是为所有的类 c 和所有的类型 t，将 0 转换为类型 t c::*（指向类 c 里类型为 t 的成员）。（如果你不懂成员指针，或者你从没听说过，或很少用，那也不要紧。成员指针可以称得上是稀有动物，是很少见，也许很多人从来没用过它。对此好奇的人可以参考条款 30，那儿对成员指针进行了较详细的讨论。）最后，要防止用户取 null 的地址，因为我们希望 null 的行为并不是象指针那样，而是要象指针的值，而指针的值（如 0x453ab002）是没有地址的。

所以，改进后的 null 的定义看起来就象这样：

```
const                                       // 这是一个 const 对象...
class {
public:
  template<class t>                         // 可以转换任何类型
    operator t*() const                     // 的 null 非成员指针
    { return 0; }                   //

  template<class t>                 // 可以转换任何类型
    operator t c::*() const         // 的 null 成员指针
    { return 0; }

private:
  void operator&() const;           // 不能取其地址
                                            // (见条款 27)

} null;                                     // 名字为 null
```

这就是所看到的真实的代码，虽然在实际编程中有可能想给类一个名字。如果不给名字，编译器里指向 null 类型的信息也确实很难理解。

成员模板的用法的另一个例子参见条款 m28。

重要的一点是，以上所有那些产生正确工作的 null 的设计方案，只有在你自己是调用者的时候才有意义。如果你是设计被调用函数的人，写这样一个给别人使用的 null 其实没有多大的用处，因为你不能强迫你的调用者去使用它。例如，即使为你的用户提供了上面开发的那个 null，你还是不能防止他们这样做：

```
f(0);
```

```
// 还是调用 f(int),
// 因为 0 还是 int
```

它还是和本条款最前面的出现的问题一样。

所以，作为重载函数的设计者，归根结底最基本的一条是，只要有可能，就要避免对一个数字和一个指针类型重载。

条款 26: 当心潜在的二义性

每个人都有思想。有些人相信自由经济学，有些人相信来生。有些人甚至相信 COBOL 是一种真正的程序设计语言。C++也有一种思想：它认为潜在的二义性不是一种错误。

这是潜在二义性的一个例子：

```
class
B;                      // 对类 B 提前声明
                        //
class A {
public:
  A(const B&);          // 可以从 B 构造而来的类 A
};

class B {
public:
  operator A() const;           // 可以从 A 转换而来的类 B
};
```

这些类的声明没一点错——他们可以在相同的程序中共存而没一点问题。但是，看看下面，当把这两个类结合起来使用，在一个输入参数为 A 的函数里实际传进了一个 B 的对象，这时将会发生什么呢？

```
void f(const A&);
B b;
f(b);                           // 错误!——二义
```

一看到对 f 的调用，编译器就知道它必须产生一个类型 A 的对象，即使它手上拿着的是一个类型 B 的对象。有两种都很好的方法来实现（见条款 M5）。一种方法是调用类 A 的构造函数，它以 b 为参数构造一个新的 A 的对象。另一种方法是调用类 B 里自定义的转换运算符，它将 b 转换成一个 A 的对象。因为这两个途径都一样可行，编译器拒绝从他们中选择一个。

当然，在没碰上二义的情况下，程序可以使用。这正是潜在的二义所具有的潜伏的危害性。它可以长时期地潜伏在程序里，不被发觉也不活动；一旦某一天某位不知情的程序员真的做了什么具有二义性的操作，混乱就会爆发。这导致有这样一种令人担心的可能：你发布了一个函数库，它可以在二义的情况下被调用，而你却不知道自己正在这么做。

另一种类似的二义的形式源于 C++语言的标准转换——甚至没有涉及到类：

```
void f(int);
void f(char);

double d = 6.02;
```

```
f(d);                          // 错误!——二义
```

d 是该转换成 int 还是 char 呢？两种转换都可行，所以编译器干脆不去做结论。幸运的是，可以通过显式类型转换来解决这个问题：

```
f(static_cast<int>(d));        // 正确，调用 f(int)
f(static_cast<char>(d));       // 正确，调用 f(char)
```

多继承（见条款 43）充满了潜在二义性的可能。最常发生的一种情况是当一个派生类从多个基类继承了相同的成员名时：

```
class Base1 {
public:
  int doIt();
};

class Base2
{
public:
  void doIt();
};

class Derived: public Base1   // Derived 没有声明
                      public Base2 {       // 一个叫做 doIt 的函数
...
};

Derived d;
d.doIt();                     // 错误!——二义
```

当类 Derived 继承两个具有相同名字的函数时，C++没有认为它有错，此时二义只是潜在的。然而，对 doIt 的调用迫使编译器面对这个现实，除非显式地通过指明函数所需要的基类来消除二义，函数调用就会出错：

```
d.Base1::doIt();      // 正确，调用 Base1::doIt
d.Base2::doIt();      // 正确，调用 Base2::doIt
```

这不会令很多人感到麻烦，但当看到上面的代码没有用到访问权限时，一些本来很安分的人会动起心眼想做些不安分的事：

```
class Base1 { ... };  // 同上
class Base2 {
private:
void doIt(); // 此函数现在为 private
};
```

```
class Derived: public Base1, public Base2
{ ... };                    // 同上

Derived d;
int i = d.doIt();           // 错误! — 还是二义!
```

对 doIt 的调用还是具有二义性，即使只有 Base1 中的函数可以被访问。另外，只有 Base1::doIt 返回的值可以用于初始化一个 int 这一事实也与之无关——调用还是具有二义性。如果想成功地调用，就必须指明想要的是哪个类的 doIt。

C++中有一些最初看起来会觉得很不直观的规定，现在就是这种情况。具体来说，为什么消除“对类成员的引用所产生的二义”时不考虑访问权限呢？有一个非常好的理由，它可以归结为：改变一个类成员的访问权限不应该改变程序的含义。

比如前面那个例子，假设它考虑了访问权限。于是表达式 d.doIt()决定调用 Base1::doIt，因为 Base2 的版本不能访问。现在假设 Base1 的 Doit 版本由 public 改为 protected，Base2 的版本则由 private 改为 public。

转瞬之间，同样的表达式 d.doIt()将导致另一个完全不同的函数调用，即使调用代码和被调用函数本身都没有被修改！这很不直观，编译器甚至无法产生一个警告。可见，不是象你当初所想的那样，对多继承的成员的引用要显式地消除二义性是有道理的。

既然写程序和函数库时有这么多不同的情况会产生潜在的二义性，那么，一个好的软件开发者该怎么做呢？最根本的是，一定要时时小心它。想找出所有潜在的二义性的根源几乎是不可能的，特别是当程序员将不同的独立开发的库结合起来使用时（见条款 28），但在了解了导致经常产生潜在二义性的那些情况后，你就可以在软件设计和开发中将它出现的可能性降到最低。

条款 27: 如果不想使用隐式生成的函数就要显式地禁止它

假设想写一个类模板 Array，它所生成的类除了可以进行上下限检查外，其它行为和 C++标准数组一样。设计中面临的一个问题是怎么禁止掉 Array 对象之间的赋值操作，因为对标准 C++数组来说赋值是不合法的：

```
double values1[10];
double values2[10];

values1 = values2;              // 错误!
```

对很多函数来说，这不是个问题。如果你不想使用某个函数，只用简单地不把它放进类中。然而，赋值运算符属于那种与众不同的成员函数，当你没有去写这个函数时，C++会帮你写一个（见条款 45）。那么，该怎么办呢？

方法是声明这个函数（operator=），并使之为 private。显式地声明一个成员函数，就防止了编译器去自动生成它的版本；使函数为 private，就防止了别人去调用它。

但是，这个方法还不是很安全，成员函数和友元函数还是可以调用私有函数，除非——如果你够聪明的话——不去定义（实现）这个函数。这样，当无意间调用了这个函数时，程序在链接时就会报错。

对于 Array 来说，模板的定义可以象这样开始：

```
template<class T>
class Array {
private:
  // 不要定义这个函数!
  Array& operator=(const Array& rhs);

  ...

};
```

现在，当用户试图对 Array 对象执行赋值操作时，编译器会不答应；当你自己无意间在成员或友元函数中调用它时，链接器会嗷嗷大叫。

不要因为这个例子就认为本条款只适用于赋值运算符。不是这样的。它适用于条款 45 所介绍的每一个编译器自动生成的函数。实际应用中，你会发现赋值和拷贝构造函数具有行为上的相似性（见条款 11 和 16），这意味着几乎任何时候当你想禁止它们其中的一个时，就也要禁止另外一个。

条款 28: 划分全局名字空间

全局空间最大的问题在于它本身仅有一个。在大的软件项目中，经常会有不少人把他们定义的名字都放在这个单一的空间中，从而不可避免地导致名字冲突。例如，假设 library1.h 定义了一些常量，其中包括：

```
const double lib_version = 1.204;
```

类似的，library2.h 也定义了：

```
const int lib_version = 3;
```

很显然，如果某个程序想同时包含 library1.h 和 library2.h 就会有问题。对于这类问题，你除了嘴里骂几句，或给作者发报复性邮件，或自己编辑头文件来消除名字冲突外，也没其它什么办法。

但是，作为程序员，你可以尽力使自己写的程序库不给别人带来这些问题。例如，可以预先想一些不大可能造成冲突的某种前缀，加在每个全局符号前。当然得承认，这样组合起来的标识符看起来不是那么令人舒服。

另一个比较好的方法是使用 c++ namespace。namespace 本质上和使用前缀的方法一样，只不过避免了别人总是看到前缀而已。所以，不要这么做：

```
const double sdmbook_version = 2.0;      // 在这个程序库中,
                                         // 每个符号以"sdm"开头
class sdmhandle { ... };

sdmhandle& sdmgethandle();               // 为什么函数要这样声明?
                                         // 参见条款 47
```

而要这么做：

```
namespace sdm {
  const double book_version = 2.0;
  class handle { ... };
  handle& gethandle();
}
```

用户于是可以通过三种方法来访问这一名字空间里的符号：将名字空间中的所有符号全部引入到某一用户空间；将部分符号引入到某一用户空间；或通过修饰符显式地一次性使用某个符号：

```
void f1()
{
```

```
  using namespace sdm;          // 使得 sdm 中的所有符号不用加
                                // 修饰符就可以使用

  cout << book_version;         // 解释为 sdm::book_version
  ...

  handle h = gethandle();       // handle 解释为 sdm::handle,
                                // gethandle 解释为 sdm::gethandle
  ...

}

void f2()
{
  using sdm::book_version;      // 使得仅 book_version 不用加
                                // 修饰符就可以使用

  cout << book_version;         // 解释为
                                // sdm::book_version
  ...

  handle h = gethandle();       // 错误! handle 和 gethandle
                                // 都没有引入到本空间
  ...

}

void f3()
{
  cout << sdm::book_version;    // 使得 book_version
                                // 在本语句有效
  ...

  double d = book_version;      // 错误! book_version
                                // 不在本空间

  handle h = gethandle();       // 错误! handle 和 gethandle
                                // 都没有引入到本空间
  ...

}
```

（有些名字空间没有名字。这种没命名的名字空间一般用于限制名字空间内部元素的可见性。详见条款 m31。）

名字空间带来的最大的好处之一在于：潜在的二义不会造成错误（参见条款 26）。所以，从多个不同的名字空间引入同一个符号名不会造成冲突（假如确实真的从不使用这个符号的话）。例如，除了名字空间 sdm 外，假如还要用到下面这个名字空间：

```
namespace acmewindowsystem {

  ...

  typedef int handle;

  ...

}
```

只要不引用符号 handle，使用 sdm 和 acmewindowsystem 时就不会有冲突。假如真的要引用，可以明确地指明是哪个名字空间的 handle：

```
void f()
{
  using namespace sdm;                  // 引入 sdm 里的所有符号
  using namespace acmewindowsystem;     // 引入 acme 里的所有符号

  ...                                   // 自由地引用 sdm
                                        // 和 acme 里除 handle 之外
                                        // 的其它符号

  handle h;                             // 错误! 哪个 handle?

  sdm::handle h1;                       // 正确, 没有二义

  acmewindowsystem::handle h2;          // 也没有二义

  ...

}
```

假如用常规的基于头文件的方法来做，只是简单地包含 sdm.h 和 acme.h，这样的话，由于 handle 有多个定义，编译将不能通过。

名字空间的概念加入到 c++标准的时间相对较晚，所以有些人会认为它不太重要，可有可无。但这种想法是错误的，因为 c++标准库（参见条款 49）里几乎所有的东西都存在于名字空间 std 之中。这可能令你不以为然，但它却以一种直接的方式影响到你：这就是为什么 c++提供了那些看起来很有趣的、没有扩展名的头文件，如<iostream>, <string>等。详细介绍参见条款 49。

由于名字空间的概念引入的时间相对较晚，有些编译器可能不支持。就算是这样，那也没理由污染全局名字空间，因为可以用 struct 来近似实现 namespace。可以这样做：先创建一个结构用以保存全局符号名，然后将这些全局符号名作为静态成员放入结构中：

```
// 用于模拟名字空间的一个结构的定义
struct sdm {
  static const double book_version;
  class handle { ... };
  static handle& gethandle();
};

const double sdm::book_version = 2.0;      // 静态成员的定义
```

现在，如果有人想访问这些全局符号名，只用简单地在它们前面加上结构名作为前缀：

```
void f()
{
  cout << sdm::book_version;

  ...

  sdm::handle h = sdm::gethandle();

  ...
}
```

但是，如果全局范围内实际上没有名字冲突，用户就会觉得加修饰符麻烦而多余。幸运的是，还是有办法来让用户选择使用它们或忽略它们。

对于类型名，可以用类型定义（typedef）来显式地去掉空间引用。例如，假设结构 s（模拟的名字空间）内有个类型名 t，可以这样用 typedef 来使得 t 成为 s::t 的同义词：

```
typedef sdm::handle handle;
```

对于结构中的每个（静态）对象 x，可以提供一个（全局）引用 x，并初始化为 s::x：

```
const double& book_version = sdm::book_version;
```

老实说，如果读了条款 47，你就会不喜欢定义一个象 book_version 这样的非局部静态对象。（你就会用条款 47 中所介绍的函数来取代这样的对象）

处理函数的方法和处理对象一样，但要注意，即使定义函数的引用是合法的，但代码的维护者会更喜欢你使用函数指针：

```
sdm::handle& (* const gethandle)() =       // gethandle 是指向 sdm::gethandle
  sdm::gethandle;                          // 的 const 指针 (见条款 21)
```

注意 gethandle 是一个常指针。因为你当然不想让你的用户将它指向别的什么东西，而不是 sdm::gethandle，对不对？

（如果真想知道怎么定义一个函数的引用，看看下面：

```
sdm::handle& (&gethandle)() =       // gethandle 是指向
  sdm::gethandle;                   // sdm::gethandle 的引用
```

我个人认为这样的做法也很好，但你可能以前从没见到过。除了初始化的方式外，函数的引用和函数的常指针在行为上完全相同，只是函数指针更易于理解。）

有了上面的类型定义和引用，那些不会遭遇全局名字冲突的用户就会使用没有修饰符的类型和对象名；相反，那些有全局名字冲突的用户就会忽略类型和引用的定义，代之以带修饰符的符号名。还要注意的是，不是所有用户都想使用这种简写名，所以要把类型定义和引用放在一个单独的头文件中，不要把它和（模拟 namespace 的）结构的定义混在一起。

struct 是 namespace 的很好的近似，但实际上还是相差很远。它在很多方面很欠缺，其中很明显的一点是对运算符的处理。如果运算符被定义为结构的静态成员，它就只能通过函数调用来使用，而不能象常规的运算符所设计的那样，可以通过自然的中缀语法来使用：

```
// 定义一个模拟名字空间的结构，结构内部包含 widgets 的类型
// 和函数。widgets 对象支持 operator+进行加法运算
struct widgets {
  class widget { ... };


  // 参见条款 21：为什么返回 const
  static const widget operator+(const widget& lhs,
                                const widget& rhs);

  ...

};

// 为上面所述的 widge 和 operator+
// 建立全局（无修饰符的）名称

typedef widgets::widget widget;


const widget (* const operator+)(const widget&,        // 错误!
                                 const widget&);       // operator+不能是指针名


widget w1, w2, sum;
```

```
sum = w1 + w2;                          // 错误! 本空间没有声明
                                        // 参数为 widgets 的 operator+

sum = widgets::operator+(w1, w2);       // 合法, 但不是
                                        // "自然"的语法
```

正因为这些限制，所以一旦编译器支持，就要尽早使用真正的名字空间。

第五章 类和函数：实现

c++是一种高度类型化的语言，所以，给出合适的类和模板的定义以及合适的函数声明是整个设计工作中最大的一部分。按理说，只要这部分做好了，类、模板以及函数的实现就不容易出问题。但是，往往人们还是会犯错。

犯错的原因有的是不小心违反了抽象的原则：让实现细节可以提取类和函数内部的数据。有的错误在于不清楚对象生命周期的长短。还有的错误起源于不合理的前期优化工作，特别是滥用 inline 关键字。最后一种情况是，有些实现策略会导致源文件间的相互联结问题，它可能在小规模范围内很合适，但在重建大系统时会带来难以接受的成本。

所有这些问题，以及与之类似的问题，都可以避免，只要你清楚该注意哪些方面。以下的条款就指明了应该特别注意的几种情况。

条款 29: 避免返回内部数据的句柄

请看面向对象世界里发生的一幕：

对象 a：亲爱的，永远别变心！
对象 b：别担心，亲爱的，我是 const。

然而，和现实生活中一样，a 会怀疑，"能相信 b 吗？" 同样地，和现实生活中一样，答案取决于 b 的本性：其成员函数的组成结构。

假设 b 是一个 const string 对象：

```
class string {
public:
  string(const char *value);        // 具体实现参见条款 11
  ~string();                         // 构造函数的注解参见条款 m5

  operator char *() const;           // 转换 string -> char*;
                                     // 参见条款 m5
  ...

private:
  char *data;
};

const string b("hello world");       // b 是一个 const 对象
```

既然 b 为 const，最好的情况当然就是无论现在还是以后，b 的值总是"hello world"。这就寄希望于别的程序员能以合理的方式使用 b 了。特别是，千万别有什么人象下面这样残忍地将 b 强制转换掉 const（参见条款 21）：

```
string& alsob =                      // 使得 alsob 成为 b 的另一个名字，
  const_cast<string&>(b);    // 但不具有 const 属性
```

然而，即使没有人做这种残忍的事，就能保证 b 永远不会改变吗？看看下面的情形：

```
char *str = b;                 // 调用 b.operator char*()

strcpy(str, "hi mom");         // 修改 str 指向的值
```

b 的值现在还是"hello world"吗？或者，它是否已经变成了对母亲的问候语？答案完全取决于 string::operator char*的实现。

下面是一个有欠考虑的实现，它导致了错误的结果。但是，它工作起来确实很高效，所以很多程序员才掉进它的错误陷阱之中：

```
// 一个执行很快但不正确的实现
inline string::operator char*() const
{ return data; }
```

这个函数的缺陷在于它返回了一个"句柄"（在本例中，是个指针），而这个句柄所指向的信息本来是应该隐藏在被调用函数所在的 string 对象的内部。这样，这个句柄就给了调用者自由访问 data 所指的私有数据的机会。换句话说，有了下面的语句：

```
char *str = b;
```

情况就会变成这样：

```
str------------------------>"hello world\0"
              /
             /
b.data
```

显然，任何对 str 所指向的内存的修改都使得 b 的有效值发生变化。所以，即使 b 声明为 const，而且即使只是调用了 b 的某个 const 成员函数，b 也会在程序运行过程中得到不同的值。特别是，如果 str 修改了它所指的值，b 也会改变。

string::operator char*本身写的没有一点错，麻烦的是它可以用于 const 对象。如果这个函数不声明为 const，就不会有问题，因为这样它就不能用于象 b 这样的 const 对象了。

但是，将一个 string 对象转换成它相应的 char*形式是很合理的一件事，无论这个对象是否为 const。所以，还是应该使函数保持为 const。这样的话，就得重写这个函数，使得它不返回指向对象内部数据的句柄：

```
// 一个执行慢但很安全的实现
inline string::operator char*() const
{
  char *copy = new char[strlen(data) + 1];
  strcpy(copy, data);

  return copy;

}
```

这个实现很安全，因为它返回的指针所指向的数据只是 string 对象所指向数据的拷贝；通过函数返回的指针无法修改 string 对象的值。当然，安全是要有代价的：这个版本的 string::operator char* 运行起来比前面那个简单版本要慢；此外，函数的调用者还要记得 delete 掉返回的指针。

如果不能忍受这个版本的速度，或者担心内存泄露，可以来一点小小的改动：使函数返回一个指向 const char 的指针：

```
class string {
public:
  operator const char *() const;

  ...

};

inline string::operator const char*() const
{ return data; }
```

这个函数既快又安全。虽然它和最初给出的那个函数不一样，但它可以满足大多数程序的需要。这个做法还和 c++标准组织处理 string/char*难题的方案一致：标准 string 类型中包含一个成员函数 c_str，它的返回值是 string 的 const char*版本。关于标准 string 类型的更多信息参见条款 49。

指针并不是返回内部数据句柄的唯一途径。引用也很容易被滥用。下面是一种常见的用法，还是拿 string 类做例子：

```
class string {
public:

  ...

  char& operator[](int index) const
  { return data[index]; }

private:
  char *data;
};

string s = "i'm not constant";

s[0] = 'x';              // 正确, s 不是 const

const string cs = "i'm constant";

cs[0] = 'x';             // 修改了 const string，
                         // 但编译器不会通知
```

注意 string::operator[]是通过引用返回结果的。这意味着函数的调用者得到的是内部数据 data[index]的另一个名字，而这个名字可以用来修改 const 对象的内部数据。这个问题和前面看到的相同，只不过这次的罪魁祸首是引用，而不是指针。

这类问题的通用解决方案和前面关于指针的讨论一样：或者使函数为非 const，或者重写函数，使之不返回句柄。如果想让 string::operator[]既适用于 const 对象又适用于非 const 对象，可以参见条款 21。

并不是只有 const 成员函数需要担心返回句柄的问题，即使是非 const 成员函数也得承认：句柄的合法性失效的时间和它所对应的对象是完全相同的。这个时间可能比用户期望的要早很多，特别是当涉及的对象是由编译器产生的临时对象时。

例如，看看这个函数，它返回了一个 string 对象：

```
string somefamousauthor()          // 随机选择一个作家名
{                                  // 并返回之

  switch (rand() % 3) {            // rand()在<stdlib.h>中
                                   // (还有<cstdlib>。参见条款 49)
  case 0:
    return "margaret mitchell";    // 此作家曾写了 "飘"，
                                   // 一部绝对经典的作品
  case 1:
    return "stephen king";         // 他的小说使得许多人
                                   // 彻夜不眠
  case 2:
    return "scott meyers";         // 嗯...滥竽充数的一个
  }


  return "";                       // 程序不会执行到这儿，
                                   // 但对于一个有返回值的函数来说，
                                   // 任何执行途径上都要有返回值
}
```

希望你的注意力不要集中在随机数是怎样从 rand 产生的问题上，也不要嘲笑我把自己和这些作家联系在一起。真正要注意的是，somefamousauthor 的返回值是一个 string 对象，一个临时 string 对象（参见条款 m19）。这样的对象是暂时性的，它们的生命周期通常在函数调用表达式结束时终止。例如上面的情况中，包含 somefamousauthor 函数调用的表达式结束时，返回值对象的生命周期也就随之结束。

具体看看下面这个使用 somefamousauthor 的例子，假设 string 声明了一个上面的 operator const char*成员函数：

```
const char *pc = somefamousauthor();

cout << pc;
```

不论你是否相信，谁也不能预测这段代码将会做些什么，至少不能确定它会做些什么。因为当你想打印 pc 所指的字符串时，字符串的值是不确定的。造成这一结果的原因在于 pc 初始化时发生了下面这些事件：

1. 产生一个临时 string 对象用以保存 somefamousauthor 的返回值。
2. 通过 string 的 operator const char*成员函数将临时 string 对象转换为 const char*指针，并用这个指针初始化 pc。
3. 临时 string 对象被销毁，其析构函数被调用。析构函数中，data 指针被删除（代码详见条款 11）。然而，data 和 pc 所指的是同一块内存，所以现在 pc 指向的是被删除的内存--------其内容是不可确定的。

因为 pc 是被一个指向临时对象的句柄初始化的，而临时对象在被创建后又立即被销毁，所以在 pc 被使用前句柄已经是非法的了。也就是说，无论想做什么，当要使用 pc 时，pc 其实已经名存实亡。这就是指向临时对象的句柄所带来的危害。

所以，对于 const 成员函数来说，返回句柄是不明智的，因为它会破坏数据抽象。对于非 const 成员函数来说，返回句柄会带来麻烦，特别是涉及到临时对象时。句柄就象指针一样，可以是悬浮（dangle）的。所以一定要象避免悬浮的指针那样，尽量避免悬浮的句柄。

同样不能对本条款绝对化。在一个大的程序中想消灭所有可能的悬浮指针是不现实的，想消灭所有可能的悬浮句柄也是不现实的。但是，只要不是万不得已，就要避免返回句柄，这样，不但程序会受益，用户也会更信赖你。

条款 30：避免这样的成员函数：其返回值是指向成员的非 const 指针或引用，但成员的访问级比这个函数要低

使一个成员为 private 或 protected 的原因是想限制对它的访问，对吗？劳累的编译器要费九牛二虎之力来确保你设置的访问限制不被破坏，对不对？所以，写个函数来让用户随意地访问受限的成员没多大意义，对不对？如果你确实认为有意义，那么请反复阅读本段，直到你不这样认为为止。

实际编程中很容易违反这条规则，下面是个例子：

```
class address { ... };          // 某人居住在此

class person {
public:
  address& personaddress() { return address; }
  ...

private:
  address address;
  ...
};
```

成员函数 personaddress 为调用者提供的是 person 对象中所包含的 address 对象，但是，也许是出于效率上的考虑，返回结果采用的是引用，而不是值（见条款 22）。遗憾的是，这个成员函数的做法有违当初将 person::address 声明为 private 的初衷：

```
person scott(...);            // 为简化省略了参数

address& addr =               // 假设 addr 为全局变量
  scott.personaddress();
```

现在，全局对象 addr 成为了 scott.address 的另一个名字，利用它可以随意读写 scott.address。实际上，scott.address 不再为 private，而是 public，访问级提升的根源在于成员函数 personaddress。当然，本例中给出的 address 的访问级是 private，如果是 protected，情况完全一样。

不仅仅是引用，指针也会产生以上问题。下面的例子和上面相同，只不过这次用的是指针：

```
class person {
public:
  address * personaddress() { return &address; }
  ...
```

```
private:
  address address;
  ...
};

address *addrptr =
  scott.personaddress();        // 问题和上面一样
```

而且，对于指针来说，要担心的不仅仅是数据成员，还要考虑到成员函数。因为返回一个成员函数的指针也是有可能的：

```
class person;                   // 提前声明

// ppmf = "pointer to person member function"
// (指向 person 成员函数的指针)
typedef void (person::*ppmf)();

class person {
public:
  static ppmf verificationfunction()
  { return &person::verifyaddress; }

  ...

private:
  address address;

  void verifyaddress();

};
```

如果你过去没试过象上面那样将成员函数指针和 typedef 结合起来的用法，可能会觉得 person::verificationfunction 的声明有点吓人。别害怕，它的全部含义只不过是：

· verificationfunction 是一个没有输入参数的成员函数
· 它的返回值是 person 类中一个成员函数的指针
· 被指向的函数（即，verificationfunction 的返回值）没有输入参数且没有返回值，即 void。

至于 static 关键字，当它用于对成员的声明时，其含义是：整个类只有这个成员的一份拷贝，并且这个成员可以不通过类的具体对象来访问。有关 static 的完整介绍可以参考 c++教程。（如果你的 c++教程里没有介绍静态成员，请把书页撕了扔到垃圾回收站吧。注意封面一定不要乱扔以免破坏环境。最后，去借或买本更好的教程吧。）

最后一个例子中，verifyaddress 是一个私有成员函数，这意味着它只是类的一个实现细节，只有类的成员才应该知道它（当然，友员也知道）。但是，由于公有成员函数 verificationfunction 返回了指向 verifyaddress 的指针，用户就可以做这样的事:

```
ppmf pmf = scott.verificationfunction();

(scott.*pmf)();                    //  等同于调用
                                   // scott.verifyaddress
```

这里，pmf 成了 person::verifyaddress 的同义词，只是有个重要的区别：可以没有限制地使用它。

虽然前面说了那么多，有一天你可能为了程序的性能还是不得不写象上面那样的函数--------返回值是某个访问级较低的成员的指针或引用。但同时，你又不想牺牲 private 和 protected 为你提供的访问限制。这种情况下，你可以通过返回指向 const 对象的指针或引用来达到两全其美的效果。详细介绍参见条款 21。

条款 31: 千万不要返回局部对象的引用，也不要返回函数内部用 new 初始化的指针的引用

本条款听起来很复杂，其实不然。它只是一个很简单的道理，真的，相信我。

先看第一种情况：返回一个局部对象的引用。它的问题在于，局部对象 ----- 顾名思义 ---- 仅仅是局部的。也就是说，局部对象是在被定义时创建，在离开生命空间时被销毁的。所谓生命空间，是指它们所在的函数体。当函数返回时，程序的控制离开了这个空间，所以函数内部所有的局部对象被自动销毁。因此，如果返回局部对象的引用，那个局部对象其实已经在函数调用者使用它之前被销毁了。

当想提高程序的效率而使函数的结果通过引用而不是值返回时，这个问题就会出现。下面的例子和条款 23 中的一样，其目的在于详细说明什么时候该返回引用，什么时候不该：

```
class rational {          // 一个有理数类
public:
  rational(int numerator = 0, int denominator = 1);
  ~rational();

  ...

private:
  int n, d;               // 分子和分母

// 注意 operator* (不正确地)返回了一个引用
friend const rational& operator*(const rational& lhs,
                                 const rational& rhs);
};

// operator*不正确的实现
inline const rational& operator*(const rational& lhs,
                                 const rational& rhs)
{
  rational result(lhs.n * rhs.n, lhs.d * rhs.d);
  return result;
}
```

这里，局部对象 result 在刚进入 operator*函数体时就被创建。但是，所有的局部对象在离开它们所在的空间时都要被自动销毁。具体到这个例子来说，result 是在执行 return 语句后离开它所在的空间的。所以，如果这样写：

```
rational two = 2;
```

```
rational four = two * two;          // 同 operator*(two, two)
```

函数调用时将发生如下事件：

1. 局部对象 result 被创建。
2. 初始化一个引用，使之成为 result 的另一个名字；这个引用先放在另一边，留做 operator* 的返回值。
3. 局部对象 result 被销毁，它在堆栈所占的空间可被本程序其它部分或其他程序使用。
4. 用步骤 2 中的引用初始化对象 four。

一切都很正常，直到第 4 步才产生了错误，借用高科技界的话来说，产生了"一个巨大的错误"。因为，第 2 步被初始化的引用在第 3 步结束时指向的不再是一个有效的对象，所以对象 four 的初始化结果完全是不可确定的。

教训很明显：别返回一个局部对象的引用。

"那好，"你可能会说，"问题不就在于要使用的对象离开它所在的空间太早吗？我能解决。不要使用局部对象，可以用 new 来解决这个问题。"象下面这样：

```
// operator*的另一个不正确的实现
inline const rational& operator*(const rational& lhs,
                                 const rational& rhs)
{
  // create a new object on the heap
  rational *result =
    new rational(lhs.n * rhs.n, lhs.d * rhs.d);

  // return it
  return *result;
}
```

这个方法的确避免了上面例子中的问题，但却引发了新的难题。大家都知道，为了在程序中避免内存泄漏，就必须确保对每个用 new 产生的指针调用 delete，但是，这里的问题是，对于这个函数中使用的 new，谁来进行对应的 delete 调用呢？

显然，operator*的调用者应该负责调用 delete。真的显然吗？遗憾的是，即使你白纸黑字将它写成规定，也无法解决问题。之所以做出这么悲观的判断，是基于两条理由：

第一，大家都知道，程序员这类人是很马虎的。这不是指你马虎或我马虎，而是指，没有哪个程序员不和某个有这类习性的人打交道。想让这样的程序员记住无论何时调用 operator* 后必须得到结果的指针然后调用 delete，这样的几率有多大呢？也是说，他们必须这样使用 operator*：

```
const rational& four = two * two;      // 得到废弃的指针；
                                       // 将它存在一个引用中
...

delete &four;                          // 得到指针并删除
```

这样的几率将会小得不能再小。记住，只要有哪怕一个 operator*的调用者忘了这条规则，就会造成内存泄漏。

返回废弃的指针还有另外一个更严重的问题，即使是最尽责的程序员也难以避免。因为常常有这种情况，operator*的结果只是临时用于中间值，它的存在只是为了计算一个更大的表达式。例如：

```
rational one(1), two(2), three(3), four(4);
rational product;

product = one * two * three * four;
```

product 的计算表达式需要三个单独的 operator*调用，以相应的函数形式重写这个表达式会看得更清楚：

```
product = operator*(operator*(operator*(one, two), three), four);
```

是的，每个 operator*调用所返回的对象都要被删除，但在这里无法调用 delete，因为没有哪个返回对象被保存下来。

解决这一难题的唯一方案是叫用户这样写代码：

```
const rational& temp1 = one * two;
const rational& temp2 = temp1 * three;
const rational& temp3 = temp2 * four;

delete &temp1;
delete &temp2;
delete &temp3;
```

果真如此的话，你所能期待的最好结果是人们将不再理睬你。更现实一点，你将会在指责声中度日，或者可能会被判处 10 年苦力去写威化饼干机或烤面包机的微代码。

所以要记住你的教训：写一个返回废弃指针的函数无异于坐等内存泄漏的来临。

另外，假如你认为自己想出了什么办法可以避免"返回局部对象的引用"所带来的不确定行为，以及"返回堆(heap)上分配的对象的引用"所带来的内存泄漏，那么，请转到条款 23，看看为什么返回局部静态(static)对象的引用也会工作不正常。看了之后，也许会帮助你避免头痛医脚所带来的麻烦。

条款 32: 尽可能地推迟变量的定义

是的，我们同意 C 语言中变量要放在模块头部定义的规定；但在 C++中，还是取消这种做法吧，它没必要，不自然，而且昂贵。

还记得吗？如果定义了一个有构造函数和析构函数的类型的变量，当程序运行到变量定义之处时，必然面临构造的开销；当变量离开它的生命空间时，又要承担析构的开销。这意味着定义无用的变量必然伴随着不必要的开销，所以只要可能，就要避免这种情况发生。

正如我所知道的，你的编程方式优雅而不失老练。所以你可能会在想，你决不会定义一个无用的变量，所以本条款的建议不适用于你严谨紧凑的编程风格。但别急，看看下面这个函数：当口令够长时，它返回口令的加密版本；当口令太短时，函数抛出 logic_error 类型的异常（logic_error 类型在 C++标准库中定义，参见条款 49）：

```
// 此函数太早定义了变量"encrypted"
string encryptPassword(const string& password)
{
  string encrypted;

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

  进行必要的操作，将口令的加密版本
  放进 encrypted 之中;

  return encrypted;
}
```

对象 encrypted 在函数中并非完全没用，但如果有异常抛出时，就是无用的。但是，即使 encryptPassword 抛出异常（见条款 M15），程序也要承担 encrypted 构造和析构的开销。所以，最好将 encrypted 推迟到确实需要它时才定义：

```
// 这个函数推迟了 encrypted 的定义，
// 直到真正需要时才定义
string encryptPassword(const string& password)
{
  if (password.length() < MINIMUM_PASSWORD_LENGTH) {
    throw logic_error("Password is too short");
  }

  string encrypted;
```

```
  进行必要的操作，将口令的加密版本
  放进 encrypted 之中;

  return encrypted;
}
```

这段代码还不是那么严谨，因为 encrypted 定义时没有带任何初始化参数。这将导致它的缺省构造函数被调用。大多数情况下，对一个对象首先做的一件事是给它一个什么值，这通常用赋值来实现。条款 12 说明了为什么"缺省构造一个对象然后对它赋值"比"用真正想要的值来初始化这个对象"效率要低得多。这一论断在此一样适用。例如，假设 encryptPassword 中最难处理的部分在这个函数中进行：

```
void encrypt(string& s);      // s 在此加密
```

于是 encryptPassword 可以象这样实现（当然，它不是最好的实现方式）：

```
// 这个函数推迟了 encrypted 的定义，
// 直到需要时才定义，但还是很低效
string encryptPassword(const string& password)
{
  ...                    // 同上，检查长度

  string encrypted;        // 缺省构造 encrypted
  encrypted = password;    // 给 encrypted 赋值
  encrypt(encrypted);
  return encrypted;
}
```

更好的方法是用 password 来初始化 encrypted，从而绕过了对缺省构造函数不必要的调用：

```
// 定义和初始化 encrypted 的最好方式
string encryptPassword(const string& password)
{
  ...                          // 检查长度

  string encrypted(password);     // 通过拷贝构造函数定义并初始化

  encrypt(encrypted);
  return encrypted;
}
```

这段代码阐述了本条款的标题中"尽可能"这三个字的真正含义。你不仅要将变量的定义推迟到必须使用它的时候，还要尽量推迟到可以为它提供一个初始化参数为止。这样做，不仅可以避免对不必要的对象进行构造和析构，还可以避免无意义的对缺省构造函数的调用。而且，在对变量进行初始化的场合下，变量本身的用途不言自明，所以在这里定义变量有益于表明

变量的含义。还记得在 C 语言中的做法吗？每个变量的定义旁最好要有一条短注释，以标明这个变量将来做什么用。而现在，一个合适的名字（见条款 28），再结合有意义的初始化参数，你就可以实现每个程序员的梦想：通过可靠的变量本身来消除对它不必要的注释。

推迟变量定义可以提高程序的效率，增强程序的条理性，还可以减少对变量含义的注释。看来是该和那些开放式模块的变量定义吻别了。

条款 33: 明智地使用内联

内联函数------多妙的主意啊！它们看起来象函数，运作起来象函数，比宏(macro)要好得多（参见条款 1），使用时还不需要承担函数调用的开销。你还能对它们要求更多吗？

然而，你从它们得到的确实比你想象的要多，因为避免函数调用的开销仅仅是问题的一个方面。为了处理那些没有函数调用的代码，编译器优化程序本身进行了专门的设计。所以当内联一个函数时，编译器可以对函数体执行特定环境下的优化工作。这样的优化对"正常"的函数调用是不可能的。

我们还是不要扯得太远。程序世界和现实生活一样，从来就没有免费的午餐，内联函数也不例外。内联函数的基本思想在于将每个函数调用以它的代码体来替换。用不着统计专家出面就可以看出，这种做法很可能会增加整个目标代码的体积。在一台内存有限的计算机里，过分地使用内联所产生的程序会因为有太大的体积而导致可用空间不够。即使可以使用虚拟内存，内联造成的代码膨胀也可能会导致不合理的页面调度行为（系统颠簸），这将使你的程序运行慢得象在爬。（当然，它也为磁盘控制器提供了一个极好的锻炼方式:)）过多的内联还会降低指令高速缓存的命中率，从而使取指令的速度降低，因为从主存取指令当然比从缓存要慢。

另一方面，如果内联函数体非常短，编译器为这个函数体生成的代码就会真的比为函数调用生成的代码要小许多。如果是这种情况，内联这个函数将会确实带来更小的目标代码和更高的缓存命中率！

要牢记在心的一条是，inline 指令就象 register，它只是对编译器的一种提示，而不是命令。也就是说，只要编译器愿意，它就可以随意地忽略掉你的指令，事实上编译器常常会这么做。例如，大多数编译器拒绝内联"复杂"的函数（例如，包含循环和递归的函数）；还有，即使是最简单的虚函数调用，编译器的内联处理程序对它也爱莫能助。（这一点也不奇怪。virtual 的意思是"等到运行时再决定调用哪个函数"，inline 的意思是"在编译期间将调用之处用被调函数来代替"，如果编译器甚至还不知道哪个函数将被调用，当然就不能责怪它拒绝生成内联调用了）。以上可以归结为：一个给定的内联函数是否真的被内联取决于所用的编译器的具体实现。幸运的是，大多数编译器都可以设置诊断级，当声明为内联的函数实际上没有被内联时，编译器就会为你发出警告信息（参见条款 48）。

假设写了某个函数 f 并声明为 inline，如果出于什么原因，编译器决定不对它内联，那将会发生些什么呢？最明显的一个回答是将 f 作为一个非内联函数来处理：为 f 生成代码时就象它是一个普通的"外联"函数一样， 对 f 的调用也象对普通函数调用那样进行。

理论上来说确实应该这样发生，但理论和现实往往会偏离，现在就属于这种情况。因为，这个方案对解决"被外联的内联"（outlined inline）这一问题确实非常理想，但它加入到 C++ 标准中的时间相对较晚。较早的 C++规范（比如 ARM------参见条款 50）告诉编译器制造商去实现的是另外不同的行为，而且这一旧的行为在现在的编译器中还很普遍，所以必须理解它是怎么一回事。

稍微想一想你就可以记起，内联函数的定义实际上都是放在头文件中。这使得多个要编译的单元（源文件）可以包含同一个头文件，共享头文件内定义的内联函数所带来的益处。下面给出了一个例子，例子中的源文件名以常规的".cpp"结尾，这应该是 C++世界最普遍的命名习惯了：

```
// 文件 example.h
inline void f() { ... }          // f 的定义

...

// 文件 source1.cpp
#include "example.h"             // 包含 f 的定义

...                              // 包含对 f 的调用

// 文件 source2.cpp
#include "example.h"             // 也包含 f 的定义

...                              // 也调用 f
```

假设现在采用旧的"被外联的内联"规则，而且假设 f 没有被内联，那么，当 source1.cpp 被编译时，生成的目标文件中将包含一个称为 f 的函数，就象 f 没有被声明为 inline 一样。同样地，当 source2.cpp 被编译时，产生的目标文件也将包含一个称为 f 的函数。当想把两个目标文件链接在一起时，编译器会因为程序中有两个 f 的定义而报错。

为了防止这一问题，旧规则规定，对于未被内联的内联函数，编译器把它当成被声明为 static 那样处理，即，使它局限于当前被编译的文件。具体到刚才看到的例子中，遵循旧规则的编译器处理 source1.cpp 中的 f 时，就象 f 在 source1.cpp 中是静态的一样；处理 source2.cpp 中的 f 时，也把它当成在 source2.cpp 中是静态的一样。这一策略消除了链接时的错误，但带来了开销：每个包含 f 的定义（以及调用 f）的被编译单元都包含自己的 f 的静态拷贝。如果 f 自身定义了局部静态变量，那么，每个 f 的拷贝都有此局部变量的一份拷贝，这必然会让程序员大吃一惊，因为一般来说，函数中的"static"意味着"只有一份拷贝"。

具体实现起来也会令人吃惊。无论新规则还是旧规则，如果内联函数没被内联，每个调用内联函数的地方还是得承担函数调用的开销；如果是旧规则，还得忍受代码体积的增加，因为每个包含（或调用） f 的被编译单元都有一份 f 的代码及其静态变量的拷贝！（更糟糕的是，每个 f 的拷贝以及每个 f 的静态变量的拷贝往往处于不同的虚拟内存页面，所以两个对 f 的不同拷贝进行调用有可能导致多个页面错误。）

还有呢！有时，可怜的随时准备为您效劳的编译器即使很想内联一个函数，却不得不为这个内联函数生成一个函数体。特别是，如果程序中要取一个内联函数的地址，编译器就必须为此生成一个函数体。编译器怎么能产生一个指向不存在的函数的指针呢？

```
inline void f() {...}          // 同上
void (*pf)() = f;              // pf 指向 f
int main()
{
  f();                       // 对 f 的内联调用
  pf();                      // 通过 pf 对 f 的非内联调用
  ...
}
```

这种情况似乎很荒谬：f 的调用被内联了，但在旧的规则下，每个取 f 地址的被编译单元还是各自生成了此函数的静态拷贝。（新规则下，不管涉及的被编译单元有多少，将只生成唯一一个 f 的外部拷贝）

即使你从来不使用函数指针，这类"没被内联的内联函数"也会找上你的门，因为不只是程序员会使用函数指针，有时编译器也这么做。特别是，编译器有时会生成构造函数和析构函数的外部拷贝，这样就可以通过得到那些函数的指针，方便地构造和析构类的对象数组（参见条款 M8）。

实际上，随便一个测试就可以证明构造函数和析构函数常常不适合内联；甚至，情况比测试结果还糟。例如，看下面这个类 Derived 的构造函数：

```
class Base {
public:
  ...
private:
  string bm1, bm2;   // 基类成员 1 和 2
};
class Derived: public Base {
public:
  Derived() {}               // Derived 的构造函数是空的，
  ...                        // ------但，真的是空的吗？
private:
  string dm1, dm2, dm3;      // 派生类成员 1-3
};
```

这个构造函数看起来的确象个内联的好材料，因为它没有代码。但外表常常欺骗人！仅仅因为它没有代码并不能说明它真的不含代码。实际上，它含有相当多的代码。

C++就对象创建和销毁时发生的事件有多方面的规定。条款 5 和 M8 介绍了当使用 new 时，动态创建的对象怎样自动地被它们的构造函数初始化，以及当使用 delete 时析构函数怎样被调用。条款 13 说明了当创建一个对象时，对象的每个基类以及对象的每个数据成员会被自动地创建；当对象被销毁时，会自动地执行相反的过程（即析构）。这些条款告诉你，C++规定了哪些必须发生，但没规定"怎么"发生。"怎么发生"取决于编译器的实现者，但要弄清楚的是，这些事件不是凭空自己发生的。程序中必然有什么代码使得它们发生，特别是

那些由编译器的实现者写的、在编译其间插入到你的程序中的代码，必然也藏身于某个地方------有时，它们就藏身于你的构造函数和析构函数。所以，对于上面那个号称为空的 Derived 的构造函数，有些编译器会为它产生相当于下面的代码：

```
// 一个 Derived 构造函数的可能的实现
 Derived::Derived()
{
  // 如果在堆上创建对象，为其分配堆内存；
  // operator new 的介绍参见条款 8
  if (本对象在堆上)
    this = ::operator new(sizeof(Derived));
  Base::Base();              // 初始化 Base 部分
  dm1.string();         // 构造 dm1
  dm2.string();         // 构造 dm2
  dm3.string();         // 构造 dm3
}
```

别指望上面这样的代码可以通过编译，因为它在 C++中是不合法的。首先，在构造函数内无法知道对象是不是在堆上。（想知道如何可靠地确定一个对象是否在堆上，请参见条款 M27）另外，对 this 赋值是非法的。还有，通过函数调用访问构造函数也是不允许的。然而，编译器工作起来没这些限制，它可以随心所欲。但代码的合法性不是现在要讨论的主题。问题的要点在于，调用 operator new（如果需要的话）的代码、构造基类部分的代码、构造数据成员的代码都会神不知鬼不觉地添加到你的构造函数中，从而增加构造函数的体积，使得构造函数不再适合内联。当然，同样的分析也适用于 Base 的构造函数，如果 Base 的构造函数被内联，添加到它里面的所有代码也会被添加到 Derived 的构造函数（Derived 的构造函数会调用 Base 的构造函数）。如果 string 的构造函数恰巧也被内联，Derived 的构造函数将得到其代码的 5 个拷贝，每个拷贝对应于 Derived 对象中 5 个 string 中的一个（2 个继承而来，3 个自己声明）。现在你应该明白，内联 Derived 的构造函数并非可以很简单就决定的！当然，类似的情况也适用于 Derived 的析构函数，无论如何都要清楚这一点：被 Derived 的构造函数初始化的所有对象都要被完全销毁。刚被销毁的对象以前可能占用了动态分配的内存，那么这些内存还需要释放。

程序库的设计者必须预先估计到声明内联函数带来的负面影响。因为想对程序库中的内联函数进行二进制代码升级是不可能的。换句话说，如果 f 是库中的一个内联函数，用户会将 f 的函数体编译到自己的程序中。如果程序库的设计者后来要修改 f，所有使用 f 的用户程序必须重新编译。这会很令人讨厌（参见条款 34）。相反，如果 f 是非内联函数，对 f 的修改仅需要用户重新链接，这就比需要重新编译大大减轻了负担；如果包含这个函数的程序库是被动态链接的，程序库的修改对用户来说完全是透明的。

内联函数中的静态对象常常表现出违反直觉的行为。所以，如果函数中包含静态对象，通常要避免将它声明为内联函数。具体介绍参见条款 M26。

为了提高程序开发质量，以上诸项一定要牢记在心。但在具体编程时，从纯实际的角度来看，有一个事实比其余的因素都重要：大多数调试器遇上内联函数都会无能为力。

这不是什么新鲜事。你想，怎么在一个不存在的函数里设置断点呢？怎么单步执行到这样一个函数呢？怎么俘获对它的调用呢？除非你是个百年一遇的怪才，或者用了暗渡陈仓之类的伎俩，否则是不可能做到的。让人高兴的是，这一点倒是可以作为决定该不该对函数声明 inline 的决策依据之一。

一般来说，实际编程时最初的原则是不要内联任何函数，除非函数确实很小很简单，象下面这个 age 函数：

```
class Person {
public:
  int age() const { return personAge; }
  ...
private:
  int personAge;
  ...
};
```

慎重地使用内联，不但给了调试器更多发挥作用的机会，还将内联的作用定位到了正确的位置：它是一个根据需要而使用的优化工具。不要忘了从无数经验得到的这条 80-20 定律（参见条款 M16）：一个程序往往花 80%的时间来执行程序中 20%的代码。这是一条很重要的定律，因为它提醒你，作为程序员的一个很重要的目标，就是找出这 20%能够真正提高整个程序性能的代码。你可以选择内联你的函数，或者没必要就不内联，但这些选择只有作用在"正确"的函数上才有意义。

一旦找出了程序中那些重要的函数，以及那些内联后可以确实提高程序性能的函数（这些函数本身依赖于所在系统的体系结构），就要毫不犹豫地声明为 inline。同时，要注意代码膨胀带来的问题，并监视编译器的警告信息（参见条款 48），看看是否有内联函数没有被编译器内联。

若能做到明智地使用，内联函数将是每个 C++程序员百宝箱中的一件无价之宝。当然，正如前面的讨论所揭示的，它们并不象所想象的那样简单和直接。

条款 34: 将文件间的编译依赖性降至最低

假设某一天你打开自己的 C++程序代码，然后对某个类的实现做了小小的改动。提醒你，改动的不是接口，而是类的实现，也就是说，只是细节部分。然后你准备重新生成程序，心想，编译和链接应该只会花几秒种。毕竟，只是改动了一个类嘛！于是你点击了一下"Rebuild"，或输入 make（或其它类似命令）。然而，等待你的是惊愕，接着是痛苦。因为你发现，整个世界都在被重新编译、重新链接！

当这一切发生时，你难道仅仅只是愤怒吗？

问题发生的原因在于，在将接口从实现分离这方面，C++做得不是很出色。尤其是，C++的类定义中不仅包含接口规范，还有不少实现细节。例如：

```
class Person {
public:
  Person(const string& name, const Date& birthday,
         const Address& addr, const Country& country);
  virtual ~Person();

  ...                    // 简化起见，省略了拷贝构造
                         // 函数和赋值运算符函数
  string name() const;
  string birthDate() const;
  string address() const;
  string nationality() const;

private:
  string name_;          // 实现细节
  Date birthDate_;       // 实现细节
  Address address_;      // 实现细节
  Country citizenship_;  // 实现细节
};
```

这很难称得上是一个很高明的设计，虽然它展示了一种很有趣的命名方式：当私有数据和公有函数都想用某个名字来标识时，让前者带一个尾部下划线就可以区别了。这里要注意到的重要一点是，Person 的实现用到了一些类，即 string， Date，Address 和 Country；Person 要想被编译，就得让编译器能够访问得到这些类的定义。这样的定义一般是通过#include 指令来提供的，所以在定义 Person 类的文件头部，可以看到象下面这样的语句：

```
#include <string>          // 用于 string 类型 (参见条款 49)
#include "date.h"
#include "address.h"
#include "country.h"
```

遗憾的是，这样一来，定义 Person 的文件和这些头文件之间就建立了编译依赖关系。所以如果任一个辅助类（即 string， Date，Address 和 Country）改变了它的实现，或任一个辅助类所依赖的类改变了实现，包含 Person 类的文件以及任何使用了 Person 类的文件就必须重新编译。对于 Person 类的用户来说，这实在是令人讨厌，因为这种情况用户绝对是束手无策。

那么，你一定会奇怪为什么 C++一定要将一个类的实现细节放在类的定义中。例如，为什么不能象下面这样定义 Person，使得类的实现细节与之分开呢？

```
class string;         // "概念上" 提前声明 string 类型
                      // 详见条款 49

class Date;           // 提前声明
class Address;        // 提前声明
class Country;        // 提前声明

class Person {
public:
  Person(const string& name, const Date& birthday,
         const Address& addr, const Country& country);
  virtual ~Person();

  ...                 // 拷贝构造函数, operator=

  string name() const;
  string birthDate() const;
  string address() const;
  string nationality() const;
};
```

如果这种方法可行的话，那么除非类的接口改变，否则 Person 的用户就不需要重新编译。大系统的开发过程中，在开始类的具体实现之前，接口往往基本趋于固定，所以这种接口和实现的分离将大大节省重新编译和链接所花的时间。

可惜的是，现实总是和理想相抵触，看看下面你就会认同这一点：

```
int main()
{
  int x;                // 定义一个 int

  Person p(...);        // 定义一个 Person
                        // (为简化省略参数)
  ...

}
```

当看到 x 的定义时，编译器知道必须为它分配一个 int 大小的内存。这没问题，每个编译器都知道一个 int 有多大。然而，当看到 p 的定义时，编译器虽然知道必须为它分配一个 Person 大小的内存，但怎么知道一个 Person 对象有多大呢？唯一的途径是借助类的定义，但如果类的定义可以合法地省略实现细节，编译器怎么知道该分配多大的内存呢？

原则上说，这个问题不难解决。有些语言如 Smalltalk，Eiffel 和 Java 每天都在处理这个问题。它们的做法是，当定义一个对象时，只分配足够容纳这个对象的一个指针的空间。也就是说，对应于上面的代码，他们就象这样做：

```
int main()
{
  int x;                  // 定义一个 int

  Person *p;              // 定义一个 Person 指针

  ...
}
```

你可能以前就碰到过这样的代码，因为它实际上是合法的 C++语句。这证明，程序员完全可以自己来做到 "将一个对象的实现隐藏在指针身后"。

下面具体介绍怎么采用这一技术来实现 Person 接口和实现的分离。首先，在声明 Person 类的头文件中只放下面的东西：

```
// 编译器还是要知道这些类型名，
// 因为 Person 的构造函数要用到它们
class string;      // 对标准 string 来说这样做不对，
                   // 原因参见条款 49
class Date;
class Address;
class Country;

// 类 PersonImpl 将包含 Person 对象的实
// 现细节，此处只是类名的提前声明
class PersonImpl;

class Person {
public:
  Person(const string& name, const Date& birthday,
         const Address& addr, const Country& country);
  virtual ~Person();

  ...                     // 拷贝构造函数, operator=
```

```
  string name() const;
  string birthDate() const;
  string address() const;
  string nationality() const;

private:
  PersonImpl *impl;              // 指向具体的实现类
};
```

现在 Person 的用户程序完全和 string，date，address，country 以及 person 的实现细节分家了。那些类可以随意修改，而 Person 的用户却落得个自得其乐，不闻不问。更确切的说，它们可以不需要重新编译。另外，因为看不到 Person 的实现细节，用户不可能写出依赖这些细节的代码。这是真正的接口和实现的分离。

分离的关键在于，"对类定义的依赖" 被 "对类声明的依赖" 取代了。所以，为了降低编译依赖性，我们只要知道这么一条就足够了：只要有可能，尽量让头文件不要依赖于别的文件；如果不可能，就借助于类的声明，不要依靠类的定义。其它一切方法都源于这一简单的设计思想。

下面就是这一思想直接深化后的含义：

· 如果可以使用对象的引用和指针，就要避免使用对象本身。定义某个类型的引用和指针只会涉及到这个类型的声明。定义此类型的对象则需要类型定义的参与。

· 尽可能使用类的声明，而不使用类的定义。因为在声明一个函数时，如果用到某个类，是绝对不需要这个类的定义的，即使函数是通过传值来传递和返回这个类：

```
  class Date;                  // 类的声明

  Date returnADate();          // 正确 ---- 不需要 Date 的定义
  void takeADate(Date d);
```

当然，传值通常不是个好主意（见条款 22），但出于什么原因不得不这样做时，千万不要还引起不必要的编译依赖性。

如果你对 returnADate 和 takeADate 的声明在编译时不需要 Date 的定义感到惊讶，那么请跟我一起看看下文。其实，它没看上去那么神秘，因为任何人来调用那些函数，这些人会使得 Date 的定义可见。"噢" 我知道你在想，"为什么要劳神去声明一个没有人调用的函数呢？" 不对！不是没有人去调用，而是，并非每个人都会去调用。例如，假设有一个包含数百个函数声明的库（可能要涉及到多个名字空间----参见条款 28），不可能每个用户都去调用其中的每一个函数。将提供类定义（通过#include 指令）的任务从你的函数声明头文件转交给包含函数调用的用户文件，就可以消除用户对类型定义的依赖，而这种依赖本来是不必要的、是人为造成的。

· 不要在头文件中再（通过#include 指令）包含其它头文件，除非缺少了它们就不能编译。相反，要一个一个地声明所需要的类，让使用这个头文件的用户自己（通过#include 指令）去包含其它的头文件，以使用户代码最终得以通过编译。一些用户会抱怨这样做对他们来说很不方便，但实际上你为他们避免了许多你曾饱受的痛苦。事实上，这种技术很受推崇，并被运用到 C++标准库（参见条款 49）中；头文件<iosfwd>就包含了 iostream 库中的类型声明（而且仅仅是类型声明）。

Person 类仅仅用一个指针来指向某个不确定的实现，这样的类常常被称为句炳类(Handle class)或信封类(Envelope class)。（对于它们所指向的类来说，前一种情况下对应的叫法是主体类(Body class)；后一种情况下则叫信件类(Letter class)。）偶尔也有人把这种类叫 "Cheshire 猫" 类，这得提到《艾丽丝漫游仙境》中那只猫，当它愿意时，它会使身体其它部分消失，仅仅留下微笑。

你一定会好奇句炳类实际上都做了些什么。答案很简单：它只是把所有的函数调用都转移到了对应的主体类中，主体类真正完成工作。例如，下面是 Person 的两个成员函数的实现：

```
#include "Person.h"          // 因为是在实现 Person 类，
                             // 所以必须包含类的定义

#include "PersonImpl.h"      // 也必须包含 PersonImpl 类的定义，
                             // 否则不能调用它的成员函数。
                             // 注意 PersonImpl 和 Person 含有一样的
                             // 成员函数，它们的接口完全相同

Person::Person(const string& name, const Date& birthday,
               const Address& addr, const Country& country)
{
  impl = new PersonImpl(name, birthday, addr, country);
}

string Person::name() const
{
  return impl->name();
}
```

请注意 Person 的构造函数怎样调用 PersonImpl 的构造函数（隐式地以 new 来调用，参见条款 5 和 M8）以及 Person::name 怎么调用 PersonImpl::name。这很重要。使 Person 成为一个句柄类并不改变 Person 类的行为，改变的只是行为执行的地点。

除了句柄类，另一选择是使 Person 成为一种特殊类型的抽象基类，称为协议类（Protocol class）。根据定义，协议类没有实现；它存在的目的是为派生类确定一个接口（参见条款 36）。所以，它一般没有数据成员，没有构造函数；有一个虚析构函数（见条款 14），还有一套纯虚函数，用于制定接口。Person 的协议类看起来会象下面这样：

```
class Person {
public:
  virtual ~Person();

  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};
```

Person 类的用户必须通过 Person 的指针和引用来使用它，因为实例化一个包含纯虚函数的类是不可能的（但是，可以实例化 Person 的派生类----参见下文）。和句柄类的用户一样，协议类的用户只是在类的接口被修改的情况下才需要重新编译。

当然，协议类的用户必然要有什么办法来创建新对象。这常常通过调用一个函数来实现，此函数扮演构造函数的角色，而这个构造函数所在的类即那个真正被实例化的隐藏在后的派生类。这种函数叫法挺多（如工厂函数(factory function)，虚构造函数(virtual constructor)），但行为却一样：返回一个指针，此指针指向支持协议类接口（见条款 M25）的动态分配对象。这样的函数象下面这样声明：

```
// makePerson 是支持 Person 接口的
//  对象的"虚构造函数" ( "工厂函数")
Person*
  makePerson(const string& name,        //  用给定的参数初始化一个
             const Date& birthday,      //  新的 Person 对象，然后
             const Address& addr,       //  返回对象指针
             const Country& country);
```

用户这样使用它：

```
string name;
Date dateOfBirth;
Address address;
Country nation;

...

//  创建一个支持 Person 接口的对象
Person *pp = makePerson(name, dateOfBirth, address, nation);

...

cout   << pp->name()              //  通过 Person 接口使用对象
       << " was born on "
```

```
      << pp->birthDate()
      << " and now lives at "
      << pp->address();

...

delete pp;                    // 删除对象
```

makePerson 这类函数和它创建的对象所对应的协议类（对象支持这个协议类的接口）是紧密联系的，所以将它声明为协议类的静态成员是很好的习惯：

```
class Person {
public:
  ...       // 同上

// makePerson 现在是类的成员
  static Person * makePerson(const string& name,
                             const Date& birthday,
                             const Address& addr,
                             const Country& country);
```

这样就不会给全局名字空间（或任何其他名字空间）带来混乱，因为这种性质的函数会很多（参见条款 28）。

当然，在某个地方，支持协议类接口的某个具体类（concrete class）必然要被定义，真的构造函数也必然要被调用。它们都背后发生在实现文件中。例如，协议类可能会有一个派生的具体类 RealPerson，它具体实现继承而来的虚函数：

```
class RealPerson: public Person {
public:
  RealPerson(const string& name, const Date& birthday,
             const Address& addr, const Country& country)
  :  name_(name), birthday_(birthday),
     address_(addr), country_(country)
  {}

  virtual ~RealPerson() {}

  string name() const;          // 函数的具体实现没有
  string birthDate() const;     // 在这里给出，但它们
  string address() const;       // 都很容易实现
  string nationality() const;

private:
  string name_;
```

```
  Date birthday_;
  Address address_;
  Country country_;
```

有了 RealPerson，写 Person::makePerson 就是小菜一碟：

```
Person * Person::makePerson(const string& name,
                            const Date& birthday,
                            const Address& addr,
                            const Country& country)
{
  return new RealPerson(name, birthday, addr, country);
}
```

实现协议类有两个最通用的机制，RealPerson 展示了其中之一：先从协议类（Person）继承接口规范，然后实现接口中的函数。另一种实现协议类的机制涉及到多继承，这将是条款 43 的话题。

是的，句柄类和协议类分离了接口和实现，从而降低了文件间编译的依赖性。"但，所有这些把戏会带来多少代价呢？"，我知道你在等待罚单的到来。答案是计算机科学领域最常见的一句话：它在运行时会多耗点时间，也会多耗点内存。

句柄类的情况下，成员函数必须通过（指向实现的）指针来获得对象数据。这样，每次访问的间接性就多一层。此外，计算每个对象所占用的内存大小时，还应该算上这个指针。还有，指针本身还要被初始化（在句柄类的构造函数内），以使之指向被动态分配的实现对象，所以，还要承担动态内存分配（以及后续的内存释放）所带来的开销 ---- 见条款 10。

对于协议类，每个函数都是虚函数，所有每次调用函数时必须承担间接跳转的开销（参见条款 14 和 M24）。而且，每个从协议类派生而来的对象必然包含一个虚指针（参见条款 14 和 M24）。这个指针可能会增加对象存储所需要的内存数量（具体取决于：对于对象的虚函数来说，此协议类是不是它们的唯一来源）。

最后一点，句柄类和协议类都不大会使用内联函数。使用任何内联函数时都要访问实现细节，而设计句柄类和协议类的初衷正是为了避免这种情况。

但如果仅仅因为句柄类和协议类会带来开销就把它们打入冷宫，那就大错特错。正如虚函数，你难道会不用它们吗？（如果回答不用，那你正在看一本不该看的书！）相反，要以发展的观点来运用这些技术。在开发阶段要尽量用句柄类和协议类来减少 "实现" 的改变对用户的负面影响。如果带来的速度和/或体积的增加程度远远大于类之间依赖性的减少程度，那么，当程序转化成产品时就用具体类来取代句柄类和协议类。希望有一天，会有工具来自动执行这类转换。

有些人还喜欢混用句柄类、协议类和具体类，并且用得很熟练。这固然使得开发出来的软件系统运行高效、易于改进，但有一个很大的缺点：还是必须得想办法减少程序重新编译时消耗的时间。

第六章 继承和面向对象设计

很多人认为，继承是面向对象程序设计的全部。这个观点是否正确还有待争论，但本书其它章节的条款数量足以证明，在进行高效的 C++程序设计时，还有更多的工具听你调遣，而不仅仅是简单地让一个类从另一个类继承。

然而，设计和实现类的层次结构与 C 语言中的一切都有着根本的不同。只有在继承和面向对象设计领域，你才最有可能从根本上重新思考软件系统构造的方法。另外，C++提供了多种很令人困惑的面向对象构造部件，包括公有、保护和私有基类；虚拟和非虚拟基类；虚拟和非虚拟成员函数。这些部件不仅互相之间有联系，还和 C++的其它部分相互作用。所以，对于每种部件的含义、什么时候该用它们、怎样最好地和 C++中非面向对象部分相结合 ---- 要想真正理解这些，就要付出艰苦的努力。

使得事情更趋复杂的另一个原因是，C++中很多不同的部件或多或少地好象都在做相同的事。例如：

· 假如需要设计一组具有共同特征的类，是该使用继承使得所有的类都派生于一个共同的基类呢，还是使用模板使得它们都从一个共同的代码框架中产生？

· 类 A 的实现要用到类 B，是让 A 拥有一个类型为 B 的数据成员呢，还是让 A 私有继承于 B？

· 假设想设计一个标准库中没有提供的、类型安全的同族容器类（条款 49 列出了标准库实际提供的容器类），是使用模板呢，还是最好为某个 "自身用普通（void*）指针来实现" 的类建立类型安全的接口呢？

在本章节的条款中，我将指导大家怎样去回答这类问题。当然，我不可能顾及到面向对象设计的方方面面。相反，我将集中解释的是：C++中不同的部件其真正含义是什么，当使用某个部件时你真正做了什么。例如，公有继承意味着 "是一个" （详见条款 35），如果使它成为别的什么意思，就会带来麻烦。相似地，虚函数的含义是 "接口必须被继承"，非虚函数的含义是 "接口和实现都要被继承"。不能区分它们之间的含义会给 C++程序员带来无尽的痛苦。

如果能理解 C++各种部件的含义，你将发现自己对面向对象设计的认识大大转变。你将不再停留在为区分 C++语言提供的不同部件而苦恼，而是在思考要为你的软件系统做些什么。一旦知道自己想做什么，将它转化为相应的 C++部件将是一件很容易的事。

做你想做的，理解你所做的！这两点的重要性绝没有过分抬高。接下来的条款将对如何高效地实现这两点进行了详细的讨论。条款 44 总结了 C++面向对象构造部件间的对应关系和它们的含义。它是本章节最好的总结，也可作为将来使用的简明参考。

条款 35: 使公有继承体现 "是一个" 的含义

在"Some Must Watch While Some Must Sleep"（ W. H. Freeman and Company, 1974）一书中，William Dement 讲了一个故事，故事说的是他如何让学生们记住他的课程中最重要的部分。"据说，"，他告诉他的学生，"一般的英国学生除了记得 Hastings 战役发生在 1066 年外，再也不记得其它历史。"， "如果一个小孩不记得别的历史，" Dement 强调说，"也一定记得 1066 这个日子。" 但对于他班上的学生来说，只有很少一些话题可以引起他们的兴趣，比如，安眠药会引起失眠之类。所以他哀求他的学生，即使忘掉他在课堂上讲授的其它任何东西，也要记住那些仅有的几个重要的历史事件。而且，他在整个学期不停地对学生灌输这一基本观点。

学期结束时，期末考试的最后一道题是，"请写下你从课程中学到的一辈子都会记住的东西"。当 Dement 评改试卷时，他大吃一惊。几乎所有学生都写下了 "1066"。

所以，在这里我也以极度颤抖的声音告诉你，C++面向对象编程中一条重要的规则是：公有继承意味着 "是一个" 。一定要牢牢记住这条规则。

当写下类 D（"Derived" ）从类 B（"Base"）公有继承时，你实际上是在告诉编译器（以及读这段代码的人）：类型 D 的每一个对象也是类型 B 的一个对象，但反之不成立；你是在说：B 表示一个比 D 更广泛的概念，D 表示一个比 B 更特定概念；你是在声明：任何可以使用类型 B 的对象的地方，类型 D 的对象也可以使用，因为每个类型 D 的对象是一个类型 B 的对象。相反，如果需要一个类型 D 的对象，类型 B 的对象就不行：每个 D "是一个" B，但反之不成立。

C++采用了公有继承的上述解释。看这个例子：

```
class Person { ... };

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

从日常经验中我们知道，每个学生是人，但并非每个人是学生。这正是上面的层次结构所声明的。我们希望，任何对 "人" 成立的事实 ---- 如都有生日 ----也对 "学生" 成立；但我们不希望，任何对 "学生" 成立的事实 ---- 如都在某一学校上学 ----也对 "人" 成立。人的概念比学生的概念更广泛；学生是一种特定类型的人。

在 C++世界中，任何一个其参数为 Person 类型的函数（或 Person 的指针或 Person 的引用）可以实际取一个 Student 对象（或 Student 的指针或 Student 的引用）：

```
void dance(const Person& p);        // 任何人可以跳舞

void study(const Student& s);       // 只有学生才学习

Person p;                           // p 是一个人
Student s;                          // s 是一个学生
```

```
dance(p);                    // 正确，p 是一个人

dance(s);                    // 正确，s 是一个学生，
                             // 一个学生"是一个"人

study(s);                    // 正确

study(p);                    // 错误! p 不是学生
```

只是公有继承才会这样。也就是说，只是 Student 公有继承于 Person 时，C++的行为才会象我所描述的那样。私有继承则是完全另外一回事（见条款 42），至于保护继承，好象没有人知道它是什么含义。另外，Student "是一个" Person 的事实并不说明 Student 的数组 "是一个" Person 数组。关于这一话题的讨论参见条款 M3。

公有继承和 "是一个" 的等价关系听起来简单，但在实际应用中，可能不会总是那么直观。有时直觉会误导你。例如，有这样一个事实：企鹅是鸟；还有这样一个事实：鸟会飞。如果想简单地在 C++中表达这些事实，我们会这样做：

```
class Bird {
public:
  virtual void fly();          // 鸟会飞

  ...

};

class Penguin:public Bird {    // 企鹅是鸟

  ...

};
```

突然间我们陷入困惑，因为这种层次关系意味着企鹅会飞，而我们知道这不是事实。发生什么了？

造成这种情况，是因为使用的语言（汉语）不严密。说鸟会飞，并不是说所有的鸟会飞，通常，只有那些有飞行能力的鸟才会飞。如果更精确一点，我们都知道，实际上有很多种不会飞的鸟，所以我们会提供下面这样的层次结构，它更好地反映了现实：

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

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

```
  ...
};

class NonFlyingBird: public Bird {

  ...                 //  没有声明 fly 函数
};

class Penguin: public NonFlyingBird {

  ...                 //  没有声明 fly 函数
};
```

这种层次就比最初的设计更忠于我们所知道的现实。

但关于鸟类问题的讨论，现在还不能完全结束。因为在有的软件系统中，说企鹅是鸟是完全合适的。比如说，如果程序只和鸟的嘴、翅膀有关系而不涉及到飞，最初的设计就很合适。这看起来可能很令人恼火，但它反映了一个简单的事实：没有任何一种设计可以理想到适用于任何软件。好的设计是和软件系统现在和将来所要完成的功能密不可分的（参见条款 M32）。如果程序不涉及到飞，并且将来也不会，那么让 Penguin 派生于 Bird 就会是非常合理的设计。实际上，它会比那个区分会飞和不会飞的设计还要好，因为你的设计中不会用到这种区分。在设计层次中增加多余的类是一种很糟糕的设计，就象在类之间制定了错误的继承关系一样。

对于 "所有鸟都会飞，企鹅是鸟，企鹅不会飞" 这一问题，还可以考虑用另外一种方法来处理。也就是对 penguin 重新定义 fly 函数，使之产生一个运行时错误：

```
void error(const string& msg);      // 在别处定义

class Penguin: public Bird {
public:
  virtual void fly() { error("Penguins can't fly!"); }

  ...

};
```

解释型语言如 Smalltalk 喜欢采用这种方法，但这里要认识到的重要一点是，上面的代码所说的可能和你所想的是完全不同的两回事。它不是说，"企鹅不会飞"，而是说，"企鹅会飞，但让它们飞是一种错误"。

怎么区分二者的不同？这可以从检测到错误发生的时间来区分。"企鹅不会飞" 的指令是由编译器发出的，"让企鹅飞是一种错误" 只能在运行时检测到。

为了表示 "企鹅不会飞" 这一事实，就不要在 Penguin 对象中定义 fly 函数：

```
class Bird {

  ...                    // 没有声明 fly 函数

};

class NonFlyingBird: public Bird {

  ...                    // 没有声明 fly 函数

};

class Penguin: public NonFlyingBird {

  ...                    // 没有声明 fly 函数

};
```

如果想使企鹅飞，编译器就会谴责你的违规行为：

```
Penguin p;

p.fly();                  // 错误!
```

用 Smalltalk 的方法得到的行为和这完全不同。用那种方法，编译器连半句话都不会说。

C++的处理方法和 Smalltalk 的处理方法有着根本的不同，所以只要是在用 C++编程，就要采用 C++的方法做事。另外，在编译时检测错误比在运行时检测错误有某些技术上的优点，详见条款 46。

也许你会说，你在鸟类方面的知识很贫乏。但你可以借助于你的初等几何知识，对不对？我是说，矩形和正方形总该不复杂吧？

那好，回答这个简单问题：类 Square（正方形）可以从类 Rectangle（矩形）公有继承吗？

```
    Rectangle
       ^
       | ?
     Square
```

"当然可以！" 你会不屑地说，"每个人都知道一个正方形是一个矩形，但反过来通常不成立。" 确实如此，至少在高中时可以这样认为。但我不认为我们还是高中生。

看看下面的代码：

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

  virtual int height() const;          // 返回当前值
  virtual int width() const;           // 返回当前值

  ...

};

void makeBigger(Rectangle& r)          // 增加 r 面积的函数
{
  int oldHeight = r.height();

  r.setWidth(r.width() + 10);          // 对 r 的宽度增加 10

  assert(r.height() == oldHeight);     // 断言 r 的高度未变
}
```

很明显，断言永远不会失败。makeBigger 只是改变了 r 的宽度，高度从没被修改过。

现在看下面的代码，它采用了公有继承，使得正方形可以被当作矩形来处理：

```
class Square: public Rectangle { ... };

Square s;

...

assert(s.width() == s.height());      // 这对所有正方形都成立


makeBigger(s);                        // 通过继承，s "是一个" 矩形
                                      // 所以可以增加它的面积

assert(s.width() == s.height());      // 这还是对所有正方形成立
```

很明显，和前面的断言一样，后面的这个断言也永远不会失败。因为根据定义，正方形的宽和高应该相等。

那么现在有一个问题。我们怎么协调下面的断言呢？

· 调用 makeBigger 前，s 的宽和高相等；
· makeBigger 内部，s 的宽度被改变，高度未变；

· 从 makeBigger 返回后，s 的高度又和宽度相等。（注意 s 是通过引用传给 makeBigger 的，所以 makeBigger 修改了 s 本身，而不是 s 的拷贝）

怎么样？

欢迎加入公有继承的精彩世界，在这里，你在其它研究领域养成的直觉 ---- 包括数学 ---- 可能不象你所期望的那样为你效劳。对于上面例子中的情况来说，最根本的问题在于：对矩形适用的规则（宽度的改变和高度没关系）不适用于正方形（宽度和高度必须相同）。但公有继承声称：对基类对象适用的任何东西 ---- 任何！---- 也适用于派生类对象。在矩形和正方形的例子（以及条款 40 中涉及到 set 的一个类似的例子）中，所声称的原则不适用，所以用公有继承来表示它们的关系只会是错误。当然，编译器不会阻拦你这样做，但正如我们所看到的，它不能保证程序可以工作正常。正如每个程序员都知道的，代码通过编译并不说明它能正常工作。

但也不要太担心你多年积累的软件开发直觉在步入到面向对象设计时会没有用武之地。那些知识还是很有价值，但既然你在自己的设计宝库中又增加了继承这一利器，你就要用新的眼光来扩展你的专业直觉，从而指导你开发出正确无误的面向对象程序。很快，你会觉得让 Penguin 从 Bird 继承或让 Square 从 Rectangle 继承的想法很可笑，就象现在某个人向你展示一个长达数页的函数你会觉得可笑一样。也许它是解决问题的正确方法，只是不太合适。

当然，"是一个" 的关系不是存在于类之间的唯一关系。类之间常见的另两个关系是 "有一个" 和 "用...来实现"。这些关系在条款 40 和 42 进行讨论。这两个关系中的某一个被不正确地表示成 "是一个" 的情况并不少见，这将导致错误的设计。所以，一定要确保自己理解这些关系的区别，以及怎么最好地用 C++来表示它们。

条款 36: 区分接口继承和实现继承

（公有）继承的概念看起来很简单，进一步分析，会发现它由两个可分的部分组成：函数接口的继承和函数实现的继承。这两种继承类型的区别和本书简介中所讨论的函数声明和函数定义间的区别是完全一致的。

作为类的设计者，有时希望派生类只继承成员函数的接口（声明）；有时希望派生类同时继承函数的接口和实现，但允许派生类改写实现；有时则希望同时继承接口和实现，并且不允许派生类改写任何东西。

为了更好地体会这些选择间的区别，看下面这个类层次结构，它用来表示一个图形程序中的几何形状：

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

  virtual void error(const string& msg);

  int objectID() const;

  ...

};

class Rectangle: public Shape { ... };

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

纯虚函数 draw 使得 Shape 成为一个抽象类。所以，用户不能创建 Shape 类的实例，只能创建它的派生类的实例。但是，从 Shape（公有）继承而来的所有类都受到 Shape 的巨大影响，因为：

· 成员函数的接口总会被继承。正如条款 35 所说明的，公有继承的含义是 "是一个" ，所以对基类成立的所有事实也必须对派生类成立。因此，如果一个函数适用于某个类，也必将适用于它的子类。

Shape 类中声明了三个函数。第一个函数，draw，在某一画面上绘制当前对象。第二个函数，error，被其它成员函数调用，用于报告出错信息。第三个函数，objectID，返回当前对象的一个唯一整数标识符（条款 17 给出了一个怎样使用这种函数的例子）。每个函数以不同的方式声明：draw 是一个纯虚函数；error 是一个简单的（非纯?）虚函数；objectID 是一个非虚函数。这些不同的声明各有什么含义呢？

首先看纯虚函数 draw。纯虚函数最显著的特征是：它们必须在继承了它们的任何具体类中重新声明，而且它们在抽象类中往往没有定义。把这两个特征放在一起，就会认识到：

· 定义纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。

这对 Shape::draw 函数来说非常有意义，因为，让所有 Shape 对象都可以被绘制是很合理，但 Shape 类无法为 Shape::draw 提供一个合理的缺省实现。例如，绘制椭园的算法就和绘制矩形的算法大不一样。打个比方来说，上面 Shape::draw 的声明就象是在告诉子类的设计者，"你必须提供一个 draw 函数，但我不知道你会怎样实现它。"

顺便说一句，为一个纯虚函数提供定义也是可能的。也就是说，你可以为 Shape::draw 提供实现，C++编译器也不会阻拦，但调用它的唯一方式是通过类名完整地指明是哪个调用：

```
Shape *ps = new Shape;          // 错误! Shape 是抽象的

Shape *ps1 = new Rectangle;     // 正确
ps1->draw();                    // 调用 Rectangle::draw

Shape *ps2 = new Ellipse;       // 正确
ps2->draw();                    // 调用 Ellipse::draw

ps1->Shape::draw();             // 调用 Shape::draw

ps2->Shape::draw();             // 调用 Shape::draw
```

一般来说，除了能让你在鸡尾酒会上给你的程序员同行留下深刻印象外，了解这种用法一般没大的作用。然而，正如后面将看到的，它可以被应用为一种机制，为简单的（非纯）虚函数提供 "比一般做法更安全" 的缺省实现。

有时，声明一个除纯虚函数外什么也不包含的类很有用。这样的类叫协议类（Protocol class），它为派生类仅提供函数接口，完全没有实现。协议类在条款 34 中介绍过，并将在条款 43 再次提及。

简单虚函数的情况和纯虚函数有点不一样。照例，派生类继承了函数的接口，但简单虚函数一般还提供了实现，派生类可以选择改写它们或不改写它们。思考片刻就可以认识到：

· 声明简单虚函数的目的在于，使派生类继承函数的接口和缺省实现。

具体到 Shape::error，这个接口是在说，每个类必须提供一个出错时可以被调用的函数，但每个类可以按它们认为合适的任何方式处理错误。如果某个类不想做什么特别的事，可以借助于 Shape 类中提供的缺省出错处理函数。也就是说，Shape::error 的声明是在告诉子类的设计者，"你必须支持 error 函数，但如果你不想写自己的版本，可以借助 Shape 类中的缺省版本。"

实际上，为简单虚函数同时提供函数声明和缺省实现是很危险的。想知道为什么，看看 XYZ 航空公司的这个飞机类的层次结构。XYZ 公司只有两种飞机，A 型和 B 型，而且两种机型的飞行方式完全一样。所以，XYZ 设计了这样的层次结构：

```
class Airport { ... };      // 表示飞机
```

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

  ...

};

void Airplane::fly(const Airport& destination)
{
  飞机飞往某一目的地的缺省代码
}

class ModelA: public Airplane { ... };

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

为了表明所有飞机都必须支持 fly 函数，而且因为不同型号的飞机原则上都需要对 fly 有不同的实现， 所以 Airplane::fly 被声明为 virtual。但是，为了避免在 ModelA 类和 ModelB 类中写重复的代码，缺省的飞行行为是由 Airplane::fly 函数提供的，ModelA 和 ModelB 继承了这一函数。

这是典型的面向对象设计。两个类享有共同的特征（实现 fly 的方式），所以这一共同特征被转移到基类，并让这两个类来继承这一特征。这种设计使得共性很清楚，避免了代码重复，将来容易增强功能，并易于长期维护 ---- 所有这一切正是面向对象技术高度吹捧的。XYZ 公司真得为此而骄傲。

现在假设 XYZ 公司发了大财，决定引进一种新型飞机，C 型。C 型和 A 型、B 型有区别，特别是，飞行方式不一样。

XYZ 的程序员在上面的层次结构中为 C 型增加了一个类，但因为急于使新型飞机投入使用，他们忘了重新定义 fly 函数：

```
class ModelC: public Airplane {

  ...                    // 没有声明 fly 函数
};
```

然后，在程序中，他们做了类似下面的事：

```
Airport JFK(...);          // JFK 是纽约市的一个机场

Airplane *pa = new ModelC;

...
```

```
pa->fly(JFK);                    // 调用 Airplane::fly!
```

这将造成悲剧：竟然试图让 ModelC 对象如同 ModelA 或 ModelB 那样飞行。这种行为可不能换来旅客对你的信任！

这里的问题不在于 Airplane::fly 具有缺省行为，而在于 ModelC 可以不用明确地声明就可以继承这一行为。幸运的是，可以很容易做到为子类提供缺省行为、同时只是在子类想要的时候才给它们。窍门在于切断虚函数的接口和它的缺省实现之间的联系。下面是一种方法：

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

  ...

protected:
  void defaultFly(const Airport& destination);
};

void Airplane::defaultFly(const Airport& destination)
{
  飞机飞往某一目的地的缺省代码
}
```

注意 Airplane::fly 已经变成了纯虚函数，它提供了飞行的接口。缺省实现还是存在于 Airplane 类中，但现在它是以一个独立函数（defaultFly）的形式存在的。ModelA 和 ModelB 这些类想执行缺省行为的话，只用简单地在它们的 fly 函数体中对 defaultFly 进行一个内联调用（关于内联和虚函数间的相互关系，参见条款 33）：

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

  ...

};

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

  ...

};
```

对于 ModelC 类来说，它不可能无意间继承不正确的 fly 实现。因为 Airplane 中的纯虚函数强迫 ModelC 提供它自己版本的 fly。

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

};

void ModelC::fly(const Airport& destination)
{
  ModelC 飞往某一目的地的代码
}
```

这个方法不会万无一失（程序员还会因为 "拷贝粘贴" 而出错），但它比最初的设计可靠多了。至于 Airplane::defaultFly 被声明为 protected，是因为它确实只是 Airplane 及其派生类的实现细节。使用 airplane 的用户只关心飞机能飞，而不会关心是怎么实现的。

Airplane::defaultFly 是一个非虚函数也很重要。因为没有子类会重新定义这个函数，条款 37 说明了这一事实。如果 defaultFly 为虚函数，就会又回到这个问题：如果某些子类应该重新定义 defaultFly 而又忘记去做，那该怎么办？

一些人反对将接口和缺省实现作为单独函数分开，例如上面的 fly 和 defaultFly。他们认为，起码这会污染类的名字空间，因为有这么多相近的函数名称在扩散。然而他们还是赞同接口和缺省实现应该分离。怎么解决这种表面上存在的矛盾呢？可以借助于这一事实：纯虚函数必须在子类中重新声明，但它还是可以在基类中有自己的实现。下面的 Airplane 正是利用这一点重新定义了一个纯虚函数：

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

  ...

};

void Airplane::fly(const Airport& destination)
{
  飞机飞往某一目的地的缺省代码
}

class ModelA: public Airplane {
public:
```

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

  ...

};

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

  ...

};

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

  ...

};

void ModelC::fly(const Airport& destination)
{
  ModelC 飞往某一目的地的代码
}
```

这一设计和前面的几乎一样，只是纯虚函数 Airplane::fly 的函数体取代了独立函数 Airplane::defaultFly。从本质上说，fly 已经被分成两个基本部分了。它的声明说明了它的接口（派生类必须使用），而它的定义说明了它的缺省行为（派生类可能会使用，但要明确地请求）。然而，将 fly 和 defaultFly 合并后，就不再能够为这两个函数声明不同的保护级别了：本来是 protected 的代码（在 defaultFly 中）现在成了 public（因为它在 fly 中）。

最后，来谈谈 Shape 的非虚函数，objectID。当一个成员函数为非虚函数时，它在派生类中的行为就不应该不同。实际上，非虚成员函数表明了一种特殊性上的不变性，因为它表示的是不会改变的行为 ---- 不管一个派生类有多特殊。所以，

· 声明非虚函数的目的在于，使派生类继承函数的接口和强制性实现。

可以认为，Shape::objectID 的声明就是在说，"每个 Shape 对象有一个函数用来产生对象的标识符，并且对象标识符的产生方式总是一样的。这种方式由 Shape::objectID 的定义决定，派生类不能改变它。" 因为非虚函数表示一种特殊性上的不变性，所以它决不能在子类中重新定义，关于这一点条款 37 进行了讨论。

理解了纯虚函数、简单虚函数和非虚函数在声明上的区别，就可以精确地指定你想让派生类继承什么：仅仅是接口，还是接口和一个缺省实现？或者，接口和一个强制实现？因为这些不同类型的声明指的是根本不同的事，所以在声明成员函数时一定要从中慎重选择。只有这样做，才可以避免没经验的程序员常犯的两个错误。

第一个错误是把所有的函数都声明为非虚函数。这就使得派生类没有特殊化的余地；非虚析构函数尤其会出问题（参见条款 14）。当然，设计出来的类不准备作为基类使用也是完全合理的（条款 M34 就给出了一个你会这样做的例子）。这种情况下，专门声明一组非虚成员函数是适当的。但是，把所有的函数都声明为非虚函数，大多数情况下是因为对虚函数和非虚函数之间区别的无知，或者是过分担心虚函数对程序性能的影响（参见条款 M24）。而事实上是：几乎任何一个作为基类使用的类都有虚函数（再次参见条款 14）。

如果担心虚函数的开销，请允许我介绍 80-20 定律（参见条款 M16）。它指出，在一个典型的程序中，80%的运行时间都花在执行 20%的代码上。这条定律很重要，因为它意味着，平均起来，80%的函数调用可以是虚函数，并且它们不会对程序的整体性能带来哪怕一丁点可以觉察到的影响。所以，在担心是否承担得起虚函数的开销之前，不妨将注意力集中在那 20%会真正带来影响的代码上。

另一个常见的问题是将所有的函数都声明为虚函数。有时这没错 ---- 比如，协议类（Protocol class）就是证据（参见条款 34）。但是，这样做往往表现了类的设计者缺乏表明坚定立场的勇气。一些函数不能在派生类中重定义，只要是这种情况，就要旗帜鲜明地将它声明为非虚函数。不能让你的函数好象可以为任何人做任何事 ---- 只要他们花点时间重新定义所有的函数。记住，如果有一个基类 B，一个派生类 D，和一个成员函数 mf，那么下面每个对 mf 的调用都必须工作正常：

```
D *pd = new D;
B *pb = pd;

pb->mf();          // 通过基类指针调用 mf

pd->mf();          // 通过派生类指针调用 mf
```

有时，必须将 mf 声明为非虚函数才能保证一切都以你所期望的方式工作（参见条款 37）。如果需要特殊性上的不变性，就大胆地说出来吧！

条款 37: 决不要重新定义继承而来的非虚函数

有两种方法来看待这个问题：理论的方法和实践的方法。让我们先从实践的方法开始。毕竟，理论家一般都很耐心。

假设类 D 公有继承于类 B，并且类 B 中定义了一个公有成员函数 mf。mf 的参数和返回类型不重要，所以假设都为 void。换句话说，我这么写：

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

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

甚至对 B，D 或 mf 一无所知，也可以定义一个类型 D 的对象 x，

```
D x;                    // x 是类型 D 的一个对象
```

那么，如果发现这么做：

```
B *pB = &x;             // 得到 x 的指针

pB->mf();               // 通过指针调用 mf
```

和下面这么做的执行行为不一样：

```
D *pD = &x;             // 得到 x 的指针

pD->mf();               // 通过指针调用 mf
```

你一定就会感到很惊奇。

因为两种情况下调用的都是对象 x 的成员函数 mf，因为两种情况下都是相同的函数和相同的对象，所以行为会相同，对吗？

对，会相同。但，也许不会相同。特别是，如果 mf 是非虚函数而 D 又定义了自己的 mf 版本，行为就不会相同：

```
class D: public B {
public:
  void mf();            // 隐藏了 B::mf; 参见条款 50

  ...
```

};

pB->mf(); // 调用 B::mf

pD->mf(); // 调用 D::mf

行为的两面性产生的原因在于，象 B::mf 和 D::mf 这样的非虚函数是静态绑定的（参见条款 38）。这意味着，因为 pB 被声明为指向 B 的指针类型，通过 pB 调用非虚函数时将总是调用那些定义在类 B 中的函数 ---- 即使 pB 指向的是从 B 派生的类的对象，如上例所示。

相反，虚函数是动态绑定的（再次参见条款 38），因而不会产生这类问题。如果 mf 是虚函数，通过 pB 或 pD 调用 mf 时都将导致调用 D::mf，因为 pB 和 pD 实际上指向的都是类型 D 的对象。

所以，结论是，如果写类 D 时重新定义了从类 B 继承而来的非虚函数 mf，D 的对象就可能表现出精神分裂症般的异常行为。也就是说，D 的对象在 mf 被调用时，行为有可能象 B，也有可能象 D，决定因素和对象本身没有一点关系，而是取决于指向它的指针所声明的类型。引用也会和指针一样表现出这样的异常行为。

实践方面的论据就说这么多。我知道你现在想知道的是，不能重新定义继承而来的非虚函数的理论依据是什么。我很高兴解答。

条款 35 解释了公有继承的含义是 "是一个"，条款 36 说明了为什么 "在一个类中声明一个非虚函数实际上为这个类建立了一种特殊性上的不变性"。如果将这些分析套用到类 B、类 D 和非虚成员函数 B::mf，那么，

· 适用于 B 对象的一切也适用于 D 对象，因为每个 D 的对象 "是一个" B 的对象。
· B 的子类必须同时继承 mf 的接口和实现，因为 mf 在 B 中是非虚函数。

那么，如果 D 重新定义了 mf，设计中就会产生矛盾。如果 D 真的需要实现和 B 不同的 mf，而且每个 B 的对象 ---- 无论怎么特殊 ---- 也真的要使用 B 实现的 mf，那么，每个 D 将不 "是一个" B。这种情况下，D 不能从 B 公有继承。相反，如果 D 真的必须从 B 公有继承，而且 D 真的需要和 B 不同的 mf 的实现，那么，mf 就没有为 B 反映出特殊性上的不变性。这种情况下，mf 应该是虚函数。最后，如果每个 D 真的 "是一个" B，并且如果 mf 真的为 B 建立了特殊性上的不变性，那么，D 实际上就不需要重新定义 mf，也就决不能这样做。

不管采用上面的哪一种论据都可以得出这样的结论：任何条件下都要禁止重新定义继承而来的非虚函数。

条款 38: 决不要重新定义继承而来的缺省参数值

让我们从一开始就把问题简化。缺省参数只能作为函数的一部分而存在；另外，只有两种函数可以继承：虚函数和非虚函数。因此，重定义缺省参数值的唯一方法是重定义一个继承而来的函数。然而，重定义继承而来的非虚函数是一种错误（参见条款 37），所以，我们完全可以把讨论的范围缩小为 "继承一个有缺省参数值的虚函数" 的情况。

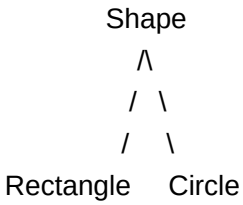
既然如此，本条款的理由就变得非常明显：虚函数是动态绑定而缺省参数值是静态绑定的。

什么意思？你可能会说你不懂这些最新的面向对象术语；或者，过度劳累的你一时想不起静态和动态绑定的区别。那么，让我们来复习一下。

对象的静态类型是指你声明的存在于程序代码文本中的类型。看下面这个类层次结构：

```
enum ShapeColor { RED, GREEN, BLUE };

// 一个表示几何形状的类
class Shape {
public:
  // 所有的形状都要提供一个函数绘制它们本身
  virtual void draw(ShapeColor color = RED) const = 0;

  ...

};

class Rectangle: public Shape {
public:
  // 注意：定义了不同的缺省参数值 ---- 不好!
  virtual void draw(ShapeColor color = GREEN) const;

  ...

};

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

  ...

};
```

用图形来表示是下面这样：



现在看看这些指针：

```
Shape *ps;                  // 静态类型 = Shape*

Shape *pc = new Circle;     // 静态类型 = Shape*

Shape *pr = new Rectangle;  // 静态类型 = Shape*
```

这个例子中， ps， pc，和 pr 都被声明为 Shape 指针类型，所以它们都以此作为自己的静态类型。注意，这和它们真的所指向的对象的类型绝对没有关系 ---- 它们的静态类型总是 Shape*。

对象的动态类型是由它当前所指的对象的类型决定的。即，对象的动态类型表示它将执行何种行为。上面的例子中，pc 的动态类型是 Circle*，pr 的动态类型是 Rectangle*。至于 ps，实际上没有动态类型，因为它（还）没有指向任何对象。

动态类型，顾名思义，可以在程序运行时改变，典型的方法是通过赋值：

```
ps = pc;                    // ps 的动态类型
                            // 现在是 Circle*

ps = pr;                    // ps 的动态类型
                            // 现在是 Rectangle*
```

虚函数是动态绑定的，意思是说，虚函数通过哪个对象被调用，具体被调用的函数就由那个对象的动态类型决定：

```
pc->draw(RED);              // 调用 Circle::draw(RED)

pr->draw(RED);              // 调用 Rectangle::draw(RED)
```

我知道这些都是老掉牙的知识了，你当然也了解虚函数。（如果想知道它们是怎么实现的，参见条款 M24）但是，将虚函数和缺省参数值结合起来分析就会产生问题，因为，如上所述，虚函数是动态绑定的，但缺省参数是静态绑定的。这意味着你最终可能调用的是一个定义在派生类，但使用了基类中的缺省参数值的虚函数：

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

这种情况下，pr 的动态类型是 Rectangle*，所以 Rectangle 的虚函数被调用 ---- 正如我们所期望的那样。Rectangle::draw 中，缺省参数值是 GREEN。但是，由于 pr 的静态类型是

Shape*，这个函数调用的参数值是从 Shape 类中取得的，而不是 Rectangle 类！所以结果将十分奇怪并且出人意料，因为这个调用包含了 Shape 和 Rectangle 类中 Draw 的声明的组合。你当然不希望自己的软件以这种方式运行啦；至少，用户不希望这样，相信我。

不用说，ps， pc，和 pr 都是指针的事实和产生问题的原因无关。如果它们是引用，问题也会继续存在。问题仅仅出在，draw 是一个虚函数，并且它的一个缺省参数在子类中被重新定义了。

为什么 C++坚持这种有违常规的做法呢？答案和运行效率有关。如果缺省参数值被动态绑定，编译器就必须想办法为虚函数在运行时确定合适的缺省值，这将比现在采用的在编译阶段确定缺省值的机制更慢更复杂。做出这种选择是想求得速度上的提高和实现上的简便，所以大家现在才能感受得到程序运行的高效；当然，如果忽视了本条款的建议，就会带来混乱。

条款 39: 避免 "向下转换" 继承层次

在当今喧嚣的经济时代，关注一下我们的金融机构是个不错的主意。所以，看看下面这个有关银行帐户的协议类(Protocol class)（参见条款 34）：

```
class Person { ... };

class BankAccount {
public:
  BankAccount(const Person *primaryOwner,
              const Person *jointOwner);
  virtual ~BankAccount();

  virtual void makeDeposit(double amount) = 0;
  virtual void makeWithdrawal(double amount) = 0;

  virtual double balance() const = 0;

  ...

};
```

很多银行现在提供了多种令人眼花缭乱的帐户类型，但为简化起见，我们假设只有一种银行帐户，称为存款帐户：

```
class SavingsAccount: public BankAccount {
public:
  SavingsAccount(const Person *primaryOwner,
                 const Person *jointOwner);
  ~SavingsAccount();

  void creditInterest();              // 给帐户增加利息

  ...

};
```

这远远称不上是一个真正的存款帐户，但还是那句话，现在什么年代？至少，它满足我们现在的需要。

银行想为它所有的帐户维持一个列表，这可能是通过标准库（参见条款 49）中的 list 类模板实现的。假设列表被叫做 allAccounts：

```
list<BankAccount*> allAccounts;        // 银行中所有帐户
```

和所有的标准容器一样，list 存储的是对象的拷贝，所以，为避免每个 BankAccount 存储多个拷贝，银行决定让 allAccounts 保存 BankAccount 的指针，而不是 BankAccount 本身。

假设现在准备写一段代码来遍历所有的帐户，为每个帐户计算利息。你会这么写：

```
// 不能通过编译的循环（如果你以前从没
// 见过使用 "迭代子" 的代码，参见下文）
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  (*p)->creditInterest();      // 错误!

}
```

但是，编译器很快就会让你认识到：allAccounts 包含的指针指向的是 BankAccount 对象，而非 SavingsAccount 对象，所以每次循环，p 指向的是一个 BankAccount。这使得对 creditInterest 的调用无效，因为 creditInterest 只是为 SavingsAccount 对象声明的，而不是 BankAccount。

如果"list<BankAccount*>::iterator p = allAccounts.begin()" 在你看来更象电话线中的噪音，而不是 C++，那很显然，你以前无缘见识过 C++标准库中的容器类模板。标准库中的这一部分通常被称为标准模板库（STL），你可以在条款 49 和 M35 初窥其概貌。但现在你只用知道，变量 p 工作起来就象一个指针，它将 allAccounts 中的元素从头到尾循环一遍。也就是说，p 工作起来就好象它的类型是 BankAccount**而列表中的元素都存储在一个数组中。

上面的循环不能通过编译很令人泄气。的确，allAccounts 是被定义为保存 BankAccount*，但要知道，上面的循环中它事实上保存的是 SavingsAccount*，因为 SavingsAccount 是仅有的可以被实例话的类。愚蠢的编译器！对我们来说这么显然的事情它竟然笨得一无所知。所以你决定告诉它：allAccounts 真的包含的是 SavingsAccount*：

```
// 可以通过编译的循环，但很糟糕
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  static_cast<SavingsAccount*>(*p)->creditInterest();

}
```

一切问题迎刃而解！解决得很清晰，很漂亮，很简明，所做的仅仅是一个简单的转换而已。你知道 allAccounts 指针保存的是什么类型的指针，迟钝的编译器不知道，所以你通过一个转换来告诉它，还有比这更合理的事吗？

在此，我要拿圣经的故事做比喻。转换之于 C++程序员，就象苹果之于夏娃。

这种类型的转换 ---- 从一个基类指针到一个派生类指针 ---- 被称为 "向下转换"，因为它向下转换了继承的层次结构。在刚看到的例子中，向下转换碰巧可以工作；但正如下面即将看到的，它将给今后的维护人员带来恶梦。

还是回到银行的话题上来。受到存款帐户业务大获成功的激励，银行决定再推出支票帐户业务。另外，假设支票帐户和存款帐户一样，也要负担利息：

```
class CheckingAccount: public BankAccount {
public:
  void creditInterest();    // 给帐户增加利息

  ...

};
```

不用说，allAccounts 现在是一个包含存款和支票两种帐户指针的列表。于是，上面所写的计算利息的循环转瞬间有了大麻烦。

第一个问题是，虽然新增了一个 CheckingAccount，但即使不去修改循环代码，编译还是可以继续通过。因为编译器只是简单地听信于你所告诉它们（通过 static_cast）的一切：*p 指向的是 SavingsAccount*。谁叫你是它的主人呢？这会给今后维护带来第一个恶梦。维护期第二个恶梦在于，你一定想去解决这个问题，所以你会写出这样的代码：

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  if (*p 指向一个 SavingsAccount)
    static_cast<SavingsAccount*>(*p)->creditInterest();
  else
    static_cast<CheckingAccount*>(*p)->creditInterest();

}
```

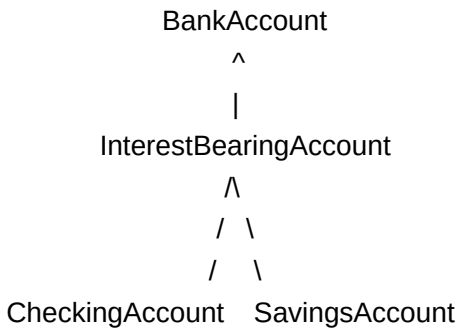
任何时候发现自己写出 "如果对象属于类型 T1，做某事；但如果属于类型 T2，做另外某事" 之类的代码，就要扇自己一个耳光。这不是 C++的做法。是的，在 C，Pascal，甚至 Smalltalk 中，它是很合理的做法，但在 C++中不是。在 C++中，要使用虚函数。

记得吗？对于一个虚函数，编译器可以根据所使用对象的类型来保证正确的函数调用。所以不要在代码中随处乱扔条件语句或开关语句；让编译器来为你效劳。如下所示：

```
class BankAccount { ... };      // 同上

// 一个新类，表示要支付利息的帐户
class InterestBearingAccount: public BankAccount {
```

```
public:
  virtual void creditInterest() = 0;

  ...

};

class SavingsAccount: public InterestBearingAccount {

  ...                        // 同上

};

class CheckingAccount: public InterestBearingAccount {

  ...                        // as above

};
```

用图形表示如下：



因为存款和支票账户都要支付利息，所以很自然地想到把这一共同行为转移到一个公共的基类中。但是，如果假设不是所有的银行帐户都需要支付利息（以我的经验，这当然是个合理的假设），就不能把它转移到 BankAccount 类中。所以，要为 BankAccount 引入一个新的子类 InterestBearingAccount，并使 SavingsAccoun 和 CheckingAccount 从它继承。

存款和支票账户都要支付利息的事实是通过 InterestBearingAccount 的纯虚函数 creditInterest 来体现的，它要在子类 SavingsAccount 和 CheckingAccount 中重新定义。

有了新的类层次结构，就可以这样来重写循环代码：

```
// 好一些，但还不完美
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  static_cast<InterestBearingAccount*>(*p)->creditInterest();
```

```
}
```

尽管这个循环还是包含一个讨厌的转换，但代码已经比过去健壮多了，因为即使又增加 InterestBearingAccount 新的子类到程序中，它还是可以继续工作。

为了完全消除转换，就必须对设计做一些改变。一种方法是限制帐户列表的类型。如果能得到一列 InterestBearingAccount 对象而不是 BankAccount 对象，那就太好了：

```
// 银行中所有要支付利息的帐户
list<InterestBearingAccount*> allIBAccounts;

// 可以通过编译且现在将来都可以工作的循环
for (list<InterestBearingAccount*>::iterator p =
        allIBAccounts.begin();
     p != allIBAccounts.end();
     ++p) {

  (*p)->creditInterest();

}
```

如果不想用上面这种 "采用更特定的列表" 的方法，那就让 creditInterest 操作使用于所有的银行帐户，但对于不用支付利息的帐户来说，它只是一个空操作。这个方法可以这样来表示：

```
class BankAccount {
public:
  virtual void creditInterest() {}

  ...

};

class SavingsAccount: public BankAccount { ... };
class CheckingAccount: public BankAccount { ... };
list<BankAccount*> allAccounts;
// 看啊，没有转换!
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  (*p)->creditInterest();

}
```

要注意的是，虚函数 BankAccount::creditInterest 提供一个了空的缺省实现。这可以很方便地表示，它的行为在缺省情况下是一个空操作；但这也会给它本身带来难以预见的问题。想

知道内幕，以及如何消除这一危险，请参考条款 36。还要注意的是，creditInterest 是一个（隐式的）内联函数，这本身没什么问题；但因为它同时又是一个虚函数，内联指令就有可能被忽略。条款 33 解释了为什么。

正如上面已经看到的，"向下转换" 可以通过几种方法来消除。最好的方法是将这种转换用虚函数调用来代替，同时，它可能对有些类不适用，所以要使这些类的每个虚函数成为一个空操作。第二个方法是加强类型约束，使得指针的声明类型和你所知道的真的指针类型之间没有出入。为了消除向下转换，无论费多大工夫都是值得的，因为向下转换难看、容易导致错误，而且使得代码难于理解、升级和维护（参见条款 M32）。

至此，我所说的都是事实；但，不是全部事实。有些情况下，真的不得不执行向下转换。

例如，假设还是面临本条款开始的那种情况，即，allAccounts 保存 BankAccount 指针，creditInterest 只是为 SavingsAccount 对象定义，要写一个循环来为每个帐户计算利息。进一步假设，你不能改动这些类；你不能改变 BankAccount，SavingsAccount 或 allAccounts 的定义。（如果它们在某个只读的库中定义，就会出现这种情况）如果是这样的话，你就只有使用向下转换了，无论你认为这个办法有多丑陋。

尽管如此，还是有比上面那种原始转换更好的办法。这种方法称为 "安全的向下转换"，它通过 C++的 dynamic_cast 运算符（参见条款 M2）来实现。当对一个指针使用 dynamic_cast 时，先尝试转换，如果成功（即，指针的动态类型(见条款 38)和正被转换的类型一致），就返回新类型的合法指针；如果 dynamic_cast 失败，返回空指针。

下面就是加上了 "安全向下转换" 的例子：

```
class BankAccount { ... };          // 和本条款开始时一样

class SavingsAccount:               // 同上
  public BankAccount { ... };

class CheckingAccount:              // 同上
  public BankAccount { ... };

list<BankAccount*> allAccounts;     // 看起来应该熟悉些了吧...

void error(const string& msg);      // 出错处理函数;
                                    // 见下文

// 嗯，至少转换很安全
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  // 尝试将*p 安全转换为 SavingsAccount*;
  // psa 的定义信息见下文
  if (SavingsAccount *psa =
```

```
        dynamic_cast<SavingsAccount*>(*p)) {
    psa->creditInterest();
  }

  // 尝试将它安全转换为 CheckingAccount
  else if (CheckingAccount *pca =
           dynamic_cast<CheckingAccount*>(*p)) {
    pca->creditInterest();
  }

  // 未知的帐户类型
  else {
    error("Unknown account type!");
  }
}
```

这种方法远不够理想，但至少可以检测到转换失败，而用 dynamic_cast 是无法做到的。但要注意，对所有转换都失败的情况也要检查。这正是上面代码中最后一个 else 语句的用意所在。采用虚函数，就不必进行这样的检查，因为每个虚函数调用必然都会被解析为某个函数。然而，一旦打算进行转换，这一切好处都化为乌有。例如，如果某个人在类层次结构中增加了一种新类型的帐户，但又忘了更新上面的代码，所有对它的转换就会失败。所以，处理这种可能发生的情况十分重要。大部分情况下，并非所有的转换都会失败；但是，一旦允许转换，再好的程序员也会碰上麻烦。

上面 if 语句的条件部分，有些看上去象变量定义的东西，看到它你是不是慌张地擦了擦眼镜？如果真这样，别担心，你没看错。这种定义变量的方法是和 dynamic_cast 同时增加到 C++语言中的。这一特性使得写出的代码更简洁，因为对 psa 或 pca 来说，它们只有在被 dynamic_cast 成功初始化的情况下，才会真正被用到；使用新的语法，就不必在（包含转换的）条件语句外定义这些变量。（条款 32 解释了为什么通常要避免多余的变量定义）如果编译器尚不支持这种定义变量的新方法，可以按老方法来做：

```
for (list<BankAccount*>::iterator p = allAccounts.begin();
     p != allAccounts.end();
     ++p) {

  SavingsAccount *psa;        // 传统定义
  CheckingAccount *pca;       // 传统定义

  if (psa = dynamic_cast<SavingsAccount*>(*p)) {
    psa->creditInterest();
  }

  else if (pca = dynamic_cast<CheckingAccount*>(*p)) {
    pca->creditInterest();
  }
```

```
  else {
    error("Unknown account type!");
  }
}
```

当然，从处理事情的重要性来说，把 psa 和 pca 这样的变量放在哪儿定义并不十分重要。重要之处在于：用 if-then-else 风格的编程来进行向下转换比用虚函数要逊色得多，应该将这种方法保留到万不得已的情况下使用。运气好的话，你的程序世界里将永远看不到这样悲惨荒凉的景象。

条款 40: 通过分层来体现 "有一个" 或 "用...来实现"

使某个类的对象成为另一个类的数据成员，从而实现将一个类构筑在另一个类之上，这一过程称为 "分层"(Layering)。例如：

```
class Address { ... };          // 某人居住之处

class PhoneNumber { ... };

class Person {
public:
  ...

private:
  string name;                  // 下层对象
  Address address;              // 同上
  PhoneNumber voiceNumber;      // 同上
  PhoneNumber faxNumber;        // 同上
};
```

本例中，Person 类被认为是置于 string，Address 和 PhoneNumber 类的上层，因为它包含那些类型的数据成员。"分层" 这一术语有很多同义词，它也常被称为：构成(composition)，包含(containment)或嵌入(embedding)。

条款 35 解释了公有继承的含义是 "是一个"。对应地，分层的含义是 "有一个" 或 "用...来实现"。

上面的 Person 类展示了 "有一个" 的关系。一个 Person 对象 "有一个" 名字，地址，电话号码和传真号码。你不能说，一个人 "是一个" 名字或一个人 "是一个" 地址；你得说，一个人 "有一个" 名字， "有一个" 地址，等等。大多数人对区分这些没什么困难，所以混淆 "是一个" 和 "有一个" 的情况相对来说比较少见。

稍微有点麻烦的是区分 "是一个" 和 "用...来实现"。例如，假设需要一个类模板，用来表示任意对象的集合，并且集合中没有重复元素。程序设计中，重用（Reuse）是再好不过的一件事了，而且你也许已经读过条款 49 中关于 C++标准库的总体介绍，那么，你的第一反应一定是想采用标准库中的 set 模板。是啊，既然可以使用别人所写的东西，为什么还要再去写一个新的模板呢？

但是，深入研究 set 的帮助文档后，你会发现，set 的下述限制将不能满足你的程序要求：set 要求包含在它内部的元素必须是完全有序的，即，对 set 中的任两个元素 a 和 b 来说，一定可以确定：要么 a<b，要么 b<a。对许多类型来说，这个要求很容易满足，而且，对象间完全有序使得 set 可以在性能方面提供某些保证，这一点很吸引人。（参见条款 49 了解标准库在性能上更多的保证）然而，你所需要的是更广泛的东西：一个类似 set 的类，但对象不必完全有序；用 C++标准所包装的术语来说，它们只需要所谓的 "相等可比较性"：对

于同种类型的 a 和 b 对象来说，要能确定是否 a==b。这种要求更简单，它更适合于那些表示颜色这类东西的类型。总不能说红色比绿色更少或绿色比红色更少吧？看来，对你的程序来说，还是得需要自己来写个模板。

当然，重用还是件好事。作为数据结构专家，你知道，在实现集合的众多选择中，一个最简单的办法是采用链表。你一定猜到了什么。对，标准库中正有这么一个 list 模板（用来产生链表类）！所以可以重用它。

具体来说，你决定让自己的 Set 模板从 list 继承。即，Set<T>将从 list<T>继承。因为，在你的实现中，Set 对象实际上将是 list 对象。于是你这样声明 Set 模板：

```
// Set 中错误地使用了 list
template<class T>
class Set: public list<T> { ... };
```

至此，一切好象都很正确，但实际上错误不小。正如条款 35 所说明的，如果 D "是一个" B，对 B 成立的所有事实对 D 也成立。但是，list 对象可以包含重复元素，所以如果 3051 这个值被增加到 list<int>中两次，list 中将包含 3051 的两个拷贝。相反，Set 不可以包含重复元素，所以如果 3051 被增加到 Set<int>中两次，Set 中将只包含这个值的一个拷贝。于是，说一个 Set "是一个" list 就是弥天大谎，因为如上所述，有一些在 list 对象中成立的事实在 Set 对象中不成立。

因为这两个类的关系并非 "是一个"，所以用公有继承来表示它们的关系就是一个错误。正确的方法是让 Set 对象 "用 list 对象来实现"：

```
// Set 中使用 list 的正确方法
template<class T>
class Set {
public:
  bool member(const T& item) const;

  void insert(const T& item);
  void remove(const T& item);

  int cardinality() const;

private:
  list<T> rep;                    // 表示一个 Set
};
```

Set 的成员函数可以利用 list 以及标准库其它部分所提供的大量功能，所以，实现代码既不难写也很易读：

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

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

template<class T>
void Set<T>::remove(const T& item)
{
  list<T>::iterator it =
    find(rep.begin(), rep.end(), item);

  if (it != rep.end()) rep.erase(it);
}

template<class T>
int Set<T>::cardinality() const
{ return rep.size(); }
```

这些函数很简单，所以很自然地想到将它们作为内联函数；但在做最后决定前，还是回顾一下条款 33 所做的讨论。（上面的代码中，find, begin, end, push_back 等函数是标准库基本框架的一部分，它们可用来对 list 这样的容器模板进行操作。标准库框架的总体介绍参见条款 49 和 M35。）

值得指出的是，Set 类的接口没有做到完整并且最小（参见条款 18）。从完整性上来说，它最大的遗漏在于不能对 Set 中的内容进行循环，而这一功能对很多程序来说是必需的（标准库中的所有成员都提供了这一功能，包括 set）。Set 的另一个缺陷是没有遵循标准库所采用的容器类常规（见条款 49 和 M35），从而造成使用 Set 时更难以利用库中其它的部分。

Set 的接口尽管有这些瑕疵，但下面这一点不能被掩盖：Set 在理解它和 list 的关系上，具有无可辩驳的正确性。这种关系并非 "是一个"（虽然初看会以为是），而是 "用...来实现"，通过分层来实现这种关系是类的设计者应该感到自豪的。

顺便说一句，当通过分层使两个类产生联系时，实际上在两个类之间建立了编译时的依赖关系。关于为什么要考虑到这一点以及如何减少这方面的麻烦，参见条款 34。

条款 41: 区分继承和模板

考虑下面两个设计问题：

· 作为一位立志献身计算机科学的学生，你想设计一个类来表示对象的堆栈。这将需要多个不同的类，因为每个堆栈中的元素必须是同类的，即，它里面包含的必须只是同种类型的对象。例如，会有一个类来表示 int 的堆栈，第二个类来表示 string 的堆栈，第三个类来表示 string 的堆栈的堆栈，等等。你也许对设计一个最小的类接口（参见条款 18）很感兴趣，所以会将对堆栈的操作限制在：创建堆栈，销毁堆栈，将对象压入堆栈，将对象弹出堆栈，以及检查堆栈是否为空。设计中，你不会借助标准库中的类（包括 stack ---- 参见条款 49），因为你渴望亲手写这些代码。重用（Reuse）是一件美事，但当你的目标是探究事情的工作原理时，那就只有挖地三尺了。

· 作为一位爱猫的宠物迷，你想设计一个类来表示猫。这也将需要多个不同的类，因为每个品种的猫都会有点不同。和所有对象一样，猫可以被创建和销毁，但，正如所有猫迷所知道的，猫所做的其它事不外乎吃和睡。然而，每一种猫吃和睡都有各自惹人喜爱的方式。

这两个问题的说明听起来很相似，但却导致完全不同的两种设计。为什么？

答案涉及到"类的行为" 和 "类所操作的对象的类型"之间的关系。对于堆栈和猫来说，要处理的都是各种不同的类型（堆栈包含类型为 T 的对象，猫则为品种 T），但你必须问自己这样一个问题：类型 T 影响类的行为吗？如果 T 不影响行为，你可以使用模板。如果 T 影响行为，你就需要虚函数，从而要使用继承。

下面的代码通过定义一个链表来实现 Stack 类，假设堆栈的对象类型为 T：

```
class Stack {
public:
  Stack();
  ~Stack();

  void push(const T& object);
  T pop();

  bool empty() const;            // 堆栈为空?

private:
  struct StackNode {             // 链表节点
    T data;                    // 此节点数据
    StackNode *next;             // 链表中下一节点

    // StackNode 构造函数，初始化两个域
    StackNode(const T& newData, StackNode *nextNode)
```

```
    : data(newData), next(nextNode) {}
  };

  StackNode *top;                // 堆栈顶部

  Stack(const Stack& rhs);              // 防止拷贝和
  Stack& operator=(const Stack& rhs);    // 赋值(见条款 27)
};
```

于是，Stack 对象将构造如下所示的数据结构：

Stack 对象 top--> data+next--> data+next--> data+next--> data+next

---

StackNode 对象

链表本身是由 StackNode 对象构成的，但那只是 Stack 类的一个实现细节，所以 StackNode 被声明为 Stack 的私有类型。注意 StackNode 有一个构造函数，用来确保它所有的域都被正确初始化。即使你闭着眼睛都可以写出一个链表，但也不要忽视了 C++的一些新特性，如 struct 中的构造函数。

下面看看你对 Stack 成员函数的实现。和许多原型（prototype）的实现（离制作成软件产品相差太远）一样，这里没有错误检查，因为在原型世界里，没有东西会出错。

```
Stack::Stack(): top(0) {}      // 顶部初始化为 null

void Stack::push(const T& object)
{
  top = new StackNode(object, top);    // 新节点放在
}                                // 链表头部

T Stack::pop()
{
  StackNode *topOfStack = top;    // 记住头节点
  top = top->next;

  T data = topOfStack->data;      // 记住节点数据
  delete topOfStack;

  return data;
}

Stack::~Stack()                  // 删除堆栈中所有对象
{
  while (top) {
    StackNode *toDie = top;       // 得到头节点指针
    top = top->next;              // 移向下一节点
```

```
    delete toDie;                // 删除前面的头节点
  }
}

bool Stack::empty() const
{ return top == 0; }
```

这些代码毫无吸引人之处。实际上，唯一有趣的一点在于：即使对 T 一无所知，你还是能够写出每个成员函数。（上面的代码中实际上有个假设，即，假设可以调用 T 的拷贝构造函数；但正如条款 45 所说明的，这是一个绝对合理的假设）不管 T 是什么，对构造，销毁，压栈，出栈，确定栈是否为空等操作所写的代码不会变。除了 "可以调用 T 的拷贝构造函数" 这一假设外，stack 的行为在任何地方都不依赖于 T。这就是模板类的特点：行为不依赖于类型。

将 stack 类转化成一个模板就很简单了，即使是 Dilbert 的老板都会写：

```
template<class T> class Stack {

  ...                        // 完全和上面相同

};
```

但是，猫呢？为什么猫不适合模板？

重读上面的说明，注意这一条："每一种猫吃和睡都有各自惹人喜爱的方式"。这意味着必须为每种不同的猫实现不同的行为。不可能写一个函数来处理所有的猫，所能做的只能是制定一个函数接口，所有种类的猫都必须实现它。啊哈！衍生一个函数接口的方法只能是去声明一个纯虚函数（参见条款 36）：

```
class Cat {
public:
  virtual ~Cat();                  // 参见条款 14

  virtual void eat() = 0;          // 所有的猫吃食
  virtual void sleep() = 0;        // 所有的猫睡觉
};
```

Cat 的子类 ---- 比如，Siamese 和 BritishShortHairedTabby ---- 当然得重新定义继承而来的 eat 和 sleep 函数接口：

```
class Siamese: public Cat {
public:
  void eat();
  void sleep();

  ...
```

```
};

class BritishShortHairedTabby: public Cat {
public:
  void eat();
  void sleep();

  ...

};
```

好了，现在知道了为什么模板适合 Stack 类而不适合 Cat 类，也知道了为什么继承适合 Cat 类。唯一剩下的问题是，为什么继承不适合 Stack 类。想知道为什么，不妨试着去声明一个 Stack 层次结构的根类 ---- 所有其它的堆栈类都从这个唯一的类继承：

```
class Stack {       // a stack of anything
public:
  virtual void push(const ??? object) = 0;
  virtual ??? pop() = 0;

  ...

};
```

现在问题很明显了。该为纯虚函数 push 和 pop 声明什么类型呢？记住，每一个子类必须重新声明继承而来的虚函数，而且参数类型和返回类型都要和基类的声明完全相同。不幸的是，一个 int 堆栈只能压入和弹出 int 对象，而一个 Cat 堆栈只能压入和弹出 Cat 对象。Stack 类要怎样声明它的纯虚函数才能使用户既可以创建出 int 堆栈又可以创建出 Cat 堆栈呢？冷酷而严峻的事实是，做不到。这就是为什么说继承不适合创建堆栈。

但也许你做事喜欢偷偷摸摸。或许你认为自己可以通过使用通用(void*)指针来骗过编译器。但事实证明，现在这种情况下，通用指针也帮不上忙。因为你无法避开这一条件：派生类虚函数的声明永远不能和它在基类中的声明相抵触。但是，通用指针可以帮助解决另外一个不同的问题，它和模板所生成的类的效率有关。详细介绍参见条款 42。

讲完了堆栈和猫，下面将本条款得到的结论总结如下：

· 当对象的类型不影响类中函数的行为时，就要使用模板来生成这样一组类。
· 当对象的类型影响类中函数的行为时，就要使用继承来得到这样一组类。

真正消化了以上两点的含义，你就可以在设计中游刃于继承或模板之间。

条款 42: 明智地使用私有继承

条款 35 说明，C++将公有继承视为 "是一个" 的关系。它是通过这个例子来证实的：假如某个类层次结构中，Student 类从 Person 类公有继承，为了使某个函数成功调用，编译器可以在必要时隐式地将 Student 转换为 Person。这个例子很值得再看一遍，只是现在，公有继承换成了私有继承：

```
class Person { ... };

class Student:                    // 这一次我们
  private Person { ... };         // 使用私有继承

void dance(const Person& p);      // 每个人会跳舞

void study(const Student& s);     // 只有学生才学习


Person p;                         // p 是一个人
Student s;                        // s 是一个学生

dance(p);                         // 正确, p 是一个人

dance(s);                         // 错误！一个学生不是一个人
```

很显然，私有继承的含义不是 "是一个"，那它的含义是什么呢？

"别忙!" 你说。"在弄清含义之前，让我们先看看行为。私有继承有那些行为特征呢？" 那好吧。关于私有继承的第一个规则正如你现在所看到的：和公有继承相反，如果两个类之间的继承关系为私有，编译器一般不会将派生类对象（如 Student）转换成基类对象（如 Person）。这就是上面的代码中为对象 s 调用 dance 会失败的原因。第二个规则是，从私有基类继承而来的成员都成为了派生类的私有成员，即使它们在基类中是保护或公有成员。行为特征就这些。

这为我们引出了私有继承的含义：私有继承意味着 "用...来实现"。如果使类 D 私有继承于类 B，这样做是因为你想利用类 B 中已经存在的某些代码，而不是因为类型 B 的对象和类型 D 的对象之间有什么概念上的关系。因而，私有继承纯粹是一种实现技术。用条款 36 引入的术语来说，私有继承意味着只是继承实现，接口会被忽略。如果 D 私有继承于 B，就是说 D 对象在实现中用到了 B 对象，仅此而已。私有继承在软件 "设计" 过程中毫无意义，只是在软件 "实现" 时才有用。

私有继承意味着 "用...来实现" 这一事实会给程序员带来一点混淆，因为条款 40 指出，"分层" 也具有相同的含义。怎么在二者之间进行选择呢？答案很简单：尽可能地使用分层，必须时才使用私有继承。什么时候必须呢？这往往是指有保护成员和/或虚函数介入的时候 ---- 但这个问题过一会儿再深入讨论。

条款 41 提供了一种方法来写一个 Stack 模板，此模板生成的类保存不同类型的对象。你应该熟悉一下那个条款。模板是 C++最有用的组成部分之一，但一旦开始经常性地使用它，你会发现，如果实例化一个模板一百次，你就可能实例化了那个模板的代码一百次。例如 Stack 模板，构成 Stack<int>成员函数的代码和构成 Stack<double>成员函数的代码是完全分开的。有时这是不可避免的，但即使模板函数实际上可以共享代码，这种代码重复还是可能存在。这种目标代码体积的增加有一个名字：模板导致的 "代码膨胀"。这不是件好事。

对于某些类，可以采用通用指针来避免它。采用这种方法的类存储的是指针，而不是对象，实现起来就是：

· 创建一个类，它存储的是对象的 void*指针。
· 创建另外一组类，其唯一目的是用来保证类型安全。这些类都借助第一步中的通用类来完成实际工作。

下面的例子使用了条款 41 中的非模板 Stack 类，不同的是这里存储的是通用指针，而不是对象：

```
class GenericStack {
public:
  GenericStack();
  ~GenericStack();

  void push(void *object);
  void * pop();

  bool empty() const;

private:
  struct StackNode {
    void *data;                 // 节点数据
    StackNode *next;            // 下一节点

    StackNode(void *newData, StackNode *nextNode)
    : data(newData), next(nextNode) {}
  };

  StackNode *top;                     // 栈顶

  GenericStack(const GenericStack& rhs);   // 防止拷贝和
  GenericStack&                       // 赋值(参见
    operator=(const GenericStack& rhs);    // 条款 27)
};
```

因为这个类存储的是指针而不是对象，就有可能出现一个对象被多个堆栈指向的情况（即，

被压入到多个堆栈）。所以极其重要的一点是，pop 和类的析构函数销毁任何 StackNode 对象时，都不能删除 data 指针 ---- 虽然还是得要删除 StackNode 对象本身。毕竟，StackNode 对象是在 GenericStack 类内部分配的，所以还是得在类的内部释放。所以，条款 41 中 Stack 类的实现几乎完全满足 the GenericStack 的要求。仅有的改变只是用 void* 来替换 T。

仅仅有 GenericStack 这一个类是没有什么用处的，但很多人会很容易误用它。例如，对于一个用来保存 int 的堆栈，一个用户会错误地将一个指向 Cat 对象的指针压入到这个堆栈中，但编译却会通过，因为对 void*参数来说，指针就是指针。

为了重新获得你所习惯的类型安全，就要为 GenericStack 创建接口类（interface class），象这样：

```
class IntStack {                // int 接口类
public:
  void push(int *intPtr) { s.push(intPtr); }
  int * pop() { return static_cast<int*>(s.pop()); }
  bool empty() const { return s.empty(); }

private:
  GenericStack s;               //  实现
};

class CatStack {                // cat 接口类
public:
  void push(Cat *catPtr) { s.push(catPtr); }
  Cat * pop() { return static_cast<Cat*>(s.pop()); }
  bool empty() const { return s.empty(); }

private:
  GenericStack s;               //  实现
};
```

正如所看到的，IntStack 和 CatStack 只是适用于特定类型。只有 int 指针可以被压入或弹出 IntStack，只有 Cat 指针可以被压入或弹出 CatStack。IntStack 和 CatStack 都通过 GenericStack 类来实现，这种关系是通过分层（参见条款 40）来体现的，IntStack 和 CatStack 将共享 GenericStack 中真正实现它们行为的函数代码。另外，IntStack 和 CatStack 所有成员函数是（隐式）内联函数，这意味着使用这些接口类所带来的开销几乎是零。

但如果有些用户没认识到这一点怎么办？如果他们错误地认为使用 GenericStack 更高效，或者，如果他们鲁莽而轻率地认为类型安全不重要，那该怎么办？怎么才能阻止他们绕过 IntStack 和 CatStack 而直接使用 GenericStack（这会让他们很容易地犯类型错误，而这正是设计 C++所要特别避免的）呢？

没办法！没办法防止。但，也许应该有什么办法。

在本条款的开始我就提到，要表示类之间 "用...来实现" 的关系，有一个选择是通过私有继承。现在这种情况下，这一技术就比分层更有优势，因为通过它可以让你告诉别人：GenericStack 使用起来不安全，它只能用来实现其它的类。具体做法是将 GenericStack 的成员函数声明为保护类型：

```
class GenericStack {
protected:
  GenericStack();
  ~GenericStack();

  void push(void *object);
  void * pop();

  bool empty() const;

private:
  ...                          // 同上
};

GenericStack s;                // 错误! 构造函数被保护


class IntStack: private GenericStack {
public:
  void push(int *intPtr) { GenericStack::push(intPtr); }
  int * pop() { return static_cast<int*>(GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};

class CatStack: private GenericStack {
public:
  void push(Cat *catPtr) { GenericStack::push(catPtr); }
  Cat * pop() { return static_cast<Cat*>(GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};

IntStack is;                   // 正确

CatStack cs;                   // 也正确
```

和分层的方法一样，基于私有继承的实现避免了代码重复，因为这个类型安全的接口类只包含有对 GenericStack 函数的内联调用。

在 GenericStack 类之上构筑类型安全的接口是个很花俏的技巧，但需要手工去写所有那些接口类是件很烦的事。幸运的是，你不必这样。你可以让模板来自动生成它们。下面是一个模板，它通过私有继承来生成类型安全的堆栈接口：

```
template<class T>
class Stack: private GenericStack {
public:
  void push(T *objectPtr) { GenericStack::push(objectPtr); }
  T * pop() { return static_cast<T*>(GenericStack::pop()); }
  bool empty() const { return GenericStack::empty(); }
};
```

这是一段令人惊叹的代码，虽然你可能一时还没意识到。因为这是一个模板，编译器将根据你的需要自动生成所有的接口类。因为这些类是类型安全的，用户类型错误在编译期间就能发现。因为 GenericStack 的成员函数是保护类型，并且接口类把 GenericStack 作为私有基类来使用，用户将不可能绕过接口类。因为每个接口类成员函数被（隐式）声明为 inline，使用这些类型安全的类时不会带来运行开销；生成的代码就象用户直接使用 GenericStack 来编写的一样（假设编译器满足了 inline 请求 ---- 参见条款 33）。因为 GenericStack 使用了 void*指针，操作堆栈的代码就只需要一份，而不管程序中使用了多少不同类型的堆栈。简而言之，这个设计使代码达到了最高的效率和最高的类型安全。很难做得比这更好。

本书的基本认识之一是，C++的各种特性是以非凡的方式相互作用的。这个例子，我希望你能同意，确实是非凡的。

从这个例子中可以发现，如果使用分层，就达不到这样的效果。只有继承才能访问保护成员，只有继承才使得虚函数可以重新被定义。（虚函数的存在会引发私有继承的使用，例子参见条款 43）因为存在虚函数和保护成员，有时私有继承是表达类之间 "用...来实现" 关系的唯一有效途径。所以，当私有继承是你可以使用的最合适的实现方法时，就要大胆地使用它。同时，广泛意义上来说，分层是应该优先采用的技术，所以只要有可能，就要尽量使用它。

条款 43: 明智地使用多继承

要看是谁来说，多继承（MI）要么被认为是神来之笔，要么被当成是魔鬼的造物。支持者宣扬说，它是对真实世界问题进行自然模型化所必需的；而批评者争论说，它太慢，难以实现，功能却不比单继承强大。更让人为难的是，面向对象编程语言领域在这个问题上至今仍存在分歧：C++，Eiffel 和 the Common LISP Object System (CLOS)提供了 MI；Smalltalk，Objective C 和 Object Pascal 没有提供；而 Java 只是提供有限的支持。可怜的程序员该相信谁呢？

在相信任何事情之前，首先得弄清事实。C++中，关于 MI 一条不容争辩的事实是，MI 的出现就象打开了潘朵拉的盒子，带来了单继承中绝对不会存在的复杂性。其中，最基本的一条是二义性（参见条款 26）。如果一个派生类从多个基类继承了一个成员名，所有对这个名字的访问都是二义的；你必须明确地说出你所指的是哪个成员。下面的例子取自 ARM（参见条款 50）中的一个专题讨论：

```
class Lottery {
public:
  virtual int draw();

  ...

};

class GraphicalObject {
public:
  virtual int draw();

  ...

};

class LotterySimulation: public Lottery,
                         public GraphicalObject {

  ...                          // 没有声明 draw

};

LotterySimulation *pls = new LotterySimulation;

pls->draw();                   // 错误! ---- 二义
pls->Lottery::draw();          // 正确
pls->GraphicalObject::draw();  // 正确
```

这段代码看起来很笨拙，但起码可以工作。遗憾的是，想避免这种笨拙很难。即使其中一个被继承的 draw 函数是私有成员从而不能被访问，二义还是存在。（对此有一个很好的理由来解释，但完整的说明在条款 26 中提供，所以此处不再重复。）

显式地限制修饰成员不仅很笨拙，而且还带来限制。当显式地用一个类名来限制修饰一个虚函数时，函数的行为将不再具有虚拟的特征。相反，被调用的函数只能是你所指定的那个，即使调用是作用在派生类的对象上：

```
class SpecialLotterySimulation: public LotterySimulation {
public:
  virtual int draw();

  ...

};

pls = new SpecialLotterySimulation;

pls->draw();                    // 错误! ---- 还是有二义
pls->Lottery::draw();           // 调用 Lottery::draw
pls->GraphicalObject::draw();   // 调用 GraphicalObject::draw
```

注意，在这种情况下，即使 pls 指向的是 SpecialLotterySimulation 对象，也无法（没有 "向下转换" ---- 参见条款 39）调用这个类中定义的 draw 函数。

没完，还有呢。Lottery 和 GraphicalObject 中的 draw 函数都被声明为虚函数，所以子类可以重新定义它们（见条款 36），但如果 LotterySimulation 想对二者都重新定义那该怎么办？令人沮丧的是，这不可能，因为一个类只允许有唯一一个没有参数、名称为 draw 的函数。（这个规则有个例外，即一个函数为 const 而另一个不是的时候 ---- 见条款 21）

从某一方面来说，这个问题很严重，严重到足以成为修改 C++语言的理由。ARM 中就讨论了一种可能，即，允许被继承的虚函数可以 "改名" ；但后来又发现，可以通过增加一对新类来巧妙地避开这个问题：

```
class AuxLottery: public Lottery {
public:
  virtual int lotteryDraw() = 0;

  virtual int draw() { return lotteryDraw(); }
};

class AuxGraphicalObject: public GraphicalObject {
public:
  virtual int graphicalObjectDraw() = 0;
```

```
  virtual int draw() { return graphicalObjectDraw(); }
};


class LotterySimulation: public AuxLottery,
                         public AuxGraphicalObject {
public:
  virtual int lotteryDraw();
  virtual int graphicalObjectDraw();

  ...

};
```

这两个新类， AuxLottery 和 AuxGraphicalObject，本质上为各自继承的 draw 函数声明了新的名字。新名字以纯虚函数的形式提供，本例中即 lotteryDraw 和 graphicalObjectDraw；函数是纯虚拟的，所以具体的子类必须重新定义它们。另外，每个类都重新定义了继承而来的 draw 函数，让它们调用新的纯虚函数。最终效果是，在这个类体系结构中，有二义的单个名字 draw 被有效地分成了无二义但功能等价的两个名字：lotteryDraw 和 graphicalObjectDraw：

```
LotterySimulation *pls = new LotterySimulation;

Lottery *pl = pls;
GraphicalObject *pgo = pls;

// 调用 LotterySimulation::lotteryDraw
pl->draw();

// 调用 LotterySimulation::graphicalObjectDraw
pgo->draw();
```

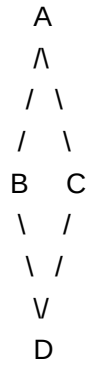
这是一个集纯虚函数，简单虚函数和内联函数（参见条款 33）综合应用之大成的方法，值得牢记在心。首先，它解决了问题，这个问题说不定哪天你就会碰到。其次，它可以提醒你，使用多继承会导致复杂性。是的，这个方法解决了问题，但仅仅为了重新定义一个虚函数而不得不去引入新的类，你真的愿意这样做吗？AuxLottery 和 AuxGraphicalObject 类对于保证类层次结构的正确运转是必需的，但它们既不对应于问题范畴（problem domain ）的某个抽象，也不对应于实现范畴（implementation domain）的某个抽象。它们单纯是作为一种实现设备而存在，再没有别的用处。你一定知道，好的软件是 "设备无关" 的，这条法则在此也适用。

将来使用 MI 还会面临更多的问题，二义性问题（尽管有趣）只不过是刚开始。另一个问题基于这样一个实践经验：一个起初象下面这样的继承层次结构：

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

往往最后悲惨地发展成象下面这样：

```
class A { ... };
class B : virtual public A { ... };
class C : virtual public A { ... };
class D: public B, public C { ... };
```



钻石可能是女孩最好的朋友，也许不是；但肯定的是，象这样一种钻石形状的继承结构绝对不可能成为我们的朋友。如果创建了象这样的层次结构，就会立即面临这样一个问题：是不是该让 A 成为虚基类呢？即，从 A 的继承是否应该是虚拟的呢？现实中，答案几乎总是 ---- 应该；只有极少数情况下会想让类型 D 的对象包含 A 的数据成员的多个拷贝。正是认识到这一事实，上面的 B 和 C 将 A 声明为虚基类。

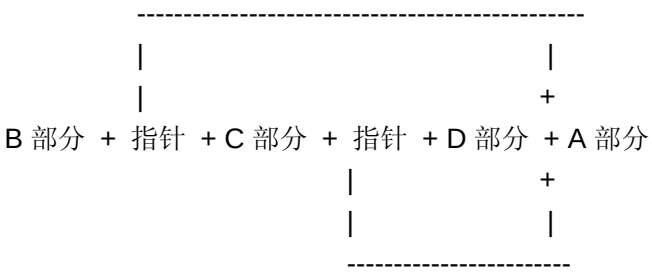
遗憾的是，在定义 B 和 C 的时候，你可能不知道将来是否会有类去同时继承它们，而且知不知道这一点实际上对正确地定义这两个类没有必要。对类的设计者来说，这实在是进退两难。如果不将 A 声明为 B 和 C 的虚基类，今后 D 的设计者就有可能需要修改 B 和 C 的定义，以便更有效地使用它们。通常，这很难做到，因为 A，B 和 C 的定义往往是只读的。例如这样的情况：A，B 和 C 在一个库中，而 D 由库的用户来写。

另一方面，如果真的将 A 声明为 B 和 C 的虚基类，往往会在空间和时间上强加给用户额外的开销。因为虚基类常常是通过对象指针来实现的，并非对象本身。自不必说，内存中对象的分布是和编译器相关的，但一条不变的事实是：如果 A 作为 "非虚" 基类，类型 D 的对象在内存中的分布通常占用连续的内存单元；如果 A 作为 "虚" 基类，有时，类型 D 的对象在内存中的分布占用连续的内存单元，但其中两个单元包含的是指针，指向包含虚基类数据成员的内存单元：

A 是非虚基类时 D 对象通常的内存分布：

A 部分+ B 部分+ A 部分 + C 部分 + D 部分

A 是虚基类时 D 对象在某些编译器下的内存分布：



即使编译器不采用这种特殊的实现策略，使用虚继承通常也会带来某种空间上的惩罚。

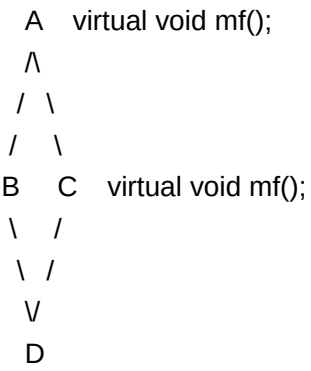
考虑到这些因素，看来，在进行高效的类设计时如果涉及到 MI，作为库的设计者就要具有超凡的远见。然而现在的年代，常识都日益成为了稀有品，因而你会不明智地过多依赖于语言特性，这就不仅要求设计者能够预计得到未来的需要，而且简直就是要你做到彻底的先知先觉（参见条款 M32）。

当然，这也可以说成是在虚函数和非虚函数间选择，但还是有重大的不同。条款 36 说明，虚函数具有定义明确的高级含义，非虚函数也同样具有定义明确的高级含义，而且它们的含义有显著的不同，所以在清楚自己想对子类的设计者传达什么含义的基础上，在二者之间作出选择是可能的。但是，决定基类是否应该是虚拟的，则缺乏定义明确的高级含义；相反，决定通常取决于整个继承的层次结构，所以除非知道了整个层次结构，否则无法做出决定。如果正确地定义出个类之前需要清楚地知道将来怎么使用它，这种情况下将很难设计出高效的类。

就算避开了二义性问题，并且解决了是否应该从基类虚拟继承的疑问，还是会有许多复杂性问题等着你。为了长话短说，在此我仅提出应该记住的其它两点：

· 向虚基类传递构造函数参数。非虚继承时，基类构造函数的参数是由紧临的派生类的成员初始化列表指定的。因为单继承的层次结构只需要非虚基类，继承层次结构中参数的向上传递采用的是一种很自然的方式：第 n 层的类将参数传给第 n-1 层的类。但是，虚基类的构造函数则不同，它的参数是由继承结构中最底层派生类的成员初始化列表指定的。这就造成，负责初始化虚基类的那个类可能在继承图中和它相距很远；如果有新类增加到继承结构中，执行初始化的类还可能改变。（避免这个问题的一个好办法是：消除对虚基类传递构造函数参数的需要。最简单的做法是避免在这样的类中放入数据成员。这本质上是 Java 的解决之道：Java 中的虚基类（即，"接口"）禁止包含数据）

· 虚函数的优先度。就在你自认为弄清了所有的二义之时，它们却又在你面前摇身一变。再次看看关于类 A，B，C 和 D 的钻石形状的继承图。假设 A 定义了一个虚成员函数 mf，C 重定义了它；B 和 D 则没有重定义 mf：



根据以前的讨论，你会认为下面有二义：

```
D *pd = new D;
pd->mf();                    // A::mf 或者 C::mf?
```

该为 D 的对象调用哪个 mf 呢，是直接从 C 继承的还是间接（通过 B）从 A 继承的那个呢？答案取决于 B 和 C 如何从 A 继承。具体来说，如果 A 是 B 或 C 的非虚基类，调用具有二义性；但如果 A 是 B 和 C 的虚基类，就可以说 C 中 mf 的重定义优先度高于最初 A 中的定义，因而通过 pd 对 mf 的调用将（无二义地）解析为 C::mf。如果你坐下来仔细想想，这正是你想要的行为；但需要坐下仔细想想才能弄懂，也确实是一种痛苦。

也许至此你会承认 MI 确实会导致复杂化。也许你认识到每个人其实都不想使用它。也许你准备建议国际 C++标准委员会将多继承从语言中去掉；或者至少你想向你的老板建议，全公司的程序员都禁止使用它。

也许你太性急了。

请记住，C++的设计者并没有想让多继承难以使用；恰恰是，想让一切都能以更合理的方式协调工作，这本身会带来某些复杂性。上面的讨论中你会注意到，这些复杂性很多是由于使用虚基类引起的。如果能避免使用虚基类 ---- 即，如果能避免产生那种致命的钻石形状继承图 ---- 事情就好处理多了。

例如，条款 34 中讲到，协议类（Protocol class）的存在仅仅是为派生类制定接口；它没有数据成员，没有构造函数，有一个虚析构函数（参见条款 14），有一组用来指定接口的纯虚函数。一个 Person 协议类看起来象下面这样：

```
class Person {
public:
  virtual ~Person();

  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};
```

这个类的用户在编程时必须使用 Person 的指针或引用，因为抽象类不能被实例化。

为了创建 "可以作为 Person 对象而使用" 的对象，Person 的用户使用工厂函数（factory function，参见条款 34）来实例化具体的子类：

```
// 工厂函数，从一个唯一的数据库 ID
// 创建一个 Person 对象
Person * makePerson(DatabaseID personIdentifier);

DatabaseID askUserForDatabaseID();


DatabaseID pid = askUserForDatabaseID();

Person *pp = makePerson(pid);    // 创建支持 Person
                                 // 接口的对象

...                              // 通过 Person 的成员函数
                                 // 操作*pp

delete pp;                       // 删除不再需要的对象
```

这就带来一个问题：makePerson 返回的指针所指向的对象如何创建呢？显然，必须从 Person 派生出某种具体类，使得 makePerson 可以对其进行实例化。

假设这个类被称为 MyPerson。作为一个具体类，MyPerson 必须实现从 Person 继承而来的纯虚函数。这可以从零做起，但如果已经存在一些组件可以完成大多数或全部所需的工作，那么从软件工程的角度来说，能利用这些组件将再好不过。例如，假设已经有一个和数据库有关的旧类 PersonInfo，它提供的功能正是 MyPerson 所需要的：

```
class PersonInfo {
public:
  PersonInfo(DatabaseID pid);
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  virtual const char * theAddress() const;
  virtual const char * theNationality() const;

  virtual const char * valueDelimOpen() const;      // 看下文
  virtual const char * valueDelimClose() const;

  ...

};
```

可以断定这是一个很旧的类，因为它的成员函数返回的是 const char*而不是 string 对象。但是，如果鞋合脚，为什么不穿呢？这个类的成员函数名暗示，这双鞋穿上去会很舒服。

随之你会发现，当初设计 PersonInfo 是用来方便地以各种不同格式打印数据库字段，每个字段值的开头和结尾用特殊字符串分开。默认情况下，字段值的起始分隔符和结束分隔符为括号，所以字段值 "Ring-tailed Lemur" 将会这样被格式化：

[Ring-tailed Lemur]

因为括号不是所有 PersonInfo 的用户都想要的，虚函数 valueDelimOpen 和 valueDelimClose 允许派生类指定它们自己的起始分隔符和结束分隔符。PersonInfo 类的 theName，theBirthDate，theAddress 以及 theNationality 的实现将调用这两个虚函数，在它们的返回值中添加适当的分隔符。拿 PersonInfo::name 作为例子，代码看起来象这样：

```
const char * PersonInfo::valueDelimOpen() const
{
  return "[";                  // 默认起始分隔符
}

const char * PersonInfo::valueDelimClose() const
{
  return "]";                  // 默认结束分隔符
}

const char * PersonInfo::theName() const
{
  // 为返回值保留缓冲区。因为是静态
  // 类型，它被自动初始化为全零。
  static char value[MAX_FORMATTED_FIELD_VALUE_LENGTH];

  // 写起始分隔符
  strcpy(value, valueDelimOpen());

  将对象的名字字段值添加到字符串中

  // 写结束分隔符
  strcat(value, valueDelimClose());

  return value;
}
```

有些人会挑剔 PersonInfo::theName 的设计（特别是使用了固定大小的静态缓冲区 ---- 参见条款 23），但请将你的挑剔放在一边，关注这一点：首先，theName 调用 valueDelimOpen，生成它将要返回的字符串的起始分隔符；然后，生成名字值本身；最后，调用

valueDelimClose。因为 valueDelimOpen 和 valueDelimClose 是虚函数，theName 返回的结果既依赖于 PersonInfo，也依赖于从 PersonInfo 派生的类。

作为 MyPerson 的实现者，这是条好消息，因为在研读 Person 文档的细则时你发现，name 及其相关函数需要返回的是不带修饰的值，即，不允许带分隔符。也就是说，如果一个人来自 Madagascar，调用这个人的 nationality 函数将返回"Madagascar"，而不是"[Madagascar]"。

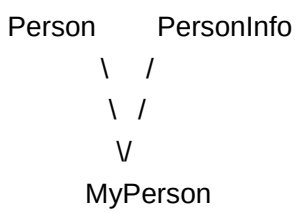
MyPerson 和 PersonInfo 之间的关系是，PersonInfo 刚好有些函数使得 MyPerson 易于实现。仅次而已。没看到有 "是一个" 或 "有一个" 的关系。它们的关系是 "用...来实现"，而且我们知道，这可以用两种方式来表示：通过分层（见条款 40）和通过私有继承（见条款 42）。条款 42 指出，分层一般来说是更好的方法，但在有虚函数要被重新定义的情况下，需要使用私有继承。现在的情况是，MyPerson 需要重新定义 valueDelimOpen 和 valueDelimClose，所以不能用分层，而必须用私有继承：MyPerson 必须从 PersonInfo 私有继承。

但 MyPerson 还必须实现 Person 接口，因而需要公有继承。这导致了多继承一个很合理的应用：将接口的公有继承和实现的私有继承结合起来：

```
class Person {                    // 这个类指定了
public:                           // 需要被实现
  virtual ~Person();              // 的接口

  virtual string name() const = 0;
  virtual string birthDate() const = 0;
  virtual string address() const = 0;
  virtual string nationality() const = 0;
};

class DatabaseID { ... };         // 被后面的代码使用；
                                  // 细节不重要

class PersonInfo {                // 这个类有些有用
public:                           // 的函数，可以用来
  PersonInfo(DatabaseID pid);     // 实现 Person 接口
  virtual ~PersonInfo();

  virtual const char * theName() const;
  virtual const char * theBirthDate() const;
  virtual const char * theAddress() const;
  virtual const char * theNationality() const;

  virtual const char * valueDelimOpen() const;
  virtual const char * valueDelimClose() const;
```

```
  ...

};


class MyPerson: public Person,        // 注意，使用了
                private PersonInfo {   // 多继承
public:
  MyPerson(DatabaseID pid): PersonInfo(pid) {}

  // 继承来的虚分隔符函数的重新定义
  const char * valueDelimOpen() const { return ""; }
  const char * valueDelimClose() const { return ""; }

  // 所需的 Person 成员函数的实现
  string name() const
  { return PersonInfo::theName(); }

  string birthDate() const
  { return PersonInfo::theBirthDate(); }

  string address() const
  { return PersonInfo::theAddress(); }

  string nationality() const
  { return PersonInfo::theNationality(); }
};
```

用图形表示，看起来象下面这样：



这种例子证明，MI 会既有用又易于理解，尽管可怕的钻石形状继承图不会明显消失。

然而，必须当心诱惑。有时你会掉进这样的陷阱中：对某个需要改动的继承层次结构来说，本来用一个更基本的重新设计可以更好，但你却为了追求速度而去使用 MI。例如，假设为可以活动的卡通角色设计一个类层次结构。至少从概念上来说，让各种角色能跳舞唱歌将很有意义，但每一种角色执行这些动作时方式都不一样。另外，跳舞唱歌的缺省行为是什么也不做。

所有这些用 C++来表示就象这样：

```
class CartoonCharacter {
public:
  virtual void dance() {}
  virtual void sing() {}
};
```

虚函数自然地体现了这样的约束：唱歌跳舞对所有 CartoonCharacter 对象都有意义。什么也不做的缺省行为通过类中那些函数的空定义来表示（参见条款 36）。假设有一个特殊类型的卡通角色是蚱蜢，它以自己特殊的方式跳舞唱歌：

```
class Grasshopper: public CartoonCharacter {
public:
  virtual void dance();    // 定义在别的什么地方
  virtual void sing();     // 定义在别的什么地方
};
```

现在假设，在实现了 Grasshopper 类后，你又想为蟋蟀增加一个类：

```
class Cricket: public CartoonCharacter {
public:
  virtual void dance();
  virtual void sing();
};
```

当坐下来实现 Cricket 类时，你意识到，为 Grasshopper 类所写的很多代码可以重复使用。但这需要费点神，因为要到各处去找出蚱蜢和蟋蟀唱歌跳舞的不同之处。你猛然间想出了一个代码复用的好办法：你准备用 Grasshopper 类来实现 Cricket 类，你还准备使用虚函数以使 Cricket 类可以定制 Grasshopper 的行为。

你立即认识到这两个要求 ---- "用...来实现" 的关系，以及重新定义虚函数的能力 ---- 意味着 Cricket 必须从 Grasshopper 私有继承，但蟋蟀当然还是一个卡通角色，所以你通过同时从 Grasshopper 和 CartoonCharacter 继承来重新定义 Cricket：

```
class Cricket: public CartoonCharacter,
               private Grasshopper {
public:
  virtual void dance();
  virtual void sing();
};
```

然后准备对 Grasshopper 类做必要的修改。特别是，需要声明一些新的虚函数让 Cricket 重新定义：

```
class Grasshopper: public CartoonCharacter {
public:
```

```
  virtual void dance();
  virtual void sing();

protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};
```

蚱蜢跳舞现在被定义成象这样：

```
void Grasshopper::dance()
{
  执行共同的跳舞动作;

  danceCustomization1();

  执行更多共同的跳舞动作;

  danceCustomization2();

  执行最后共同的跳舞动作;
}
```

蚱蜢唱歌的设计与此类似。

很明显，Cricket 类必须修改一下，因为它必须重新定义新的虚函数：

```
class Cricket:public CartoonCharacter,
      private Grasshopper {
public:
  virtual void dance() { Grasshopper::dance(); }
  virtual void sing() { Grasshopper::sing(); }

protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};
```

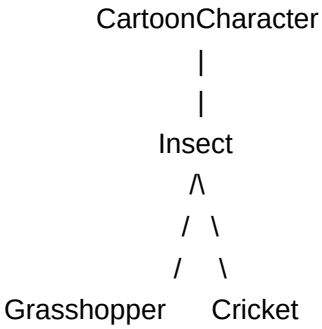
这看来很不错。当需要 Cricket 对象去跳舞时，它执行 Grasshopper 类中共同的 dance 代码，然后执行 Cricket 类中定制的 dance 代码，接着继续执行 Grasshopper::dance 中的代码，等等。

然而，这个设计中有个严重的缺陷，这就是，你不小心撞上了 "奥卡姆剃刀" ---- 任何一种奥卡姆剃刀都是有害的思想，William of Occam 的尤其如此。奥卡姆者鼓吹：如果没有必要，就不要增加实体。现在的情况下，实体就是指的继承关系。如果你相信多继承比单继承更复杂的话（我希望你相信），Cricket 类的设计就没必要复杂。（译注：1) William of Occam(1285-1349)，英国神学家，哲学家。2) 奥卡姆剃刀(Occam's razor)是一种思想，主要由 William of Occam 提出。之所以将它称为 "奥卡姆剃刀"，是因为 William of Occam 经常性地、很锐利地运用这一思想。）

问题的根本之处在于，Cricket 类和 Grasshopper 类之间并非 "用...来实现" 的关系。而是，Cricket 类和 Grasshopper 类之间享有共同的代码。特别是，它们享有决定唱歌跳舞行为的代码 ---- 蚱蜢和蟋蟀都有这种共同的行为。

说两个类具有共同点的方式不是让一个类从另一个类继承，而是让它们都从一个共同的基类继承，蚱蜢和蟋蟀之间的公共代码不属于 Grasshopper 类，也不属于 Cricket，而是属于它们共同的新的基类，如，Insect:

```
class CartoonCharacter { ... };

class Insect: public CartoonCharacter {
public:
  virtual void dance();    // 蚱蜢和蟋蟀
  virtual void sing();     // 的公共代码

protected:
  virtual void danceCustomization1() = 0;
  virtual void danceCustomization2() = 0;

  virtual void singCustomization() = 0;
};

class Grasshopper: public Insect {
protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};

class Cricket: public Insect {
protected:
  virtual void danceCustomization1();
  virtual void danceCustomization2();

  virtual void singCustomization();
};
```



可以看到，这个设计更清晰。只是涉及到单继承，此外，只是用到了公有继承。Grasshopper 和 Cricket 定义的只是定制功能；它们从 Insect 一点没变地继承了 dance 和 sing 函数。William of Occam 一定会很骄傲。

尽管这个设计比采用了 MI 的那个方案更清晰，但初看可能会觉得比使用 MI 的还要逊色。毕竟，和 MI 的方案相比，这个单继承结构中引入了一个全新的类，而使用 MI 就不需要。如果没必要，为什么要引入一个额外的类呢？

这就将你带到了多继承诱人的本性面前。表面看来，MI 好象使用起来更容易。它不需要增加新的类，虽然它要求在 Grasshopper 类中增加一些新的虚函数，但这些函数在任何情况下都是要增加的。

设想有个程序员正在维护一个大型 C++类库，现在需要在库中增加一个新的类，就象 Cricket 类要被增加到现有的的 CartoonCharacter/Grasshopper 层次结构中一样。程序员知道，有大量的用户使用现有的层次结构，所以，库的变化越大，对用户的影响越大。程序员决心将这种影响降低到最小。对各种选择再三考虑之后，程序员认识到，如果增加一个从 Grasshopper 到 Cricket 的私有继承连接，层次结构中将不需要任何其它变化。程序员不禁因为这个想法露出了微笑，暗自庆幸今后可以大量地增加功能，而代价仅仅只是增加很小一点复杂性。

现在设想这个负责维护的程序员是你。那么，请抵御这一诱惑！

条款 44: 说你想说的；理解你所说的

在本章关于 "继承和面向对象设计" 的简介中，我曾强调，理解不同的面向对象构件在 C++中的含义十分重要。这和仅仅知道 C++语言的规则有很大的不同。例如，C++规则说，如果类 D 从类 B 公有继承，从 D 的指针到 B 的指针就有一个标准转换；B 的公有成员函数将被继承为 D 的公有成员函数，等等。这些规则都是正确的，但在将设计思想转化为 C++的过程中，它们起不到任何作用。相反，你需要知道，公有继承意味着 "是一个"，如果 D 从 B 公有继承，类型 D 的每一个对象也 "是一个" 类型 B 的对象。因而，如果想在设计中表示 "是一个"，就自然会想到使用公有继承。

"说出你想说的" 只是成功的一半。事情的另一面是 "理解你所说的"，这一点同样重要。例如，将成员函数声明为非虚函数会给子类带来限制，如果没有认识到这一点就随便这样做将是不负责任的行为 ---- 除非你完全是有意这么做。声明一个非虚成员函数，你实际上是在说这个函数表示了一种特殊性上的不变性；如果不明白这一点，将会给程序带来灾难。

公有继承和 "是一个" 的等价性，以及非虚成员函数和 "特殊性上的不变性" 的等价性，是 C++构件如何和设计思想相对应的例子。下面的列表总结了这些对应关系中最重要的几个。

· 共同的基类意味着共同的特性。如果类 D1 和类 D2 都把类 B 声明为基类，D1 和 D2 将从 B 继承共同的数据成员和/或共同的成员函数。见条款 43。
· 公有继承意味着 "是一个"。如果类 D 公有继承于类 B，类型 D 的每一个对象也是一个类型 B 的对象，但反过来不成立。见条款 35。
· 私有继承意味着 "用...来实现"。如果类 D 私有继承于类 B，类型 D 的对象只不过是用类型 B 的对象来实现而已；类型 B 和类型 D 的对象之间不存在概念上的关系。见条款 42。
· 分层意味着 "有一个" 或 "用...来实现"。如果类 A 包含一个类型 B 的数据成员，类型 A 的对象要么具有一个类型为 B 的部件，要么在实现中使用了类型 B 的对象。见条款 40。

下面的对应关系只适用于公有继承的情况：

· 纯虚函数意味着仅仅继承函数的接口。如果类 C 声明了一个纯虚函数 mf，C 的子类必须继承 mf 的接口，C 的具体子类必须为之提供它们自己的实现。见条款 36。
· 简单虚函数意味着继承函数的接口加上一个缺省实现。如果类 C 声明了一个简单（非纯）虚函数 mf，C 的子类必须继承 mf 的接口；如果需要的话，还可以继承一个缺省实现。见条款 36。
· 非虚函数意味着继承函数的接口加上一个强制实现。如果类 C 声明了一个非虚函数 mf，C 的子类必须同时继承 mf 的接口和实现。实际上，mf 定义了 C 的 "特殊性上的不变性"。见条款 36。

第七章 杂项

进行高效的 C++程序设计有很多准则，其中有一些很难归类。本章就是专门为这些准则而安排的。不要因此而小看了它们的重要性。要想写出高效的软件，就必须知道：编译器在背后为你（给你?）做了些什么，怎样保证非局部的静态对象在被使用前已经被初始化，能从标准库得到些什么，从何处着手深入理解语言底层的设计思想。本书最后的这个章节，我将详细说明这些问题，甚至更多其它问题。

条款 45: 弄清 C++在幕后为你所写、所调用的函数

一个空类什么时候不是空类？ ---- 当 C++编译器通过它的时候。如果你没有声明下列函数，体贴的编译器会声明它自己的版本。这些函数是：一个拷贝构造函数，一个赋值运算符，一个析构函数，一对取址运算符。另外，如果你没有声明任何构造函数，它也将为你声明一个缺省构造函数。所有这些函数都是公有的。换句话说，如果你这么写：

```
class Empty{};
```

和你这么写是一样的：

```
class Empty {
public:
  Empty();                    // 缺省构造函数
  Empty(const Empty& rhs);    // 拷贝构造函数

  ~Empty();                   // 析构函数 ---- 是否
                              // 为虚函数看下文说明
  Empty&
  operator=(const Empty& rhs);  // 赋值运算符

  Empty* operator&();         // 取址运算符
  const Empty* operator&() const;
};
```

现在，如果需要，这些函数就会被生成，但你会很容易就需要它们。下面的代码将使得每个函数被生成：

```
const Empty e1;               // 缺省构造函数
                              // 析构函数

Empty e2(e1);                 // 拷贝构造函数

e2 = e1;                      // 赋值运算符

Empty *pe2 = &e2;             // 取址运算符
                              // (非 const)

const Empty *pe1 = &e1;       // 取址运算符
                              // (const)
```

假设编译器为你写了函数，这些函数又做些什么呢？是这样的，缺省构造函数和析构函数实际上什么也不做，它们只是让你能够创建和销毁类的对象（对编译器来说，将一些 "幕后" 行为的代码放在此处也很方便 ---- 参见条款 33 和 M24。）。注意，生成的析构函数一般是

非虚拟的（参见条款14），除非它所在的类是从一个声明了虚析构函数的基类继承而来。缺省取址运算符只是返回对象的地址。这些函数实际上就如同下面所定义的那样：

```
inline Empty::Empty() {}

inline Empty::~Empty() {}

inline Empty * Empty::operator&() { return this; }

inline const Empty * Empty::operator&() const
{ return this; }
```

至于拷贝构造函数和赋值运算符，官方的规则是：缺省拷贝构造函数（赋值运算符）对类的非静态数据成员进行 "以成员为单位的" 逐一拷贝构造（赋值）。即，如果 m 是类 C 中类型为 T 的非静态数据成员，并且 C 没有声明拷贝构造函数（赋值运算符），m 将会通过类型 T 的拷贝构造函数（赋值运算符）被拷贝构造（赋值）---- 如果 T 有拷贝构造函数（赋值运算符）的话。如果没有，规则递归应用到 m 的数据成员，直至找到一个拷贝构造函数（赋值运算符）或固定类型（例如，int，double，指针，等）为止。默认情况下，固定类型的对象拷贝构造（赋值）时是从源对象到目标对象的 "逐位" 拷贝。对于从别的类继承而来的类来说，这条规则适用于继承层次结构中的每一层，所以，用户自定义的构造函数和赋值运算符无论在哪一层被声明，都会被调用。

我希望这已经说得很清楚了。

但怕万一没说清楚，还是给个例子。看这样一个 NamedObject 模板的定义，它的实例是可以将名字和对象联系起来的类：

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

  ...

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

因为 NamedObject 类声明了至少一个构造函数，编译器将不会生成缺省构造函数；但因为没有声明拷贝构造函数和赋值运算符，编译器将生成这些函数（如果需要的话）。

看下面对拷贝构造函数的调用：

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

```
NamedObject<int> no2(no1);      // 调用拷贝构造函数
```

编译器生成的拷贝构造函数必须分别用 no1.nameValue 和 no1.objectValue 来初始化 no2.nameValue 和 no2.objectValue。nameValue 的类型是 string，string 有一个拷贝构造函数（你可以在标准库中查看 string 来证实 ---- 参见条款 49），所以 no2.nameValue 初始化时将调用 string 的拷贝构造函数，参数为 no1.nameValue。另一方面，NamedObject<int>::objectValue 的类型是 int（因为这个模板实例中，T 是 int），int 没有定义拷贝构造函数，所以 no2.objectValue 是通过从 no1.objectValue 拷贝每一个比特(bit)而被初始化的。

编译器为 NamedObject<int>生成的赋值运算符也以同样的方式工作，但通常，编译器生成的赋值运算符要想如上面所描述的那样工作，与此相关的所有代码必须合法且行为上要合理。如果这两个条件中有一个不成立，编译器将拒绝为你的类生成 operator=，你就会在编译时收到一些诊断信息。

例如，假设 NamedObject 象这样定义，nameValue 是一个 string 的引用，objectValue 是一个 const T：

```
template<class T>
class NamedObject {
public:
  // 这个构造函数不再有一个 const 名字参数，因为 nameValue
  // 现在是一个非 const string 的引用。char*构造函数
  // 也不见了，因为引用要指向的是 string
  NamedObject(string& name, const T& value);

  ...                       // 同上，假设没有
                            // 声明 operator=
private:
  string& nameValue;          // 现在是一个引用
  const T objectValue;        // 现在为 const
};
```

现在看看下面将会发生什么：

```
string newDog("Persephone");
string oldDog("Satch");

NamedObject<int> p(newDog, 2);      // 正在我写本书时，我们的
                                    // 爱犬 Persephone 即将过
                                    // 她的第二个生日

NamedObject<int> s(oldDog, 29);     // 家犬 Satch 如果还活着，
                                    // 会有 29 岁了（从我童年时算起）
```

```
p = s;                          // p 中的数据成员将会发生
                                // 些什么呢？
```

赋值之前，p.nameValue 指向某个 string 对象，s.nameValue 也指向一个 string，但并非同一个。赋值会给 p.nameValue 带来怎样的影响呢？赋值之后，p.nameValue 应该指向 "被 s.nameValue 所指向的 string" 吗，即，引用本身应该被修改吗？如果是这样，那太阳从西边出来了，因为 C++没有办法让一个引用指向另一个不同的对象（参见条款 M1）。或者，p.nameValue 所指的 string 对象应该被修改吗？ 这样的话，含有 "指向那个 string 的指针或引用" 的其它对象也会受影响，也就是说，和赋值没有直接关系的其它对象也会受影响。这是编译器生成的赋值运算符应该做的吗？

面对这样的难题，C++拒绝编译这段代码。如果想让一个包含引用成员的类支持赋值，你就得自己定义赋值运算符。对于包含 const 成员的类（例如上面被修改的类中的 objectValue）来说，编译器的处理也相似；因为修改 const 成员是不合法的，所以编译器在隐式生成赋值函数时也会不知道怎么办。还有，如果派生类的基类将标准赋值运算符声明为 private， 编译器也将拒绝为这个派生类生成赋值运算符。因为，编译器为派生类生成的赋值运算符也应该处理基类部分（见条款 16 和 M33），但这样做的话，就得调用对派生类来说无权访问的基类成员函数，这当然是不可能的。

以上关于编译器生成函数的讨论引发了这样的问题：如果想禁止使用这些函数，那该怎么办呢？也就是说，假如你永远不想让类的对象进行赋值，所以有意不声明 operator=，那该怎么做呢？这个小难题的解决方案正是条款 27 讨论的主题。指针成员和编译器生成的拷贝构造函数及赋值运算符之间的相互影响经常被人忽视，关于这个话题的讨论请查看条款 11。

条款 46: 宁可编译和链接时出错，也不要运行时出错

除了极少数情况下会使 C++抛出异常（例如，内存耗尽 ---- 见条款 7）外，运行时错误的概念和 C++没什么关系，就象在 C 中一样。没有下溢，上溢，除零检查；没有数组越界检查，等等。一旦程序通过了编译和链接，你就得靠自己了 ---- 一切后果自负。这很象跳伞运动，一些人从中找到了刺激，另一些人则吓得摔成了残废。这一思想背后的动机当然在于效率：没有运行时检查，程序会更小更快。

处理这类事情有另一个不同的方法。一些语言如 Smalltalk 和 LISP 通常在编译链接期间只是检查极少一些错误，但却提供了强大的运行时系统来处理执行期间的错误。不象 C++，这些语言几乎都是解释型的，在提供额外灵活性的同时，它们也带来了性能上的损失。

不要忘了你是在用 C++编程。即使发现 Smalltalk/LISP 的方法很吸引人，也要忘掉它们。常说要坚持党的路线，现在的情况下，它的含义就是要避免运行时错误。只要有可能，就要让出错检查从运行时退回到链接时，或者，最理想的是，编译时。

这种方法带来的好处不仅仅在于程序的大小和速度，还有可靠性。如果程序通过了编译和链接而没有产生错误信息，你就可以确信程序中没有编译器和链接器能检查得到的任何错误，仅此而已。（当然，另一个可能性是，编译器或链接器有问题，但不要拿这种可能性来困扰我们。）

对于运行时错误来说，情况大不一样。在某次运行期间程序没有产生任何运行时错误，你就能确信另一次不同的运行期内不会产生错误吗？比如：在另一次运行中，你以不同的顺序做事，或者采用不同的数据，或者运行更长或更短时间，等等。你可以不停地测试自己的程序直到面色发紫，但你还是不能覆盖所有的可能性。因而，运行时发现错误比在编译链接期间检查错误更不能让人放心。

通常，对设计做一点小小的改动，就可以在编译期间消除可能产生的运行时错误。这常常涉及到在程序中增加新的数据类型（参见条款 M33）。例如，假设想写一个类来表示时间中的日期，最初的做法可能象这样：

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

  ...

};
```

准备实现这个构造函数，面临的一个问题是对 day 和 month 值的合法性检查。让我们来看看，对于传给 month 的值来说，怎么做可以免于对它进行合法性检查呢？

一个明显的办法是采用枚举类型而不用整数：

```
enum Month { Jan = 1, Feb = 2, ... , Nov = 11, Dec = 12 };
```

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

  ...

};
```

遗憾的是，这不会换来多少好处，因为枚举类型不需要初始化：

```
Month m;
Date d(22, m, 1857);      // m 是不确定的
```

所以，Date 构造函数还是得验证 month 参数的值。

既想免除运行时检查，又要保证足够的安全性，你就得用一个类来表示 month，你就得保证只有合法的 month 才被创建：

```
class Month {
public:
  static const Month Jan() { return 1; }
  static const Month Feb() { return 2; }
  ...
  static const Month Dec() { return 12; }

  int asInt() const           // 为了方便，使 Month
  { return monthNumber; }     // 可以被转换为 int

private:
  Month(int number): monthNumber(number) {}

  const int monthNumber;
};

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

这个设计在几个方面的特点综合确定了它的工作方式。首先，Month 构造函数是私有的。这防止了用户去创建新的 month。可供使用的只能是 Month 的静态成员函数返回的对象，再加上它们的拷贝。第二，每个 Month 对象为 const，所以它们不能被改变（否则，很多地方会忍不住将一月转换成六月，特别是在北半球）。最后一点，得到 Month 对象的唯一办法是调用函数或拷贝现有的 Month（通过隐式 Month 拷贝构造函数 ---- 见条款 45）。这样，

就可以在任何时间任何地方使用 Month 对象；不必担心无意中使用了没有被初始化的对象。（否则就可能有问题。条款 47 进行了说明）

有了这些类，用户几乎不可能指定一个非法的 month，甚至完全不可能 ---- 如果不出现下面这种可恶的情况的话：

```
Month *pm;                  // 定义未被初始化的指针

Date d(1, *pm, 1997);       // 使用未被初始化的指针!
```

但这种情况所涉及的是另一个问题，即通过未被初始化的指针取值，其结果是不可确定的。（参见条款 3，看看我对 "不确定行为" 的感受）遗憾的是，我没有办法来防止或检查这种异端行为。但是，如果假设这种情况永远不会发生，或者如果我们不考虑这种情况下软件的行为，Date 构造函数对它的 Month 参数就可以免于合法性检查。另一方面，构造函数还是必须检查 day 参数的合法性 ---- 九月，四月，六月和十一月各有多少天呢？

Date 的例子将运行时检查用编译时检查来取代。你可能想知道什么时候可以使用链接时检查。实际上，不是经常这么做。C++用链接器来保证所需要的函数只被定义一次（参见条款 45，"需要" 一个函数会带来什么）。它还使用链接器来保证静态对象（参见条款 47）只被定义一次。你可以用同样的方法使用链接器。例如，条款 27 说明，对于一个显式声明的函数，如果想有意禁止对它进行定义，链接器检查就很有用。

但不要过于强求。想消除所有的运行检查是不切实际的。例如，任何允许交互式输入的程序都要进行输入验证。同样地，某个类中如果包含需要执行上下限检查的数组，每次访问数组时就要对数组下标进行检查。尽管如此，将检查从运行时转移到编译或链接时一直是值得努力的目标，只要实际可行，就要追求这一目标。这样做的奖赏是，程序会更小，更快，更可靠。

条款 47: 确保非局部静态对象在使用前被初始化

大家都是成年人了，所以用不着我来告诉你们：使用未被初始化的对象无异于蛮干。事实上，关于这个问题的整个想法会让你觉得可笑；构造函数可以确保对象在创建时被初始化，难道不是这样吗？

唔，是，也不是。在某个特定的被编译单元（即，源文件）中，可能一切都不成问题；但如果在某个被编译单元中，一个对象的初始化要依赖于另一个被编译单元中的另一个对象的值，并且这第二个对象本身也需要初始化，事情就会变得更复杂。

例如，假设你已经写了这样一个程序库，它提供一个文件系统的抽象，其中可能包括一个功能，使得互联网上的文件看起来就象在本地一样。既然程序库使得整个世界看起来象一个单独的文件系统，你就可以在程序库的名字空间（见条款 28）中创建一个专门的对象，theFileSystem，这样，用户任何时候需要和程序库所提供的文件系统交互，都可以使用它：

```
class FileSystem { ... };          // 在个类在你
                                   // 的程序库中

FileSystem theFileSystem;          // 程序库用户
                                   // 和这个对象交互
```

因为 theFileSystem 表示的是很复杂的东西，所以它的构造重要而且必需；在 theFileSystem 还没构造之前就使用它会造成不可确定的行为。（然而，参考条款 M17，象 theFileSystem 这样的对象，其初始化可以被有效、安全地延迟。）

现在假设某个程序库的用户创建了一个类，表示文件系统中的目录。很自然地，这个类使用了 theFileSystem：

```
class Directory {                  // 由程序库的用户创建
public:
  Directory();
  ...
};

Directory::Directory()
{
  通过调用 theFileSystem 的成员函数
  创建一个 Directory 对象;
}
```

进一步假设用户想为临时文件专门创建一个全局 Directory 对象：

```
Directory tempDir;                 // 临时文件目录
```

现在，初始化顺序的问题变得很明显了：除非 theFileSystem 在 tempDir 之前被初始化，否则，tempDir 的构造函数将会去使用还没被初始化的 theFileSystem。但 theFileSystem 和 tempDir 是由不同的人在不同的时间、不同的文件中创建的。怎么可以确认 theFileSystem 在 tempDir 之前被创建呢？

任何时候，如果在不同的被编译单元中定义了 "非局部静态对象" ，并且这些对象的正确行为依赖于它们被初始化的某一特定顺序，这类问题就会产生。非局部静态对象指的是这样的对象：

· 定义在全局或名字空间范围内（例如：theFileSystem 和 tempDir），
· 在一个类中被声明为 static，或，
· 在一个文件范围被定义为 static。

很抱歉，"非局部静态对象" 这个术语没有简称，所以你要让自己习惯这种有点咬口的句子。

对于不同被编译单元中的非局部静态对象，你一定不希望自己的程序行为依赖于它们的初始化顺序，因为你无法控制这种顺序。让我再重复一遍：你绝对无法控制不同被编译单元中非局部静态对象的初始化顺序。

很自然地想知道，为什么无法控制？

这是因为，确定非局部静态对象初始化的 " 正确" 顺序很困难，非常困难，极其困难。即使在它最普通的形式下 ---- 多个被编译单元，多个通过隐式模板实例化所生成的非局部静态对象（隐式模板实例化时，它们本身可能都会产生这样的问题） ---- 不仅不可能确定正确的初始化顺序，往往连找一个可以确定正确顺序的特殊情况都不值得。

在 "混沌理论" 领域，有一个原理称为 "蝴蝶效应" 。这条原理声称，世界某个角落的一只蝴蝶拍动翅膀，会对大气产生微小的影响，从而导致某个遥远的地方天气模式的深刻变化。稍微准确一点来说也就是：对于某种系统，输入的微小干扰会导致输出彻底的变化。

软件系统的开发也表现了自身的 "蝴蝶效应"。一些系统对需求的细节高度敏感，需求发生细小的变化，实现系统的难易程度就会发生巨大的变化。例如，条款 29 说明，将一个隐式转换的要求从 "String 到 char*" 改为 "String 到 const char*"，就可以将一个运行慢、容易出错的函数用一个运行快并且安全的函数来代替。

确保非局部静态对象在使用前被初始化的问题也和上面一样，它对你的实现细节十分敏感。但是，如果你不强求一定要访问 "非局部静态对象"，而愿意访问具有和非局部静态对象 " 相似行为" 的对象（不存在初始化问题），难题就消失了。取而代之的是一个很容易解决的问题，甚至称不上是一个问题。

这种技术 ---- 有时称为 "单一模式"（译注：即 Singleton pattern，参见 "Design Patterns" 一书）---- 本身很简单。首先，把每个非局部静态对象转移到函数中，声明它为 static。其次，让函数返回这个对象的引用。这样，用户将通过函数调用来指明对象。换句话说，用函数内部的 static 对象取代了非局部静态对象。（参见条款 M26）

这个方法基于这样的事实：虽然关于"非局部"静态对象什么时候被初始化，C++几乎没有做过说明；但对于函数中的静态对象（即，"局部"静态对象）什么时候被初始化，C++却明确指出：它们在函数调用过程中初次碰到对象的定义时被初始化。所以，如果你不对非局部静态对象直接访问，而用返回局部静态对象引用的函数调用来代替，就能保证从函数得到的引用指向的是被初始化了的对象。这样做的另一个好处是，如果这个模拟非局部静态对象的函数从没有被调用，也就永远不会带来对象构造和销毁的开销；而对于非局部静态对象来说就没有这样的好事。

下面的代码对 theFileSystem 和 tempDir 都采用了这一技术：

```
class FileSystem { ... };          // 同前
FileSystem& theFileSystem()        // 这个函数代替了
{                                  // theFileSystem 对象

  static FileSystem tfs;           // 定义和初始化
                                   // 局部静态对象
                                   // (tfs = "the file system")

  return tfs;                      // 返回它的引用
}

class Directory { ... };           // 同前

Directory::Directory()
{
  同前，除了 theFileSystem 被
  theFileSystem()代替;
}

Directory& tempDir()               // 这个函数代替了
{                                  // tempDir 对象

  static Directory td;             // 定义和初始化
                                   // 局部静态对象

  return td;                       // 返回它的引用
}
```

系统被修改后，用户还是完全和以前一样编程，只是现在他们用的是 theFileSystem()和 tempDir()，而不是 theFileSystem 和 tempDir。即，他们所用的是返回对象引用的函数，而不是对象本身。

这种返回引用的函数虽然采用了上面所讨论的技术，但函数本身总是很简单：第一行定义并初始化一个局部静态对象，第二行返回它，仅此而已。因为太简单，你可能很想把它声明为 inline。条款 33 指出，对于 C++语言规范的最新修订版本来说，这是一个非常有效的实现

策略；但它同时指出，在使用之前，一定要确认你的编译器和标准中的相关要求要一致。如果编译器不符合最新标准，你又象上面那样使用内联，就可能造成函数以及函数内部静态对象有多份拷贝。这足以让一个成年的程序员哭泣。

至此已没有什么神秘之处了。为了使这一技术有效，一定要给对象一个合理的初始化顺序。如果你让对象 A 必须在对象 B 之前初始化，同时又让 A 的初始化依赖于 B 已经被初始化，你就会惹上麻烦，坦白说，是罪有应得。如果能避开这种不合理的情况，本条款所介绍的方案将会很好地为你提供帮助。

条款 48: 重视编译器警告

很多程序员日常总是不理睬编译器警告。毕竟，如果问题很严重，就会是个错误，不是吗？这种想法在其它语言中相对来说没什么害处，但在 C++中，可以肯定的一点是，编译器的设计者肯定比你更清楚到底发生了什么。例如，大家可能都犯过这个错误：

```
class B {
public:
  virtual void f() const;
};

class D: public B {
public:
  virtual void f();
};
```

本来是想用 D::f 重新定义虚函数 B::f，但有个错误：在 B 中，f 是一个 const 成员函数，但在 D 中没有被声明为 const。据我所知，有个编译器会这么说：

```
warning: D::f() hides virtual B::f()
```

对于这条警告，很多缺乏经验的程序员会这样自言自语，"D::f 当然会隐藏 B::f ---- 本来就应该是这样！" 错了。编译器想告诉你的是：声明在 B 中的 f 没有在 D 中重新声明，它被完全隐藏了（参见条款 50：为什么这样）。忽视这条编译器警告几乎肯定会导致错误的程序行为。你会不停地调试去找原因，而这个错误实际上早就被编译器发现了。

当然，在对某个编译器的警告信息积累了经验之后，你会真正理解不同的信息所表示的含义（唉，往往和它们表面看上去的意思不同）。一旦有了这些经验，你会对很多警告不予理睬。这没问题，但重要的是，在忽略一个警告之前，你一定要准确理解它想告诉你的含义。

只要谈到警告，就要想到警告是和编译器紧密相关的，所以在编程时不要马马虎虎，寄希望于编译器为你找出每一条错误。例如上面隐藏了函数的那段代码，当它通过不同的（但使用很广泛的）编译器时可能不会产生警告。编译器是用来将 C++转换成可执行格式的，并不是你的私人保镖。你想得到那样的安全？去用 Ada 吧。

条款 49: 熟悉标准库

C++标准库很大。非常大。难以置信的大。怎么个大法？这么说吧：在 C++标准中，关于标准库的规格说明占了密密麻麻 300 多页，这还不包括标准 C 库，后者只是 "作为参考"（老实说，原文就是用的这个词）包含在 C++库中。

当然，并非总是越大越好，但在现在的情况下，确实越大越好，因为大的库会包含大量的功能。标准库中的功能越多，开发自己的应用程序时能借助的功能就越多。C++库并非提供了一切（很明显的是，没有提供并发和图形用户接口的支持），但确实提供了很多。几乎任何事你都可以求助于它。

在归纳标准库中有些什么之前，需要介绍一下它是如何组织的。因为标准库中东西如此之多，你（或象你一样的其他什么人）所选择的类名或函数名就很有可能和标准库中的某个名字相同。为了避免这种情况所造成的名字冲突，实际上标准库中的一切都被放在名字空间 std 中（参见条款 28）。但这带来了一个新问题。无数现有的 C++代码都依赖于使用了多年的伪标准库中的功能，例如，声明在<iostream.h>，<complex.h>，<limits.h>等头文件中的功能。现有软件没有针对使用名字空间而进行设计，如果用 std 来包装标准库导致现有代码不能用，将是一种可耻行为。（这种釜底抽薪的做法会让现有代码的程序员说出比 "可耻" 更难听的话）

慑于被激怒的程序员会产生的破坏力，标准委员会决定为包装了 std 的那部分标准库构件创建新的头文件名。生成新头文件的方法仅仅是将现有 C++头文件名中的 .h 去掉，方法本身不重要，正如最后产生的结果不一致也并不重要一样。所以<iostream.h>变成了<iostream>，<complex.h>变成了<complex>，等等。对于 C 头文件，采用同样的方法，但在每个名字前还要添加一个 c。所以 C 的<string.h>变成了<cstring>，<stdio.h>变成了<cstdio>，等等。最后一点是，旧的 C++头文件是官方所反对使用的（即，明确列出不再支持），但旧的 C 头文件则没有（以保持对 C 的兼容性）。实际上，编译器制造商不会停止对客户现有软件提供支持，所以可以预计，旧的 C++头文件在未来几年内还是会被支持。

所以，实际来说，下面是 C++头文件的现状：

· 旧的 C++头文件名如<iostream.h>将会继续被支持，尽管它们不在官方标准中。这些头文件的内容不在名字空间 std 中。

· 新的 C++头文件如<iostream>包含的基本功能和对应的旧头文件相同，但头文件的内容在名字空间 std 中。（在标准化的过程中，库中有些部分的细节被修改了，所以旧头文件和新头文件中的实体不一定完全对应。）

· 标准 C 头文件如<stdio.h>继续被支持。头文件的内容不在 std 中。

· 具有 C 库功能的新 C++头文件具有如<cstdio>这样的名字。它们提供的内容和相应的旧 C 头文件相同，只是内容在 std 中。

所有这些初看有点怪，但不难习惯它。最大的挑战是把字符串头文件理清楚：<string.h>是旧的 C 头文件，对应的是基于 char*的字符串处理函数；<string>是包装了 std 的 C++头文件，对应的是新的 string 类（看下文）；<cstring>是对应于旧 C 头文件的 std 版本。如果能掌握这些（我相信你能），其余的也就容易了。

关于标准库，需要知道的第二点是，库中的一切几乎都是模板。看看你的老朋友 iostream。（如果你和 iostream 不是朋友，转到条款 2，看看你为什么要和它发展关系）iostream 帮助你操作字符流，但什么是字符？是 char 吗？是 wchar_t？是 Unicode 字符？一些其它的多字节字符？没有明显正确的答案，所以标准库让你去选。所有的流类（stream class）实际上是类模板，在实例化流类的时候指定字符类型。例如，标准库将 cout 类型定义为 ostream，但 ostream 实际上是一个 basic_ostream<char>类型定义（typedef ）。

类似的考虑适用于标准库中其它大部分类。string 不是类，它是类模板：类型参数限定了每个 string 类中的字符类型。complex 不是类，它是类模板：类型参数限定了每个 complex 类中实数部分和虚数部分的类型。vector 不是类，它是类模板。如此不停地进行下去。

在标准库中你无法避开模板，但如果只是习惯于和 char 类型的流和字符串打交道，通常可以忽略它们。这是因为，对这些组件的 char 实例，标准库都为它们定义了 typedef，这样你就可以在编程时继续使用 cin，cout，cerr 等对象，以及 istream，ostream，string 等类型，不必担心 cin 的真实类型是 basic_istream<char>以及 string 的真实类型是 basic_string<char>。

标准库中很多组件的模板化和上面所建议的大不相同。再看看那个概念上似乎很直观的 string。当然，可以基于 "它所包含的字符类型" 确定它的参数，但不同的字符集在细节上有不同，例如，特殊的文件结束字符，拷贝它们的数组的最有效方式，等等。这些特征在标准中被称为 traits，它们在 string 实例中通过另外一个模板参数指定。此外，string 对象要执行动态内存分配和释放，但完成这一任务有很多不同的方法（参见条款 10）。哪一个最好？你得选择：string 模板有一个 Allocator 参数，Allocator 类型的对象被用来分配和释放 string 对象所使用的内存。

这里有一个 basic_string 模板的完整声明，以及建立在它之上的 string 类型定义（typedef）；你可以在<string>头文件中找到它（或与之相当的什么东西）：

```
namespace std {

  template<class charT,
           class traits = char_traits<charT>,
           class Allocator = allocator<charT> >
    class basic_string;

  typedef basic_string<char> string;

}
```

注意，basic_string 的 traits 和 Allocator 参数有缺省值。这在标准库中是很典型的做法。它为使用者提供了灵活性， 但对于这种灵活性所带来的复杂性，那些只想做 "正常" 操作的" 典型" 用户却又可以避开。换句话说，如果只想使用象 C 字符串那样的字符串对象，就可以使用 string 对象，而不用在意实际上是在用 basic_string<char, char_traits<char>, allocator<char> >类型的对象。

是的，通常可以这么做，但有时还是得稍稍看看底层。例如，条款 34 指出，声明一个类而不提供定义具有优点；它还指出，下面是一种声明 string 类型的错误方法：

```
class string;                    // 会通过编译，但
                                 // 你不会这么做
```

先不要考虑名字空间，这里真正的问题在于：string 不是一个类，而是一个 typedef。如果可以通过下面的方法解决问题就太好了：

```
typedef basic_string<char> string;
```

但这又不能通过编译。"你所说的 basic_string 是什么东西？" 编译器会奇怪 ---- 当然，它可能会用不同的语句来问你。所以，为了声明 string，首先得声明它所依赖的所有模板。如果可以这么做的话，就会象下面这样：

```
template<class charT> struct char_traits;

template<class T> class allocator;

  template<class charT,
           class traits = char_traits<charT>,
           class Allocator = allocator<charT> >
    class basic_string;

typedef basic_string<char> string;
```

然而，你不能声明 string。至少不应该。这是因为，标准库的实现者声明的 stirng（或 std 名字空间中任何其它东西）可以和标准中所指定的有所不同，只要最终提供的行为符合标准就行。例如，basic_string 的实现可以增加第四个模板参数，但这个参数的缺省值所产生的代码的行为要和标准中所说的原始的 basic_string 一致。

那到底该怎么办？不要手工声明 string（或标准库中其它任何部分）。相反，只用包含一个适当的头文件，如<string>。

有了头文件和模板的这些知识，现在可以看看标准 C++库中有哪些主要组件：

· 标准 C 库。它还在，你还可以用它。虽然有些地方有点小的修修补补，但无论怎么说，还是那个用了多年的 C 库。

· Iostream。和 "传统" Iostream 的实现相比，它已经被模板化了，继承层次结构也做了修改，增强了抛出异常的能力，可以支持 string（通过 stringstream 类）和国际化（通过 locales ---- 见下文）。当然，你期望 Iostream 库所具有的东西几乎全都继续存在。也就是说，它还是支持流缓冲区，格式化标识符，操作子和文件，还有 cin，cout，cerr 和 clog 对象。这意味着可以把 string 和文件当做流，还可以对流的行为进行更广泛的控制，包括缓冲和格式化。

· String。string 对象在大多数应用中被用来消除对 char*指针的使用。它们支持你所期望的那些操作（例如，字符串连接，通过 operator[]对单个字符进行常量时间级的访问，等等），它们可以转换成 char*，以保持和现有代码的兼容性，它们还自动处理内存管理。一些 string 的实现采用了引用计数（参见条款 M29），这会带来比基于 char*的字符串更佳的性能（时间和空间上）。

· 容器。不要再写你自己的基本容器类！标准库提供了下列高效的实现：vector（就象动态可扩充的数组），list（双链表），queue， stack，deque，map，set 和 bitset。唉，竟然没有 hash table（虽然很多制造商作为扩充提供），但多少可以作为补偿的一点是， string 是容器。这很重要，因为它意味着对容器所做的任何操作（见下文）对 string 也适用。

什么？你不明白我为什么说标准库的实现很高效？很简单：标准库规定了每个类的接口，而且每条接口规范中的一部分是一套性能保证。所以，举例来说，无论 vector 是如何实现的，仅仅提供对它的元素的访问是不够的，还必须提供 "常量时间" 内的访问。如果不这样，就不是一个有效的 vector 实现。

很多 C++程序中，动态分配字符串和数组导致大量使用 new 和 delete，new/delete 错误 ---- 尤其是没有 delete 掉 new 出来的内存而导致的泄漏 ---- 时常发生。如果使用 string 和 vector 对象（二者都执行自身的内存管理）而不使用 char*和动态分配的数组的指针，很多 new 和 delete 就可以免于使用，使用它们所带来的问题也会随之消失（例如，条款 6 和 11）。

· 算法。标准容器当然好，如果存在易于使用它们的方法就更好。标准库就提供了大量简易的方法（即，预定义函数，官方称为算法(algorithm) ---- 实际上是函数模板），其中的大多数适用于库中所有的容器 ---- 以及内建数组（built-in arrays）！

算法将容器的内容当作序列（sequence），每个算法可以应用于一个容器中所有值所对应的序列，或者一个子序列（subsequence）。标准算法有 for_each（为序列中的每个元素调用某个函数），find（在序列中查找包含某个值的第一个位置 ---- 条款 M35 展示了它的实现），count_if（计算序列中使得某个判定为真的所有元素的数量），equal（确定两个序列包含的元素的值是否完全相同），search（在一个序列中找出某个子序列的起始位置），copy（拷贝一个序列到另一个），unique（在序列中删除重复值），rotate（旋转序列中的值），sort（对序列中的值排序）。注意这里只是抽取了所有算法中的几个；标准库中还包括其它很多算法。

和容器操作一样，算法也有性能保证。例如，stable_sort 算法执行时要求不超过 0 比较级(N log N) 。（如果不理解上面句子中符号 "0" 的意思，不要紧张。概括的说，它的意思实际上是，stable_sort 提供的性能必须和最高效的通用排序算法在同一个级别。）

· 对国际化的支持。不同的文化以不同的方式行事。和 C 库一样，C++库提供了很多特性有助于开发出国际化的软件。但虽然从概念上来说和 C 类似，其实 C++的方法还是有所不同。例如，C++为支持国际化广泛使用了模板，还利用了继承和虚函数，这些一定不会让你感到奇怪。

支持国际化最主要的构件是 facets 和 locales。facets 描述的是对一种文化要处理哪些特性，包括排序规则（即，某地区字符集中的字符应该如何排序），日期和时间应该如何表示，数字和货币值应该如何表示，怎样将信息标识符映射成（自然的）明确的语言信息，等等。locales 将多组 facets 捆绑在一起。例如，一个关于美国的 locale 将包括很多 facets，描述如何对美国英语字符串排序，如何以适合美国人的方式读写日期和时间，读写货币和数字值，等等。而对于一个关于法国的 locales 来说，它描述的是怎么以法国人所习惯的方式完成这些任务。C++允许单个程序中同时存在多个 locales，所以一个应用中的不同部分可能采用的是不同的规范。

· 对数字处理的支持。FORTRAN 的末日可能就快到了。C++库为复数类（实数和虚数部分的精度可以是 float，double 或 long double）和专门针对数值编程而设计的特殊数组提供了模板。例如，valarray 类型的对象可用来保存可以任意混叠(aliasing)的元素。这使得编译器可以更充分地进行优化，尤其是对矢量计算机来说。标准库还对两种不同类型的数组片提供了支持，并提供了算法计算内积(inner product)，部分和(partial sum)，临差(adjacent difference)等。

· 诊断支持。标准库支持三种报错方式：C 的断言（参见条款 7），错误号，例外。为了有助于为例外类型提供某种结构，标准库定义了下面的例外类（exception class）层次结构：

```
                                  |---domain_error
                |----- logic_error<---- |---invalid_argument
                |                       |---length_error
                |                       |---out_of_range
exception<--|
                |                        |--- range_error
                |-----runtime_error<--|---underflow_error
                                        |---overflow_error
```

logic_error（或它的子类）类型的例外表示的是软件中的逻辑错误。理论上来说，这样的错误可以通过更仔细的程序设计来防止。runtime_error（或它的子类）类型的例外表示的是只有在运行时才能发现的错误。

可以就这样使用它们，可以通过继承它们来创建自己的例外类，或者可以不去管它。没有人强迫你使用它。

上面列出的内容并没有涵盖标准库中的一切。记住，规范有 300 多页。但它还是为你初步展现了标准库的基本概貌。

标准库中容器和算法这部分一般称为标准模板库（STL---- 参见条款 M35）。STL 中实际上还有第三个构件 ---- 迭代子（Iterator） ---- 前面没有介绍过。迭代子是指针似的对象，它

让 STL 算法和容器共同工作。不过现在不需要弄清楚迭代子，因为我这里所介绍的是标准库的高层描述。如果你对它感兴趣，可以在条款 39 和 M35 中找到使用它的例子。

STL 是标准库中最具创新的部分，这并不是因为它提供了容器和算法（虽然它们非常有用），而是因为它的体系结构。简单来说，它的体系结构具有扩展性：你可以对 STL 进行添加。当然，标准库中的组件本身是固定的，但如果遵循 STL 构建的规范，你可以写出自己的容器，算法和迭代子，使它们可以和标准 STL 组件一起工作，就象标准组件自身之间相互工作一样。你还可以利用别人所写的符合 STL 规范的容器，算法和迭代子，就象别人利用你的一样。使得 STL 具有创新意义的原因在于它实际上不是软件，而是一套规范（convention）。标准库中的 STL 组件只是具体体现了遵循这种规范所能带来的好处。

通过使用标准库中的组件，通常可以让你避免从头到尾来设计自己的 IO 流，string，容器，国际化，数值数据结构以及诊断等机制。这就给了你更多的时间和精力去关注软件开发中真正重要的部分：实现那些有别于你的竞争对手的软件功能。

条款 50: 提高对 C++的认识

C++中有很多 "东西"：C，重载，面向对象，模板，例外，名字空间。这么多东西，有时让人感到不知所措。怎么弄懂所有这些东西呢？

C++之所以发展到现在这个样子，在于它有自己的设计目标。理解了这些设计目标，就不难弄懂所有这些东西了。C++最首要的目标在于：

· 和 C 的兼容性。很多很多 C 还存在，很多很多 C 程序员还存在。C++利用了这一基础，并建立在 ---- 我是指 "平衡在" ---- 这一基础之上。

· 效率。作为 C++的设计者和第一个实现者，Bjarne Stroustrup 从一开始就清楚地知道，要想把 C 程序员争取过来，就要避免转换语言会带来性能上的损失，否则他们不会对 C++再看第二眼。结果，他确信 C++在效率上可以和 C 匹敌 ---- 二者相差大约在 5%之内。

· 和传统开发工具及环境的兼容性。各色不同的开发环境到处都是，编译器、链接器和编辑器则无处不在。从小型到大型的所有开发环境，C++都要轻松应对，所以带的包袱越轻越好。想移植 C++？你实际上移植的只是一种语言，并利用了目标平台上现有的工具。（然而，往往也可能带来更好的实现，例如，如果链接器能被修改，使得它可以处理内联和模板在某些方面更高的要求）

· 解决真实问题的可应用性。C++没有被设计为一种完美的，纯粹的语言，不适于用它来教学生如何编程。它是设计为专业程序员的强大工具，用它来解决各种领域中的真实问题。真实世界都有些磕磕碰碰，因此，程序员们所依赖的工具如果偶尔出点问题，也不值得大惊小怪。

以上目标阐明了 C++语言中大量的实现细节，如果没有它们作指导，就会有摩擦和困惑。为什么隐式生成的拷贝构造函数和赋值运算符要象现在这样工作呢，尤其是指针（参见条款 11 和 45）？因为这是 C 对 struct 进行拷贝和赋值的方式，和 C 兼容很重要。为什么析构函数不自动被声明为 virtual（参见条款 14），为什么实现细节必须出现在类的定义中（参见条款 34）呢？因为不这样做就会带来性能上的损失，效率很重要。为什么 C++不能检测非局部静态对象之间的初始化依赖关系（参见条款 47）呢？因为 C++支持单独编译（即，分开编译源模块，然后将多个目标文件链接起来，形成可执行程序），依赖现有的链接器，不和程序数据库打交道。所以，C++编译器几乎不可能知道整个程序的一切情况。最后一点，为什么 C++不让程序员从一些繁杂事务如内存管理（参见条款 5-10）和低级指针操作中解脱出来呢？因为一些程序员需要这些处理能力，一个真正的程序员的需要至关重要。

关于 C++身后的设计目标如何影响语言行为的形成，以上介绍远远不够。要想覆盖所有的内容，将需要一整本书；方便的是，Stroustrup 写了一本。这本书是 "The Design and Evolution of C++" (Addison-Wesley, 1994)，有时简称为 "D&E"。读了它，你会了解到有哪些特性被增加到 C++中，以什么顺序，以及为什么。你还会知道哪些特性被放弃了，以及为什么。你甚至可以了解到一些幕后故事，如 dynamic_cast（参见条款 39 和 M2）如何被考虑，被放弃，又被考虑，最后被接受 ---- 以及为什么。如果你理解 C++有困难，D&E 将为你驱散心头的疑云。

对于 C++如何成为现在的样子，"The Design and Evolution of C++" 提供了丰富的资料和见解，但它绝对不是正式的语言规格说明。对此你得求助于 C++国际标准，一本令人印象深刻的长达 700 多页的正式文本。在那儿你可以读到象下面这样刻板的句子：

一个虚函数调用所使用的缺省参数是表示对象的指针或引用的静态类型所决定的虚函数所声明的缺省参数。派生类中的重载函数不获取它重载的函数中的缺省值。

这段话是条款 38（"决不要重新定义继承而来的缺省参数值"）的基础，但我期望我对这个论题的论述比上面的原文多少更让人容易理解一些。

C++标准不是临睡前的休闲读物，而是你最好的依靠 ---- 你的 "标准" 依靠 ---- 如果你和其他人（比如，编译器供货商，或采用其它工具编程的开发人员）对什么东西是或不是 C++有分歧的话。标准的全部目的在于，为解决这类争议提供权威信息。

C++标准的官方名称很咬口，但如果你需要知道，就得知道。这就是：International Standard for Information Systems----Programming Language C++。它由International Organization for Standardization (ISO)第 21 工作组颁布。（如果你爱钻牛角尖，它实际上是由ISO/IEC JTC1/SC22/WG21 颁布的----我没有添油加醋）你可以从你的国家标准机构（在美国，是ANSI，即American National Standards Institute）定购正式C++标准的副本，但C++标准的最新草稿副本 ---- 和最终文件十分相近（虽然不完全一样）---- 在互联网上是免费提供的。可以找到它的一个好地方是 "the Cygnus Solutions Draft Standard C++ Page"（http://www.cygnus.com/misc/wp/），互联网上变化速度很快，如果你发现这个网站不能连接也不要奇怪。如果是这样，搜索引擎一定会帮你找到一个正确的URL。

我说过，"The Design and Evolution of C++" 对于了解 C++语言的设计思想很有好处，C++标准则明确了语言的具体细节；如果在 "D&E 千里之外的视野" 和 "C++标准的微观世界" 之间存在承上启下的桥梁那就太好了。教程应当适合于这个角色，但它们的视角往往偏向于标准，更侧重于说明什么是语言，而没有解释为什么。

进入 ARM 吧。ARM 是另一本书，"The Annotated C++ Reference Manual" (Addison-Wesley, 1990)，作者是 Margaret Ellis 和 Bjarne Stroustrup。这本书一出版就成为了 C++的权威，国际标准就是基于 ARM（和已有的 C 标准）开始制定的。这几年间，C++标准和 ARM 中的说明在某些方面有分歧，所以 ARM 不再象过去那样具有权威性了。但它还是很具参考价值，因为它所说的大多数还是正确的；所以，在 C++领域中，有些厂家还是坚持采用 ARM 规范，这并不少见，毕竟，标准只是最近才定下来。

然而，使得 ARM 真正有用的不是它的 RM 部分（the Reference Manual），而是 A 部分（the annotations）：注释。针对 C++的很多特性 "为什么" 要象现在这样工作，ARM 提供了全面的解释。这些解释 D&E 中也有一些，但大多数没有，你确实需要了解它们。例如，第一次碰到下面这段代码，大部分人会为它发疯：

```
class Base {
public:
  virtual void f(int x);
};
```

```
class Derived: public Base {
public:
  virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                    // 错误!
```

问题在于 Derived::f 隐藏了 Base::f，即使它们取的是不同的参数类型；所以编译器要求对 f 的调用取一个 double*，而 10 这个数字当然不行。

这不很合理，但 ARM 对这种行为提供了解释。假设调用 f 时，你真的是想调用 Derived 中的版本，但不小心用错了参数类型。进一步假设 Derived 是在继承层次结构的下层，你不知道 Derived 间接继承了某个基类 BaseClass，而且 BaseClass 中声明了一个带 int 参数的虚函数 f。这种情况下，你就会无意中调用了 BaseClass::f，一个你甚至不知道它存在的函数！在使用大型类层次结构的情况下，这种错误会时常发生；所以，为了防患于未然，Stroustrup 决定让派生类成员按名字隐藏掉基类成员。

顺便指出，如果想让 Derived 的用户可以访问 Base::f，可以很容易地通过一个 using 声明来完成：

```
class Derived: public Base {
public:
  using Base::f;              // 将 Base::f 引入到
                              // Derived 的空间范围
  virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                    // 正确，调用 Base::f
```

对于尚不支持 using 声明的编译器，另一个选择是采用内联函数：

```
class Derived: public Base {
public:
  virtual void f(int x) { Base::f(x); }
  virtual void f(double *pd);
};

Derived *pd = new Derived;
pd->f(10);                    // 正确，调用 Derived::f(int)，
                              // 间接调用了 Base::f(int)
```

借助于 D&E 和 ARM，你会对 C++的设计和实现获得透彻理解，从而可能参悟到：有时候，看似巴洛克风格的建筑外观之后，是合理严肃的结构设计。（译注：巴洛克风格的建筑极尽

富丽堂皇、粉装玉琢，因而结构复杂，甚至有点怪异）将这些理解和 C++标准的具体细节结合起来，你就矗立于软件开发的坚实基础之上，从而走向真正有效的 C++程序设计之路。