《C++ 编程规范：101条规则、准则最佳实践》读书笔记

整理日期：20120617

主页：http://chuanqi.name
E-mail：chuanqi.tan@gmail.com
光是 Herb Sutter & Anderi Alexandrescu 合著，就已经没有理由不读本书了！

By 谭川奇

前 言

自从Scott Meyers开创了Effective C++系列之后，这种以条款为组织方式的C++书籍便一发不可收拾，而且很多也成为了经典。Herb Sutter & Anderi Alexandrescu 合著的本书号称是全球C++界20年集体智慧和经验的凝结，自然是不可多得的好书。两位大师对规范尺度的拿捏、对经验的总结、对语言的深刻理解，都是我们很难达到的境界，我们只要做到仔细聆听并记住理解就好了。

但愿本笔记能为你的学习提供一些帮助！

谭川奇 2012.06.17 于北京
chuanqi.tan@gmail.com

本文采用的约定

绿色：我写的评论

蓝色：关键字、行业用语、缩写

下划线：值得多看一眼（的经验）

暗蓝色：书中的摘要

红色：非常重要的警告、经验

目 录

组织和策略问题

0. 不要拘泥于小节（又名：了解哪些东西不应该标准化）

只规定需要规定的事情：不要强制施加个人喜好或者过时的做法。

----不要规定缩进多少，应该规定要用缩进来体现代码的结构。

----不要强制行的具体长度，应该保证代码行的长度有利于阅读。

----不要在命名方面规定太多，应该规定的是使用一致的命名规范。

----不要规定注释体例（除非需要使用工具从特定的体例中提取出文档。

例子

----括号的位置。

----空格与制表符。

----匈牙利记法（有害无益，不要这样做）。

----单入口，单出口(single entry,single exit,SESE)。

1. 在高警告级别干净利落地进行编译

高度重视警告：使用编译器的最高警告级别。应该要求构建是干净利落的（没有警告）。理解所有的警告。通过修改代码而不是降低警告级别来排除警告。

排除警告的正确做法：

----把它弄清楚；然后

----改写代码以排除警告，并且

----使代码阅读者和编译器都能更加清楚，代码是按编写者的意图执行的。

例子

----第三方头文件

----“未使用的函数参数”（Unused function parameter）

----“定义了从未使用过的变量”（Variable defined but never used）

----“变量使用前可能未经初始化”（Variable may be used without being initialized）

----“遗漏了return语句”（Missing return）

----“有符号数/无符号数不匹配”（Signed/unsigned mismatch）

2. 使用自动化构建系统

一次按键就解决问题：使用完全自动化（“单操作”）的构建系统，无需用户干预即可构建整个项目。

可以把这里的自动构建系统理解为像VS这样的IDE，能够自动的编译连接，而无须像在GCC中那样手动的make,like

3. 使用版本控制系统

好记性不如烂笔头（中国谚语）：请使用版本控制系统（version control system，VCS）。永远不要让文件长时间地登出。在新的单元测试通过之后，应该频繁登入。确保登入的代码不会影响构建功能。

4. 在代码审查上投入

审查代码：更多的关注有助于提高质量。亮出自己的代码，阅读别人的代码。互相学习，彼此都会受益。

设计风格

5. 一个实体应该只有一个紧凑的职责

一次只解决一个问题：只给一个实体（变量、类、函数、名称空间、模块和库）赋予一个定义良好的职责。随着实体变大，其职责范围自然也会扩大，但是职责不应该发散。

例子

----realloc（在标准C语言中，realloc是一个臭名昭著的不良设计担当太多的任务。）

----basic_string（在标准C++语言中，basic_string是一个臭名昭著的不良设计巨大的类设计太臃肿。）

6. 正确、简单和清晰第一

软件简单为美（Keep It Simple Software,KISS）：正确优于速度。简单优于复杂。清晰优于机巧。安全优于隐患。

格言警句：

----简单设计和清晰代码的价值怎么强调都不过分。不知道

----程序必须为阅读它的人而缩写，只是顺便用于机器执行。一行写不下

----编写程序应该以人为本，计算机为二。理解就好

----计算机系统中最便宜、最快速、最可靠的组件都还不存在。等你开发

----所缺的恰是最精确（永不出错），最安全（坚不可摧）的简单设计。改动了一些

----使设计、文档编写、测试和维护起来尽量变容易，简单化也是一门学问

例子：

----不要使用不必要的或者小聪明式的操作符重载。

----应该使用命名变量，而不要使用临时变量，作为构造函数的参数。

7. 编程中应知道何时和如何考虑可伸缩性

小心数据的爆炸性增长：不要进行不成熟的优化，但是要密切关注渐进复杂性。处理用户数据的算法对所

处理的数据量耗费的时间应该是可预测的，最好不差于线性 关系。如果能够证明优化必要而且非常重要，尤其在数据量逐渐增长的情况下，那么应该集中精力改善算法的O（N）复杂性，而不是进行小型的优化，比如节省一 个多余的加法运算。

请预先做好下面事情：

----使用灵活的、动态分配的数据，不要使用固定大小的数组。

----了解算法的实际复杂性。

----优先使用线性算法或者尽可能快的算法。

----尽可能避免于线性复杂性的算法。

----永远不要使用指数复杂性的算法，除非你已经山穷水尽，确实别无选择。

8．不要进行不成熟的优化

拉丁谚语云，快马无需鞭策：不成熟优化的诱惑非常大，而它的无效性也同样严重。优化的第一原则就是，不要优化。优化的第二原则（仅使用于专家）是：还是不要优化。再三测试，而后优化。

请记住：

----请在优化前确保你将所做的优化是确实有价值的。

----让一个正确的程序更快速，比让一个快速的程序正确，要容易的太多，太多。

----为人编写代码。

----某一天需要优化代码，首先要考虑算法优化，并尝试将优化封装和模块化，然后注释修改历史。

例子：

----inline悖论

编译器可以通过函数的命中计数出色地告诉我们哪些函数应该但是没有被标记为inline，它可以自动的去优化。然后分析器在寻找哪些函数不应该却已经标记为inline时极不擅长。所以，除非非常确定，不要盲目用inline。

9．不要进行不成熟的劣化

放松自己，轻松编程：在所有其他事情特别是代码复杂性和可读性都相同的情况下，一些高效的设计模式和编程惯用法会从你的指尖自然流出，而且不会比悲观的替代方案更难写。这并不是不成熟的优化，而是避免不必要的劣化（pessimization）。

例子：

----在可以用通过引用传递的时候，却定义了通过值传递的参数。

----在使用前缀++操作符很合适的场合，却使用了后缀版本。

----在构造函数中使用赋值操作而不是初始化列表。

10．尽量减少全局和共享数据

共享会导致冲突：避免共享数据，尤其是全局数据。共享数据会增加耦合度，从而降低可维护性，通常还会降低性能。

例子：

----避免使用名字空间作用域中具有外部连接的数据或者作为静态类成员的数据。

----全局名称空间中的对象名称还会污染全局名字空间。

----名字空间作用域中的对象、静态成员对象或者跨线程或跨进程共享的对象会减少多线程和多处理器环境中的并行性，往往是产生性能和可伸缩性瓶颈的源头。

----应尽量降低类之间的偶合，尽量减少交互。

用通信方式（比较消息队列）代替数据共享。

跨线程共享对象的代码应该总是将对这些共享对象的所有访问序列化。

11. 隐藏信息

不要泄密：不要公开提供抽象的实体的内部信息。

例子：

----应该公开抽象，而不是数据。

----不要从任何提供抽象的实体中公开数据。

----绝对不要将类的数据成员设为public。

----绝对不要公开指向它们的指针或句柄。

值的聚合struct只是简单的将数据绑定在一起，并不提供任何抽象，所以它不需要隐藏数据。

12. 懂得何时和如何进行并发性编程

线程安全地：如果应用程序使用了多个线程或者进程，应该知道如何尽量减少共享对象，以及如何安全地共享必须共享的对象。

最重要的问题：避免死锁，活锁（livelock）和恶性的竞争条件（包括加锁不足导致的崩溃）。

若应用程序需要跨线程共享数据，请：

----参考目标平台的文档，了解该平台的同步化原语。

----最好将平台的原语用自己设计的抽象包装起来。

----确保正在使用的类型在多线程程序中使用是安全的。

----参阅更多书籍。

13. 确保资源为对象所拥有。使用显式的RAII和智能指针。

利器在手，不要在徒手为之：C++的"资源获取即初始化"（rscourse acquisition is initialization，RAII）惯用法是正确处理资源的利器。RAII使编译器能够提供强大且自动的保证，这在其他语言中是需要脆弱的手工编写 的惯用法才能实现的。分配原始资源的时候，应该立刻将其传递给属主对象。永远不要在

一条语句中分配一个以上的资源。

应该在自己的代码语句中执行显式的资源分配（比如new）而且每次都应该马上将分配的资源赋予管理对象（比如shared_ptr）。

C++可以确保对象在何时被分配在何时被释放，这是它最强大的地方也是最危险的地方。任何时候直接使用原始的指针都是一件极其危险的事情。

在实现RAII时，要小心复制构造和赋值，编译器生成的版本可能并不正确。如果复制没有意义，一定要明确的将复制构造和赋值设为私有从而明确禁止二者。

编程风格

14. 宁要编译时和连接时错误，也不要运行时错误

能够在编译时做的事情，就不要推迟到运行时：编写代码时，应该在编译期间使用编译器检查不变式（invariant），而不应该在运行时在进行检查。运行 时检查取决于控制流和数据的具体情况，这意味者难知道检查是否彻底。相比而言，编译时检查与控制流和数据无关，一般情况下能够获得更高的可信度。

充分利用C++静态检查功能：

----静态检查与数据和控制流无关。

----静态表示的模型更加可靠。

----静态检查不会带来运行时开销。

可用编译时检查代替运行时检查的一些情况：

----编译时布尔条件。

----编译时多态。考虑用编译时多态（模板）来代替运行时多态（虚拟函数）。前者产生的代码能够更好的进行静态检查。

----枚举。

----向下强制。

15. 积极使用const

const是我们的朋友：不变的值更易于理解、跟踪和分析，所以应该尽可能地使用常量代替变量，定义值的时候，应该把const作为默认的选项：常量很安 全，在编译时会对其进行检查，而且它与C++的类型系统一浑然一体。不要强制转换const的类型，除非要调用常量不正确的函数。

常量正确性是值得实现的。

不要强制转换const，除非要调用常量不正确的函数，或者在一些很罕见的情况下，为了解决老编译器中不支持mutable的问题。

例子：

----在函数声明中，要避免将通过值传递的函数参数声明为const，因为如果这样声明的const完全是多余

的。

16. 避免使用宏

实_不_相_瞒：宏是C和C++语言的抽象设施中最生硬的工具，它是披着函数外衣的饥饿的狼，很难驯服，它会我行我素地游走于各处。要避免使用宏。

知道吗？

----在C++中几乎从不需要定义宏

----可用const或enum定义易于理解的常量。

----用inline避免函数调用的开销。

----用template指定函数系列和类型系列。

----用namespace避免名称冲突。

----几乎每个宏都说明程序设计语言、程序或程序员存在缺陷。

----宏表面上看起来很好，实际上却是另一回事。

----宏会忽略作用域，忽略类型系统，忽略所有其他的语言特定和规则。

----宏还会劫持它为文件其余部分所定义（#define）的符号。

----宏中的错误可能只有在宏展开之后才能被报告出来，而不是在定义时。

例子：

----将模板实例化传给宏。

使用宏的情况：

----#include保护符（guard）

----条件编译中的#的ifdef和#if defined

----assert的实现

宏最凶狠的地方在于：它的效果在预处理阶段就产生了，而此时C++的语法和语义规则都还没有起作用。

关于宏的第一条规则就是：不要使用它，除非不得不用。几乎每个宏都说明程序设计语言、程序或者程序员存在缺陷。--Stroustrup

17. 避免使用"魔数"

程序设计并非魔术，所以不要故弄玄虚：要避免在代码中使用诸如42和3.14159这样的文字常量。它们本身没有提供任何说明，并且因为增加了难于检测的重复而使维护更加复杂。可以用符号名称和表达式替换它们，比如width*aspectRatio。

例子：

----重要的特定领域的常量应该放在名字空间一级。

----特定于类的常量。

应该用符号常量替换直接写死的字符串。将字符串与代码分开（比如将字符串放入一个专门的cpp文件或者

资源文件中），这样非程序员也能对其进行审查和更新，而且能够减少重复，还有助于国际化。

18．尽可能局部地声明变量

避免作用域膨胀，对于需求如此，对于变量也是如此。变量将引入状态，而我们应该尽可能少地处理状态，变量的生存期也是越短越好。

在能够合理地初始化（并非赋值，只要能合理的初始化就可以了）一个变量之前，决不要声明它。未初始化的变量是所有C和C++程序中普通的错误来源。

19．总是初始化变量

一切从白纸开始：未初始化的变量是C和C++程序中错误的常见来源。养成在使用内存之前先清除的习惯，可以避免这种错误，在定义变量的时候就将其初始化。

例子：

----使用默认初始值或？：减少数据流和控制流的混合。

----用函数替代复杂的计算流。有时候计算值的最好方式是将计算封装在一个函数中。

----初始化数组。

20．避免函数过长，避免嵌套过深

短胜于长，平优于深：过长的函数和嵌套过深的代码块的出现，经常是因为没能赋予一个函数以一个紧凑的职责所致，这两种情况通常都能通过更好的重构予以解决。

建议：

----尽量紧凑：对一个函数只赋予一种职责。

----不要自我重复：优先 使用命名函数，而不要让相似的代码片断反复出现。

----优先使用&&：在可以使用&&条件判断的地方要避免使用连续嵌套的if。

----不要过分使用try：优先使用析构函数进行自动清除而避免使用try代码块。

----优先使用标准算法：算法比循环嵌套要少，通常也更好。

----不要根据类型标签（type tag）进行分支（switch）：优先使用多态函数。

21．避免跨编译单元的初始化依赖

保持（初始化）顺序：不同编译单元中的名字空间级对象决不应该在初始化上相互依赖，因为其初始化顺序是未定义的。这样做会惹出很多麻烦，轻则在项目中稍做修改就会引发奇怪的崩溃，重则出现严重的不可移植问题----即使是同一编译器的新版本也不行。

因此，在任何名字空间级对象的初始化代码中，不能假设其它编译单元中定义的任何其它对象都已经初始化了。

如果确实需要可能依赖于另一个变量的此种变量，可以考虑使用Singleton（单体）设计模式。（项目中就

出现了）

22. 尽量减少定义性依赖。避免循环依赖

不要过分依赖：如果用前向声明（forward declaration）能够实现，那么就不要包含（#include）定义。

不要相互依赖：循环依赖是指两个模块直接或者间接地相互依赖。所谓模块就是一个紧凑的发布单元。相互依赖的多个模块并不是真正的独立模块，而是紧紧胶着在一起的一个更大的模块，一个更大的发布单元。因此，循环依赖有碍于模块性，是大型项目的祸根。请避免循环依赖。

23. 头文件应自给自足

各司其责：应该确保所编写的每个头文件都能独立进行编译，为此需要包含其内容所依赖的所有头文件。

建议：

----不要包含并不需要的头文件，它们只会带来零乱的依赖性。

----构建时，独立编译每个头文件，并确认没有产生错误或者警告。

例子：

----非独立名称。

----只在使用时才实例化成员函数模板和模板的成员函数。

eg：如果a.h依赖于b.h, c.h，而b.h依赖于c.h。

由于b.h依赖于c.h，所以在b.h中#include "c.h"。此时：如果a.h只#include "b.h"就可以完成编译了，但是它实际上还依赖了c.h所以应该在a.h中完整的包含两个头文件，明确的写上：

#include "b.h"

#include "c.h"

24. 总是编写内部#include保护符，决不要编写外部#include保护符函数与操作符

为头（文件）添加保护：在所有头文件中使用带有唯一名称的包含保护符号（#include guard）防止无意的多次包含。

定义包含保护符时应遵守以下规则：

----保护符使用统一名称。

----不要自作聪明地在受保护部分的前后放置代码或注释。

函数与操作符

25. 正确地选择通过值、（智能）指针或者引用传递参数

正确选择参数：分清输入参数、输出参数和输入/输出参数，分清值参数和引用参数。正确地传递参数。

正确选择参数是通过值、通过引用还是通过指针传递，是一种能够最大程度提高安全性和效率的好习惯。选择如何传递参数时，应遵循以下准则。对于只输入（input-only）参数：

----始终用const限制所有指向只输入参数的指针和引用。

----优先通过值来取得原始类型（如char、float）和复制开销比较低的值对象（如Point、complex<float>）的输入。

----如果函数需要其参数的副本，则可以考虑通过传递代替通过引用传递。

对于输入参数或者输入/输出参数：

----如果参数是可选的或者函数需要保存这个指针的副本或者操控参数的所有权，那么应该优先通过（智能）指针传递。

----如果参数是必需的，而且函数无需保存指针指向参数的指针，或者无需操控其所有权，那么应该优先通过引用传递。

引用和指针的区别就是是否可以为空。应该用是否可以为空来区别使用引用或者指针。

26. 保持重载操作符的自然语义

程序员讨厌意外情况：只在有充分理由时才重载操作符，而且应该保持其自然语义；如果做到这一点很困难，那么你可能已经误用了操作符重载。

如果有疑问，就按int类型那样去设计操作符重载。

27. 优先使用算术操作符和赋值操作符的标准形式

如果要定义a+b，也应该定义a+=b：在定义二元操作符时，也应该提供操作符的赋值形式，并且应该尽量减少重复，提高效率。

用@=来定义@操作，如下所示：

```
T& T::operator@=(const T&){
        //...
        return *this;
}

T operator@(const T& lhs, const T& rhs){
        T temp(lhs);
        return tmp @= rhs;
}
```

28. 优先使用++和—的标准形式。优先调用前缀形式

如果定义++c，也要定义c++：递增和递减操作符很麻烦，因为它们都有前缀和后缀形式，而两种形式语

义有略有不同。定义operator++和operator—时，应该模仿她们对应的内置操作符，如果不需要原值，应该优先调用前缀版本。
对于++和—而言，后缀形式返回的是原值，而前缀形式返回的是新值。
应该用前缀形式实现后缀形式。

29．考虑重载以避免隐含类型转换

如无必要勿增对象：隐式类型转换提供了语法上的便利，但是如果创建对象时对象的工作并必要而适于优化，那么可以提供签名与常见参数类型精确匹配的重载函数，而且不会导致转换。
运算符重载经常被过度的使用。用int.tostring()来代替int -> string的隐式类型转换几乎总是更好的设计。

30．避免重载&&，||或，（逗号）

明智就是知道何时应该适可而止：内置的&&、||和，（逗号）得到了编译器的特殊照顾，如果重载它们，它们就会变成普通函数，具有完全不同的语义，这肯定会引入微妙的错误和缺陷。不要轻率地重载这些操作符。
例子：
----用带有重载operator，的初始化库，用于对序列进行初始化。
例外：
----表达式模板库是一个例外，设计它的目的就是用来捕获所有操作符。
重载这三个操作符会改变它们本来的语义（短路求值原则），这是被编译器特别照顾的。

31．不要编写依赖于函数参数求值顺序的代码

保持（求值）顺序：函数参数的求值顺序是不确定的，因此不要依赖具体的顺序。
解决方案：用命名对象控制求值顺序。

类的设计与继承

32．弄清所要编写的是哪种类

了解自我：有很多种不同的类。弄清楚要编写的是那一种。
一个值类应该：

- 有一个公用析构函数，复制构造函数和带有值语义的赋值。
- 没有虚拟函数（包括析构函数）。
- 是用作具体类，而不是基类。

- 总是在栈中实例化，或者作为另一个类直接包含的成员实例化。

基类是类层次结构的构造要素。一个基类应该：

- 有一个公用而且虚拟，或者保护而且非虚拟的析构函数，和一个非公用复制构造函数和赋值操作符。
- 通过虚拟函数建立接口。
- 总是动态地在堆中实例化为具体派生对象，并通过一个（智能）指针来使用。

不严格地说，traits类是携带有关类型信息的模板。一个traits类应该：

- 只包含typedef和静态函数。没有可修改的状态或者虚拟函数。
- 通常不实例化（其构造一般是被禁止的）。

策略类（通常是模板）是可插拔行为的片段。一个策略类应该：

- 可能有也可以没有状态或者虚拟函数。
- 通常不独立实例化，只作为基类或者成员。

一个异常类应该：

- 有一个公用析构函数和不会失败（no-fail）的构造函数（特别是一个不会失败的复制构造山书，从异常的复制构造函数抛出将使程序中止）。
- 有虚拟函数，经常实现克隆和访问。
- 从std::exceptio虚拟派生更好。

33. 用小类代替巨类

分而治之：小类更容易编写，更容易保证正确，测试和使用。小类更有可能适用于各种不同情况。应该用这种小类体现简单概念，不要用大杂烩式的类，它们要实现的概念既多又复杂。
用巨类来完成整个项目的设计的唯一结果就是失败！因为人的需求总是在变的。

34. 用组合代替继承

避免继承带来的重负：继承是C++中第二个紧密的耦合关系，仅次于友元关系。紧密的偶合是一种不良现

象，应该尽量避免。因此，应该用组合代替继承，除非知道后者确实对设计者有好处。

这里的"组合"就是指在一个类型中嵌入另一个类型的成员变量。

与继承相比，组合有以下重要优点：

----在不影响调用代码的情况下具有更大的灵活性。

----更好的编译时隔离，更短的编译时间。能够减少头文件的依赖，因为声明对象的指针不需要对对象的完整定义（因为指针在内存中的大小是固定的）。相反，继承则总是要求基类的完整定义可见。

----减少奇异现象。从一个类型继承，会导致名字查找涉及与该类型同一名字空间中定义的函数和函数模板。

----更广的实用性。

----更健壮、更安全。

----复杂性和脆弱性降低。

使用公用继承模仿可替换性。

要用非公用的继承的情况（从常用到罕用排序）：

----如果需要改写虚拟函数。

----如果需要访问保护成员。

----如果需要在基类之前构造已使用的对象，或者在基类之后销毁此对象。

----如果需要操心虚拟基类。

----如果能够确定空基类优化能带来好处，包括这种情况下优化的确很重要，以及这种情况下目标编译器确实能实施这种优化。

----如果需要控制多态。相当于说，如果需要可替换性关系，但是关系应该只对某些代码可见（通过友元）。

35. 避免从并非要设计成基类的类中继承

有些人并不想生孩子：本意是要独立使用的类所遵守的设计蓝图与基类不同。将独立类用作基类是一种严重的设计错误，应该避免。要添加行为，应该添加非成员函数而不是成员函数。要添加状态，应该使用组合而不是继承。要避免从具体的基类中继承。

例子：

----用组合代替公用继承或者私有继承。

----std::unary_function

36. 优先提供抽象接口

偏爱抽象艺术吧：抽象接口有助于我们集中精力保证抽象的正确性，不至于受到实现或者状态管理细节的干扰。优先采用实现了（建模抽象概念的）抽象接口的设计层次结构。

应该定义和继承抽象接口。

抽象接口是完全由（纯）虚拟函数构成的抽象类，没有状态（成员函数），通常也没有成员函数实现。

应遵循依赖倒置原理（Dependency Inversion Principle,DIP）

----高层模块不应该依赖于低层模块。相反，两者都应该依赖抽象（这点就对映了35点的层次结构应该以抽象类而不是具体类为根）。

----抽象不应该依赖细节。相反，细节应该依赖抽象。

DIP三个基本的设计优点：

----更强的健壮性。

----更大的灵活性。

----更好的模块性。

例子：

----备份程序

二次机会定律：需要保证正确的最重要的东西是接口。其它所有的东西以后都可以修改。如果接口弄错了，可能就再也不允许修改了。

空基类优化是一个纯粹为了优化使用继承（最好是非公用的）的实例。

37．公用继承即可替换性。继承，不是为了重用，而是为了被重用

知其然：公用继承能够使基类的指针或者引用实际指向某个派生类的对象，既不会破坏代码的正确性，也不需要改变已有代码。

还要知其所以然：不要通过公用继承重要（基类中的已有）代码，公用继承是为了被（已经多态地使用了对象的已有代码）重用的。

继承是为了多态的，而不是重用。绝不是为了重用基类的代码，而是为了被其它代码使用。（用基类代码实现自己）

理解这句话（是为了被重要，而不是为了重用基类代码），这点很重要！

当动态多态正确而且适合时，组合是自私的，而继承是慷慨的。

公用继承所建模的必须总是（is-a）原则：所有的基类约定必须满足这一点。

eg:建模鸟和鸵鸟时，由于鸵鸟不能飞，所以它们并不满足is-a关系（鸵鸟不能做所有鸟应该做的事）。

38．实施安全的改写

负责任地进行改写：改写一个虚拟函数时，应该保持可替换性：说得更具体一些，就是要保持基类中函数的前后条件。不要改变虚拟函数的默认参数。应该显式地将改写函数重新声明为virtual。谨防不小心在虚拟类中隐藏重载函数。

改写函数可以要求更少而提供更多，但绝不能要求更多而提供更少，因为这将违反已向调用代码保证过的约定。

在改写的时候，永远不要修改默认参数。

39．考虑将虚拟函数声明为非公用的，将公用函数生命为非虚拟的

在基类中进行修改代价昂贵（尤其是库中和框架中的基类）：请将公用函数设为非虚拟的。应该将虚拟函数设为私有的，或者如果派生类需要调用基类版本，则设为保护的。（请注意，此建议不适用与析构函数）。

非虚拟接口（Nonvirtual Interface,NVI）模式：将公用函数设为非虚拟的，将虚拟函数设为私有的（或者设为保护的，如果派生类需要调用基类的话）。

通过将公用函数与虚拟函数分离，好处是：

----每个接口都能自然成形。

----基类拥有控制权。

----基类能够健壮地适用变化。

NVI对析构函数不适用，因为它们的执行顺序很特殊。

NVI不直接支持调用者的协变返回类型。

40．要避免提供隐式转换

并非所有的变化都是进步：隐式转换所带来的影响经常是弊大于利。在为自定义类型提供隐式转换之前，请三思而行，应该依赖的是显式转换（explicit构造函数和命名转换函数）。

例子：

----重载。

----错误都变得可行了。

41．将数据成员设为私有的，无行为的聚集（C语言形式的struct）除外

它们不关调用者的事：将数据成员设为私有的。简单的C语言形式的struct类型只是将一组值聚集在一起，并不封装或者提供行为，只有在这种struct 类型中才可以将所有数据成员都设成公有的。要避免将公有数据和非公有数据混合在一起，因为这几乎总是是设计混乱的标志。（要么就全公用struct，要么就全私有class）

例子：

----正确封装。

----TreeNode。

----获取函数和设置函数。

用Pimpl惯用法来隐藏类的私有成员。

42．不要公开内部数据

不要过于自动自发：避免返回类所管理的内部数据的句柄，这样类的客户就不会不受控制地修改对象自己拥有的状态。

数据隐藏是一种强大的抽象方式，也是强大的模块化机制。

隐藏数据却又暴露句柄的做法只会弄巧成拙，就像你锁上了自己家的门，却把钥匙留在锁里或吊在门口。

一个常见的错误就是：const是浅的，不会通过指针传播。如下所示：

```
class Socket{
public:
        SystemHandle* GetHandle() const {return handle_;}     //违反了常量约定，而编译器还不知道
private:
        SystemHandle* handle_;                                 //这是一个可以操作系统中某些量的句柄
}
```

代码可以通过GetHandle()返回的值修改私有变量，原因就是因为const是浅的。

43．明智地使用Pimpl

抑制语言的分离欲望：C++将私有成员指定为不可访问的，但并没有指定为不可见的。虽然这样自有其好处，但是可以考虑通过Pimpl惯用法使私有成员真正不可见，从而实现编译器防火墙，并提高信息隐藏度。

》》关于Pimpl惯用法，我转载了一篇文章，基本把Pimpl的好处讲清楚了《《

Pimpl惯用法：将私有部分隐藏在一个不透明的指针（即指向已经声明但尚未定义的类的指针，最好是选择合适的智能指针）

C++可以将类的私有成员封装起来，从而防止未授权的访问，但在类的头文件中是可以看到类的私有部分的，因此客户代码就必须依赖在这些私有部分中使用到的所有类型。想将客户代码与类的私有实现细节分离开来，就要用到Pimpl手法。所谓Pimpl，就是一个不透明的指针，指向一个进行了前置声明但又没有定义的辅助类，用来隐藏类的私有成员。举个简单的例子如下：

```
//File Widget.h
class Widget
{
public:
        Widget(int x, int y);
    ~Widget();                                  //这个析构函数必须提供，因为隐式生成的版本会导致使用上的问题
        int getSum() const;
private:
        struct WidgetImpl;
        WidgetImpl* impl;
};
```

```
//File Widget.cpp
#include "Widget.h"
struct Widget::WidgetImpl                    //在实现文件里进行定义，这样就减少了依赖，隐藏了实现细节
{
        WidgetImpl(int x, int y) : a(x), b(y) {}
        int a;
        int b;
};

Widget::Widget(int x, int y) : impl(new WidgetImpl(x, y)) {}
Widget::~Widget() { delete impl; impl = 0; } //而且必须在.h文件中定义，在cpp文件中实现。

int Widget::getSum() const {
        return impl->a + impl->b;
}
```

这个手法带来的一个明显的好处是将对象占用的内存缩减到一个指针的大小，但它的主要优点还是在于它打破了编译期的依赖性。在私有部分中所使用的类型定义，只有在类的实现中才会需要，而在客户代码中是不需要的，这就可以去掉额外的#include并提高编译速度。而且，现在我们还可以自由地增加或者删除类的私有成员，而无需重新编译客户代码。

而Pimpl的主要开销在于程序的执行性能，在每个构造过程要分配内存，每个析构过程要释放内存，同时我们每次访问被隐藏起来的成员都至少需要一个额外的间接操作。

```
struct Impl;
shared_ptr<Impl> pimpl;
```

一定要分成二行写，因为只有这样，struct Impl类才是类中的嵌套类型，在类的定义域外是不可见的。

Pimpl用法利用了C++中（可见性）和（可访问性）之间的差别：

类的所有私有成员在成员函数和友元之外是不可以访问的，但是对所有看到得类的定义的代码来说，都是可见的。

Pimpl用法的三个好处：

- 减少了编译依赖，加快编译时间。
- 减少了名字查找时的二义性和名字隐藏。
- 对错误处理和异常安全的影响，可以写出更健壮的代码。

~Widget()必须在Widget.h中定义而且必须在Widget.cpp中实现（Pimpl手法时的析构函数一定要小心）：

因为在使用Pimpl手法时，WidgetImpl的完整定义是在Widget.cpp所在的编译单元的，而C++标准规定调用析构函数时必须在当前编译单元知道类的完整的定义，所以~Widget()必须在Widget.cpp是实现。

44．优先编写非成员非友元函数

要避免交成员费：尽可能将函数指定为非成员非友元函数。

非成员非友元函数可提高封装性，减少耦合，提高通用性。

45．总是一起提供new和delete

它们是一揽子交易：每个类专门的重载void* operator new(parms)都必须与对应的重载void operator delete(void*,parms)相随相伴，其中parms是额外参数类型的一个列表（第一个是std::size_t）。数组形式的new[]和 delete[]也同样如此。

46．如果提供类专门的new，应该提供所有标准形式（普通，就地和不抛出）

不要隐藏好的new：如果类定义了perator new的重载，则应该提供operator new所有三种形式普通（plain），就地（in-place）和不抛出（nothrow）的重载。不然，类的用户就无法看到和使用它们。

应该总是避免隐藏就地new，因为它在STL容器中有广泛的使用。

避免在客户代码中调用new（nothrow）版本，但仍要为客户提供，以免客户一旦要用到时感到奇怪。

析构与复制

47．　以同样的顺序定义和初始化成员变量

与编译器一致：成员变量初始化的顺序要与类定义中声明的顺序始终保持一致：不用考虑构造函数初始化列表中编写的顺序。要确保构造函数代码不会导致混淆地指定不同的顺序。

48．在构造函数中用初始化代替赋值

设置一次，到处使用：在构造函数中，使用初始化代替赋值来设置成员变量，能够防止发生不必要的运行操作，而输入代码的工作量则保持不变。

有话直说最好：在初始化列表中初始化成员变量，代码表达意图更加明确，而且锦上添花的是，代码通常还更小，更快。（但是：应该总是在构造函数体内而不是初始化列表中执行非托管资源的获取）

49. 避免在构造函数和析构函数中调用虚拟函数

虚拟函数仅仅"几乎"总是表现的虚拟：在构造函数和析构函数中，它们并不虚拟。更糟糕的是，从构造函数或析构函数直接或者间接调用未实现的纯虚拟函数，会 导致未定义的行为。如果设计方案希望从基类构造函数或析构函数虚拟分派到派生类，那么需要采用其他技术，比如后构造函数（post- constructor）。
从构造函数调用还完全没有定义的纯虚拟函数将是给伤口撒盐，这中情况下的行为是未定义的，代码令人糊涂，维护时显得更加脆弱。（因为此时的对象是不完整的，这个时间调用虚函数一切都是不确定的，绝对禁止）

50. 将基类析构函数设为公用且虚拟的，或者保护且非虚拟的

删除，还是不删除，这是个问题：如果允许通过基类Base的指针执行删除操作，则Base的析构函数必须是公用且虚拟的。否则，就应该是保护且非虚拟的。
结论：总是为基类编写析构函数，因为隐含生成的析构函数是公用且非虚拟的（因为大多数类都不是基类）。
析构函数是非常特殊的：子类会自动的去调用基类的析构函数，所以基类的析构函数最少应该是保护的，绝不能是私有的。（私有析构函数有两个特别的用处：一、基本能保证对象只分配在堆上。二、禁止从该类再派生子类）
这两种选择隐含着两种特定的设计：

- 含有多态删除的基类：如果允许多态删除，则析构函数必须是公用且虚拟的。
- 不含多态删除的基类：如果不允许多态删除，则析构函数必须是非公用（保护）且非虚拟的。（代理类）

总结一句：析构函数应该这样设计：

- 非基类：公用且非虚拟
- 基类：公用且虚拟 或者 保护且非虚拟。

51. 析构函数、释放和交换绝对不能失败

它们的一切尝试都必须成功：决不允许析构函数，资源释放（deallocation）函数（如operator delete）或者交换函数报告错误。说得更具体一些，就是绝对不允许将哪些析构函数可能会抛出异常的类型用于C++标准库。
析构函数应该设计的总能捕获异常，而且不会让异常传播到析构函数之外。
当使用异常作为错误处理机制时，建议用一个注释掉的空异常规范/*throw（）*/来声明这些函数，通过这种方式说明这一方式。

对于异常而言：只需要将析构函数所做的一切都包装在一个try/catch(...)块中即可。

52. 一致地进行复制和销毁

既要创建，也要清除：如果定义了复制构造函数、复制赋值操作符或者析构函数中的任何一个，那么可能也需要定义另一个或者另外两个。

53. 显式地启用或禁止复制

清醒地进行复制：在下述三种行为之间谨慎选择使用编译器生成的复制构造函数和赋值操作符；编写自己的版本；如果不允许复制的话，显式地禁用前两者。

54. 避免切片。在基类中考虑用克隆代替复制

切片面包很好：切片对象则不然：对象切片是自动的、不可见的，而且可能会使得漂亮的多态设计嘎然而止。在基类中，如果客户需要进行多态（完整的、深度的）复制的话，那么请考虑禁止复制构造函数和复制赋值操作符，而改为提供虚拟的Clone成员函数。

55. 使用域值的标准形式

赋值，你的任务：在实现operator=时，应该使用标准形式具有特定签名的非虚拟形式。
请：
----要避免将赋值操作符设为虚拟的。
----要始终保证复制赋值错误安全的，最好是提供强有力的保证。
----要确保赋值操作符是错误安全的，最好是提供强有力的保证。
----要显式调用所有基类赋值操作符，并为所有数据成员赋值。
要确保赋值操作符对于自我赋值是安全的。（swap惯用法）

56. 只要可行，就提供不会失败的swap（而且要正确地提供）

swap既可无关痛痒，又能举足轻重：应该考虑提供一个swap函数，高效且绝对无误地交换两个对象。这样的函数便于实现许多惯用法，从流畅地将对象四处移动以轻易地实现赋值，到提供一个有保证的、能够提供强大防错调用代码的提交函数。
在很多高级技巧里面：swap惯用法非常的常见且有用。

名字空间与模块

57. 将类型及其非成员函数接口置于同一名字空间中

非成员也是函数：如果要将非成员函数（特别是操作符和辅助函数）设计成类X的接口的一部分，那么就必须在与X相同的名字空间中定义它们，以便正确调用。
名字空间是管理名字和减少名字冲突的重要工具。模块也是如此，它还是管理发布和版本化的重要工具。
接口原则是这样表述的：对于一个类X而言，所有在同一个名字空间中"提及"X和随X"一起提供的"函数（包括非成员函数）逻辑上都是X的一部分，因为它们形成了X接口的组成部分。

58．应该将类型和函数分别置于不同的名字空间中，除非有意让它们一起工作

协助防止名字查找问题：通过将类型（以及与其直接相关的非成员函数）置于自己单独的名字空间中，可以使类型与无意的ADL（参数依赖查找，也称Koenig查找）隔离开来，促进有意的ADL。要避免将类型和模块化函数或者操作符放在相同的名字空间中。
这点我需要注意，我从来都没有做到过！
避免将不属于类型X的接口的非成员函数与X放在同一个名字空间中，尤其绝对不要将模板化函数或者操作符与用户定义的类型放在同一名字空间中。（基本上是由于模板和操作符重载引起的微妙错误）

59．不要在头文件中或者#include之前编写名字空间using

名字空间using是为了使我们更方便，而不是让我们用来叨扰别人的：绝对不要编写suing声明或者在#include之前编写using指令。
相反，应该显式地用名字空间限定所有的名字。

60．要避免在不同的模块中分配和释放内存

物归原位：在一个模块中分配内存，而在另一个模块中释放它，会在这两个模块之间产生微妙的远距离依赖，使程序变得脆弱。必须用相同版本的编译器、同样的标志（比较著名的如用debug还是NDEBUG）和相同的标准库实现对它们进行编译，实践中，在释放内存中，用来分配内存的模块最好仍在内存中。
安逸往往会导致健忘。

61．不要在头文件中定义具有链接的实体

重复会导致膨胀：具有链接的实体（entity with linkage），包括名字空间级的变量或函数，都需要分配内存。在头文件中定义这样的实体将导致连接时错误或者内存的浪费。请将所有具有链接的实体放入实现文件。
不要在头文件中定义名字空间级的static实体。
以下具有外部链接的实体可以放入头文件中：
----内联函数
----函数模板
----类模板的静态数据成员

这个错误我碰见过很多次，连接时发生的多重定义多是这样引起的。稍微大点的项目都有可能遇见这个错误。

62. 不要允许异常跨越模块边界传播

不要想邻家的花园抛掷石头：C++异常处理没有普遍通用的二进制标准。不要在两段代码之间传播异常，除非能控制用来构建两段代码的编译器和编译选项；否则模块可能无法支持可兼容地实现异常传播。这通常可以一言以蔽之：不要允许异常跨越模块或子系统边界传播。

最低限度，应用程序必须在以下位置有捕获所有异常的catch（...）兜底语句，其中大多数都直接使用于模块。

----在main函数的附近。

----在从无法控制的代码中执行回调附近。

----在线程边界的附近。

----在模块接口边界的附近。

----在析构函数内部。

63. 在模块的接口中使用具有良好可移植性的类型

生在（模块的）边缘，必须格外小心：不要让类型出现在模块的外部接口中，除非能够确保所有的客户代码能够正确地理解该类型。应该使用客户代码能够理解的最高层抽象。

模板与泛型

64. 理智地结合静态多态性和动态多态性

1+1可远远不止是2：静态多态性和动态多态性是相辅相成的。理解它们的优缺点，善用它们的长处，结合两者获得两方面的优势。

在C++中动态多态性最擅长于以下几个方面：

----基于超集/子集关系的统一操作。

----静态类型检查。

----动态绑定和分别编译。

----二进制接口。

而在C++中静态多态性最擅长于以下几个方面

----基于语法和语义接口的统一操作

----静态类型检查

----静态绑定（防止分别编译）

----效率

65. 有意地进行显式自定义

有意胜过无意，显式强似隐式：在编写模板时，应该有意地、正确地提供自定义点，并清晰地记入文档。在使用模板时，应该了解模板想要你如何进行自定义以将其用于你的类型，并且正确地自定义。
为了避免无意地提供自定义点，应该：
----将模板内部使用的任何辅助函数都放入其自己的内嵌名字空间，并用显式的限定调用它们以禁用ADL。
----要避免依靠依赖名。

66. 不要特化函数模板

只有在能够正确实施的时候，特化才能起到好作用：在扩展其他人的函数模板（包括std::swap）时，要避免尝试编写特化代码；相反，要编写函数模板的重载，将其放在重载所用的类型的名字空间中。编写自己的函数模板时，要避免鼓励其他人直接特化函数模板本身。
这和操作符重载是一个道理，不是非常明确是正确的时候就不要使用。这些语言甜点有时非常难以理解，太容易出些微妙的错误了。

67. 不要无意地编写不通用的代码

依赖抽象而非细节：使用最通用、最抽象的方法来实现一个功能。

错误处理与异常

问题不在于我们是否会犯编程错误，而在于我们是否会安排编译器和工具寻找错误。

68. 广泛地使用断言记录内部假设和不变式

使用断言吧！广泛地使用assert或者等价物记录模块内部（也就是说，调用代码和被调用代码由同一个人或小组维护）的各种假设，这些假设是必须成立的，否则就说明存在编程错误（例如，函数的调用代码检查到函数的后条件不成立）。当然，要确保断言不会产生任何副作用。
千万不要在断言中编写具有副作用的表达式。如：assert(++i < limit);
技巧：还可以考虑在更复杂的断言中加入&&"informational message"，尤其是这样还可以取代注释。
用抛出异常来代替断言是不可取的，因为异常会导致栈的展开，而调试的时候，我们需要保持原样。
断言用在调试时、异常用到运行时。

69. 建立合理的错误处理策略，并严格遵守

应该在设计早期开发实际、一致、合理的错误处理策略，并予以严格遵守。许许多多的项目对这一点的考虑（或者错误估计（都相当草率，应该对此有意思地规定，并认真应用。策略必须包含以下内容：

----鉴别：哪些情况属于错误。
----严重程度：每个错误的严重性或紧急性。
----检查：哪些代码负责检查错误。
----传递：用什么机制在模块中报告和传递错误通知。
----处理：哪些代码负责处理错误。
----报告：怎样将错误记入日志，或通知用户。
只在模块边界处改变错误处理机制。

70. 区别错误与非错误

违反约定就是错误：函数是一个工作单元。因此，失败应该视为错误，或根据其对函数的影响而定。在函数f中，当且仅当失败违反了f的一个前条件，或者阻碍了f满足其调用代码的任何前条件，实现f自己的任何后条件或者重新建立f有责任维持的不变式时，失败才是一个错误。
错误就是阻止函数成功操作的任何失败。有三种类型：
----违反或者无法满足前条件。
----无法满足后条件。
----无法重新建立不变式。
例子：
----std::string::insert（前条件错误）
----std::string::append（后条件错误）
----无法生成返回值（后条件错误）
----std::string::find_first_of（在string的上下文中不是错误）
----同一函数中的不同错误情况。
----同一情况的不同状况。

71. 设计和编写错误安全代码

承诺，但是不是惩罚：在所有函数中，都应该提供最强的安全保证，而且不应该惩罚不需要这种保证的调用代码。至少要提供基本保证。确保出现错误时程序会处于 有效状态。这是所谓的基本保证（basic guarantee）。
要小心会破坏不变式的错误（包括但是不限于泄露），它们肯定都是bug。
应该进一步保证最终状态要么是最初状态（如果是错误，则回 滚操作），要么是所希望的目标状态（如果没有错误，则提交操作）。这就是所谓的强保证（strong guarantee）。应该进一步保证操作永远不会失败。虽然这对于大多数函数来说是不可能的，但是对于析构函数释放函数这样的函数来说则是必须的。这就 是所谓的不会失败保证（no-fail guarantee）。
当有可能出现错误时，最安全的方法就是确保函数支持事务性的行为。

将能提供异常安全和不能提供异常安全的代码分开来。

72. 优先使用异常报告错误

出现问题时，就使用异常：应该使用异常而不是错误来报告错误。但不能使用异常时，对于错误以及不是错误的情况，可以使用状态码（比如返回码，errno）来报告异常。当不可能从错误中恢复或者不需要恢复时，可以使用其他方法，比如正常终止或者非正常终止。
关于异常：
----异常不能不加修改地忽略。
----异常是自动传播的。
----有了异常处理，就不必在控制流的主线中加入错误处理和恢复了。
----对于从构造函数和操作符报告错误来说，异常处理要优于其他方案。
异常处理潜在的主要缺点在于：它要求程序员必须熟悉一些会反复遇到的惯用法，这些管用法来源于异常的特殊控制流。

73. 通过值抛出，通过引用捕获

学会正确捕获（catch）：通过值（而非指针）抛出异常，通过引用（通常是const的引用）捕获异常。这是与异常语义配合最佳的组合。当重新抛出相同的异常时，应该优先使用throw；，避免使用throw e；（又多了一次包装，会产生一个新的对象）。

- 在抛出异常时，要通过值抛出对象。要避免抛出指针，因为如果抛出指针，就需要处理内存管理问题。
- 在捕获异常时，要通过引用捕获。因为如果通过值捕获普通值将在捕获处引起切片问题，这会粗暴的去除通常至关重要的异常对象的多态性。

74. 正确地报告、处理和转换错误

什么时候说什么话：在检查出并确认是错误时报告错误。在能够正确处理错误的最近一层处理或者转换每个错误。

75. 避免使用异常规范

对异常规范说不：不要在函数中编写异常规范，除非不得以而为之。
C++的异常规范是有缺陷的，不要使用。

STL：容器

76. 默认时使用vector。否则，选择其他合适的容器

使用"正确的容器"才是正道：如果有充分的理由才使得某个特定容器类型，那就用好了，因为我们心中有数：自己做出了正确的选择。使用vector同样如此：如果没有充分理由，那就编写vector，继续前进，无需停顿，我们同样心中有数：自己做出了正确的选择。

三个基本议题：

----编程时正确、简单和清晰是第一位的。

----编程时只在必要时才考虑效率。

----尽可能编写事务性的，强错误安全的代码，而且不使用失效对象。

vector不能处理其中类型在构造和析构时的异常，但list能对这些类型集合的插入操作提供强安全保证，在此情况下，这一点有可能非常重要。（但是对我们编程来说，99%不需要处理这样的情况）

77. 用vector和string代替数组

何必用贵重的明代花瓶玩耍杂技呢？不要使用C语言风格的数组、指针运算和内存管理原语操作实现数组抽象。使用vector或者string不仅更轻松，而且还有助于编写更安全、伸缩性更好的软件。

78. 使用vector（和string::c_str）与非C++API交换数据

vector不会在转换中迷失：vector和string::c_str是与非C++API通信的通道。但是不要将迭代器当作指针。要获取vector<T>::iterator iter所引用的元素地址,应该使用&*iter。

79. 在容器中只存储值和智能指针

在容器中存储值对象:容器假设它们所存放的是类似值的类型,包括值类型(直接存放),智能指针和迭代器。

个人观点：所谓类似于值的类型，就是可以随意复制的类型

例子：

----auto_ptr（用shared_ptr代替）

----异构容器

----非值类型的容器

----可选值

----索引容器

80. 用push_back代替其他扩展序列的方式

尽可能使用push_back:如果不需要操心插入位置,就应该使用push_back在序列中添加元素。其他方法可能极慢而且不简明。

81．多用范围操作，少用单元素操作

顺风顺水无需浆（拉丁谚语）：在序列容器中添加元素时，应该多用范围操作（例如接受一对迭代器为参数的insert形式），而不要连续用该操作的单元素形式。调用范围操作通常更易于编写，也更易于阅读，而且比显式循环的效率更高。（for_each，还有操作一个范围的标准算法）

例子：

----vector：：insert

----范围构造和赋值：调用以迭代器范围为参数的构造函数（或assign函数），通常比调用默认构造函数（或clear）然后再单独插入容器的性能更好。

82．使用公认的惯用法真正地压缩容量，真正地删除元素

使用有效减肥法：要真正地压缩容器的多余容量，应该使用"swap魔术"惯用法。要真正地删除容器中的元素，应该使用eraser-remove惯用法。

```
c.eraser( remove(c.begin(), c.end(), value), c.end() );
```

STL：算法

开始使用算法的那一刻起，也将开始使用"函数对象"和"谓词"！

83．使用带检查STL实现

安全第一：即使只在其中的一个编译器平台上可用，即使只能在发行前的测试中使用，也仍然要使用带检查的STL实现。

STL错误：

----使用已失效的或未初始化的迭代器。

----传递越界索引。

----使用并非真是"范围"的迭代器。

----传递无效的迭代器位置。

----使用无效顺序。

84．用算法调用代替手工编写的循环

明智地使用函数对象：对非常简单的循环而言，手工编写的循环有可能是最简单也是最有效的解决方案。但是编写算法调用代替手工编写的循环，可以表达力更强、 维护性更好、更不易出错，而且同样高效。调用算法时，应该考虑编写自定义的函数对象以封装所需的逻辑。不要将参数帮定器（parameter- binder）和简单的函数对象凑在一起，通常这会降低清晰性。还可以考虑尝试[Boost]的Lambda库，这个库自动化了函数对象的编写过程。

特别是在有了Lambdas表达式之后，用算法代替手工循环更方便和高效了。

85．使用正确的STL查找算法

选择查找方式应"恰到好处"正确的查找方式应该使用STL（虽然比光速慢，但已经非常快了）：本条款适用于在一个范围内查找某个特定值，或者查找某个 值的位置（如果它处在范围内的话）。查找无序范围，应使用find/find_if或者count/count_if 。查找有序范围，应使用lower_bound、upper_bound、equal-range或者（在少数情况下）binary_search（尽管 binary_search有一个通行的名字，但是选择它通常并不一定正确）。
因为binary_search只能返回一个bool值表示是否找到了匹配。

86．使用正确的STL排序算法

选择排序方式应"恰到好处"：理解每个排序算法的作用，选择能够实现所需而开销最低的算法。
接开销从低到高的大致顺序：partition、stable_partition、nth_element、partial_sort(partial_sort_copy)、sort、stable_sort。
索引容器惯用法：如果在需要随机访问迭代器时（使用强的排序算法）只有双向迭代器的话，可以创建一个支持随机访问迭代器的迭代器容器（如：vector），用其中的迭代器指向原范围，然后对此迭代器容器使用更强的算法，同时使用谓词的析值版本（在进行通常的比较之前析值迭代器）

87．使谓词成为纯函数

保持谓词纯洁性：谓词就是返回是或否（返回值通常为bool类型）的函数对象。从数学的意义上来说，如果函数的结果只取决于其参数，则该函数就是一个纯函 数。不要让谓词保存或访问对其operator（）结果有影响的状态，包括成员状态和全局状态。应该使operator（）谓词的const成员函数。

88．算法和比较器的参数应多用函数对象少用函数

对象的适配性比函数好：应该向算法传递函数对象，而非函数。关联容器的比较器比较是函数对象。函数对象的适配性好（可适配性：可以使用一些绑定器and、or、非、绑定参数什么的。普通函数是不可适配的），而且与直觉相反，它们产生的代码一般比函数更快。

89．正确编写函数对象

成本要低，而且要可适配：将函数对象设计为复制成本很低的值类型。尽可能地让它们从unary_function或binary_function继承，从而能够适配。

类型安全

90. 避免使用类型分支，多使用多态

切勿分支：避免通过对象类型分支来定制行为。使用模板和虚函数，让类型自己（而不是调用它们的代码）来决定行为。
抽象之所以称之为抽象，就是因为它们比"细节"稳定的多。而软件设计的原则之一就是：对于任何可能会变的东西，用一个中间层封装起来。

91. 依赖类型，而非其表示方式

不要企图给对象拍X光片：不要对对象在内存中的准确表示方式做任何假设。相反，应该让类型决定如何在内存中读写其对象。
C++现行标准对类型的内存表示方式规定的非常非常少，内存中的布局全部依赖于编译器的具体实现。基本上所有看似合理的假设都有可能是错误的。

92. 避免使用reinterpret_cast

谎言总是站不住脚的：不要尝试使用reinterpret_cast强制编译器将某个类型 的内存表示重新解释成另一种类型的对象。这违反了维护类型安全性的原则，尤其可怕的是，reinterpret_cast甚至不能保证是否能够达到这一目 的，也无法保证其他功能。
欺骗编译器的人，最终将自食其果。

93. 避免对指针使用static_cast

不要对动态对象的指针使用static_cast：安全的替代方法有很多，包括使用dynamic_cast，重构，乃至重新设计。

94. 避免强制转换const

莫因恶小而为之：强制转换const有时会导致未定义的行为，即使合法，也是不良编程风格的主要表现。
C++有一个隐式的const_cast

```
char *p = "Hello World";
```

编译器将不加提示的执行const_cast，将const char*强制转换为char*。之所以允许这样，是为了操作与C语言的API兼容性，但是这是C++类型系统的一个漏洞。

95. 不要使用C风格的强制转换

年纪并不意味着智慧：C语言风格的强制转换根据上下文具有不同（而且经常很危险）的语义，而所有这些都隐藏在相同的语法背后。用C++风格的强制转换代替C风格的强制转换有助于防范意想不到的错误。

96. 不要对非POD进行memcpy操作或者memcmp操作

不要企图给对象拍X光片：不要用memcpy或memcmp来复制或比较任何对象，除非有什么对象的布局就是原始内存。

POD类型：C++标准给出的定义：将对象的各字节拷贝到一个字节数组中，然后再将它重新拷贝到原先的对象所占的存储区中，此时该对象应该具有它原来的值。

其实POD本质就是与C兼容的数据类型。可以随意进行内存级别的复制和比较的类型（memcpy,memcmp）。

所有POD类型都可以作为union的成员，反之，所有非POD类型都不能作为union的成员。

再比如：shared_ptr就不是POD类型，因为对它进行内存级别的复制后，它的计数器将是错误的。

97. 不要使用联合重新解释表示方式

偷梁换柱也是一种欺骗：通过在union中写入一个成员而度取另一个的滥用方式可以获得"无需强制转换的强制转换"。这比起reinterpret_cast更阴险，也更难预测。

98. 不要使用可变长参数（...）

省略会导致崩溃：省略号(...)是来自C语言的危险遗产。要避免使用可变长参数，应改用高级的C++结构和库。

99. 不要使用失效对象。不要使用不安全函数

不要使用失效药：失效对象和老的但是不安全的函数会对程序的健康产生极大的破坏。

失效对象主要有三种：

----已销毁对象。

----语义失效对象。

----从来都有效的对象。

100. 不要多态地处理数组

数组的可调整性很差：多态地处理数组是绝对的类型错误，而且编译器有可能不会做出任何提示。不要掉入这一陷阱。

指针可同时满足两种目的：

----一种是作为别名（对象的小标识符）

----一种是作为数组迭代器（以用指针运算遍历对象数组）

在接口中应该使用引用而不是指针。

在对指向Base的指针p进行指针运算时，编译器会将p[n]计算为*(p + n * sizeof(Base))，因此是将内

存中的对象都当作Base类型处理了--而决不会考虑可能是大小不同的派生类对象。因此如果：将标记为Derived数组的起始指针转换为Base*（不加提示的允许），然后再对此指针进行指针运算（还是不加提示允许），这样进行指针运算得到的结果一定是错误的。**绝不要对数组使用多态。**

要存储多态对象的数组，其实也非常简单，就是使用基类指针的数组（而不是具体对象的数组）。加上这么一个中间转换就完美解决问题了。