



从语法、编码习惯和编程规范、程序架构和思想3个方面深入探讨  
编写高质量C++代码的技巧、禁忌和最佳实践



李健 著

*Writing Solid C++ Code: 150 Suggestions to Improve Your C++ Programs*

# 编写高质量代码 改善C++程序的150个建议



YZLI0890119376



机械工业出版社  
China Machine Press



在程序员中，曾经有一个广为流传的段子，一位程序员抱怨：“这段代码是谁写的？非傻即呆！”结果他在代码结尾的注释中发现，这正是自己几年前的“杰作”。同样的功能，实现的代码可以千差万别，大师级的程序员可能只需要写两行代码，但程序却近乎完美，一般的程序员可能会敲数百甚至数千行代码，而且还漏洞百出。如何才能编写出高质量的代码呢？这是每个程序员所关心的问题。本书就如何编写出高质量的C++代码，从语法、编码规范和编程思想三大方面给出了大量的最佳实践，极具参考价值。强烈推荐！

—— 51CTO ([www.51cto.com](http://www.51cto.com), 中国领先的IT技术网站)

每个程序员都希望自己能编写出优质高效的代码，但真正能做到的少之又少，因为这不仅需要对技术有深入的钻研，而且需要大量经验的积累。本书作者将自己和C++领域的前辈们在大量编程实践中总结出来的经验，从语法、编码习惯和规范、程序设计思想三个方面进行了分类梳理，一共总结出了150条极具参考价值的建议。如果能将本书的内容吃透并融会贯通，不仅能让自己的编程功力大增。

—— 钱林松 资深C++技术专家，著有畅销书《C++反汇编与逆向分析技术揭秘》

C++语言以多范型见长，掌握和应用都需要下不小的功夫。然而一旦学成，就如侯捷老师曾经说过的那样有着“妙用无穷”的旨趣和力量。本书从语言、编程规范和程序设计思想三个方面对C++编程中的疑点和难点进行了归纳与分析。实例丰富，语言通俗易懂，为C++程序员巧学和妙用C++指点迷津。这表明国内的作者已经开始摆脱人云亦云的思想枷锁，而开始进行独立思考和写作的实践，这是非常难能可贵的。希望读者们能够从本书中学有所获。

—— 高博 盛大创新院技术骨干、知名译者（译有《设计原本》等多本经典著作）

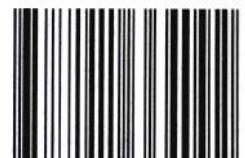
客服热线：(010) 88378991, 88361066  
购书热线：(010) 68326294, 88379649, 68995259  
投稿热线：(010) 88379604  
读者信箱：[hzjsj@hzbook.com](mailto:hzjsj@hzbook.com)

华章网站 <http://www.hzbook.com>

网上购书：[www.china-pub.com](http://www.china-pub.com)



上架指导：程序设计/C++  
ISBN 978-7-111-36409-2



9 787111 364092

定价：59.00元



*Writing Solid C++ Code: 150 Suggestions to Improve Your C++ Programs*

# 编写高质量代码 改善C++程序的150个建议

李健 著



机械工业出版社  
China Machine Press

本书是 C++ 程序员进阶修炼的必读之作，包含的全部都是 C++ 编码的最佳实践，从语法、编码规范和编程习惯、程序架构和设计思想等三大方面对 C++ 程序和设计中的疑难问题给出了经验性的解决方案，为 C++ 程序员编写更高质量的 C++ 代码提供了 150 条极为宝贵的建议。每个问题都来自于实践，都极具代表性，本书不仅以建议的方式正面为每个问题给出了被实践证明为十分优秀的解决方案，而且还从反面给出了被实践证明为不好的解决方案，从正反两个方面进行了分析和对比。

全书在逻辑上一共分为三个部分：语法部分涵盖 C++ 从 C 语言继承而来的一些极为重要但又极容易被误解和误用的一些语法特性，从 C 语言到 C++ 的改变，以及内存管理、类、模板、异常处理、STL 等方面的内容；编码习惯和编程规范部分则主要讨论了如何提高程序的正确性、可读性、程序性能和编码效率方面的问题；程序架构和思想部分则从更高的高度对 C++ 程序设计思维和方法进行了审视，给出了一些颇具价值的观点和最佳实践。

这是一本关于如何提高 C++ 程序设计效率与质量的工具书，希望书中的每条建议都能引起你的思考，对于有难度的内容，建议大家消化理解，切勿死记硬背，同时也希望大家能悟出更好的解决方案。希望本书中的每条建议所传递的思想和理念能够渗透到大家的编码实践中，进而帮助大家真正具备编写高质量 C++ 代码的能力。

**封底无防伪标均为盗版**

**版权所有，侵权必究**

**本书法律顾问 北京市展达律师事务所**

#### **图书在版编目 (CIP) 数据**

**编写高质量代码：改善 C++ 程序的 150 个建议 / 李健著 —北京：机械工业出版社，2012.1**

**ISBN 978-7-111-36409-2**

**I. 编… II. 李… III. C 语言－程序设计 IV TP312**

**中国版本图书馆 CIP 数据核字 (2011) 第 230078 号**

**机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)**

**责任编辑：杨绣国 陈佳媛**

**北京京北印刷有限公司印刷**

**2012 年 1 月第 1 版第 1 次印刷**

**186mm×240mm • 22 印张**

**标准书号：ISBN 978-7-111-36409-2**

**定价：59.00 元**

**凡购本书，如有缺页、倒页、脱页，由本社发行部调换**

**客服热线：(010) 88378991；88361066**

**购书热线：(010) 68326294；88379649；68995259**

**投稿热线：(010) 88379604**

**读者信箱：hzjsj@hzbook.com**



## 为什么要写这本书

一直以来，C++ 就是一门富有争议的编程语言。一方面它是一门复杂的语言，由于 C++ 语言提供了复杂的语法规则，被看作“拙劣工程学”的成果，甚至引来了 Linux 之父 Linus Benedict Torvalds 的炮轰，称其为“糟糕程序员的垃圾语言”。另一方面它又是一门流行的语言，正如 C++ 之父 Stroustrup 所说的，“在这 12 年里，C++ 用户数大约每 7 个半月增加一倍”。虽然有些言过其实，但确实说明了 C++ 的流行程度。在 Tiobe 编程语言热度排行榜中，C++ 稳居前三位，成绩斐然（如图 0-1 所示）。

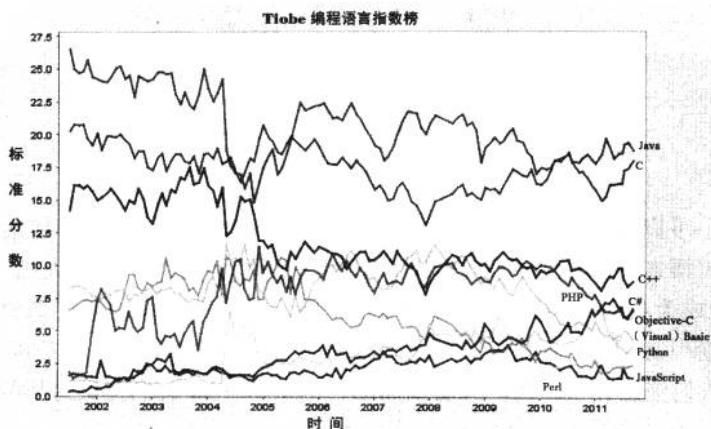


图 0-1 Tiobe 编程语言排行榜

究其原因，主要在于 C++ 不仅具有面向对象编程语言的逻辑表达优势，还具有和 C 语言不相上下的时间效率和空间效率。换言之，C++ 可以让我们用清晰的代码实现高效的程序设计，很好地体现了 KISS (Keep It Simple and Stupid) 设计之精髓。

C++ 语言的发展大概可以分为三个阶段：

第一阶段是从 20 世纪 80 年代到 1995 年。这一阶段 C++ 语言基本上是传统类型上的面向对象语言，并且凭借着接近 C 语言的运行效率，在工业界使用的开发语言中占据了相当大的份额。

第二阶段是从 1995 年到 2000 年，在这一阶段，由于标准模板库 (STL) 和后来的 Boost 等程序库的出现，使得泛型程序设计在 C++ 中占据了越来越多的比重。当然，同时由于 Java、C# 等语言的出现和硬件价格的大规模下降，C++ 也受到了一定的冲击。

第三阶段是从 2000 年至今，由于以 Loki、MPL 等程序库为代表的产生式编程和模板元编程的出现，C++ 出现了发展历史上的又一个新高峰。这些新技术的出现以及和原有技术的融合，使 C++ 成为当今主流程序设计语言中最复杂的一员。

经过多年的技术沉淀，C++ 在现代软件领域中已经占据了举足轻重的地位，特别是在系统级复杂应用程序、高性能并行计算以及对灵活性和底层操作要求较高的软件开发中占据主导地位和绝对优势。C++ 主要应用领域如图 0-2 所示。

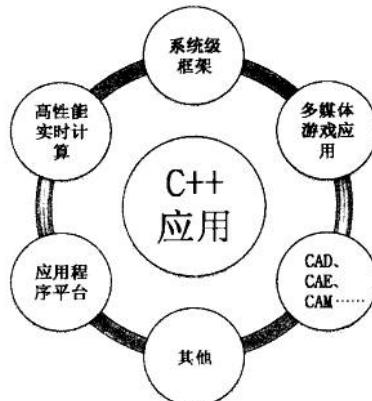


图 0-2 C++ 语言主要应用领域

难学易用，可以说是 C++ 语言的最大特点。C++ 语言具有较高的复杂度和较陡峭的学习曲线。C++ 语言的“难学”似乎成为了众多程序员面前的一座“大山”。然而这样的“大山”并不能成为我们抛弃 C++ 语言的理由，因为大山的后面是我们期望已久的美丽桃花源：C++ 难学但易用，正如侯捷老师所言：“一旦学成，妙用无穷。”在艰苦的登山途中，一根拐杖会令我们倍感轻松，为广大聪明而刻苦的 C++ 程序员们提供这样的一根拐杖便是本书的目的。

对于那些刚刚进入 C++ 世界的程序员来说，别人的经验教训就是他的前车之鉴，能够帮助他快速、准确地掌握 C++ 的要点。为此，我将自己和前辈们的“工程经验”进行总结、归纳、升华，最后整理成 150 条意见和建议，以期帮助大家更好地理解、运用 C++ 语言。

## 读者对象

本书适用于具有一定 C/C++ 语言基础的程序员。书中的 150 条建议能够帮助读者更加深入地理解 C++ 语言，用好 C++ 语言，提升 C++ 程序设计的整体品质。

## 如何阅读本书

作为一个 C++ 从业者，我十分了解初学者在学习过程中可能遇到的困惑，清楚那些“工程经验的积累”对一个初学者成长的显著作用。所以，本书采用了《Effective C++》经典的“条款式”叙述方式，对本人以及众多 C++ 前辈们的经验进行了重新的编撰整理，总结成了 150 条编程建议。

本书所包括的 150 条编程建议，可以说是散布在各个角落的语言规则、编程准则，以及最佳实践的汇总；这 150 条建议将分为三大部分：语法篇、编码习惯和规范篇、程序架构和思想篇。这三大部分从语言语法，到编码习惯，再到架构思想，由浅入深，层层递进，使读者逐渐认识并理解这门语言。

其中，语法篇主要围绕语法展开，分为从 C 继承而来的、从 C 到 C++ 的改变、内存管理、类、模板、异常处理、STL 七章；编码习惯和规范篇则集中在“习惯”二字上，建议主要集中在如何提高程序的正确性、可读性、效率等方面；而最后的程序架构和思想篇，则站在更高的高度去审视程序设计，给出一些编程规范和最佳实践。

我希望本书中的每条建议都能引起读者的思考，鼓励消化理解，杜绝死记硬背，取其精华去其糟粕；我希望本书中的每条建议所传递的理念能够渗透到读者的设计实践中，而不是邯郸学步式地模仿，使这些建议成为编写程序的束缚。

## 勘误和支持

由于作者水平有限，编写时间仓促，书中难免会出现一些错误、疏漏或者不妥之处，恳请读者批评指正。你可以将书中的错误通过邮件告知我，也可以将你遇到的其他问题通过邮件发给我，我将尽力提供最满意的解答。我的邮箱是 [lijian8409@gmail.com](mailto:lijian8409@gmail.com)，真诚期待大家

的反馈。

## 致谢

感谢伟大的 Bjarne Stroustrup 博士，是他发明了 C++ 这样一门影响深远的语言；感谢为 C++ 语言做出重大贡献的前辈们，是你们的努力让 C++ 语言不断向前发展。

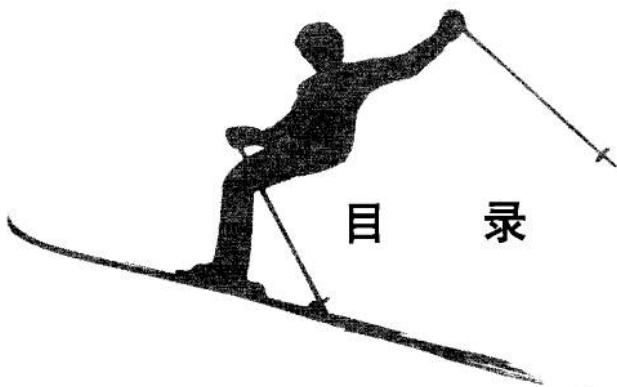
感谢机械工业出版社华章公司的杨福川编辑和杨绣国编辑，是你们一直以来的支持与帮助促进了这本书的顺利出版。

感谢远在家乡的父母，你们二十余年如一日的关爱和鼓励，是我面对困难和挫折时重新振作的力量源泉，让我不敢懈怠，不言放弃；感谢仍在继续学业的弟弟，希望你能顺利拿到博士学位，实现你的理想与人生价值；感谢亲爱的老婆的默默陪伴和支持，你是我前进的最大动力。对你们的爱永存心间。

谨以此书，献给我深爱的家人，以及众多热爱并奋斗在 C++ 第一线的朋友们。

李 健

2011 年 10 月于北京



## 前 言

# 第一部分 语 法 篇

## 第 1 章 从 C 继承而来的 /2

- 建议 0：不要让 main 函数返回 void /2
- 建议 1：区分 0 的 4 种面孔 /5
- 建议 2：避免那些由运算符引发的混乱 /8
- 建议 3：对表达式计算顺序不要想当然 /9
- 建议 4：小心宏 #define 使用中的陷阱 /12
- 建议 5：不要忘记指针变量的初始化 /14
- 建议 6：明晰逗号分隔表达式的奇怪之处 /15
- 建议 7：时刻提防内存溢出 /16
- 建议 8：拒绝晦涩难懂的函数指针 /19
- 建议 9：防止重复包含头文件 /19
- 建议 10：优化结构体中元素的布局 /21
- 建议 11：将强制转型减到最少 /23
- 建议 12：优先使用前缀操作符 /26
- 建议 13：掌握变量定义的位置与时机 /28
- 建议 14：小心 typedef 使用中的陷阱 /30

建议 15：尽量不要使用可变参数 /32

建议 16：慎用 goto /36

建议 17：提防隐式转换带来的麻烦 /38

建议 18：正确区分 void 与 void\*/42

## 第 2 章 从 C 到 C++，需要做出一些改变 /45

建议 19：明白在 C++ 中如何使用 C /45

建议 20：使用 memcpy() 系列函数时要足够小心 /48

建议 21：尽量用 new/delete 替代 malloc/free/49

建议 22：灵活地使用不同风格的注释 /52

建议 23：尽量使用 C++ 标准的 iostream/55

建议 24：尽量采用 C++ 风格的强制转型 /58

建议 25：尽量用 const、enum、inline 替换 #define/59

建议 26：用引用代替指针 /62

## 第 3 章 说一说“内存管理”的那点事儿 /66

建议 27：区分内存分配的方式 /67

建议 28：new/delete 与 new[]/delete[] 必须配对使用 /69

建议 29：区分 new 的三种形态 /71

建议 30：new 内存失败后的正确处理 /75

建议 31：了解 new\_handler 的所作所为 /78

建议 32：借助工具监测内存泄漏问题 /81

建议 33：小心翼翼地重载 operator new/ operator delete /84

建议 34：用智能指针管理通过 new 创建的对象 /88

建议 35：使用内存池技术提高内存申请效率与性能 /91

## 第 4 章 重中之重的类 /95

建议 36：明晰 class 与 struct 之间的区别 /95

建议 37：了解 C++ 悄悄做的那些事 /99

建议 38：首选初始化列表实现类成员的初始化 /101

建议 39：明智地拒绝对象的复制操作 /105

建议 40：小心，自定义拷贝函数 /107

建议 41：谨防因构造函数抛出异常而引发的问题 /110

- 建议 42：多态基类的析构函数应该为虚 /113  
建议 43：绝不让构造函数为虚 /116  
建议 44：避免在构造 / 析构函数中调用虚函数 /117  
建议 45：默认参数在构造函数中给你带来的喜与悲 /120  
建议 46：区分 Overloading、Overriding 及 Hiding 之间的差异 /122  
建议 47：重载 operator= 的标准三步走 /126  
建议 48：运算符重载，是成员函数还是友元函数 /131  
建议 49：有些运算符应该成对实现 /134  
建议 50：特殊的自增自减运算符重载 /136  
建议 51：不要重载 operator&&、operator|| 以及 operator, /137  
建议 52：合理地使用 inline 函数来提高效率 /139  
建议 53：慎用私有继承 /141  
建议 54：抵制 MI 的糖衣炮弹 /144  
建议 55：提防对象切片 /147  
建议 56：在正确的场合使用恰当的特性 /150  
建议 57：将数据成员声明为 private /154  
建议 58：明晰对象构造与析构的顺序 /156  
建议 59：明了如何在主调函数启动前调用函数 /158

## 第 5 章 用好模板，向着 GP 开进 /161

- 建议 60：审慎地在动、静多态之间选择 /161  
建议 61：将模板的声明和定义放置在同一个头文件里 /164  
建议 62：用模板替代参数化的宏函数 /168  
建议 63：区分函数模板与模板函数、类模板与模板类 /169  
建议 64：区分继承与模板 /171

## 第 6 章 让神秘的异常处理不再神秘 /176

- 建议 65：使用 exception 来处理错误 /176  
建议 66：传值 throw 异常，传引用 catch 异常 /179  
建议 67：用“throw;”来重新抛出异常 /183  
建议 68：了解异常捕获与函数参数传递之间的差异 /185  
建议 69：熟悉异常处理的代价 /189  
建议 70：尽量保证异常安全 /192

## 第 7 章 用好 STL 这个大轮子 /198

- 建议 71: 尽量熟悉 C++ 标准库 /198
- 建议 72: 熟悉 STL 中的有关术语 /201
- 建议 73: 删除指针的容器时避免资源泄漏 /204
- 建议 74: 选择合适的 STL 容器 /206
- 建议 75: 不要在 STL 容器中存储 auto\_ptr 对象 /209
- 建议 76: 熟悉删除 STL 容器中元素的惯用法 /210
- 建议 77: 小心迭代器的失效 /213
- 建议 78: 尽量使用 vector 和 string 替代动态分配数组 /214
- 建议 79: 掌握 vector 和 string 与 C 语言 API 的通信方式 /216
- 建议 80: 多用算法调用, 少用手写循环 /217

## 第二部分 编码习惯和规范篇

### 第 8 章 让程序正确执行 /222

- 建议 81: 避免无意中的内部数据裸露 /222
- 建议 82: 积极使用 const 为函数保驾护航 /224
- 建议 83: 不要返回局部变量的引用 /228
- 建议 84: 切忌过度使用传引用代替传对象 /230
- 建议 85: 了解指针参数传递内存中的玄机 /231
- 建议 86: 不要将函数参数作为工作变量 /233
- 建议 87: 躲过 0 值比较的层层陷阱 /234
- 建议 88: 不要用 reinterpret\_cast 去迷惑编译器 /236
- 建议 89: 避免对动态对象指针使用 static\_cast /237
- 建议 90: 尽量少应用多态性数组 /238
- 建议 91: 不要强制去除变量的 const 属性 /240

### 第 9 章 提高代码的可读性 /242

- 建议 92: 尽量使代码版面整洁优雅 /243
- 建议 93: 给函数和变量起一个“能说话”的名字 /246
- 建议 94: 合理地添加注释 /248
- 建议 95: 为源代码设置一定的目录结构 /251
- 建议 96: 用有意义的标识代替 Magic Numbers /252

- 建议 97：避免使用“聪明的技巧” /253
- 建议 98：运算符重载时坚持其通用的含义 /254
- 建议 99：避免嵌套过深与函数过长 /255
- 建议 100：养成好习惯，从现在做起 /256

## 第 10 章 让代码运行得再快些 /258

- 建议 101：用移位实现乘除法运算 /258
- 建议 102：优化循环，提高效率 /259
- 建议 103：改造 switch 语句 /260
- 建议 104：精简函数参数 /261
- 建议 105：谨慎使用内嵌汇编 /262
- 建议 106：努力减少内存碎片 /263
- 建议 107：正确地使用内联函数 /263
- 建议 108：用初始化取代赋值 /264
- 建议 109：尽可能地减少临时对象 /266
- 建议 110：最后再去优化代码 /267

## 第 11 章 零碎但重要的其他建议 /269

- 建议 111：采用相对路径包含头文件 /269
- 建议 112：让条件编译为开发出力 /270
- 建议 113：使用 .inl 文件让代码整洁可读 /272
- 建议 114：使用断言来发现软件问题 /274
- 建议 115：优先选择编译和链接错误 /275
- 建议 116：不放过任何一条编译器警告 /277
- 建议 117：尽量减少文件之间的编译依赖 /278
- 建议 118：不要在头文件中使用 using /280
- 建议 119：划分全局名空间避免名污染 /282

# 第三部分 程序架构和思想篇

## 第 12 章 面向对象的类设计 /286

- 建议 120：坚持“以行为为中心”的类设计 /286
- 建议 121：用心做好类设计 /287

- 建议 122: 以指针代替嵌入对象或引用 /289
- 建议 123: 努力将接口最小化且功能完善 /291
- 建议 124: 让类的数据隐藏起来 /292
- 建议 125: 不要让成员函数破坏类的封装性 /294
- 建议 126: 理解“virtual + 访问限定符”的深层含义 /295
- 建议 127: 谨慎恰当地使用友元机制 /297
- 建议 128: 控制对象的创建方式 /299
- 建议 129: 控制实例化对象的个数 /301
- 建议 130: 区分继承与组合 /303
- 建议 131: 不要将对象的继承关系扩展至对象容器 /307
- 建议 132: 杜绝不良继承 /308
- 建议 133: 将 RAII 作为一种习惯 /310
- 建议 134: 学习使用设计模式 /311
- 建议 135: 在接口继承和实现继承中做谨慎选择 /314
- 建议 136: 遵循类设计的五项基本原则 /315

## 第 13 章 返璞归真的程序设计 /318

- 建议 137: 用表驱动取代冗长的逻辑选择 /318
- 建议 138: 为应用设定特性集 /324
- 建议 139: 编码之前需三思 /324
- 建议 140: 重构代码 /326
- 建议 141: 透过表面的语法挖掘背后的语义 /328
- 建议 142: 在未来时态下开发 C++ 程序 /330
- 建议 143: 根据你的目的决定造不造轮子 /331
- 建议 144: 谨慎在 OO 与 GP 之间选择 /331
- 建议 145: 让内存管理理念与时俱进 /332
- 建议 146: 从大师的代码中学习编程思想与技艺 /334
- 建议 147: 遵循自然而然的 C++ 风格 /335
- 建议 148: 了解 C++ 语言的设计目标与原则 /335
- 建议 149: 明确选择 C++ 的理由 /338



# 第一部分

# 语 法 篇

## 本部分内容

- 第 1 章 从 C 继承而来的
- 第 2 章 从 C 到 C++, 需要做出一些改变
- 第 3 章 说一说“内存管理”的那点事儿
- 第 4 章 重中之重的类
- 第 5 章 用好模板, 向着 GP 开进
- 第 6 章 让神秘的异常处理不再神秘
- 第 7 章 用好 STL 这个大轮子

# 第 1 章 从 C 继承而来的

C 和 C++ 可以说是所有编程语言中关系最为紧密的两个。在目标上，C++ 被定位为“a better C”；在名称上，C++ 有一个乳名叫做“C with classes”；在语法上，C 更是 C++ 的一个子集，C++ 几乎支持 C 语言的全部功能。如果采用 C++ 的方法来描述，以下方式恰如其分：

```
class C{};  
class CPlusPlus : public C {};
```

C++ 继承自 C。

正是这种难以割舍的紧密联系使得 C/C++ 程序员必须对 C 有所重视。所以，本章就从 C++ 的前身——C 语言说起。

在开始这段学习旅程前，先分享一个只有程序员才明白的幽默：

有一次，她开玩笑似地问他：“我在你心里排第几？”他回头微笑着摸了摸她的头，用手指比划了个鸡蛋。她知道他在开玩笑，打了他一巴掌，尽管有些郁闷，但还是尽量避免流露出失望的神色。其实，因为她是文科生，所以她并不知道：在程序员的眼中，所有的数组、列表、容器的下标都是从 0 开始的。

所以，我们的建议也从 0 开始。

## 建议 0：不要让 main 函数返回 void

同 C 程序一样，每个 C++ 程序都包含一个或多个函数，而且必须有一个函数命名为 main，并且每个函数都由具有一定功能的语句序列组成。操作系统将 main 作为程序入口，调用 main 函数来执行程序；main 函数执行其语句序列，并返回一个值给操作系统。在大多数系统中，main 函数的返回值用于说明程序的退出状态。如果返回 0，则代表 main 函数成功执行完毕，程序正常退出，否则代表程序异常退出。

然而在编写 C++ 程序入口函数 main 的时候，很多程序员，特别是一些具有 C 基础的 C++ 程序员时经常会写出如下格式的 main 函数：

```
void main()  
{  
    // some code ...  
}
```

上述代码在VC++中是可以正确编译、链接、执行的。编译信息如下所示：

```
1>----- 已启动生成：项目：MainCpp，配置：Debug Win32 -----
1> main.cpp
1> MainCpp.vcxproj -> G:\MainCpp\Debug\MainCpp.exe
===== 生成：成功 1 个，失败 0 个，最新 0 个，跳过 0 个 =====
```

但是当你将代码放在Linux环境下，采用GCC编译器进行编译时，你会吃惊地发现编译器抛出了如下的错误信息：

```
[develop@localhost ~] g++ main.cpp
main.cpp:2: 错误：'::main' 必须返回 'int'
```

为什么同样的代码会出现两种不同的结果呢？这还是跨平台的C/C++语言吗？不要对C/C++的跨平台性产生质疑，之所以会这样，很大程度上要归结于市面上一些书的“误导”，以及微软对VC++编译器main返回值问题的过分纵容。

在C和C++中，不接收任何参数也不返回任何信息的函数原型为“void f(void);”。所以很多人认为，不需要程序返回值时可以把main函数定义成void main(void)，然而这种想法是非常错误的！

有一点你必须明确：在C/C++标准中从来没有定义过void main()这样的代码形式。C++之父Bjarne Stroustrup在他的主页FAQ中明确地写着这样一句话：

在C++中绝对没有出现过void main()/\* ... \*/这样的函数定义，在C语言中也是。

main函数的返回值应该定义为int类型，在C和C++标准中都是这样规定的。在C99标准中规定，只有以下两种定义方式是正确的<sup>①</sup>：

```
int main( void )
int main( int argc, char *argv[] )
```

在C++03中也给出了如下两种main函数的定义方式<sup>②</sup>：

```
int main()
int main( int argc, char *argv[] )
```

虽然在C和C++标准中并不支持void main()，但在部分编译器中void main()依旧是可以通过编译并执行的，比如微软的VC++。由于微软产品的市场占有率与影响力很大，因此在某种程度上加剧了这种不良习惯的蔓延。不过，并非所有的编译器都支持void main()，gcc就站在了VC++的对立面，它是这一不良习气的坚定抵制者，它会在编译时就明确地给出一个错误。

如果你坚持在某些编译器中使用void main()这种非标准形式的代码，那么当你把程序从

<sup>①</sup> 参考资料：ISO/IEC 9899:1999 (E) Programming languages—C 5.1.2.2.1 Program startup。

<sup>②</sup> 参考资料：ISO C++03 3.6.1 Main function。

一个编译器移植到另一个编译器时，你就要对可能出现的错误负责。

除了有 void main() 这样的不规范格式外，在 C 语言程序中，尤其是一些老版本的 C 代码中，你还会经常看到 main() 这样的代码形式。

一些老的 C 标准（诸如 C90）是支持 main() 这样的形式的。之所以支持，是因为在第一版的 C 语言中只有 int 一种数据类型，并不存在 char、long、float、double 等这些内置数据类型。既然只有 int 一种类型，也就不必显式地为 main 函数标明返回类型了。在 Brian W. Kernighan 和 Dennis M. Ritchie 的经典巨著《The C Programming Language, Second Edition》<sup>Θ</sup> 中用的就是 main()。后来，在 C 语言的改进版中数据类型得到了扩充，为了能兼容以前的代码，标准委员会就做出了如下规定：不明确标明返回值的，默认返回值为 int。在 C99 标准中，则要求编译器对于 main() 这种用法至少要抛出一个警告。

main 函数返回值的作用，可以采用下面的方法加以验证。

首先，编写 main.cpp 文件，文件内容如下所示：

```
int main()
{
    return 0;
}
```

在 Linux 环境下，采用命令：

```
g++ main.cpp
```

生成可执行文件 a.out。然后，执行命令：

```
./a.out && echo "success"
```

结果输出 success。

修改上述程序：

```
int main()
{
    return -1;
}
```

做同样测试，无输出。

命令 A && B 中的 && 类似于 C++ 中的并操作 (&&)，如果 A 命令正确执行，接着就会执行命令 B；如果 A 出现异常，则 B 不执行。通过以上分析可知，当 main() 返回 0 时，a.out 正确执行并返回；但是如果返回 -1，程序就不能正常返回了。

最后，还得说明一下 C++ 标准中一个“好坏难定”的规定：

在 main 函数中，return 语句的作用在于离开 main 函数（析构掉所有具有动态生存时间

<sup>Θ</sup> 本书已由机械工业出版社引进出版，中文书名为《C 程序设计语言（第 2 版·新版）》（ISBN 7-111-12806-0）和《C 程序设计语言（英文版·第 2 版）》（ISBN 7-111-19626-0）。——编辑注

的对象), 并将其返回值作为参数来调用 exit 函数。如果函数执行到结尾而没有遇到 return 语句, 其效果等同于执行了 return 0。<sup>Θ</sup>

也就是说, 如果函数执行到 main 结束处时没有遇到 return 语句, 编译器会隐式地为你加上 return 0;, 效果与返回 0 相同。之所以说这条规定“好坏难定”, 一方面是因为它让你省去了多敲几个字的麻烦; 另一方面是因为这种便捷会让某些程序员忽视编译器代替他做的工作, 而在思维中形成一种错误的认识: 此函数可以无返回。

在应用这一规则时, 你还得注意以下这两点:

- main 函数的返回类型是 int, 不是 void 或其他类型。
- 该规则仅仅对 main 函数适用。

按照以上标准得到了一个完全合乎 C/C++ 标准的最小化的完整 C++ 程序:

```
int main() { }
```

本人不推荐使用上述这条规则, 建议加上 return 0;, 杜绝那些不必要的误解。

**请记住:**

要想保证程序具有良好的可移植性能, 就要标明 main 函数返回 int, 而不是 void。强烈建议使用以下形式:

```
int main()
{
    // some processing codes
    return 0;
}
```

## 建议 1: 区分 0 的 4 种面孔

0 在 C/C++ 语言中绝对是一个多面手, 它扮演着多样的角色, 拥有着多种面孔。总结起来包括以下几种角色: 整型 0、空指针 NULL、字符串结束标志 '\0'、逻辑 FALSE/false, 不同的角色适用于不同的情形, 下面我们按照上述顺序一一介绍。

### □ 整型 0

这是我们最熟悉的一个角色。作为一个 int 类型, 整型 0 占据 32 位的空间, 其二进制表示为:

<sup>Θ</sup> A return statement in main has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling exit with the return value as the argument. If control reaches the end of main without encountering a return statement, the effect is that of executing return 0.—ISO C++03 3.6.1 Main function.

00000000 00000000 00000000 00000000

它的使用方式最为简单直接，未经修饰，如下所示：

```
int nNum = 0; // 赋值
if( nNum == 0 ) // 比较
```

#### □ 空指针 NULL

NULL 是一个表示空指针常量的宏，在 C/C++ 标准中有如下阐述：

在文件 `<locale>`、`<cstddef>`、`<stdio>`、`<stdlib>`、`<string>`、`<time>` 或者 `<cwchar>` 中定义的 NULL 宏，在国际标准中被认为是 C++ 空指针常量。<sup>Θ</sup>

指针与 int 类型所占空间是一样的，都是 32 位。那么，空指针 NULL 与 0 又有什么区别呢？还是让我们看一下 `windef.h` 中 NULL 的定义吧：

```
#ifndef NULL
#define __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
#endif
```

通过定义我们可以看出，它们之间其实是没有太大的区别，只不过在 C 语言中定义 NULL 时会进行一次强制转型。我想之所以创造出 NULL，大概是为了增强代码的可读性，但这只是我的臆测，无从考究。

需要注意的是，这里的 0 与整型的 0 还是存在区别的。例如，`int* pValue = 0;` 是合法的，而 `int* pValue = 1;` 则是不合法的。这是因为 0 可以用来表示地址，但常数 1 绝对不行。

作为指针类型时，推荐按照下面的方式使用 0：

```
float* pNum = NULL; // 赋值
if( pNum == NULL ) // 比较
```

#### □ 字符串结束标志 '\0'

'\0' 与上述两种情形有所不同，它是一个字符。作为字符，它仅仅占 8 位，其二进制表示为：

00000000

因为字符类型中并没有与 0000 0000 对应的字符，所以就创造出了这么一个特殊字符。（对于类似 '\0' 这样的特殊字符，我们称之为转义字符。）在 C/C++ 中，'\0' 被作为字符串结束标志来使用，具有唯一性，与 '0' 是有区别的。

---

<sup>Θ</sup> The macro NULL, defined in any of `<locale>`, `<cstddef>`, `<stdio>`, `<stdlib>`, `<string>`, `<time>`, or `<cwchar>`, is an implementation-defined C++ null pointer constant in this International Standard.—— ISO C++03 C.2.2.3 Macro NULL

作为字符串结束符，0 的使用有些特殊。不必显式地为字符串赋值，但是必须明确字符串的大小。例如，在下面的代码中，“Hello C/C++”只有 11 个字符，却要分配 12 个字符的空间。

```
char sHello[12] = {"Hello C/C++"}; // 赋值
if( sHello[11] == '\0' ) // 比较
```

#### □ 逻辑 FALSE/false

虽然将 FALSE/false 放在了一起，但是你必须清楚 FALSE 和 false 之间不只是大小写这么简单的差别。false/true 是标准 C++ 语言里新增的关键字，而 FALSE/TRUE 是通过 #define 定义的宏，用来解决程序在 C 与 C++ 环境中的差异。以下是 FALSE/TRUE 在 windef.h 中的定义：

```
#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif
```

换言之，FALSE/TRUE 是 int 类型，而 false/true 是 bool 类型，两者是不一样的，只不过 C++ 帮我们完成了相关的隐式转换，以至于我们在使用中没有任何感觉。bool 在 C++ 里占用的是 1 个字节，所以 false 也只占用 1 个字节。

其二进制表示如下：

```
false -> 0
FALSE -> 00000000 00000000 00000000 00000000
```

在 C++ 中，推荐使用 bool 类型的 false/true，其使用方式如下：

```
bool isReady = false; // 赋值
if( isReady ) // 判断
```

如果不细心，0 的多重性可能会让程序产生一些难以发现的 Bug，比如：

```
// 把 pSrc 指向的源字符串复制到 pDes 指向的内存块
while(pSrc)
{
    * pDes ++ = * pSrc ++;
}
```

正常情况下，当 pSrc 指向的字符为字符串结束符 '\0' 时，while 循环终止；但不幸的是，这里的条件写错了，while 终止条件变成了 pSrc 指向地址 0。结果 while 循环写入到内存中了，直至程序崩溃。

正确的写法应该是：

```
// 把 pSrc 指向的源字符串复制到 pDes 指向的内存块中
while(*pSrc)
{
```

```
* pDes ++ = * pSrc ++;
}
```

**请记住：**

由于 0 存在多种面孔，容易让不细心的程序员产生混乱。唯一的解决办法就是在使用 0 的时候小心一点，再小心一点。

---

**建议 2：避免那些由运算符引发的混乱**

一般，C++ 被认为是 C 的超集。C++ 确实从它的前辈 C 那里继承了很多东西，比如一套含义相当混乱模糊的运算符。由于 C/C++ 语法规则的灵活性，以致那些粗心的程序员常会使用错误的运算符，进而引发不必要的麻烦。下面的代码就是一个典型的例子：

```
if(nValue = 0)
{
    // do something if nValue is not zero.
}
```

显然，程序员的本意是要写 `if( nValue == 0 )`。不幸的是，上述语句虽未达成程序员的本意，但它却完全是合法的，编译器不会给出任何错误提示。C++ 语句首先会将 `nValue` 赋值为 0，然后再判断 `nValue` 是否为非零。结果就是 `if` 条件始终不能被满足，大括号中的代码永远不会被执行。

针对 = 和 == 之间的问题，通过良好的代码习惯可以避免，代码如下所示：

```
if(0 == nValue)
{
    // do something if nValue is not zero.
}
```

换句话说，就是将 0 和 `nValue` 的位置交换。此时，如果你再写出 `if( 0 = nValue)` 这样的代码，编译器会直截了当地提示，发生了错误，编译失败。原因在于 `0 = nValue` 这样的代码在 C++ 语法中是不允许的，常数 0 不能作为左值来使用。

除了上述运算符，其他几对容易弄错的运算符是 &（按位与）和 &&（与），以及 |（按位或）和 ||（或）。对于这两对运算符，能够避免错误的只有细心。

**请记住：**

不要混淆 = 和 ==、& 和 &&、| 与 || 这三对运算符之间的差别，用细心和良好的代码习惯避免由于运算符混乱带来的麻烦。

---

## 建议3：对表达式计算顺序不要想当然

一条一条的表达式构成了C/C++代码的主体。接下来我们就来说说表达式的计算顺序。这些都是很琐碎的事情，但不可否认却又是非常有价值的。也许你会觉得下面的代码片段很熟悉：

```
if( nGrade & MASK == GRADE_ONE )
    ... // processing codes
```

很明显，当grade等于GRADE\_ONE时if条件成立才是程序员的本意。可是上面的代码并没有正确地表达程序员的意思。这是因为位运算符（&和|）的优先级低于关系运算符（比如==、<、>），所以上述代码的真实效果是：

```
if( nGrade & (MASK == GRADE_ONE) )
    ... // processing codes
```

这是很多人都容易犯的错误，我也有类似的经历，想当然地认为程序会按照设想的顺序来执行。这样的错误是很难发现的，调试起来也相当的费劲。C++/C语言的运算符多达数十个，而这数十个运算符又具有不同的优先级与结合律，熟记它们确实比较困难，不过，可以用括号把意图表示得更清楚，所以不要吝啬使用括号，即使有时并不必要：

```
if( (nGrade & MASK) == GRADE_ONE )
    ... // processing codes
```

这样，代码就没有了歧义。

C/C++语言中存在着“相当险恶”的优先级问题，很多人很容易在这方面犯错误。如果代码表达式中包含较多的运算符，为了防止产生歧义并提高可读性，那么可以用括号确定表达式中每一个子表达式的计算顺序，不要过分自信地认为自己已经熟悉了所有运算符的优先级、结合律，多写几个括号确实是个好主意。例如：

```
COLOR rgb = (red<<16) | (green<<8) | blue;
bool isGradeOne = ((nGrade & MASK) == GRADE_ONE);
```

上面所说的计算顺序其实就是运算符的优先级，它只是一个“开胃菜”。接下来要说的是最为诡异的表达式评估求值的顺序问题。

因为C++与C语言之间“剪不断理还乱”的特殊关系，C语言中的好多问题也被带入到C++的世界里了，包括表达式评估求值顺序问题。在C语言诞生之初，处理器寄存器是一种异常宝贵的资源；而复杂的表达式对寄存器的要求很高，这使得编译器承受着很大的压力。为了能够使编译器生成高度优化的可执行代码，C语言创造者们就赋予了寄存器分配器这种额外的能力，使得它在表达式如何评估求值的问题上留有很大的处理余地。虽然当今寄存器有了极大的进步，对复杂表达式的求值不再有什么压力，但是赋予寄存器分配器的这种能力

却一直没有收回，所以在 C++ 中评估求值顺序的不确定性仍然存在，而且这很大程度上决定于你所使用的编译器。这就要求软件工程师更加认真仔细，以防对表达式设定了无依据、先入为主的主观评估顺序。

这其实也是 C 语言的陷阱之一，《The C Programming Language》（程序员亲切地称此书为“K & R”）中反复强调，函数参数也好，某个操作符中的操作数也罢，表达式求值次序是不一定的，每个特定机器、操作系统、编译器也都不一样。就像《The C Programming Language》影印版第 2 版的 52 页所说的那样：

如同大多数语言一样，C 语言也不能识别操作符的哪一个操作数先被计算（&&、||、?: 和，四种操作符除外），例如 `x=f()+g()`。<sup>Θ</sup>

这里所说的求值顺序主要包括以下两个方面：

#### □ 函数参数的评估求值顺序

分析下面代码片段的输出结果：

```
int i = 2010;
printf("The results are: %d %d", i, i+1);
```

函数参数的评估求值并没有固定的顺序，所以，`printf()` 函数的输出结果可能是 2010、2011，也可能是 2011、2011。

类似的还有：

```
printf("The results are: %d %d", p(), q());
```

`p()` 和 `q()` 到底谁先被调用，这是一个只有编译器才知道的问题。

为了避免这一问题的发生，有经验的工程师会保证凡是在参数表中出现过一次以上的变量，在传递时不改变其值。即使如此也并非万无一失，如果不是足够小心，错误的引用同样会使努力前功尽弃，如下所示：

```
int para = 10;
int &rPara = para;
int f(int, int);
int result = f(para, rPara *= 2);
```

推荐的形式应该是：

```
int i = 2010;
printf("The results are: %d %d", i, i+1);

int para = p();
printf("The results are: %d %d", para, q());
```

<sup>Θ</sup> (C, like most languages, does not specify the order in which the operands of an operator are evaluated. (exceptions are && || ?: and ",") for example `x = f() + g();`) —— K&R Second Edition P52

```
int para = 10;
int f(int, int);
int result = f(para, para*2);
```

#### □ 操作数的评估求值顺序

操作数的评估求值顺序也不固定，如下面的代码所示：

```
a = p() + q() * r();
```

三个函数p()、q()和r()可能以6种顺序中的任何一种被评估求值。乘法运算符的高优先级只能保证q()和r()的返回值首先相乘，然后再加到p()的返回值上。所以，就算加上再多的括号依旧不能解决问题。

幸运的是，使用显式的、手工指定的中间变量可以解决这一问题，从而保证固定的子表达式评估求值顺序：

```
int para1 = p();
int para2 = q();
a = para1 + para2 * r();
```

这样，上述代码就为p()、q()和r()三个函数指定了唯一的计算顺序：p() → q() → r()。

另外，有一些运算符自诞生之日起便有了明确的操作数评估顺序，有着与众不同的可靠性。例如下面的表达式：

```
(a < b) && (c < d)
```

C/C++语言规定， $a < b$ 首先被求值，如果 $a < b$ 成立， $c < d$ 则紧接着被求值，以计算整个表达式的值。但如果 $a$ 大于或等于 $b$ ，则 $c < d$ 根本不会被求值。类似的还有 $\|$ 。这两个运算符的短路算法特性可以让我们有机会以一种简约的、符合习惯用法的方式表达出很复杂的条件逻辑。

三目条件运算符?:也起到了把参数的评估求值次序固定下来的作用：

```
expr1 ? expr2 : expr3
```

第一个表达式会首先被评估求值，然后第二个和第三个表达式中的一个会被选中并评估求值，被选中并评估求值的表达式所求得的结果就会作为整个条件表达式的值。

此外，在建议6中将会详细介绍的逗号运算符也有固定的评估求值顺序。

#### 请记住：

表达式计算顺序是一个很繁琐但是很有必要的话题：

- 针对操作符优先级，建议多写几个括号，把你的意图表达得更清晰。
- 注意函数参数和操作数的评估求值顺序问题，小心其陷阱，让你的表达式不要依赖计算顺序。

## 建议 4：小心宏 #define 使用中的陷阱

C 语言宏因为缺少必要的类型检查，通常被 C++ 程序员认为是“万恶之首”，但就像硬币的两面一样，任何事物都是利与弊的矛盾混合体，宏也不例外。宏的强大作用在于在编译期自动地为我们产生代码。如果说模板可以通过类型替换来为我们产生类型层面上多样的代码，那么宏就可以通过符号替换在符号层面上产生的多样代码。正确合理地使用宏，可以有效地提高代码的可读性，减少代码的维护成本。

不过，宏的使用中存在着诸多的陷阱，如果不注意，宏就有可能真的变成 C++ 代码的“万恶之首”。

### (1) 用宏定义表达式时，要使用完备的括号。

由于宏只是简单的字符替换，宏的参数如果是复合结构，那么替换之后要是不用括号保护各个宏参数，可能会由于各个参数之间的操作符优先级高于单个参数内部各部分之间相互作用的操作符优先级，而产生意想不到的情形。但并不是使用了括号就一定能避免出错，我们需要完备的括号去完备地保护宏参数。

如下代码片段所定义的宏要实现参数 a 和参数 b 的求和，但是这三种定义都存在一定风险：

```
#define ADD( a, b ) a + b
#define ADD( a, b ) (a + b)
#define ADD( a, b ) (a) + (b)
```

例如， $\text{ADD}(a,b) * \text{ADD}(c,d)$  的本意是对  $(a+b)*(c+d)$  求值，在采用了上面定义的宏之后，代码展开却变成了如下形式，其中只有第 2 种方式“碰巧”实现了原本意图：

```
a + b * c + d
(a + b) * (c + d)
(a) + (b) * (c) + (d)
```

之所以说“碰巧”，是因为第 2 种方式中括号的使用也非完备的。例如：

```
#define MULTIPLE( a, b ) (a * b)
```

在计算  $(a+b) \times c$  时，如果采用上述宏  $\text{MULTIPLE}(a+b, c)$ ，代码展开后，我们得到的却是  $a+b \times c$  的结果。

要避免这些问题，要做的就是：用完备的括号完备地保护各个宏参数。正确的定义应为：

```
#define ADD( a, b ) ((a)+(b))
#define MULTIPLE( a, b ) ((a)*(b))
```

### (2) 使用宏时，不允许参数发生变化。

宏参数始终是一个比较敏感、容易引发错误的东西。有很多人认为，在某种程度上带参的宏定义与函数有几分类似。但是必须注意它们的区别，正如下面代码片段所示：

```
#define SQUARE( a ) ((a) * (a))
int Square(int a)
{
    return a*a;
}

int nValue1 = 10, nValue2 = 10;
int nSquare1 = SQUARE(nValue1++); // nSquare1=110, nValue1=12
int nSquare2 = Square(nValue2++); // nSquare2=100, nValue2=11
```

类似的定义，却产生了不同的结果，究其原因还是宏的字符替换问题。正如上面的示例一样，两处的a都被参数nValue1++替换了，所以nValue1自增操作也就被执行了两回。

这就是宏在展开时对其参数的多次取值替换所带来的副作用。为了避免出现这样的副作用，最简单有效的方法就是保证宏参数不发生变化，如下所示。

```
#define SQUARE( a ) ((a) * (a))

int nValue1 = 10;
int nSquare1 = SQUARE(nValue1); // nSquare1=100
nValue1++; // nValue1=11
```

### (3) 用大括号将宏所定义的多条表达式括起来。

如果宏定义包含多条表达式，一定要用大括号将其括起来。如果没有这个大括号，宏定义中的多条表达式很有可能只有第一句会被执行，正如下面的代码片段：

```
#define CLEAR_CUBE_VALUE( l, w, h )\
    l = 0; \
    w = 0; \
    h = 0;

int i = 0;
for (i = 0; i < CUBE_ACCOUNT; i++)
    CLEAR_CUBE_VALUE( Cubes[i].l, Cubes[i].w, Cubes[i].h );
```

简单的字符替代，并不能保证多条表达式都会放入for循环的循环体内，因为没有将它包围在循环体内的大括号中。正确的做法应该是用大括号将多条表达式括起来，这样就能保证多条表达式全部执行了，如下面的代码片段所示：

```
#define CLEAR_CUBE_VALUE( l, w, h )\
{\
    l = 0; \
    w = 0; \
    h = 0; \
}
```

---

### 请记住：

正确合理使用C语言中的宏，能有效地增强代码的可读性。但是也要遵守一定的规则，

避免踏入其中的陷阱：(1) 用宏定义表达式时，要使用完备的括号。(2) 使用宏时，不允许参数发生变化。(3) 用大括号将宏所定义的多条表达式包括起来。

---

## 建议 5：不要忘记指针变量的初始化

可以说指针是 C/C++ 语言编程中最给力的工具。指针，让我们直接去面对最为神秘的内存空间，赋予我们对内存进行直接操作的能力。由于指针操作执行速度快、占用内存少，众多程序员对它深爱不已。但是，它的灵活性和难控制性也让许多程序员觉得难以驾驭，以致到了谈指针色变的程度。

指针就是一把双刃剑。用好了它，会给你带来诸多便利，反之，则往往会引发意想不到的问题。

其中，指针的初始化就是我们应当重视的问题之一。指针应当被初始化，这是一个毋庸置疑的问题，关键是应该由谁来负责初始化，是编译器，还是程序员自己？

为了更好地贯彻零开销原则（C++ 之父 Bjarne 在设计 C++ 语言时所遵循的原则之一，即“无须为未使用的东西付出代价”），编译器一般不会对一般变量进行初始化，当然也包括指针。所以负责初始化指针变量的只有程序员自己。

使用未初始化的指针是相当危险的。因为指针直接指向内存空间，所以程序员很容易通过未初始化的指针改写该指针随机指向的存储区域。而由此产生的后果却是不确定的，这完全取决于程序员的运气。例如下面的程序片段：

```
#include <iostream>
int main()
{
    int *pInt;
    std::cout<<pInt<<"\n";
    return 0;
}
```

在 VC++ 中，程序在 Release 模式下输出 0x004080d0，而在 Debug 模式下输出 0xcccccccc。很明显未初始化的指针指向的是一个随机的地址。如果对其执行写操作会怎样？那很有可能会直接导致程序崩溃。

可以将指针初始化为某个变量的地址。需要注意的是，当用另一个变量的地址初始化指针变量时，必须在声明指针之前声明过该变量。代码片段如下所示：

```
int number = 0;      // Initialized integer variable
int* pNumber = &number; // Initialized pointer
```

当然，我们在必要时也可以将其初始化为空指针 0 (NULL)：

```
int* pNumber = NULL;      // Initialized pointer as NULL
```

如果使用未初始化的局部变量，程序编译时会给出警告 C4700：

```
warning C4700: 使用了未初始化的局部变量 "***"
```

需要注意警告中的四个字“局部变量”。因为对于全局变量来说，在声明的同时，编译器会悄悄完成对变量的初始化。代码片段如下所示：

```
#include <iostream>
int *pInt;
int main()
{
    std::cout<<pInt<<"\n";
    return 0;
}
```

此时，程序编译不会再出现警告，程序输出：00000000。

#### 请记住：

使用未初始化的局部指针变量是件很危险的事，所以，在使用局部指针变量时，一定要及时将其初始化。

## 建议 6：明晰逗号分隔表达式的奇怪之处

逗号分隔的表达式是从 C 继承而来的。它用一种特殊的运算符——逗号运算符将多个表达式连接起来。逗号表达式的一般形式为：

表达式 1, 表达式 2, 表达式 3..... 表达式 n

需要注意的是，整个逗号分隔表达式的值为表达式 n 的值。

在使用 for- 循环和 while- 循环时，经常会使用这样的表达式。然而，由于语言规则不直观，因此理解这样的语句存在一定的困难。例如：

```
if (++x, --y, x<20 && y>0) /* 三个表达式 */
```

if 条件包含由逗号分隔的三个表达式。C++ 确保每个表达式都会被执行，并产生作用。不过，整个表达式的值仅是最右边表达式的结果。因此，只有当 x 小于 20 且 y 大于 0 时才会返回 true，上述条件也才会为真。再举一个逗号表达式的例子：

```
int j=10;
int i=0;
while( ++i, --j)
{
```

```
/* 只要 j 不为 0 就会循环执行 */
}
```

其实，逗号表达式无非是把若干个表达式“串联”起来。在许多情况下，使用逗号表达式的目的只是想分别得到各个表达式的值，而并非一定需要得到和使用整个逗号表达式的值。

当然并不是所有地方出现的逗号都是逗号运算符，例如用逗号分隔的函数参数：

```
printf("%d - %s - %f", count, str, PI);
```

“count, str, PI”并非逗号分隔表达式，而是 printf 的三个输入参数。

另外一个需要注意的问题就是，在 C++ 中，逗号分隔表达式既可以用作左值，又可以用作右值。

#### 请记住：

逗号分隔的表达式由于语言规则的不直观，容易产生理解上的误差。在使用逗号分隔表达式时，C++ 会确保每个表达式都被执行，而整个表达式的值则是最右边表达式的结果。

## 建议 7：时刻提防内存溢出

作为一个程序员，对内存溢出问题肯定不陌生，它已经是软件开发历史上存在了近 40 年的大难题。在内存空间中，当要表示的数据超出了计算机为该数据分配的空间范围时，就产生了溢出，而溢出的多余数据则可以作为指令在计算机中大摇大摆地运行。不幸的是，一不小心这就成了黑客们可利用的秘密后门，“红色代码”病毒事件就是黑客利用内存溢出攻击企业网络的“经典案例”。甚至有人称，操作系统中超过 50% 的安全漏洞都是由内存溢出引起的。

众所周知，C/C++ 语言虽然是一种高级语言，但是其程序的目标代码却非常接近机器内核，它能够直接访问内存和寄存器，这种特性大大提升了 C/C++ 语言代码的性能，同时也提高了内存溢出问题出现的可能性。内存溢出问题可以说是 C/C++ 语言所固有的缺陷，因为它们既不检查数组边界，也不检查类型可靠性。

假设代码申请了 X 字节大小的内存缓冲区，随后又向其中复制超过 X 字节的数据，那么多出来的字节会溢出原本的分配区。最重要的是，C/C++ 编译器开辟的内存缓冲区常常邻近重要的数据结构。如果恶意攻击者用“别有用心”的东西刻意地覆盖原本安全可信的数据，那么后果就是此机器将会成为他们肆意攻击的“肉鸡”。下面将介绍常见的缓冲区溢出，以及预防措施。

C语言中的字符串库没有采用相应的安全保护措施，所以在使用时要特别小心。例如，在执行 strcpy、strcat 等函数操作时没有检查缓冲区大小，就会很容易引起安全问题。

现在分析下面的代码片段：

```
const int MAX_DATA_LENGTH = 32;
void DataCopy (char *szSrcData)
{
    char szDestData[MAX_DATA_LENGTH];
    strcpy(cDest,szData);
    // processing codes
    ...
}
```

似乎这段代码不存在什么问题，但是细心的读者还是会发其中的危险。如果数据源 szSrcData 的长度不超过规定的长度，那么这段代码确实没什么问题。strcpy() 不会在乎数据来源，也不会检查字符串长度，唯一能让它停下来只有字符串结束符 '\0'。不过，如果没有遇到这个结束符，它就会一个字节一个字节地复制 szSrcData 的内容，在填满 32 字节的预设空间后，溢出的字符就会取代缓冲区后面的数据。如果这些溢出的数据恰好覆盖了后面 DataCopy 函数的返回地址，在该函数调用完毕后，程序就会转入攻击者设定的“返回地址”中，乖乖地进入预先设定好的陷阱。

为了避免落入这样的圈套，给作恶者留下可乘之机，当 C/C++ 代码处理来自用户的数据时，应该处处留意。如果一个函数的数据来源不可靠，又要用到内存缓冲区，那么必须提高警惕，必须知道内存缓冲区的总长度，并检验内存缓冲区。

```
const int MAX_DATA_LENGTH = 32;
void DataCopy (char *szSrcData, DWORD nDataLen)
{
    char szDestData[MAX_DATA_LENGTH];
    if(nDataLen < MAX_DATA_LENGTH)
        strcpy(cDest,szData);
    szDestData[nDataLen] = '\0'; // 0x42;
    // processing code
    ...
}
```

首先，要获得 szSrcData 的长度，保证数据长度不大于最大缓冲区长度 MAX\_DATA\_LENGTH；其次，要保证参数传来的数据长度真实有效，方法就是向内存缓冲区的末尾写入数据。因为，当缓冲区溢出时，一旦向其中写入常量值，代码就会出错，终止运行。与其落入阴谋家的陷阱，还不如及时终止程序运行。

虽然上述方法能够有效地降低内存溢出的危害，却不能从根本上避免对内存溢出的攻击。所以在调用 strcpy、strcat、gets 等经典函数时，你要从源代码开始就提高警惕，尽量追踪传入数据的流向，向代码中的每一个假设提出质疑，包括对那些所谓相对可靠的改良

版 N-Versions (strncpy 或 strncat) 也不可轻信。

访问边界数据同样可能引起缓冲区溢出。在这种情况下的内存溢出不会像第一种那么危险，但同样令人讨厌。就如下面的代码片段：

```
const int DATA_LENGTH = 16;
Int data[16] = {1,9,8,4,0,9,1,7,1,9,8,7,0,3,0,9};
void PrintData()
{
    for(int i=0; a[i]!=0&&i<DATA_LENGTH; i++)
    {
        cout<<data[i]<<endl;
    }
}
```

这也是一个隐藏很深、难以发现的问题：当  $i==16$  的时候，在判断  $i < \text{DATA\_LENGTH}$  的同时需要判断  $\text{data}[16]$ 。而  $\text{data}[16]$  已经访问到了非法区域，可能引起缓冲区溢出。正确的方式应该是不要将索引号  $i$  与数据本身  $\text{data}[i]$  的判断放在一起，而是将判断条件分成两句：

```
const int DATA_LENGTH = 16;
int data[16] = {1,9,8,4,0,9,1,7,1,9,8,7,0,3,0,9};
void PrintData()
{
    for(int i=0; i<DATA_LENGTH; i++)
    {
        if(a[i]!=0)
            cout<<data[i]<<endl;
    }
}
```

类似的问题还有可能发生在访问未初始化指针或失效指针时。未初始化的指针和失效后未置 NULL 的指针指向的是未知的内存空间，所以对这样的指针进行操作很有可能访问或改写未知的内存区域，也就可能引起缓冲区溢出的问题了。

#### 请记住：

因为内存溢出潜在的危害很大，所以必须注意和面对这个问题，特别是在网络相关的应用程序中。在调用 C 语言字符串经典函数（如 strcpy、strcat、gets 等）时，要从源代码开始就提高警惕，尽量追踪传入数据的流向，向代码中的每一个假设提出质疑。在访问数据时，注意对于边界数据要特殊情况特殊处理，还要对杜绝使用未初始化指针和失效后未置 NULL 的“野指针”。

## 建议 8：拒绝晦涩难懂的函数指针

在 C/C++ 程序中，数据指针是最直接也是最常用的，理解起来也相对简单容易，但是函数指针理解起来却并不轻松。函数指针在运行时的动态调用中应用广泛，是一种常见而有效的手段。但是，如果不注重一定的使用技巧，函数指针也会变得晦涩难懂。

告诉我下面定义的含义是什么？

```
void (*p[10]) (void (*)());
```

如此繁琐的语法定义几乎难以辨认，这与我们提倡的可读性背道而驰了。这样的函数指针之所以让程序员发愁，最主要的原因是它的括号太多了，往往会让程序员陷在括号堆中理不清头绪。下面一层一层地来分析吧。第一个括号中的 `*p[10]` 是一个指针数组，数组中的指针指向的是一些函数，这些函数参数为 `void (*)()`，返回值为空；参数部分的 `void (*)()` 是一个无参数、返回值为空的函数指针。

分析这样的代码简直是一种折磨。如何有效地提高函数指针定义的可读性呢？那就是使用 `typedef`。`typedef` 方法可以有效地减少括号的数量，可以通过 `typedef` 来合理地简化这些声明，理清层次，所以它的使用倍受推荐。

以上面的定义为例。首先，声明一个无参数、返回空的函数指针的 `typedef`，如下所示：

```
typedef void (*pfv)();
```

接下来，声明另一个 `typedef`，一个指向参数为 `pfv` 且返回为空的函数指针：

```
typedef void (*pFun_taking_pfv) (pfv);
```

现在，再去声明一个含有 10 个这样指针的数组就变得轻而易举了，而且可读性有了很大的提升：

```
pFun_taking_pfv p[10]; /* 等同于 void (*p[10]) (void (*)()); */
```

### 请记住：

函数指针在运行时的动态调用（例如函数回调）中应用广泛。但是直接定义复杂的函数指针会由于有太多的括号而使代码的可读性下降。使用 `typedef` 可以让函数指针更直观和易维护。拒绝晦涩难懂的函数指针定义，拒绝函数定义中成堆的括号。

## 建议 9：防止重复包含头文件

假设，我们的工程中有如下三个文件：`a.h`、`b.h` 和 `c.cpp`，其中 `b` 文件中包含了 `a.h`，`c` 文件中又分别包含了 `a.h` 和 `b.h` 两个文件，如图 1-1 所示。

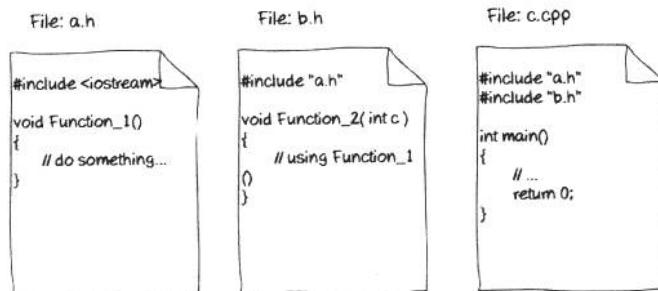


图 1-1 工程文件示例

在编译整个工程时，编译器会出现“multiple definition of”错误。原因在于 a.h 文件被包含了两次。为了避免同一个文件被包含多次，C/C++ 中有两种处理方式，一种是 `#ifndef` 方式，另一种是 `#pragma once` 方式。

方式 1：

```

#ifndef __SOMEFILE_H__
#define __SOMEFILE_H__
... ... // 声明、定义语句
#endif

```

方式 2：

```

#pragma once
... ... // 声明、定义语句

```

C/C++ 语言标准支持第一种方式。这种方式不仅可以保证同一个文件不会被包含多次，也能保证内容完全相同的两个文件不会被同时包含。当然，其缺点就是如果一不小心在不同头文件中定义了相同的宏名，造成了宏名“撞车”，那就可能会导致明明看到头文件存在，编译器却硬说找不到声明，这确实会令人非常恼火。为了避免宏名“撞车”，保证宏的唯一性，建议按照 Google 公司建议的那样，头文件基于其所在项目源代码树的全路径而命名。命名格式为：

```
<PROJECT>_<PATH>_<FILE>.h
```

由于编译器在每次编译时都需要打开头文件才能判定是否有重复定义，因此在编译大型项目时，`ifndef` 会使编译时间相对较长。

`#pragma once` 方式一般由编译器提供，它保证同一个文件不会被包含多次。这里所说的“同一个文件”指的是物理上的一个文件，而不是指内容相同的两个文件。`#pragma once` 声明只针对文件，而不能针对某一文件中的一段代码。这种方式避免了因想方设法定义一个独一无二的宏而产生的烦恼；另外，针对大型项目的编译速度也有了提升。但是这种方式因为不受 C/C++ 语言标准支持，所以受到了编译器的限制，它在兼容性方面表现得不是很好。因

此很多程序员为了代码的兼容性，宁肯降低一些编译性能，而选择遵循C/C++标准，采用第一种方式。

---

**注意** 针对#pragma once，GCC已经取消了对其的支持，而微软的VC++却仍在坚持。

---

### 请记住：

为了避免重复包含头文件，建议在声明每个头文件时采用“头文件卫士”加以保护，比如采用如下的形式：

```
#ifndef _PROJECT_PATH_FILE_H_
#define _PROJECT_PATH_FILE_H_
... ... // 声明、定义语句
#endif
```

---

## 建议 10：优化结构体中元素的布局

下面的代码片段定义了结构体A和B：

```
struct A // 结构体A
{
    int a;
    char b;
    short c;
};

struct B // 结构体B
{
    char b;
    int a;
    short c;
};
```

在32位机器上，char、short、int三种类型的大小分别是1、2、4。那么上面两个结构体的大小如何呢？

结构体A中包含了一个4字节的int，一个1字节的char和一个2字节的short，B也一样，所以A、B的大小应该都是 $4+2+1=7$ 字节。但是，实验给出的却是另外的结果：

```
sizeof(struct A) = 8, sizeof(struct B) = 12
```

其原因还要从字节对齐说起。

现代计算机中内存空间都是按照字节来划分的，从理论上来讲，对变量的访问可以从任何地址开始；但在实际情况中，为了提升存取效率，各类型数据需要按照一定的规则在空间上排列，这使得对某些特定类型的数据只能从某些特定地址开始存取，以空间换取时间，这

就是字节对齐。

结构体默认的字节对齐一般满足三个准则：

- (1) 结构体变量的首地址能够被其最宽基本类型成员的大小所整除。
- (2) 结构体每个成员相对于结构体首地址的偏移量 (offset) 都是成员自身大小的整数倍，如有需要，编译器会在成员之间加上填充字节 (Internal Adding)。
- (3) 结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要，编译器会在最末一个成员之后加上填充字节 (Trailing Padding)。

按照这三条规则再去分析结构体 A 和 B，就不会对于上述的结果一脸诧异了。这两个结构体在内存空间中的排列如图 1-2 所示（灰色网格表示的字节为填充字节）。

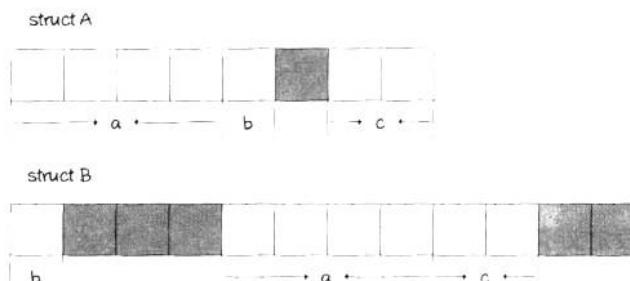


图 1-2 结构体 A 和 B 的内存分布

在编程应用中，如果空间紧张，需要考虑节约空间，那么就需要将结构体中的各个变量按照上面的原则进行排列。基本的原则是：把结构体中的变量按照类型大小从小到大依次声明，尽量减少中间的填充字节。

也可以采用保留字节的形式显式地进行字节填充实现对齐，以提高存取效率。其实这就是时间与空间的博弈。如下面的代码片段所示，其中的 reserved 成员对程序没有什么意义，它只是填补空间以达到字节对齐的目的：

```
struct A // 结构体 A
{
    int a;
    char b;
    char reserved; // 保留字节，空间换时间
    short c;
};

struct B // 结构体 B
{
    char b;
    char reserved1[3]; // 保留字节 1，空间换时间
    int a;
    short c;
```

```
char reserved2[2]; // 保留字节2, 空间换时间
};
```

在某些时候，还可以通过编译器的 pack 指令调整结构体的对齐方式。#pragma pack 的基本用法为：

```
#pragma pack( n )
```

n 为字节对齐数，其取值为 1、2、4、8、16，默认是 8。

```
#pragma pack(1) // 设置1字节对齐
struct A // 结构体A
{
    int a;
    char b;
    short c;
};
```

将结构体 A 的对齐方式设为 1 字节对齐，那么 A 就不再有填充字节了，sizeof(A) 的结果即为各元素所占字节之和 7。

**请记住：**

了解结构体中元素的对齐规则，合理地为结构体元素进行布局。这样不仅可以有效地节约空间，还可以提高元素的存取效率。

## 建议 11：将强制转型减到最少

C++ 在设计中一直强调类型安全，而且也采取了一定的措施来保障这条准则的执行。但是，从 C 继承而来的强制转型却破坏了 C++ 类型系统，C 中的强制转型可谓是“无所不能”，其超强的能力给 C++ 带来了很大的安全隐患。强制转型会引起各种各样的麻烦，有时这些麻烦很容易被察觉，有时它们却又隐藏极深，难以察觉。

在 C/C++ 语言中，强制转型是“一个你必须全神贯注才能正确使用”的特性。所以一定要慎用强制转型。

首先来回顾一下 C 风格（C-style）的强制转型语法，如下所示：

```
// 将表达式的类型转换为 T
(T) expression
T(expression)
```

这两种形式之间没有本质上的区别。在 C++ 中一般称为旧风格的强制转型。

在赋值时，强制类型的转换形式会让人觉得不精密、不严格，缺乏安全感，主要是因为不管表达式的值是什么类型，系统都自动将其转为赋值运算符左侧变量的类型。而转变后数

据可能会有所不同，若不加注意，就可能产生错误。

将较大的整数转换为较短的数据类型时，会产生无意义的结果，而程序员可能被蒙在鼓里。正如下面的代码片段所示：

```
unsigned i = 65535;
int j = (int) i;
```

输出结果竟然成了 -1。较长的无符号类型在转换为较短的有符号类型时，其数值很可能超出较短类型的数值表示范围。编译器不会监测这样的错误，它所能做的仅仅是抛出一条非安全类型转换的警告信息。如果这样的问题发生在运行时，那么一切会悄无声息，系统既不会中断，也不会出现任何的出错信息。

类似的问题还会发生在有符号负数转化为无符号数、双精度类型转化为单精度类型、浮点数转化为整型等时候。以上这些情况都属于数值的强制转型，在转换过程中，首先生成临时变量，然后会进行数值截断。

在标准 C 中，强制转型还有可能导致内存扩张与截断。这是因为在标准 C 中，任何非 void 类型的指针都可以和 void 类型的指针相互指派，也就可以通过 void 类型指针这个中介，实现不同类型的指针间接相互转换了。代码如下所示：

```
double PI = 3.1415926;
double *pd = &PI;
void *temp = pd;
int *pi = temp; // 转换成功
```

指针 pd 指向的空间本是一个双精度数据，8 字节。但是经过转换后，pi 却指向了一个 4 字节的 int 类型。这种发生内存截断的设计缺陷会在转换后进行内存访问时存在安全隐患。不过，这种情况只会发生在标准 C 中。在 C++ 中，设计者为了杜绝这种错误的出现，规定了不同类型的指针之间不能相互转换，所以在使用纯 C++ 编程时大可放心。而如果 C++ 中嵌入了部分 C 代码，就要注意因强制转型而带来的内存扩张或截断了。

与旧风格的强制转型相对应的就是新风格的强制转型了，在 C++ 提供了如下四种形式：

```
const_cast(expression)
dynamic_cast(expression)
reinterpret_cast(expression)
static_cast(expression)
```

新风格的强制转型针对特定的目的进行了特别的设计，如下所示。

□ `const_cast<T*> (a)`

它用于从一个类中去除以下这些属性：`const`、`volatile` 和 `_unaligned`。

```
class A { // ... };
void Function()
{
```

```

const A *pConstObj = new A;
A *pObj = pConstObj; //ERROR: 不能将 const 对象指针赋值给非 const 对象
pObj = const_cast<A*>(pConstObj); // OK
//...
}

```

这种强制转型的目的简单明确，使用情形比较单一，易于掌握。

#### □ `dynamic_cast<T*>(a)`

它将 a 值转换成类型为 T 的对象指针，主要用来实现类层次结构的提升，在很多书中它被称做“安全的向下转型（Safe Downcasting）”，用于继承体系中的向下转型，将基类指针转换为派生类指针，这种转换较为严格和安全。如下面的代码片段所示：

```

class B { /*...*/ };
class D : public B { /*...*/ };
void Function(D *pObjD)
{
    D *pObj = dynamic_cast<D*>(pObjD);
    //...
}

```

如果 pObjD 指向一个 D 类型的对象，pObj 则指向该对象，所以对该指针执行 D 类型的任何操作都是安全的。但是，如果 pObjD 指向的是一个 B 类型的对象，pObj 将是一个空指针，这在一定程度上保证了程序员所需要的“安全”，只是，它也付出了一定的运行时代价，而且代价非常大，实现相当慢。有一种通用实现是通过对类名称进行字符串比较来实现的，只是其在继承体系中所处的位置越深，对 strcmp 的调用就越多，代价也就越大。如果应用对性能要求较高，那么请放弃 `dynamic_cast`。

#### □ `reinterpret_cast<T*>(a)`

它能够用于诸如 `One_class*` 到 `Unrelated_class*` 这样的不相关类型之间的转换，因此它是不安全的。其与 C 风格的强制转型很是相似。

```

class A { /* ... */ };
class B { /*... */ };
void f()
{
    A* pa = new A;
    B* pb = reinterpret_cast<B*>(pa);
    // ...
}

```

在不了解 A、B 内存布局的情况下，强行将其进行转换，很有可能出现内存膨胀或截断。

#### □ `static_cast<T*>(a)`

它将 a 的值转换为模板中指定的类型 T。但是，在运行时转换过程中，它不会进行类型检查，不能确保转换的安全性。如下面的代码片段所示：

```

class B { ... };
class D : public B { ... };
void Function(B* pb, D* pd)
{
    D* pd2 = static_cast<D*>(pb); // 不安全
    B* pb2 = static_cast<B*>(pd); // 安全的
}

```

之所以说第一种是不安全的，是因为如果 pb 指向的仅仅是一个基类 B 的对象，那么就会凭空生成继承信息。至于这些信息是什么、正确与否，无从得知。所以对它进行 D 类型的操作将是不安全的。

C++ 是一种强类型的编程语言，其规则设计为“保证不会发生类型错误”。在理论层面上，如果希望程序顺利地通过编译，就不应该试图对任何对象做任何不安全的操作。不幸的是，继承自 C 语言的强制转型破坏了类型系统，所以建议尽量少地使用强制转型，无论是旧的 C 风格的还是新的 C++ 风格的。如果发现自己使用了强制转型，那么一定要小心，这可能就是程序出现问题的一个信号。

#### 请记住：

由于强制转型无所不能，会给 C++ 程序带来很大的安全隐患，因此建议在 C++ 代码中，努力将强制转型减到最少。

## 建议 12：优先使用前缀操作符

也许从开始接触 C/C++ 程序的那天起，就记住了前缀和后缀运算，知道了 ++ 的前缀形式是“先加再用”，后缀形式是“先用再加”。前缀和后缀运算是 C 和 C++ 语言中的基本运算，它们具有类似的功能，区别也很细微，主要体现在运行效率上。

分析下面的代码片段：

```

int n=0, m=0;
n = ++m; /* m 先加 1，之后赋给 n */
cout << n << m; /* 结果：1 1 */

```

在这个例子中，赋值之后，n 等于 1，因为它是在将 m 赋予 n 之前完成的自增操作。再看下面的代码：

```

int n=0, m=0;
n = m++; /* 先将 m 赋予 n，之后 m 加 1 */
cout << n << m; /* 结果：0 1 */

```

这个例子中，赋值之后，n 等于 0，因为它是先将 m 赋予 n，之后 m 再加 1 的。

为了更好地理解前缀操作符和后缀操作符之间的区别，可以查看这些操作的反汇编代

码。即使不了解汇编语言，也可以很清楚地看到二者之间的区别，注意 inc 指令出现的位置：

```
/* m=n++; 的反汇编代码 */
mov ecx, [ebp-0x04] /*store n's value in ecx register*/
mov [ebp-0x08], ecx /*assign value in ecx to m*/
inc dword ptr [ebp-0x04] /*increment n*/
/*m=++n; 的反汇编代码 */
inc dword ptr [ebp-0x04] /*increment n*/
mov eax, [ebp-0x04] /*store n's value in eax register*/
mov [ebp-0x08], eax /*assign value in eax to m*/
```

从汇编代码可以看出，两者采取了相同的操作，只是顺序稍有不同而已。但是，前缀操作符的效率要优于后缀操作符，这是因为在运行操作符之前编译器需要建立一个临时的对象，而这还要从函数重载说起。

重载函数间的区别取决于它们在参数类型上的差异，但不论是自增的前缀还是后缀，都只有一个参数。为了解决这个语言问题，C++ 规定后缀形式有一个 int 类型的参数，当函数被调用时，编译器传递一个 0 作为 int 类型参数的值给该函数：

```
// 成员函数形式的重载
< Type > ClassName :: operator ++ ( ); // 前缀
< Type > ClassName :: operator ++ ( int ); // 后缀
// 非成员函数形式的重载
< Type > operator ++ (ClassName &); // 前缀
< Type > operator ++(ClassName &,int); // 后缀
```

在实现中，后缀操作会先构造一个临时对象，并将原对象保存，然后完成自增操作，最后将保存对象原值的临时对象返回。代码如下所示：

```
ClassName & ClassName::operator++()
{
    ClassAdd (1); //increment current object
    return *this; //return by reference the current object
}

ClassName ClassName::operator++(int unused)
{
    ClassName temp(*this); //copy of the current object
    ClassAdd (1); //increment current object
    return temp; //return copy
}
```

由于前缀操作省去了临时对象的构造，因此它在效率上优于后缀操作。不过，在应用到整型和长整型的操作时，前缀和后缀操作在性能上的区别通常是可以忽略的。但对于用户自定义类型，这还是非常值得注意的。当然就像 80-20 规则<sup>⊖</sup>告诉我们的那样，如果在

---

<sup>⊖</sup> 80-20 规则：一个典型的程序将花去 80% 的时间仅仅运行 20% 的代码。

一个很大的程序里，程序数据结构和算法不够优秀，它所能带来的效率提升也是微不足道的，不能使大局有所改变。但是既然它们有差异，我们为什么不在必要的时候采用更有效率的呢？

---

**请记住：**

对于整型和长整型的操作，前缀操作和后缀操作的性能区别通常是可以忽略的。对于用户自定义类型，优先使用前缀操作符。因为与后缀操作符相比，前缀操作符因为无须构造临时对象而更具性能优势。

---

### 建议 13：掌握变量定义的位置与时机

在 C/C++ 代码中，变量是一个不得不提的关键词。变量在程序中起着不同寻常的作用。所有的代码中肯定离不开各种类型变量的影子，既有内置类型的，又有自定义类型的。虽然常见，但使用它也是有一定的技巧与玄机的。掌握了这些技巧与玄机，在合适的时机将变量定义在合适的位置上，会使代码变得更具可读性与高效性。

C++ 规则允许在函数的任何位置定义变量，当程序执行到变量定义的位置，并接收到这一变量的定义时，就会调用相应的构造函数，完成变量的构造。当程序控制点超出变量的作用域时，析构函数就会被调用，完成对该变量的清理。而对象的构造和析构不可避免地会带来一定的开销，无论该变量在程序中有没有发挥作用，所以建议在需要使用变量时再去定义。

分析下面代码片段中定义的函数：

```
std::string ChangToUpper(const std::string& str)
{
    using namespace std;
    string upperStr;
    if (str.length() <= 0)
    {
        throw error("String to be changed is null");
    }
    ... // 将字符变为大写
    return upperStr;
}
```

在上面的代码中，变量 `upperStr` 定义的时机有点早。如果输入字符串为空，函数抛出异常，这个变量就不会被使用。所以，如果函数抛出了异常，就要为 `upperStr` 的构造与析构付出代价，而这些代价完全完全是可以避免的。所以，变得精明些，把握变量定义的时机：尽量晚地去定义变量，直到不得不定义时。代码如下所示。

```
std::string ChangToUpper(const std::string& str)
{
    using namespace std;
    if (str.length() <= 0 )
    {
        throw error("String to be changed is null");
    }
    string upperStr;
    ...      // 将字符变为大写
    return upperStr;
}
```

关于变量定义的位置，建议变量定义得越“local”越好，尽量避免变量作用域的膨胀。这样做不仅可以有效地减少变量名污染，还有利于代码阅读者尽快找到变量定义，获悉变量类型与初始值，使阅读代码更容易。

针对“变量名污染”，最臭名昭著的例子就是在VC 6.0环境的for语句中声明变量i：

```
for( int i=0; i<N; i++)
{
    ...// do something
}
... // some code
for( int i=0; i<M; i++)
{
    ...// do another thing
}
```

上述代码在VC 6.0中是不能通过编译的，编译器会提示变量i重复定义。不熟悉VC 6.0环境的人肯定会很诧异。这是因为在VC 6.0中，i的作用域超出了本身的循环。幸好，微软意识到了这个问题，在其后续的VC++系列产品中，i的作用域重新被限定在了for循环体中。

不过在这条规则中，还有一个小小的例外，如下所示：

```
for (int i = 0; i < 1000000; ++i)
{
    ClassName obj;
    obj.DoSomething();
}
```

以上变量的定义遵循了“尽可能晚，尽可能local”的规则，但是ClassName的构造和析构却因此被调用了1 000 000次。更高效的方式就是将obj的定义放在循环之外，构造函数和析构函数的调用次数则会减少到1次：

```
ClassName obj;
for (int i = 0; i < 1000000; ++i)
{
```

```

    obj.DoSomething();
}

```

**请记住：**

在定义变量时，要三思而后行，掌握变量定义的时机与位置，在合适的时机于合适的位置上定义变量。尽可能推迟变量的定义，直到不得不需要该变量为止；同时，为了减少变量名污染，提高程序可读性，尽量缩小变量的作用域。

## 建议 14：小心 `typedef` 使用中的陷阱

`typedef` 本来是很好理解的一个概念，但是因为与宏并存，理解起来就有点困难了。再加上部分教材以偏概全，更是助长了错误认识的产生。某些教材介绍 `typedef` 时会给出类似以下的形式，但是缺少进一步的解释：

```

typedef string NAME;
typedef int AGE;

```

这种形式让我不由地想起 C 语言中著名的宏定义：

```

#define MAC_NAME string
#define MAC_AGE   int

```

因为二者的声明方式太相似了，所以很多人习惯用 `#define` 的思维方式来看待 `typedef`，认为应当把 `int` 与 `AGE` 看成独立的两部分。实际情况是怎样的呢？首先分析下面的代码片段：

```

typedef int* PTR_INT1;
#define int* PTR_INT2
int main()
{
    PTR_INT1 pNum1, pNum2;
    PTR_INT2 pNum3, pNum4;
    int year = 2011;
    pNum1 = &year;
    pNum2 = &year;
    pNum3 = &year;
    pNum4 = &year;
    cout<<pNum1<<" "<<pNum2<<" "<<pNum3<<" "<<pNum4;
    return 0;
}

```

输出为：2011 2011 2011 0E8951241。通过程序执行结果可以看出 `typedef` 与 `#define` 的不同：`typedef` 后面是一个整体声明，是不能分割的部分，就像整型变量声明 `int i;`，只不过 `typedef` 声明的是一个别名。宏定义只是简单的字符串替换，不过，`typedef` 并不是原地扩展，它的新名称具有一定的封装性，更易于定义变量，它可以同时声明指针类型的多个对象，而

宏则不能。使用 `typedef` 声明多个指针对象，形式直观，方便省事：

```
char *pa, *pb, *pc, *pd; // 方式1

typedef char* PTR_CHAR;
PTR_CHAR pa, pb, pc, pd; // 方式2, 直观省事
```

除此之外，`typedef` 还有多种用途，下面来看看。

(1) 在部分较老的 C 代码中，声明 `struct` 对象时，必须带上 `struct` 关键字，即采用“`struct` 结构体类型 `结构体对象`”的声明格式。例如：

```
struct tagRect
{
    int width;
    int length;
};

struct tagRect rect;
```

为了在结构体使用过程中，少写声明头部的 `struct`，于是就有人使用了 `typedef`：

```
typedef struct tagRect
{
    int width;
    int length;
}RECT;
RECT rect;
```

在现在的 C++ 代码中，这种方式已经不常见，因为对于结构体对象的声明已经不需要使用 `struct` 了，可以采用“`结构体类型 结构体对象`”的形式。

(2) 用 `typedef` 定义一些与平台无关的类型。例如在标准库中广泛使用的 `size_t` 的定义：

```
#ifndef _SIZE_T_DEFINED
#define _WIN64
typedef unsigned __int64    size_t;
#else
typedef _W64 unsigned int   size_t;
#endif
#define _SIZE_T_DEFINED
#endif
```

(3) 为复杂的声明定义一个简单的别名。这一点将在建议 93 中详细介绍。它可以增强程序的可读性和标识符的灵活性，这也是它最突出的作用。

在 `typedef` 的使用过程中，还必须记住：`typedef` 在语法上是一个存储类的关键字，类似于 `auto`、`extern`、`mutable`、`static`、`register` 等，虽然它并不会真正影响对象的存储特性，如：

```
typedef static int INT2; // 不可行, 编译将失败
```

编译器会提示“指定了一个以上的存储类型”。

---

请记住：

区分 `typedef` 与 `#define` 之间的不同：不要用理解宏的思维方式对待 `typedef`，`typedef` 声明的新名称具有一定的封装性，更易定义变量。同时还要注意它是一个无“现实意义”的存储类关键字。

---

## 建议 15：尽量不要使用可变参数

在某些情况下我们希望函数参数的个数可以根据实际需要来确定，所以 C 语言中就提供了一种长度不确定的参数，形如：“...”，C++ 语言也继承了这一语言特性。在采用 ANSI 标准形式时，参数个数可变的函数的原型是：

```
type funcname(type para1, type para2, ...);
```

这种形式至少需要一个普通的形式参数，后面的省略号 (...) 不能省去，它是函数原型必不可少的一部分。典型的例子有大家熟悉的 `printf()`、`scanf()` 函数，如下所示的就是 `printf()` 的原型：

```
int printf( const char *format , ... );
```

除了参数 `format` 固定以外，其他参数的个数和类型是不确定的。在实际调用时可以有以下形式：

```
int year = 2011;
char str[] = "Hello 2011";
printf("This year is %d", year);
printf("The greeting words are %s", str);
printf("This year is %d ,and the greeting words are:%s", year, str);
```

也许这些已经为大家所熟知，但是可变参数的实现原理却是 C 语言中比较难理解的一部分。在标准 C 语言中定义了一个头文件，专门用来对付可变参数列表，其中，包含了一个 `va_list` 的 `typedef` 声明和一组宏定义 `va_start`、`va_arg`、`va_end`，如下所示：

```
// File: VC++2010 中的 stdarg.h
#include <vadefs.h>

#define va_start _crt_va_start
#define va_arg _crt_va_arg
#define va_end _crt_va_end

// File: VC++2010 中的 vadefs.h
#ifndef _VA_LIST_DEFINED
```

```

typedef char * va_list;
#define _VA_LIST_DEFINED
#endif

#ifndef __cplusplus
#define _ADDRESSOF(v)    ( reinterpret_cast<const char &>(v) )
#else
#define _ADDRESSOF(v)    ( &(v) )
#endif

#ifdef(_M_IX86)
#define _INTSIZEOF(n)   ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )
#define _crt_va_start(ap,v) ( ap = (va_list)_ADDRESSOF(v) + _INTSIZEOF(v) )
#define _crt_va_arg(ap,t)  ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
#define _crt_va_end(ap)   ( ap = (va_list)0 )

```

定义 `_INTSIZEOF(n)` 是为了使系统内存对齐；`va_start(ap, v)` 使 `ap` 指向第一个可变参数在堆栈中的地址，`va_arg(ap,t)` 使 `ap` 指向下一个可变参数的堆栈地址，并用 `*` 取得该地址的内容；最后变参获取完毕，通过 `va_end(ap)` 让 `ap` 不再指向堆栈，如图 1-3 所示。

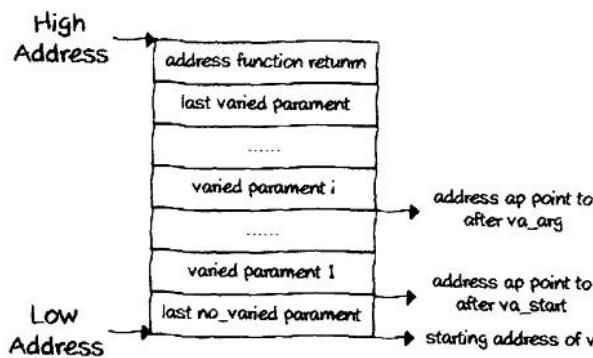


图 1-3 可变参数存储示意图

由于将 `va_start`、`va_arg`、`va_end` 定义成了宏，可变参数的类型和个数在该函数中完全由程序代码控制，并不能智能地进行识别，所以导致编译器对可变参数的函数原型检查不够严格，难于查错，不利于写出高质量的代码。

参数个数可变具有很多的优点，为程序员带来了很多的方便，但是上面 C 风格的可变参数却存在着如下的缺点：

(1) 缺乏类型检查，类型安全性无从谈起。“省略号的本质是告诉编译器‘关闭所有检查，从此由我接管，启动 `reinterpret_cast`’”，强制将某个类型对象的内存表示重新解释成另外一种对象类型，这是违反“类型安全性”的，是大忌。

例如，自定义的打印函数。

```

void UserDefinedPrintFun(char* format, int i, ...)
{
    va_list arg_ptr;
    char *s = NULL;
    int *i = NULL;
    float *f = NULL;

    va_start(arg_ptr, i);
    while(*format != '\0')
    {
        format++;
        if(*(format-1) == '%' && *format == 's')
        {
            s = va_arg(arg_ptr, char*);
            ..... // 输出至屏幕
        }
        else if(*(format-1) == '%' && *format == 'd')
        {
            i = va_arg(arg_ptr, int*);
            ..... // 输出至屏幕
        }
        else if(*(format-1) == '%' && *format == 'f')
        {
            f = va_arg(arg_ptr, float*);
            ..... // 输出至屏幕
        }
    }
    va_end(arg_ptr);
    return;
}

```

如果采用下面三种方法调用，合法合理：

```

UserDefinedPrintFun ("%d", 2010); // 结果 2010
UserDefinedPrintFun ("%d%d", 2010, 2011); // 结果 20102011
UserDefinedPrintFun ("%s%d", "Hello", 2012); // 结果 Hello2012

```

但是，当给定的格式字符串与参数类型不对应时，强制转型这个“怪兽”就会被唤醒，悄悄地毁坏程序的安全性，这可不是什么高质量的程序，如下所示：

```

UserDefinedPrintFun ("%d", 2010.80f);
// 结果 2010
UserDefinedPrintFun ("%d%d", "Hello", 2012);
// 结果 150958722015 (这是什么结果？？？)

```

(2) 因为禁用了语言类型检查功能，所以在调用时必须通过其他方式告诉函数所传递参数的类型，以及参数个数，就像很多人熟知的 printf() 函数中的格式字符串 char\* format。这种方式需要手动协调，既易出错，又不安全，上面的代码片段已经充分说明了

这一点。

(3) 不支持自定义数据类型。自定义数据类型在 C++ 中占有较重的地位，但是长参数只能传递基本的内置类型。还是以 printf() 为例，如果要打印出一个 Student 类型对象的内容，对于这样的自定义类型，该用什么格式的字符串去传递参数类型呢？如下所示：

```
class Student
{
public:
    Student();
    ~Student();
private:
    string m_name;
    char   m_age;
    int    m_scoer;
};

Student XiaoLi;
printf(format, XiaoLi); // format 应该是什么呢
```

上述缺点足以让我们有了拒绝使用 C 风格可变参数的念头，何况 C++ 的多态性已经为我们提供了实现可变参数的安全可靠的有效途径呢！如下所示：

```
class PrintFunction
{
public:
    void UserDefinedPrintFun(int i);
    void UserDefinedPrintFun(float f);
    void UserDefinedPrintFun(int i, char* s);
    void UserDefinedPrintFun(float f, char* s);
private:
    .....
};
```

虽然上述设计不能像 printf() 函数那样灵活地满足各种各样的需求，但是可以根据需求适度扩充函数定义，这样不仅能满足需求，其安全性也是毋庸置疑的。舍安全而求危险，这可不是明白人所为。如果还对 printf() 的灵活性念念不忘，我告诉大家，有些 C++ 库已经使用 C++ 高级特性将类型安全、速度与使用方便很好地结合在一起了，比如 Boost 中的 format 库，大家可以尝试使用。

### 请记住：

编译器对可变参数函数的原型检查不够严格，所以容易引起问题，难于查错，不利于写出高质量的代码。所以应当尽量避免使用 C 语言方式的可变参数设计，而用 C++ 中更为安全的方式来完美代替之。

## 建议 16：慎用 goto

如果说有一个关键字在 C/C++ 语言程序中备受争议，那么非程序跳转关键字 `goto` 莫属。在早期的 BASIC 和 FORTRAN 语言中，`goto` 备受依赖，为了照顾部分程序员的设计习惯，在 C 语言中 `goto` 关键字依然得到了保留。然而，与前面两种语言有所不同，`goto` 在 C/C++ 语言中就像一个多余的外来户，有没有它几乎不影响 C 语言程序的设计与运行，它没有带来太大的正面作用，相反却容易破坏程序的结构性。所以，Kernighan 和 Ritchie 认为 `goto` 语句“非常容易被滥用”，并且建议“一定要谨慎使用，或者干脆不用”。

之所以建议避免使用 `goto`，是因为 C/C++ 语言中提供了更好的方式去实现 `goto` 的功能。为了帮助部分程序员克服 `goto` 依赖症，接下来我会分别介绍以下情形下 `goto` 语句的替换方式：

### □ if 控制内的多条语句

如果熟悉旧风格的 BASIC 和 FORTRAN，会知道只有紧跟在 `if` 条件后的那一条语句才属于该 `if` 的控制域，所以就出现了以下 `goto` 的使用形式：

```
Size = 20;
Flag = 1;
if( price > 15)
    goto A_PLAN;
goto B_PLAN;
A_PLAN:
    Size /=2;
    Flag = 3;
B_PLAN:
    Money = price * Size * Flag;
```

而在 C/C++ 语言中，复合语句或代码块很容易实现上述目的，而且使代码更加清晰可读，如下所示：

```
Size = 20;
Flag = 1;
if( price > 15)
{
    Size /=2;
    Flag = 3;
}
Money = price * Size * Flag;
```

### □ 不确定循环

首先请看如下代码：

```
ReadScore:
```

```

scanf( " %d " , &Score );
if(Score < 0 )
    goto ErrorStage;
... // Processing codes
goto ReadScore;
ErrorStage:
... // Error Processing

```

这种情形可以用我们熟知的 while 循环来完美代替：

```

scanf( " %d " , &Score );
while(Score >= 0)
{
    ... // Processing codes
    scanf( " %d " , &Score );
}

```

此外，如果跳转到循环结尾会开始新一轮循环，可以使用 continue 代替；如果要跳出循环，那就使用 break。

上述 goto 语句破坏了程序的结构性，影响了程序的可读性，这在 C/C++ 程序员看来是难以容忍的。然而，有一种 goto 的使用情形为许多 C/C++ 程序员所接受，那就是程序在一组嵌套循环中出现错误，无路可走时的跳转处理，如下所示：

```

while(... )
{
    for(... )
    {
        for(... )
        {
            Processing statement;
            if(error)
                goto ERROR;
        }
        More processing statement;
    }
    Yet more processing statement;
}
And more processing statement;
ERROR:
    Deal_With_Error Statement;

```

#### 请记住：

过度使用 goto 会使代码流程错综复杂，难以理清头绪。所以，如果不熟悉 goto，不要使用它；如果已经习惯使用它，试着不去使用。

## 建议 17：提防隐式转换带来的麻烦

在 C/C++ 语言的表达式中，允许在不同类型的数据之间进行某一操作或混合运算。当对不同类型的数据进行操作时，首先要做就是将数据转换成为相同的数据类型。C/C++ 语言中的类型转换可以分为两种，一种为隐式转换，而另一种则为建议 11 中提及的显式强制转型。显式强制转型在某种程度上还有一定的优点，对于编写代码的人来说使用它能够很容易地获得所需类型的数据，对于阅读代码的人来说可以从代码中获知作者的意图。而隐式转换则不然，它让发生的一切变得悄无声息，在编译时这一切由编译程序按照一定规则自动完成，不需任何的人为干预。

存在大量的隐式转换也是 C/C++ 常受人诟病的焦点之一。隐式转换虽然带来了一定的便利，使编码更加简洁，减少了冗余，但是这些并不足以让我们完全接受它，因为隐式转换所带来的副作用不可小觑，它通常会使我们在调试程序时毫无头绪。就像下面的代码片段所示：

```
void Function(char c);

int main()
{
    long para = 256;
    Function(para);
    return 0;
}
```

上述代码片段中的函数调用不会出现任何错误，编译器给出的仅仅是一个警告。可是细心的程序员一眼就能看出问题：函数 Function (char c) 的参数 c 是一个 char 型，256 绝不会出现在其取值区间内。但是编译器会自动地完成数据截断处理。编译器悄悄完成的这种转换存在着很大的不确定性：一方面它可能是合理的，因为尽管类型 long 大于 char，但 para 中很可能存放着 char 类型范围内的数值；另一方面 para 的值的确可能是 char 无法容纳的数据，这种“暗地里的勾当”一不小心便会造成一个非常隐蔽、难以捉摸的错误。

C/C++ 隐式转换主要发生在以下几种情形。

### □ 基本类型之间的隐式转换

```
int ival = 3;
double dval = 3.1415
cout<<(ival + dval)<<endl; //ival 被提升为 double 类型 :3.0

extern double sqrt(double);
sqrt(2); //2 被提升为 double 类型 : 2.0
```

在编译这段代码时，编译器会按照规则自动地将 ival 转换为与 dval 相同的 double 类型。

C语言规定的转换规则是由低级向高级转换。两个通用的转换原则是：

- (1) 为防止精度损失，类型总是被提升为较宽的类型。
- (2) 所有含有小于整型类型的算术表达式在计算之前其类型都会被转换成整型。

这两点在C++中依旧有效，这已无须多言。它最直接的害处就是有可能导致重载函数产生二义性，如下所示：

```
void Print(int ival);
void Print(float fval);

int ival = 2;
float fval = 2.0f;
Print(ival); // OK, int-version
Print(fval); // OK, float-version
Print(1); // OK, int-version
Print(0.5); // ERROR!!
```

参数0.5应该转换为ival还是fval？这是编译器没法搞明白的一个问题。

#### □ T\*指针到void\*的隐式转换

在C语言中，标准允许T\*与void\*之间的双向转换，这也就间接导致了各种数据类型之间的隐式转换是被允许的，无论是从低级到高级，还是从高级到低级。这样的转换存在着太多的不安全因素，所以到了C++中，双向变单向，只允许T\*隐式地转换为void\*了，示例代码如下所示：

```
char* pChar = new char[20];
void * pVoid = pChar;
```

□ non-explicit constructor 接受一个参数的用户定义类对象之间隐式转换先看如下代码：

```
class A
{
public:
    A(int x) : m_data(x) {}
private:
    int m_data;
}
void DoSomething(A aObject);
DoSomething(20);
```

在上面的代码中，调用DoSomething()函数时会发现实参与形参类型不一致，但是因为类A的构造函数只含有一个int类型的参数，所以编译器会以20为参数调用A的构造函数，以便构造临时对象，然后传给DoSomething()函数。不要为此而感到惊讶，其实编译器比想像的还要聪明：当无法完成直接隐式转换的时候，它不会罢休，它会尝试使用间接的方式。所以，下面的代码也是可以被编译器接受的：

```
void DoSomething(A aObject);
```

```
float fval = 20.0f;
DoSomething(fval);
```

这是一个多么奇妙的世界。这样的隐式转换在某些时候会变得相当微妙，一个误用也许会引起难以捉摸的错误。另外，由于在隐式转换过程中需要调用类的构造函数、析构函数，如果这种转换代价很大，那么这样的隐式转换将会影响系统性能。

当然，我们熟知的隐式转换还包括“子类到基类的隐式转换”和“const 到 non-const 的同类型隐式转换”。不过这两种转换是比较安全的，所以在这里就不再详细讨论。

如果试图禁止所有的隐式类型转换，那么为了维持函数使用代码的简洁性，函数必须对所有的类型执行重载。这将是一个十分庞大且毫无技术含量的重复性工程，这不仅大大增加了函数实现的负担，重复的代码也严重偏离了 DRY 原则。

---

**说明** DRY——Don't Repeat Yourself Principle，直译为“不要重复自己”。简而言之，就是不要写重复的代码。DRY 利用的方法就是抽象：把共同的事物抽象出来，把代码抽取到一个地方去，这样就可以避免重复写代码。

---

C/C++ 对于这个问题采取的策略是“把问题交给程序员全权处理”。程序员既然享受了隐式转换所带来的便利，那么如果出现错误也是程序员需要负责的问题。权利与义务对等，这也算得上合情合理。但在程序员的眼里，这样的处理方式却不能让他们满意。后来 C++ 设计者意识到了这个问题，于是提供了控制隐式转换的两条有效途径：

#### □ 使用具名转换函数

来看一段代码：

```
class Rational
{
public:
    Rational(int numerator = 0, int denominator = 1)
        :m_num(numerator), m_den(denominator) {}

    operator double() const
    {
        return ((double)m_num/(double)m_den);
    }
private:
    int m_num;
    int m_den;
};

Rational r(1,2);
cout<<r<<endl;
```

上面代码的本意是打印类似 n/m 的形式，可是结果输出的却是 0.5。问题出现在哪里？

当调用 operator<< 时，编译器会发现没有合适的函数存在，所以它就试图找到一个合适的隐式类型转换顺序，以使函数得到正常调用。本来程序中并不存在将 Rational 转为其他类型的转换规则，但是 Rational::operator double 函数告诉编译器 Rational 类型可以转换为 double 类型，所以就有了上述结果的出现。为了避免此类问题的出现，建议使用非 C/C++ 关键字的具名函数，代码如下所示：

```
class Rational
{
public:
    Rational(int numerator = 0, int denominator = 1);
    operator as_double() const;
private:
    int m_num;
    int m_den;
};

Rational r(1,2);
cout<<r<<endl; // 提示无 operator<< Rational 重载函数
```

#### □ 使用 explicit 限制的构造函数

这种方式针对的是具有一个单参数构造函数的用户自定义类型。代码如下所示：

```
class Widget
{
public:
    Widget( unsigned int factor );
    Widget( const char* name, const Widget* other = NULL );
};
```

上述代码片段中，用户自定义类型 Widget 的构造函数可以是一个参数，也可以是两个参数。具有一个参数时，其参数类型可以是 unsigned int，亦可以是 char\*。所以这两种类型的数据均可以隐式地转换为 Widget 类型。控制这种隐式转换的方法很简单：为单参数的构造函数加上 explicit 关键字：

```
class Widget
{
    explicit Widget(unsigned int factor);
    explicit Widget(const char* name, const Widget* other = NULL);
};
```

#### 请记住：

提防隐式转换所带来的微妙问题，尽量控制隐式转换的发生；通常采用的方式包括：(1) 使用非 C/C++ 关键字的具名函数，用 operator as\_T() 替换 operator T() (T 为 C++ 数据类型)。(2) 为单参数的构造函数加上 explicit 关键字。

## 建议 18：正确区分 void 与 void\*

void 及 void 指针类型对于许多 C/C++ 语言初学者，甚至是部分有经验的程序员来说都是一个谜，它让人云里雾里，不甚清晰，因此在使用时也会出现一些这样那样的问题。也许在进入 C/C++ 语言精彩世界的第一刻就认识了 void 和 void\*，可是它们的具体含义到底是什么呢？

void 是“无类型”，所以它不是一种数据类型；void \* 则为“无类型指针”，即它是指向无类型数据的指针，也就是说它可以指向任何类型的数据。

从来没有人会定义一个 void 变量，如果真的这么做了，编译器会在编译阶段清晰地提示，“illegal use of type 'void'"。void 体现的是“有与无”的问题，要先“有”了，在非 void 的前提下才能去讨论这个变量是什么类型的，此哲学思想渗透于小小 void 的使用与设计中。

void 发挥的真正作用是限制程序的参数与函数返回值。在 C/C++ 语言中，对 void 关键字的使用做了如下规定：

(1) 如果函数没有返回值，那么应将其声明为 void 类型。

在 C 语言中，凡不加返回值类型限定的函数，就会被编译器作为返回整型值处理。但是许多程序员却误以为其为 void 类型。例如：

```
Add ( int a, int b );
int main()
{
    printf ( "1010 + 1001 = %d", Add ( 1010, 1001 ) );
    return 0;
}
Add ( int a, int b )
{
    return a + b;
}
```

程序运行的结果为：2 + 3 = 5。这个结果更加明确地说明了函数返回值为 int 类型，而非 void。

在林锐博士的《高质量程序设计指南——C++/C 语言（第 3 版）》一书中曾提到：“C++ 语言有很严格的类型安全检查，不允许上述情况（指函数不加类型声明）发生”。但是在一些较老的编译器（比如 VC++6.0）中上述 Add 函数的编译无错也无警告且运行正确，所以不能将严格的类型检查这样的重任完全交给编译器。

为了避免出现混乱，在编写 C/C++ 程序时，必须对任何函数都指定其返回值类型。如果函数没有返回值，则要声明为 void。这既保证了程序良好的可读性，也满足了编程规范性的要求。

(2) 如果函数无参数，那么声明函数参数为 void。

正如我们原先遇到的情况一样，如果在调用一个无参数函数时，一不小心为其设定了参数：

```

int TestFunction(void)
{
    return 2012;
}
int main()
{
    int thisYear = TestFunction(2011);
    // processing code
    return 0;
}

```

那么在 C++ 编译器中编译代码时则会出错，提示“'TestFunction' : function does not take 1 parameters”。而在 C 语言中，据说它能编译通过且能正确执行。之所以说是据说，是因为本人没有在 C 环境下实验证实这种情况，请原谅我的懒惰，因为我真的不想去碰 Turbo C，虽然那也曾经是我的入门开发环境。

所以，在 C/C++ 中，若函数不接受任何参数，一定要指明参数为 void。就算写的是 C 函数，为了将来的兼容性，请不要省略这个 void。

接下来说说特殊指针类型 void\*。

众所周知，如果存在两个类型相同的指针 pInt1 和 pInt2，那么我们可以直接在二者间互相赋值；如果是两个指向不同数据类型的指针 pInt 和 pFloat，直接相互赋值则会编译出错，必须使用强制运算符把赋值运算符右侧的指针类型转换为左侧的指针类型，这一点在建议 11 中已经解释得很清晰，代码如下所示：

```

int *pInt;
float *pFloat;
pInt = pFloat; // 编译出错，提示“'=': cannot convert from 'int *' to 'float *'”
pInt = (float *)pFloat; // 正确，需强制转型

```

而 void \* 则不同，任何类型的指针都可以直接赋值给它，无须强制转型，如下所示：

```

void *pVoid;
float *pFloat;
pVoid = pFloat; // 正确，无需强制转型

```

但这种转换在 C++ 中并不是双向的，在不使用强制转型的前提下，不允许将 void \* 赋给其他类型的指针，如下所示：

```

void *pVoid;
float *pFloat;
pFloat = pVoid; // 错误，编译失败，提示“'=': cannot convert from 'void *' to 'float *'”

```

对于一般数据类型的指针，我们可以进行加减等算法操作，但是按照 ANSI 标准，对 void 指针进行算法操作是不合法的：

```
// 分别采用 VC++ 编译器和 Gcc 编译器进行验证
```

```

int * pInt;
pInt ++;           // 正确, pInt 指针增大 sizeof(int)
pInt += 2;          // 正确, pInt 指针增大 2*sizeof(int)

void * pVoid;
pVoid ++;          // 错误, error C2036: "pVoid*": 未知的大小
pVoid += 1;         // 错误

```

ANSI 标准之所以这样认定, 是因为只有在确定了指针指向数据类型的大小之后, 才能进行算法操作。但是大名鼎鼎的 GNU 则有不同的规定, 它指定 void \* 的算法操作与 char \* 一致。所以在上面代码片段中出现错误的代码在 GNU 编译器中能顺利通过编译, 并且能正确执行。虽然 GNU 较 ANSI 更开放, 提供了对更多语法的支持, 但是 ANSI 标准更加通用, 更加“标准”, 所以在实际设计中, 还是应该尽可能地迎合 ANSI 标准。在实际的程序设计中, 为迎合 ANSI 标准, 并提高程序的可移植性, 可以采用以下方式进行代码设计:

```

void * pVoid;
(char *)pVoid ++;      // ANSI: 正确; GNU: 正确
(char *)pVoid += 2;     // ANSI: 错误; GNU: 正确

```

如果函数的参数可以是任意类型指针, 那么应声明其参数为 void \*, 最典型的例子就是我们熟知的内存操作函数 memcpy 和 memset 的原型:

```

void * memcpy(void *dest, const void *src, size_t len);
void * memset ( void * buffer, int c, size_t num );

```

仔细品味, 就会发现这样的函数设计是多么富有学问, 任何类型的指针都可以传入 memcpy 和 memset 中, 传出的则是一块没有具体数据类型规定的内存, 这也真实地体现了内存操作函数的意义。如果类型不是 void \*, 而是 char \*, 那么这样的 memcpy 和 memset 函数就会与数据类型产生明显联系, 纠缠不清, 这不是一个通用的、“纯粹的、脱离低级趣味”的函数设计!

### 请记住:

void 与 void\* 是一对极易混淆的双胞胎兄弟, 但是它们在骨子里却存在着质的不同, 区分它们, 按照一定的规则使用它们, 可以提高程序的可读性、可移植性。仔细体会, 还会发现隐藏在它们背后的设计哲学。

## 第 2 章 从 C 到 C++, 需要做出一些改变

C++ 语言之父当初设计该语言的初衷是“a better C”，所以 C++ 一般被认为是 C 的超集合，但是不要因此而误以为，“这意味着 C++ 兼容 C 语言的所有东西”。作为一种欲与 C 兼容的语言，C++ 保留了一部分过程式语言的特点，大部分的 C 代码可以很轻易地在 C++ 中正确编译，但仍有少数差异，导致某些有效的 C 代码在 C++ 中无法通过编译。

因此，从 C 到 C++，我们要因为这些差异而做出一些改变，我们应当熟悉这些差异，使用原有的丰富的 C 库为现在的 C++ 工程更好地服务。

### 建议 19：明白在 C++ 中如何使用 C

首先，分析下面的代码片段：

```
// Demo.h
#ifndef SRC_DEMO_H
#define SRC_DEMO_H
extern "C"
{
    ... // do something
}
#endif // SRC_DEMO_H
```

显然，头文件中的编译宏“#ifndef SRC\_DEMO\_H、#define SRC\_DEMO\_H、#endif”的作用是防止该头文件被重复引用（详见建议 9）。那么，extern "C" 又有什么特殊的作用呢？暂且先留着这个疑问。

C++ 语言被称做“C with classes”、“a better C”或“C 的超集合”，但是并非兼容 C 语言的所有东西，两者之间的“大同”并不能完全抹杀其中的“小异”。最常见的差异就是，C 允许从 void 类型指针隐式转换成其他类型的指针，但 C++ 为了安全考虑明令禁止了此种行为。比如：如下代码在 C 语言中是有效的：

```
// 从 void* 隐式转换为 double*
double *pDouble = malloc(nCount * sizeof(double));
```

但要使其在 C++ 中正确运行，就需要显式地转换：

```
double *pDouble = (double *)malloc(nCount * sizeof(double));
```

除此之外，还有一些其他的可移植问题，比如 new 和 class 在 C++ 中是关键字，而在 C

中，却可以作为变量名。

若想在 C++ 中使用大量现成的 C 程序库，就必须把它放到 `extern "C" { /* code */ }` 中。到这里，也许大家会茅塞顿开，明白本建议开始列出的代码片段中那些宏的真实作用了。当然，具有强烈好奇心的读者也许会有了新的问题：为什么加上 `extern "C" { /* code */ }` 就好使了呢？这是一个问题。下面就分析一下隐藏在这个现象背后的真实原因：C 与 C++ 具有不同的编译和链接方式。C 编译器编译函数时不带函数的类型信息，只包含函数符号名字；而 C++ 编译器为了实现函数重载，在编译时会带上函数的类型信息。假设某个函数的原型为：

```
int Function(int a, float b);
```

C 编译器把该函数编译成类似 `_Function` 的符号（这种符号一般被称为 mangled name），C 链接器只要找到了这个符号，就可以连接成功，实现调用。C 编译链接器不会对它的参数类型信息加以验证，只是假设这些信息是正确的，这正是 C 编译链接器的缺点所在。而在强调安全的 C++ 中，编译器会检查参数类型信息，上述函数原型会被编译成 `_Function_int_float` 这样的符号（也正是这种机制为函数重载的实现提供了必要的支持）。在连接过程中，链接器会在由函数原型所在模块生成的目标文件中寻找 `_Function_int_float` 这样的符号。

解决上述矛盾就成了设置 `extern "C"` 这一语法最直接的原因与动力。`extern "C"` 的作用就是告诉 C++ 链接器寻找调用函数的符号时，采用 C 的方式，让编译器寻找 `_Function` 而不是 `_Function_int_float`。

要实现在 C++ 中调用 C 的代码，具体方式有以下几种：

(1) 修改 C 代码的头文件，当其中含有 C++ 代码时，在声明中加入 `extern "C"`。代码如下所示：

```
/*C 语言头文件: CDemo.h */
#ifndef C_SRC_DEMO_H
#define C_SRC_DEMO_H
extern "C" int Function(int x,int y);
#endif // C_SRC_DEMO_H

/* C 语言实现文件: CDemo.c */
#include "CDemo.h"
int Function ( int x, int y )
{
    .
    . .
    ... // processing code
}

// C++ 调用文件
#include "CDemo.h"
int main()
```

```
{
    Function (2,3);
    return 0;
}
```

(2) 在C++代码中重新声明一下C函数，在重新声明时添加上extern "C"。代码如下所示：

```
/*C语言头文件: CDemo.h */
#ifndef C_SRC_DEMO_H
#define C_SRC_DEMO_H
extern int Function(int x,int y);
#endif // C_SRC_DEMO_H

/* C语言实现文件: CDemo.c */
#include "CDemo.h"
int Function ( int x, int y )
{
    ... // processing code
}

// C++ 调用文件
#include "CDemo.h"
extern "C" int Function(int x,int y);

int main()
{
    Function (2,3);
    return 0;
}
```

(3) 在包含C头文件时，添上extern "C"。代码如下所示：

```
/*C语言头文件: CDemo.h */
#ifndef C_SRC_DEMO_H
#define C_SRC_DEMO_H
extern int Function(int x,int y);
#endif // C_SRC_DEMO_H

/* C语言实现文件: CDemo.c */
#include "CDemo.h"
int Function ( int x, int y )
{
    ... // processing code
}

// C++ 调用文件
extern "C" {
#include "CDemo.h"
}
```

```
int main()
{
    Function (2, 3);
    return 0;
}
```

使用中，谨记：extern "C" 一定要加在 C++ 的代码文件中才能起作用。

#### 请记住：

若想在 C++ 中使用大量现成的 C 程序库，实现 C++ 与 C 的混合编程，那你必须了解 extern "C" 是怎么回事儿，明白 extern "C" 的使用方式。

### 建议 20：使用 memcpy() 系列函数时要足够小心

memcpy()、memset()、memcmp() 等这些内存操作函数经常会帮我们完成一些数据复制、赋值等操作。因为在 C 语言中，无论是内置类型，还是自定义的结构类型（struct），其内存模型对于我们来说都是可知的、透明的。所以，我们可以对该对象的底层字节序列一一进行操作，简单而有效。代码片段如下所示：

```
struct STUDENT
{
    char _name[32];
    int _age;
    bool _gender;
};

STUDENT a = {"Li Lei", 20, true};
STUDENT b = {"Han MeiMei", 19, false};

int len = sizeof(STUDENT);
STUDENT c;
memset(&c, 0, len);
memcpy(&c, &a, len);

char *data = (char*)malloc(sizeof(char)*len);
memcpy(data, &b, len);
```

在 C++ 中，我们把传统 C 风格的数据类型叫做 POD (Plain Old Data) 对象，即一种古老的纯数据。在 C 的世界里根本没有 POD 这一概念，因为 C 的所有对象都是 POD。一般来说，POD 对象应该满足如下特性：其二进制内容是可以随意复制的，无论在什么地方，只要其二进制内容存在，我们就能准确无误地还原出 POD 对象。正是由于这个原因，对于任何 POD 对象，我们都可以放心大胆地使用 memset()、memcpy()、memcmp() 等函数对对象的内

存数据进行操作。

然而在 C++ 中，每个人都要十二分的注意了。因为 C++ 的对象可能并不是一个 POD，所以我们无法像在 C 中那样获得该对象直观简洁的内存模型。对于 POD 对象，我们可以通过对象的基地址和数据成员的偏移量获得数据成员的地址。但是 C++ 标准并未对非 POD 对象的内存布局做任何定义，对于不同的编译器，其对象布局是不同的。而在 C 语言中，对象布局仅仅会受到底层硬件系统差异的影响。

针对非 POD 对象，其序列化会遇到一定的障碍：由于对象的不同部分可能存在与不同的地方，因而无法直接复制，只能通过手工加入序列化操作代码来处理对象数据，很麻烦。但是针对 POD 对象，这一切将变得不再困难：从基地址开始，直接按对象的大小复制数据，或传输，或存储，随意处理。

为什么 C++ 中的对象有可能不是一个 POD 呢？这还要从 C++ 的重要特征之一——动态说起。动态的一个基本支撑技术就是虚函数。在使用虚函数时，类的每一次继承都会产生一个虚函数表（vtable），其中存放的是指向虚函数的指针。这些虚函数表必须存放在对象体中，也就是和对象的数据存放在一起。因而，对象数据在内存里并不是以连续的方式存放的，而是被分割成了不同的部分，甚至“身首异处”<sup>Θ</sup>。既然对象数据不再集中在一起，如果此时再贸然使用 memcpy()、memset() 函数，那么所带来的后果将不可预计。

---

#### 请记住：

要区分哪些数据对象是 POD，哪些是非 POD。由于非 POD 对象的存在，在 C++ 中使用 memcpy() 系列函数时要保持足够的小心。

---

## 建议 21：尽量用 new/delete 代替 malloc/free

在 C 语言中，我们已经熟悉利用 malloc/free 来管理动态内存，而在 C++ 中，我们又有了新的工具：new/delete。你不禁会产生疑问——有了 malloc/free 为什么还要 new/delete 呢？使用 malloc/free 和使用 new/delete 又有什么区别呢？首先来分析一下下面的代码片段：

```
class Object
{
public:
    Object()
    {
        cout << "Hello, I was born." << endl;
    }
    ~Object()
    {
```

---

<sup>Θ</sup> 关于虚函数的布局请参见《Inside the C++ Object Model》。

```

        cout << "Bye, I am died." << endl;
    }
    void Hello()
    {
        cout << "I am Object."<<endl;
    }
};

int main()
{
    cout << " Using Malloc & Free..." << endl;
    Object* pObjectA = (Object*)malloc(sizeof(Object));
    pObjectA->Hello();
    free pObjectA;

    cout << " Using New & Delete..." << endl;
    Object* pObjectB = new Object;
    pObjectB->Hello();
    delete pObjectB;

    return 0;
}

```

代码运行的结果为：

```

Using Malloc & Free...
I am Object.
Using New & Delete...
Hello, I was born.
I am Object.
Bye, I am died.

```

通过结果我们可以得知：new/delete 在管理内存的同时调用了构造和析构函数；而 malloc/free 仅仅实现了内存分配与释放。接下来，我们进行详细讨论。

malloc/free 是 C/C++ 语言的标准库函数，而 new/delete 是 C++ 的运算符。它们都可用于申请动态内存和释放内存。

由于 malloc/free 是库函数，所以需要对应的头文件库函数支持。对于非内置数据类型的对象，用 malloc/free 无法满足创建动态对象的要求。这是因为对象在创建的同时要自动执行构造函数，对象在消亡之前则要自动执行析构函数。由于 malloc/free 不是运算符，不受编译器的控制管辖，所以不能够把执行构造函数和析构函数的任务强加于 malloc/free 上。而 new/delete 就不同了，它们是保留字，是操作符，它们和“+”、“-”、“\*”、“/”有着一样的地位。new 不仅能完成动态内存分配，还能完成初始化工作，稳妥地构造对象；delete 不仅能完成内存的释放，还能进行对象的清理。举个形象的例子：通过 new 建造出来的是一栋房子，可以直接居住；而通过 malloc 申请到的仅仅是一块地皮，要想成为房子，还需要做出另外的努力。

`malloc` 的语法是：

```
指针名 = (数据类型 *) malloc (长度) ; // (数据类型 *) 表示指针
```

`new` 的语法是：

```
指针名 = new 类型 (参数) ; // 单个对象
指针名 = new 类型 [个数] ; // 对象数组
```

`malloc` 函数返回的是 `void *` 类型，如果写成：`ClassA* p = malloc (sizeof(ClassA));`，程序则无法通过编译，会抛出这样的错误信息：“不能将 `void*` 赋值给 `ClassA *` 类型变量”。所以必须通过 `(ClassA *)` 来进行强制转型。相较而言，`new` 则不存在强制转型的问题，而且书写更为简单。总结起来，`malloc` 与 `new` 之间的区别主要有以下几点：

- `new` 是 C++ 运算符，而 `malloc` 则是 C 标准库函数。
- 通过 `new` 创建的东西是具有类型的，而 `malloc` 函数返回的则是 `void*`，需要进行强制转型。
- `new` 可以自动调用对象的构造函数，而 `malloc` 不会。
- `new` 失败时会调用 `new_handler` 处理函数，而 `malloc` 失败则直接返回 `NULL`。

`free` 与 `delete` 之间的区别则只有以下两点：

- `delete` 是 C++ 运算符，`free` 是 C 标准库函数。
- `delete` 可以自动调用对象的析构函数，而 `malloc` 不会。

针对内置类型而言，因为没有对象的构造与析构，所以 `malloc/free` 除了需要强制转型之外，和 `new/delete` 所做的工作无异，用哪一个只是涉及个人喜好而已。

```
//declaring native type
int* i1 = new int;
delete i1;

int* i2 = (int*) malloc(sizeof(int));
free(i2);

//declaring native type array
char* c1 = new char[10];
delete[] c1;

char* c2 = (char*) malloc(sizeof(char)*10);
free(c2);
```

既然提到了 `malloc/free`，不能不提一下 `realloc`。使用 `realloc` 函数可以重新设置内存块的大小，而在 C++ 中没有类似于 `realloc` 这样的替代品。如果出现上述需求，所做的就是，释放原来的内存，再重新申请。

既然 `new/delete` 的功能不仅赶上而且超越了 `malloc/free`，那为什么 C++ 标准中没有把

malloc/free 淘汰出局呢？这是因为 C++ 要遵守“对 C 兼容”的承诺，要让一些有价值的包含 malloc/free 函数库的 C 程序在 C++ 中得到重用。所以，在 C++ 中，new/delete 和 malloc/free 一直并存着。

不过，将 malloc/free 和 new/delete 混合使用绝对不是什么好主意。Remember that, to new is C++; to malloc is C; and to mix them is sin. 如果用 free 来释放通过 new 创建的动态对象，或者用 delete 释放通过 malloc 申请的动态内存，其结果都是未定义的。换句话说，不能保证它会出现什么问题。如果程序在关键时刻就因为这个在重要客户面前出现问题，那么懊悔恐怕已经来不及了。

#### 请记住：

- (1) 不要企图用 malloc/free 来完成动态对象的内存管理，应该用 new/delete。
- (2) 请记住：new 是 C++ 的，而 malloc 是 c 的。如果混淆了它们，那将是件蠢事。所以 new/delete 必须配对使用，malloc/free 也一样。

## 建议 22：灵活地使用不同风格的注释

注释，可以说是计算机程序中不可或缺的一个部分，它的存在让我们阅读程序代码、理解作者意图变得相对容易（当然，这里说的是具有良好注释的代码）。在 C/C++ 语言中，存在着两种不同的注释语法：

□ 旧有的 C 风格的注释：/\* describe your purposes \*/

□ 新式的 C++ 风格的注释：// describe your purposes

既然两种注释语法都有效，选择哪一种呢？C 风格的还是 C++ 风格的呢？

很多的 C++ 书籍推荐我们使用新式的 C++ 风格注释语法，比如受 C++ 程序员顶礼膜拜的经典书籍《Effective C++》在条款 4 中的建议就是如此。为此，Scott Meyers 还给出了一定的理由——由“内嵌注释结束符”引发的“惨案”<sup>Θ</sup>：

```
/* C 风格的注释 */
if(a>b)
{
    /*     int temp =a;    /* swap a and b */
    a = b;
    b = temp;
}
*/
```

<sup>Θ</sup> 此段代码取自 Scott Meyers 的《Effective C++》，在此表示感谢。

```
// C++ 风格的注释
if(a>b)
{
    //int temp =a;    // swap a and b
    //a = b;
    //b = temp;
}
```

当程序员因为某些特殊原因而采用C风格的注释语法将上述代码进行注释时，由于原代码中存在原有的内嵌注释，导致注释过早地找到结束匹配符，使代码注释失效，出现编译错误。而C++风格的注释则不会出现类似的麻烦。

然而，正如一个硬币有两面，任何东西都是有利有弊的。让程序员更加便利与轻松才是硬道理。使用注释亦然。还是先看下面的一段代码：

```
/*
 * new.cxx - defines C++ new routine
 *
 * Copyright (c) Microsoft Corporation. All rights reserved.
 *
 *Purpose:
 *      Defines C++ new routine.
 ****
#endif _SYSCRT
#include <cruntime.h>
#include <crtdbg.h>
#include <malloc.h>
#include <new.h>
#include <stdlib.h>
#include <winheap.h>
#include <rtcsup.h>
#include <internal.h>

void * operator new( size_t cb )
{
    void *res;
    for (;;) {
        // allocate memory block
        res = _heap_alloc(cb);
        // if successful allocation, return pointer to memory
        if (res)
            break;
        // call installed new handler
        if (!_callnewh(cb))
            break;
        // new handler was successful -- try to allocate again
    }
    RTCCALLBACK(_RTC_Allocate_hook, (res, cb, 0));
    return res;
```

```

    }
#else /* _SYSCRT */

```

这是 VC++ 库中 new.cpp 文件中的部分代码。在注释方面，这段代码中有很多值得我们学习的地方。

#### □ 版权和版本声明，使用 C 风格的 /\* \*/

标准化的代码有很多必不可少的东西，比如版权信息、文件名称、标识符、摘要、当前版本号、作者 / 修改者、完成日期、版本历史信息，等等。这些信息不会为我们的代码运行带来任何的改进，但是可以提高了代码的可读性，方便代码的维护。如此繁缛的信息，可能多达十几行，此时如果使用 C++ 风格的注释语法，那么就得记得在每一行的开始都写下两个 “/” 符。那此时何不采用更加简单便利的 /\* \*/ 呢？

#### □ 内嵌注释用 //

内嵌注释一般出现在代码主体内。此时，建议使用新式的 C++ 风格的注释语法。最直接的原因就是避免出现那些由“内嵌注释结束符”引发的“惨案”。不过，在这种情况下，出于调试原因用 /\* \*/ 注掉一块代码，也不会出现什么问题。

#### □ 宏尾端的注释用 /\* \*/

Scott Meyers 对于注释语法的使用还提出了一个问题：一些“古董”级的、只针对 C 编译器而写的预处理器不能识别 C++ 风格的注释，所以下面的代码就不能按照预期那样正常运行，它们会把注释当成宏的一部分：

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

虽然使用这样的“古董”预处理器的人近乎绝迹，但是保不齐会出现一个特例。所以为了保证百分之百不出错，建议在宏尾端的注释使用 C 风格的注释语法：

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

除此之外，还有一个特别的使用情形：默认参数函数的定义。代码片段如下所示：

```

// 声明文件
class A
{
public:
    void Function( int para1, int para2 = 0 );
};

// 实现文件
void A::Function( int para1, int para2 /* = 0 */ )
{
    // processing code
}

```

我们一般将类的声明与实现进行分离，放置在不同的文件之中。此时如果函数存在默认

参数，它只能出现在声明中，不过，在实现中缺少默认参数的说明可能会影响我们对函数的设计或理解，所以有必要在实现中对默认参数进行一些说明。使用C风格的注释语法按照上述形式进行说明确实是一个值得推荐的方式。在这种情形下，C++风格的注释变得无能为力了。

灵活地使用两种形式的注释方式，在保证代码鲁棒性<sup>⊖</sup>、可读性的同时，尽量使程序员获得更多轻松与便利。

---

#### 请记住：

C风格的注释/\* \*/与C++风格的注释//在C++语言中同时存在，所以我们可以充分地利用两种注释的长处，并注意可能存在的问题，这会让我们的编码变得更加轻松、便利、高效！

---

### 建议 23：尽量使用C++标准的iostream

IO是我们最基本的需求之一。比如当我们进入C++世界时所接触的第一个程序HelloWorld，采用printf()或operator<<都可以。所以，我们会有如下的版本：

```
//Version 1
#include < stdio.h >
int main()
{
    printf("Hello World");
    return 0;
}

//Version 2
#include < cstdio >
int main()
{
    std::printf("Hello World");
    return 0;
}

//Version 3
#include < iostream.h >
int main()
{
    cout<<"Hello World";
}
```

---

<sup>⊖</sup> 此处的鲁棒性不同于一般意义上的鲁棒性，这里主要是指代码可在不同平台上正确执行，不因注释而导致代码出现错误。

```

        return 0;
    }

//Version 4
#include <iostream>
using namespace std;
int main()
{
    cout<<"Hello World";
    return 0;
}

```

stdio.h、cstdio 两个头文件中都有 printf() 的定义，而 iostream.h 和 iostream 中也都有 operator<< 的定义。是 stdio.h 还是 cstdio？是 iostream.h 还是 iostream？是 printf() 还是 operator<<？这些都值得思考。

关于 File.h 和 File，这还得从 C++ 标准库说起。C++ 标准程序库涵盖范围相当大，包含了许多好用的功能，所以，标准库与第三方提供程序库中的类型名称和函数名称发生名称冲突的可能性大大增加。为了避免这个问题的发生，标准委员会决定让标准库中的内容都披上 std 的外衣，放在 std 名空间中。但是这么做同时又带来了一个新的兼容性问题：很多 C++ 程序代码依赖的都是没有用 std 包装的 C++ “准” 标准库，例如 iostream.h 等，如果将原有的 iostream.h 贸然代替掉，那肯定会引起众多程序员的抗议。

标准化委员会最后决定设计一种新的头文件名来解决这个问题。于是，他们把 C++ 头文件 File.h 中的 .h 去掉，将 File 这样没有后缀的头文件名分配给那些用 std 包装过的组件使用；而旧有的 File.h 保持不变，仅仅在标准中声明不再支持它，顺势把问题丢给了广大厂商，堵住了那些老程序员抗议的嘴。同样，对 C 的头文件也做了相同的处理，在前面加上了一个字母 c 以示区分。因为 C++ 标准还要遵守“对 C 兼容”这个契约，备受“歧视”的旧有的 C 头文件“侥幸存活”了下来。虽然标准化委员会选择抛弃那些旧有的 C++ 头文件，但是各大厂商为了各自的商业利益，却依然选择了对旧有 C++ 头文件的支持。

因此就出现了类似 stdio.h 和 cstdio、iostream.h 和 iostream 这样的双胞胎：

```

// 标准化以前 C++ 中的 C 标准库，标准 C 的头文件继续获得支持，这类文件的内容并
// 未放在 std 中
#include<stdio.h>

// 标准化后经过改进的 C 的标准库，C 的标准库对应的新式 C++ 版本，这类头文件的
// 内容也有幸穿上了 std 的外衣
#include<cstdio>

// 标准化以前的头文件，这些头文件的内容将不处于 namespace std 中
#include<iostream.h>

// 标准化以后的标准头文件，它提供了和旧有的头文件相同的功能，但它的内容都并

```

```
// 入了namespace std 中, 从而有效避免了名称污染的问题
#include<iostream>
```

其实标准化以后标准程序库的改动并不只有这些，很多标准化的组件都被模板化了。具体参见侯捷翻译的《C++ 标准程序库》。

接下来再说说 printf() 和 operator<< 的问题。首先通过上面的讲解我们可以知道 printf() 函数继承自 C 标准库，而 operator<< 是标准 C++ 所独享的。对于 printf() 函数，大家肯定很熟悉，它是可移植的、高效的，而且是灵活的；但是正如建议 15 中所说的那样，因为 printf()、scanf() 函数不具备类型安全检查，也不能扩充，所以并不完美；而 C 语言遗留的问题在 C++ 的 operator<< 中得到了很好的解决，换句话说，printf() 的缺点正是 operator<< 的长处。现在再去回顾建议 15 中提到的关于 Student 类型对象打印的问题，用 operator<< 就可以完美解决了：

```
class Student
{
public:
    Student(string& name, int age, int score);
    ~Student();
private:
    string m_name;
    int    m_age;
    int    m_score;
friend ostream& operator<<( ostream& s, const Student& p );
};

ostream& operator<<( ostream& s, const Student& p )
{
    s<<p. m_name<<" "<<p. m_age<<" "<<p. m_score;
    return s;
}
// 调用 operator<<
Student XiaoLi("LiLei", 23, 97);
cout<< XiaoLi;
```

当然了，相比 C++ iostream 程序库中的类，C 中的 stream 函数也并不是一无是处的：

- (1) 一般认为 C stream 函数生成的可执行文件更小，有着更高的效率；Scott Meyers 在《More Effective C++》的条款 23 中的测试也很好地证明了这一点；
- (2) C++ iostream 程序库中的类会涉及对象构造、析构的问题，而 C stream 函数没有这些，所以不会像前者那样因为构造函数带来不必要的麻烦。
- (3) C stream 函数有着更强的可移植能力。

对于一般应用程序而言，这三条优点还不足以打动他们，抛弃 C++ iostream 程序库，转而投向 C stream 函数。

---

请记住：

C++ iostream 程序库中的类与 C stream 函数虽然各有优点，但是一般推荐使用前者，因为类型安全与可扩充性对于我们更有吸引力，所以，建议使用 #include< iostream >，而不是 #include< stdio.h >、#include< cstdio >、#include< iostream.h >。

---

## 建议 24：尽量采用 C++ 风格的强制转型

在建议 11 中，我们详细讲述了强制转型存在的一些问题，并建议在代码编写过程中尽量避免使用这个招人讨厌的东西。然而，正如哲学中所讲的一样：存在的即是合理的。强制转型肯定具有它存在的意义。在某些情形下我们必须求助于这个“讨厌鬼”，以帮助我们更好地完成程序设计。

比如，const 属性的去除（请不要纠结于下面示例函数的“不良”设计）：

```
class CStudent{};
const CStudent* GetCertainStudent(const std::string& name)
{
    CStudent* p = new CStudent(name);
    return p;
}

CStudent* p = GetCertainStudent("Li Lei");
```

在 VC++ 下编译，编译器会报错：

```
error C2440: "初始化": 无法从 "const CStudent *" 转换为 "CStudent **"
```

此时我们就只能求助于 const\_cast 了：

```
CStudent* p = const_cast<CStudent*>(GetCertainStudent("Li Lei"));
```

这里需要提醒的是，不要随意去除变量的 const 属性，除非是经过深思熟虑后不得不这样做。

在 C/C++ 编程中，新旧两种风格的强制转型同时存在。当强制转型已成为不可避免的定局时，安全性相对高的 C++ 风格的强制转型更为可取。

首先，新风格的强制转型不再像 C 风格的强制转型那样简单粗暴，在代码中它们更容易识别，更容易找到这些类型系统破坏者的藏匿之处。

其次，新风格的强制转型针对性更强，它针对特定的目的进行了特别的设计。如果对这些特别设计的理解不是很清晰，请返回去看看建议 11。这样能让程序员更清晰地了解强制转型的目的，同时使利用编译器诊断使用错误成为可能。

---

**请记住：**

如果实在不能避免，建议采用安全性较高的C++风格的强制转型形式。新风格更容易被注意，而且具有一定的针对性。

---

## 建议 25：尽量用 const、enum、inline 替换#define

在建议4中，我们已经详细说明了在使用宏时应注意的一些问题。“表面似和善、背后一长串”绝对是对宏的形象表述。宏的使用具有一些优点：能减少代码量（比如简单字符替换重复的代码），在某种程度上提供可阅读性（比如MFC的消息映射），提高运行效率（比如没有函数调用开销）。

然而谈到宏，绝对绕不开预处理器。把C/C++源码从源文件的形式变成可执行的二进制文件通常需要三个主要步骤：预处理→编译→链接。在预处理阶段，预处理器会完成宏替换。因为此过程并不在编译过程中进行，所以难以发现潜在的错误及其他代码维护问题，这会使代码变得难以分析，繁于调试。所以，宏——这个C语言中的“大明星”在C++的世界里却变成了程序员深恶痛绝的东西。因为#define的内容不属于语言自身的范畴，所以C++设计者为我们提供了替代宏的几大利器，建议我们尽量使用编译器管制下的const、enum、inline来实现#define的几大功能。如此看来，本建议的名称换做“尽量把工作交给编译器而非预处理器”或许更合适。

接下来分析一下#define的弊端，请看下面的代码片段：

```
#define PI 3.1415926
```

在预处理阶段，预处理器就完成了代码中符号PI的全部替换，因为这个过程发生在源代码编译以前，所以编译器根本接触不到PI这个符号名，这个符号名更不会被编译器列入到符号表中。如果因为在代码中使用了这个常量PI而引起问题，那这个错误将可能变得不易察觉，难以找到问题，出错信息只会涉及3.1415926，对PI则只字未提。

如果PI是在某个大家并不熟悉的或出自别人之手的头文件中定义的，那么寻找数值3.1415926的出处就如同大海捞针，费时费力。不过这一切也并非是不可避免的，解决的办法很简单，就是“使用常量来代替宏定义”：

```
const double PI = 3.1415926;
```

作为语言层面的常量，PI肯定会被编译器看到，并且会确保其进入符号表中，也就不会出现类似“3.1415926有错误”这样模糊不清的错误信息了。当出现问题时，我们也会有章可循，可以通过符号名顺藤摸瓜，消灭错误。另外，使用常量可以避免目标码的多份复制，也就是说生成的目标代码会更小。这是由于预处理器会对目标代码中出现的所有宏PI复制出

一份 3.1415926，而使用常量时只会为其分配一块内存。

在使用普通常量时，有一种特殊情形会让我们感觉棘手，那就是常量指针。用 `const` 去修饰指针的方式有多种，诸如：

```
const char* bookName = "150 C++ Tips";
char* const bookName = "150 C++ Tips";
const char* const bookName = "150 C++ Tips";
```

应该使用哪一种方式确实是一个需要明确的问题。`const` 修饰指针的规则可以简单地描述为：如果 `const` 出现在 \* 左边，表示所指数据为常量；如果出现在 \* 右边，表示指针自身是常量。需要注意的是，在头文件中定义常量指针时，是将指针声明为 `const` 了，而不是指针指向的数据。所以，如果定义一个指向常量字符串的常量指针，我们选择的就是最后一种，需要用两个 `const` 进行修饰。然而在定义指向常量字符串的常量指针时，用两个 `const` 修饰并不是我们推荐的形式。我们推荐使用更加安全、更加高级的 `const string` 形式：

```
const string bookName("150 C++ Tips");
```

作为 C++ 中最重要的概念，`class` 与很多其他的关键字都产生了联系，`const` 也肯定不会放过纠缠这个 C++ 主角的机会，所以就有了常量数据成员。定义常量数据成员的主要目的就是为了将常量的作用域限制在一个特定的类里，为了让限制常量最多只有一份，还必须将该常量用 `static` 进行修饰，例如：

```
class CStudent
{
private:
    static const int NUM_LESSONS = 5; // 声明常量
    int scores[NUM_LESSONS];           // 使用常量
};
```

注意，上述注释中说的是“声明常量”，而非“定义常量”，并且在声明的同时，完成了“特殊形式”的初始化。之所以谓之“特殊形式”，是因为我们熟悉的一般形式的初始化是不允许放在声明里的。这种“特殊形式”的初始化在 C++ 中被称为“类内初始化”。还有一点需要明确的是，在不同的编译器中对类内初始化的支持情况也不尽相同。在 VC++ 2010 中，并不是所有的内置类型都可以实现类内初始化，它只对整数类型（比如 `int`、`char`、`bool`）的静态成员常量才有效。如果静态成员变量是上述类型之外的其他类型，如 `double` 型，那么需要将该类的初始化放到其实现文件该变量的定义处，如下所示：

```
/* VC++ 2010 */
// CMathConstants 声明文件 (.h)
class CMathConstants
{
private:
    static const double PI;
```

```
};

// CMathConstants 实现文件 (.cpp)
const double CMathConstants::PI = 3.1415926;
```

而在GCC编译器中，内置的float、double类型的静态成员常量都可以采用类内初始化，如下所示：

```
/* Gcc 4.3 */
// CMathConstants 声明文件 (.h)
class CMathConstants
{
private:
    static const double PI = 3.1415926;
};
```

当然，如果不习惯类内初始化，讨厌其破坏了静态成员常量声明、定义的统一形式，可以选择将类内初始化全部搬到类实现文件中去，这也是我们比较推荐的形式。更何况早期的编译器可能不接受在声明一个静态的类成员时为其赋初值，那又何必去惹这些不必要的麻烦呢？

另外，如果编译器不支持类内初始化，而此时类在编译期又恰恰需要定义的成员常量值，身处如此左右为难的境地，我们该采取怎样的措施？那就求助于enum！巧用enum来解决这一问题。这一技术利用了这一点：枚举类型可以冒充整数给程序使用。代码如下所示：

```
// CStudent 声明文件 (.h)
class CStudent
{
private:
    enum{ NUM_LESSONS = 5 };
    int scores[NUM_LESSONS];
};
```

需要说明的一点是，类内部的静态常量是绝对不可以使用#define来创建的，#define的世界中没有域的概念。这不仅意味着#define不能用来定义类内部的常量，同时还说明它无法为我们带来任何封装效果。

#define的另一个普遍的用法是“函数宏”，即将宏定义得和函数一样，就像建议4中的：

```
#define ADD( a, b ) ((a)+(b))
#define MULTIPLE( a, b ) ((a)*(b))
```

这样的“函数宏”会起到“空间换时间”的效果，用代码的膨胀换取函数调用开销的减少。这样的宏会带来数不清的缺点，建议4中已经说得很清晰。如果使用宏，必须为此付出精力，而这是毫无意义的。幸运的是，C++中的内联函数给我们带来了福音：使用内联函数的模板，既可以得到宏的高效，又能保证类型安全，不必为一些鸡毛蒜皮的小问题耗费宝贵的精力。

```

template<typename T>
inline T Add(const T& a, const T& b)
{
    Return (a+b);
}

template<typename T>
inline T Multiple(const T& a, const T& b)
{
    Return (a*b);
}

```

这一模板创建了一系列的函数，方便高效，而且没有宏所带来的那些无聊问题。与此同时，由于 Add 和 Multiple 都是真实函数，它也遵循作用域和访问权的相关规则。宏在这个方面上确实是望尘莫及。

虽然建议尽量把工作交给编译器而非预处理器，而且 C++ 也为我们提供了足以完全替代#define 的新武器，但是预处理器并未完全退出历史舞台，并没有完全被抛弃。因为 #include 在我们的 C/C++ 程序中依旧扮演着重要角色，头文件卫士 #ifdef/#ifndef 还在控制编译过程中不遗余力地给予支持。但是如果将来这些问题有了更加优秀的解决方案，那时预处理器也许就真的该退休了。

---

#### 请记住：

对于简单的常量，应该尽量使用 const 对象或枚举类型数据，避免使用 #define。对于形似函数的宏，尽量使用内联函数，避免使用 #define。总之一句话，尽量将工作交给编译器，而不是预处理器。

---

## 建议 26：用引用代替指针

指针，可以通向内存世界，让我们具备了对硬件直接操作的超级能力。C++ 意识到了强大指针所带来的安全隐患，所以它适时地引入了一个新概念：引用。引用，从逻辑上理解就是“别名”，通俗地讲就是“外号”。在建立引用时，要用一个具有类型的实体去初始化这个引用，建立这个“外号”与实体之间的对应关系。

对于引用的理解与使用，主要存在两个的问题：

- 它与指针之间的区别。
- 未被充分利用。

引用并非指针。引用只是其对应实体的别名，能对引用做的唯一操作就是将其初始化，而且必须是在定义时就初始化。对引用初始化的必须是一个内存实体，否则，引用便成为了无根之草。一旦初始化结束，引用就是其对应实体的另一种叫法了。与指针不同，引用与地

址没有关联，甚至不占任何存储空间。代码如下所示：

```
int iNum = 12;
int &rNum = iNum;
int *pNum = &rNum; // 等同于 int *pNum = &iNum;
iNum = 2011; // rNum 的值也为 2011
```

由于引用没有地址，因此就不存在引用的引用、指向引用的指针或引用的数组这样的定义。据说尽管 C++ 标准委员会已经在讨论，认为应在某些上下文环境里允许引用的引用。但那都是将来的事，至少现在不可以，将来的事谁又说得准呢？

因为是别名，与实体所对应，所以引用不可能带有常量性和可挥发性。所以，下面的代码在编译时会出现问题：

```
int r = 10;
int & volatile s = r;
int & const m = r;
volatile int& t = r;
const int& n = r;
```

之所以说是出现问题，而不是错误，最主要的原因是各厂商编译器对于上述语法的容忍程度不同，有的会直接抛出错误并且编译失败，有的却只给出一个警告，比如：

□ gcc 4.3 给出错误

```
error: volatile/const 限定符不能应用到 'int&' 上
```

□ VC++ 2010 给出警告

```
warning C4227: 使用了记时错误：忽略引用上的限定符
```

对于加在引用类型前面的 `const` 或 `volatile` 修饰词，它们是符合编译器规则的，或者说被编译器选择性忽略了，没有什么问题（无 `error` 或 `warning`）。而对于指针，上述使用绝对不存在任何的问题。

C 阵营中那帮“顽固派”习惯在 C++ 工程里使用指针，并且以此为傲，现在该是为引用翻身的时候了。先看一个简单的示例：

```
void SwapData1(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

void SwapData2(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```

        *p = temp;
    }

void SwapData3(int& a, int& b)
{
    int temp = a;
    a = b;
    b = temp;
}

```

上述代码要实现的功能极为简单，就是交换两个数据的值。SwapData1() 是不能实现所设定的功能的，主要原因是函数内交换的只是实参的副本；而 SwapData2() 和 SwapData3() 则正确地实现了作者意图，其汇编代码如下：

```

SwapData1(a,b);
00F431EC mov      eax,dword ptr [b]
00F431EF push     eax
00F431F0 mov      ecx,dword ptr [a]
00F431F3 push     ecx
00F431F4 call    SwapData1 (0F4114Fh)
00F431F9 add     esp,8

SwapData2(&a,&b),
00F431FC lea      eax,[b]
00F431FF push     eax
00F43200 lea      ecx,[a]
00F43203 push     ecx
00F43204 call    SwapData2 (0F411EFh)
00F43209 add     esp,8

SwapData3(a,b);
00F4320C lea      eax,[b]
00F4320F push     eax
00F43210 lea      ecx,[a]
00F43213 push     ecx
00F43214 call    SwapData3 (0F4101Eh)
00F43219 add     esp,8

```

正如汇编代码中所示的那样，SwapData2() 和 SwapData3() 其实是一样的，都是对源数据进行操作。但是相较于 SwapData2() 而言，SwapData3() 的实现代码更加简洁清晰。这就是引用在传递函数参数时所具有的巨大优势。

再来看函数返回值方面，如果其返回值是引用类型，那么就意味着可以对该函数的返回值重新赋值。就像下面代码所示的数组索引函数一样：

```

template <typename T, int n>
class Array
{
public:
    T &operator [](int i)
    {

```

```
        return a_[i];
    }
    // ...
private:
    T a_[n];
};

Array<int, 10> iArray;
for(int i=0; i<10; i++)
    iArray[i] = i*2;
```

当然，上述代码可以使用指针重新实现，并且可以保证实现相同的功能，但是相比指针实现版，引用返回值使对数组索引函数的操作在语法上颇为自然，更容易让人接受。

也许有人认为，指针功能更强大，因为还有指向数组的指针、指向函数的指针，这里我要说的是，引用同样可以。引用在指向数组时还能够保留数组的大小信息，关于这方面的内容，在此就不多讲了。

---

#### 请记住：

从编码实践角度来看，指针和引用并无太多不同。在大多情况下，指针可由索引类型完美代替，并且其实现代代码更简洁清晰，更加易于理解。

---

# 第3章 说一说“内存管理”的那点事儿

在 C++ 的世界里，“烫”和“屯”是我们遇到得最多的两个汉字（限于 VC 用户）。可能有人不禁要问：这是为什么呢？

答案是：在 VC 中，栈空间未初始化的字符默认是 -52，补码是 0xCC。两个 0xCC，即 0xCCCC 在 GBK 编码中就是“烫”；堆空间未初始化的字符默认是 -51，两个 -51 在 GBK 编码中就是“屯”。二者都是未初始化的内存。

C++ 赋予了我们直接面对内存、操作内存的能力，但是内存管理却一直以来被认为是 C++ 语言的一大难点。因为在 C++ 语言中，缺少 GC（垃圾回收器），内存管理需要程序员手动完成，并且还要为可能的失误承担后果。

正如下面的“代码故事”：

```
#include <stdio.h>
#include <stdlib.h>

/*
在经历过无数的 "烫烫烫烫烫"，"屯屯屯屯屯" 之后，
我们都知道了：内存原来是可以驾驭的...
*/

int main()
{
    /*
        原来内存管理是这样的，即便结果完美无缺，
        但却危机四伏...
    */
    const char *src="Hello Csdn!";
    char *dest=(char*) malloc(strlen(src));
    memcpy(dest,src,strlen(src)+1);
    printf("%s\n",dest);

    return 0;
}
/*
    代码结束了，故事也到此为止。但是我们要做的还很多。
    希望我们能少遇到一点烫和屯...
*/
```

所以，我们要说说内存管理那点事儿，争取早日练就内存管理的高深技艺。

## 建议 27：区分内存分配的方式

在 C/C++ 语言中，用内存管理的水平去划分高手与菜鸟已经成为一种不成文的约定：可以从中获得更好的性能、更大自由的被称作 C++ 高手，而程序经常面临着莫名其妙的崩溃，一遍遍的调试，费时又费力的则可能是菜鸟级别的。而这一切都源于那让人又爱又恨的 C++ 内存管理的灵活性。其中，多样的内存分配方式就是其灵活性的最好例证之一。

一个程序要运行，就必须先将可执行的程序加载到计算机内存里，程序加载完毕后，就可以形成一个运行空间，并按照图 3-1 所示的那样进行布局。

代码区（Code Area）存放的是程序的执行代码；数据区（Data Area）存放的是全局数据、常量、静态变量等；堆区（Heap Area）存放的则是动态内存，供程序随机申请使用；而栈区（Stack Area）则存放着程序中所用到的局部数据。这些数据可以动态地反应程序中对函数的调用状态，通过其轨迹也可以研究其函数机制。其中，除了代码区不是我们能在代码中直接控制的，剩余三块都是我们编码过程中可以利用的。在 C++ 中，数据区又被分成自由存储区、全局/静态存储区和常量存储区，再加上堆区、栈区，也就是说内存被分成了 5 个区。这 5 种不同的分区各有所长，适用于不同的情况。

### □ 栈（Stack）区

在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元将自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是所分配的内存容量有限。

### □ 堆（Heap）区

堆就是那些由 new 分配的内存块，其释放编译器不会管它，而是由我们的应用程序控制它，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统就会自动回收。

### □ 自由存储区

自由存储区是那些由 malloc 等分配的内存块，它和堆十分相似，不过它是用 free 来结束自己生命的。

### □ 全局 / 静态存储区

全局变量和静态变量被分配到同一块内存中，在以前的 C 语言中，全局变量又分为初始化的和未初始化的，在 C++ 里面没有作此区分，它们共同占用同一块内存区。

### □ 常量存储区

这是一块比较特殊的存储区，里面存放的是常量，不允许修改。

上述 5 种分区中，最常用的就是堆与栈，容易混淆的也是堆与栈。在 BBS 论坛里，

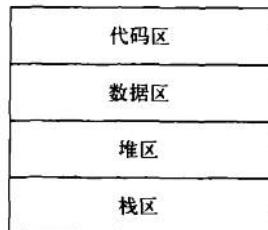


图 3-1 程序运行空间布局图

几乎到处都能看到堆与栈的争论。堆与栈的区分问题，似乎是每一个 C++ 程序员成长路上都会遇到的永恒话题。那么堆与栈之间到底有什么分别与联系呢？这就是接下来我要阐述的问题。

首先，还是分析下面的代码片段：

```
const int COUNT = 10;
void Function()
{
    string* pStr = new string[COUNT];
}
```

你是否相信就这么简单的一个函数，它却涉及了 5 种内存分区中的 3 种呢？COUNT 是一个常量，被安置在了常量存储区，不可修改；pStr 是局部变量，理所应当地放入栈里；而通过 new string[COUNT] 获得的则是一块堆空间。多么精妙，多么不可思议！当然，上述代码片段只是一个示例，是经不起推敲的，因为它会引起内存泄露（缺少与 new 对应的 delete 去释放内存）。

似乎脱离了主题，还是言归正传，说说堆与栈的区别。总的来说，二者的区别主要有以下几个方面：

#### □ 管理方式不同

对于栈来讲，它是由编译器自动管理的，无须我们手工控制；对于堆来说，它的释放工作由程序员控制，容易产生 memory leak。

#### □ 空间大小不同

一般来讲在 32 位系统下，堆内存可以达到 4GB 的空间，从这个角度来看堆内存几乎是什么限制的。但是对于栈来讲，一般都是有一定空间大小的。

#### □ 碎片问题

对于堆来讲，频繁的 new/delete 势必会造成内存空间的不连续，从而产生大量的碎片，使程序效率降低。对于栈来讲，则不存在这个问题，其原因还要从栈的特殊数据结构说起。栈是一个具有严明纪律的队列，其中的数据必须遵循先进后出的规则，相互之间紧密排列，绝不会留给其他数据可插入之空隙，所以永远都不可能有一个内存块从栈中间弹出，它们必须严格按照一定的顺序一一弹出。

#### □ 生长方向

对于堆来讲，其生长方向是向上的，也就是向着内存地址增加的方向增长；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长的。

#### □ 分配方式

堆都是动态分配的，没有静态分配的堆。栈有两种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由 alloca 函数完成，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放的，无须我们手工实现。

### □ 分配效率

栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：它会分配专门的寄存器存放栈的地址，而且压栈出栈都会有专门的指令来执行，这就决定了栈的效率比较高。堆则是C/C++函数库提供的，它的机制很复杂，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），则可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存了，然后返回。显然，堆的效率比栈要低得多。

堆和栈相比，由于堆使用了大量new/delete，容易造成大量的内存碎片，而且它没有专门的系统支持，效率很低，另外它还可能引发用户态和核心态的切换，以及内存的申请，代价会变得很高。所以栈在程序中是应用最广泛的，就算是函数的调用也会利用栈去完成，函数调用过程中的参数、返回的地址、EBP和局部变量都是采用栈的方式存放的。所以，我们推荐大家尽量多用栈，而不是用堆。

虽然栈有如此多的好处，但是由于和堆相比它不是那么灵活，有时候会分配大量的内存空间，在遇到这种情况时还是用堆好一些。

### 请记住：

内存分配具有多种不同的方式，它们各具特点，适用于不同的情形。所以，要在合适的地方采用合适的方式完成内存的分配。

## 建议 28：new/delete 与 new[]/delete[] 必须配对使用

operator new 和 operator delete 函数有两个重载版本：

```
void* operator new (size_t);           // allocate an object
void* operator new [] (size_t);         // allocate an array
void operator delete (void*);          // free an object
void operator delete [] (void*);        // free an array
```

熟悉 C 语言的朋友看到这里可能会很奇怪：在 C 语言中，无论申请的是单个对象，还是一个数组，管理内存所用的都是 malloc/free，但是为什么到了 C++ 里会出现两个呢？何况建议 21 中已经说明，new/delete 在功能上比前者更加强劲。

先分析以下代码片段存在的问题：

```
class Test
{
public:
    Test() { cout << "ctor" << endl; }
```

```

    ~Test() { cout << "dtor" << endl; }
    Hello() { cout << "Hello C++" << endl; }
};

int main()
{
    cout << "Test 1:" << endl;
    Test* p1 = new Test[3];
    delete p1;

    cout << "Test 2:" << endl;
    Test* p2 = new Test;
    delete[] p2;

    return 0;
}

```

上述代码看起来井然有序：我们采纳了建议 21，用 new 完成了内存申请，并且使用了与之对应的 delete 来释放内存。可是，执行结果却显示上述代码存在问题。在 Test 1 中，构造函数调用了 3 次，构造了 3 个 Test 类型对象，而在删除时，却只析构了一个对象 p1[0]。在 Test 2 中，构造函数调用了一次，但是析构函数却被调用了多次，将本不属于该对象的空间当成该类型对象进行了清理。也许各大厂商意识到了这个问题，于是让编译器能够检测出这样的错误。所以在 VC++2010 中，如果出现上述情形，编译器就会给出“debug assertion failed”或“堆被损坏”的错误信息。

C++ 告诉我们在回收用 new 分配的单个对象的内存空间时用 delete，在回收用 new[] 分配的一组对象的内存空间时用 delete[]。下面我们就分析一下它们的实现原理。

无论 new 还是 new[], C++ 必须知道返回指针所指向的内存块的大小，否则它就不可能正确地释放掉这块内存，这一点很像 C 语言中的 malloc。但是在用 new[] 为一个数组申请内存时，编译器还会悄悄地在内存中保存一个整数，用来表示数组中元素的个数。因为在 delete 一块内存时，我们不仅要知道指针指向多大的内存，更重要的是要知道指针指向的数据组中对象的个数。因为只有知道了对象数量才能一一调用它们的析构函数，完成对数组中所有对象的清理。如果使用的是 delete，则编译器只会将指针所指的对象当作单个对象来处理。所以对于数组，需要使用 delete[] 来处理；符号 [] 会告诉编译器在 delete 这块内存时，先去获取保存的那个元素数量值，然后再进行一一清理。如果你对汇编有所了解，那么你可以通过反汇编代码对此一探究竟。

也许你会认为 C++ 这么设计绝对是多此一举，因为单个对象只是对象数组的一个特例，无论是一个对象，还是对象数组，我们都对元素个数进行记录，这样也就不再需要两个版本的 new 和 delete 了。但是 C++ 之父之所以没有选择这么做，也许是坚持他认定的 C++ 设计风格和宗旨：决不多费一点力。殊不知，这么做的直接后果就是需要程序员付出更多的细心与努力。

需要注意的是，由于内置数据类型没有构造、析构函数，所以在针对内置数据类型时，释放内存使用 `delete` 或 `delete[]` 的效果是一样的。例如：

```
int *pArray = new int[10];
... // processing code
delete pArray; // 等同于 delete[] pArray;
```

虽然针对内置类型，`delete` 和 `delete[]` 都能正确地释放所申请的内存空间，但是如果申请的是一个数组，建议还是使用 `delete[]` 形式。

所以，使用 `new` 和 `delete` 的一个简单有效的原则就是：如果在调用 `new` 时使用了 `[]`，则你在调用 `delete` 时也使用 `[]`，如果在调用 `new` 的时候没有用 `[]`，那么也不应该在调用时使用 `[]`。`new` 和 `delete`、`new[]` 和 `delete[]` 必须对应着使用。

对于那些喜欢 `typedef` 的人，还有一点需要提醒。因为在这种情况下很容易出现 `new[]` 和 `delete` 的混用。如下面的代码片段所示：

```
typedef int scorers[LESSONS_NUM];
int *pScorer = new scorers;
```

这该使用哪一种形式的 `delete` 呢？如下所示。

```
delete pScorer; // Wrong!!!
delete[] pScorer; // Right
```

为了避免出现这样的错误，建议不要对数组类型做 `typedef`，或者采用 STL 中的 `vector` 代替数组。

#### 请记住：

`new` 和 `delete`、`new[]` 和 `delete[]` 必须对应使用，否则会出现未定义行为，导致程序崩溃。

## 建议 29：区分 `new` 的三种形态

C++ 语言一直被认为是复杂编程语言中的杰出代表之一，不仅仅是因为其繁缛的语法规则，还因为其晦涩的术语。下面要讲的就是你的老熟人——`new`：

它是一个内存管理的操作符，能够从堆中划分一块区域，自动调用构造函数，动态地创建某种特定类型的数据，最后返回该区域的指针。该数据使用完后，应调用 `delete` 运算符，释放动态申请的这块内存。

如果这就是你对 `new` 的所有认识，那么我不得不说，你依旧被 `new` 的和善外表所蒙蔽着。看似简单的 `new` 其实有着三种不同的外衣。

是的，你没有看错，也不用感到惊奇，一个简单的 new 确实有三种不同的形态，它扮演着三种不同的角色，如下所示：

- new operator
- operator new
- placement new

下面的代码片段展示的是我们印象中熟悉的那个 new：

```
string *pStr = new string("Memory Management");
int *pInt = new int(2011);
```

这里所使用的 new 是它的第一种形态 new operator。它与 sizeof 有几分类似，它是语言内建的，不能重载，也不能改变其行为，无论何时何地它所做的有且只有以下三件事，如图 3-2 所示。

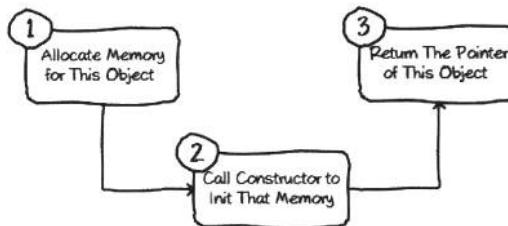


图 3-2 new operator 所完成的三件事

所以当写出 “`string *pStr = new string("Memory Management");`” 代码时，它其实做的就是以下几件事：

```
// 为 string 对象分配 raw 内存
void *memory = operator new( sizeof(string) );
// 调用构造函数，初始化内存中的对象
call string::string() on memory;
// 获得对象指针
string *pStr = static_cast<string*>(memory);
当然，对于内置类型，第二步是被忽略的，即：
// 为 int 分配 raw 内存
void *memory = operator new( sizeof(int) );
// 获得对象指针
int *pInt = static_cast<int*>(memory);
```

其实 new operator 背后还藏着一个秘密，即它在执行过程中，与其余的两种形态都发生了密切的关系：第一步的内存申请是通过 operator new 完成的；而在第二步中，关于调用什么构造函数，则由 new 的另外一种形态 placement new 来决定的。

对于 new 的第二种形态——内存申请中所调用的 operator new，它只是一个长着“明星脸”的普通运算符，具有和加减乘除操作符一样的地位，因此它也是可以重载的。

`operator new` 在默认情况下首先会调用分配内存的代码，尝试从堆上得到一段空间，同时它对事情的结果做了最充分的准备：如果成功则直接返回；否则，就转而去调用一个 `new_hander`，然后继续重复前面过程，直到异常抛出为止。所以如果 `operator new` 要返回，必须满足以下条件之一：

- 内存成功分配。
- 抛出 `bad_alloc` 异常。

通常，`operator new` 函数通过以下方式进行声明：

```
void* operator new(size_t size);
```

注意，这个函数的返回值类型是 `void*`，因为这个函数返回的是一个未经处理的指针，是一块未初始化的内存，它像极了 C 库中的 `malloc` 函数。如果你对这个过程不满意，那么可以通过重载 `operator new` 来进行必要的干预。例如：

```
class A
{
public:
    A(int a);
    ~A();
    void* operator new(size_t size);
    ...
};

void* A::operator new(size_t size)
{
    cout<<"Our operator new...";
    return ::operator new(size);
}
```

这里的 `operator new` 调用了全局的 `new` 来进行内存分配 (`::operator new(size)`)。当然这里的全局 `new` 也是可以重载的，但是在全局空间中重载 `void * operator new(size_t size)` 函数将会改变所有默认的 `operator new` 的行为方式，所以必须十二分的注意。还有一点需要注意的是，正像 `new` 与 `delete` 一一对应一样，`operator new` 和 `operator delete` 也是一一对应的；如果重载了 `operator new`，那么也得重载对应的 `operator delete`。

最后，要介绍的是 `new` 的第三种形态——`placement new`。正如前面所说的那样，`placement new` 是用来实现定位构造的，可以通过它来选择合适的构造函数。虽然通常情况下，构造函数是由编译器自动调用的，但是不排除你有时确实想直接手动调用，比如对未初始化的内存进行处理，获取想要的对象，此时就得求助于一个叫做 `placement new` 的特殊的 `operator new` 了：

```
#include <new>
#include "ClassA.h"
int main()
```

```

{
    void *s = operator new(sizeof(A));
    A* p = (A*)s;
    new(p) A(2011); //p->A::A(2011);
    ... // processing code
    return 0;
}

```

placement new 是标准 C++ 库的一部分，被声明在了头文件 <new> 中，所以只有包含了这个文件，我们才能使用它。它在 <new> 文件中的函数定义很简单，如下所示：

```

#ifndef __PLACEMENT_NEW_INLINE
#define __PLACEMENT_NEW_INLINE
inline void *_CRTDECL operator new(size_t, void *_Where) _THROW0()
{
    // construct array with placement at _Where
    return (_Where);
}

inline void __CRTDECL operator delete(void *, void *) _THROW0()
{
    // delete if placement new fails
}
#endif /* __PLACEMENT_NEW_INLINE */

```

这就是 placement new 需要完成的事。细心的你可能会发现，placement new 的定义与 operator new 声明之间的区别：placement new 的定义多一个 void\* 参数。使用它有一个前提，就是已经获得了指向内存的指针，因为只有这样我们才知道该把 placement new 初始化完成的对象放在哪里。

在使用 placement new 的过程中，我们看到的却是 "new(p) A(2011)" 这样奇怪的调用形式，它在特定的内存地址上用特定的构造函数实现了构造一个对象的功能，A(2011) 就是对构造函数 A(int a) 的显式调用。当然，如果显式地调用 placement new，那么也得本着负责任的态度显式地调用与之对应的 placement delete : p->~A();。这部分工作本来可以由编译器独自完成的：在使用 new operator 的时候，编译器会自动生成调用 placement new 的代码，相应的，在调用 delete operator 时同样会生成调用析构函数的代码。所以，除非特别必要，不要直接使用 placement new。但是要清楚，它是 new operator 的一个不可或缺的步骤。当默认的 new operator 对内存的管理不能满足我们的需要，希望自己手动管理内存时，placement new 就变得有用了。就像 STL 中的 allocator 一样，它借助 placement new 来实现更灵活有效的内存管理。

最后，总结一下：

- 如果是在堆上建立对象，那么应该使用 new operator，它会为你提供最为周全的服务。
- 如果仅仅是分配内存，那么应该调用 operator new，但初始化不在它的工作职责之内。如果你对默认的内存分配过程不满意，想单独定制，重载 operator new 是不二选择。

- 如果想在一块已经获得的内存里建立一个对象，那就应该用 placement new。但是通常情况下不建议使用，除非是在某些对时间要求非常高的应用中，因为相对于其他两个步骤，选择合适的构造函数完成对象初始化是一个时间相对较长的过程。

---

**请记住：**

不要自信地认为自己对 new 很熟悉，要正确区分 new 所具有的三种不同形态，并能在合适的情形下选择合适的形态，以满足特定需求。

---

## 建议 30：new 内存失败后的正确处理

应该有很多的程序员对比尔·盖茨的这句话有所耳闻：

对于任何一个人而言，640KB 应当是足够的了。（640K ought to be enough for everybody.）

不幸的是，伟大的比尔·盖茨也失言了。随着硬件水平的发展，内存变得越来越大，但是似乎仍不能满足人们对内存日益增长的需求。所以呢，我们 C/C++ 程序员在写程序时也必须考虑一下内存申请失败时的处理方式。

通常，我们在使用 new 进行内存分配的时候，会采用以下的处理方式：

```
char *pStr = new string[SIZE];
if(pStr == NULL)
{
    ... // Error processing
    return false;
}
```

你能发现上述代码中存在的问题吗？这是一个隐蔽性极强的臭虫（Bug）。

我们沿用了 C 时代的良好传统：使用 malloc 等分配内存的函数时，一定要检查其返回值是否为“空指针”，并以此作为检查分配内存操作是否成功的依据，这种 Test-for-NULL 代码形式是一种良好的编程习惯，也是编写可靠程序所必需的。可是，这种完美的处理形式必须有一个前提：若 new 失败，其返回值必须是 NULL。只有这样才能保证上述看似“逻辑正确、风格良好”的代码可以正确运行。

那么 new 失败后编译器到底是怎么处理的？在很久之前，即 C++ 编译器的蛮荒时代，C++ 编译器保留了 C 编译器的处理方式：当 operator new 不能满足一个内存分配请求时，它返回一个 NULL 指针。这曾经是对 C 的 malloc 函数的合理扩展。然而，随着技术的发展，标准的更新，编译器具有了更强大的功能，类也被设计得更漂亮，新时代的 new 在申请内存失败时具备了新的处理方式：抛出一个 bad\_alloc exception（异常）。所以，在新的标准里，上述 Test-for-NULL 处理方式不再被推荐和支持。

如果再回头看看本建议开头的代码片段，其中的 if (pStr == 0) 从良好的代码风格突然一下变成了毫无意义。在 C++ 里，如果 new 分配内存失败，默认是抛出异常。所以，如果分配成功，pStr == 0 就绝对不会成立；而如果分配失败了，也不会执行 if (pStr == 0)，因为分配失败时，new 就会抛出异常并跳过后面的代码。

为了更加明确地理解其中的玄机，首先看看相关声明：

```
namespace std
{
    class bad_alloc
    {
        // ...
    };
}

// new and delete
void *operator new(std::size_t) throw(std::bad_alloc);
void operator delete(void *) throw();

// array new and delete
void *operator new[](std::size_t) throw(std::bad_alloc);
void operator delete[](void *) throw();

// placement new and delete
void *operator new(std::size_t, void *) throw();
void operator delete(void *, void *) throw();

// placement array new and delete
void *operator new[](std::size_t, void *) throw();
void operator delete[](void *, void *) throw();
```

在以上的 new 操作族中，只有负责内存申请的 operator new 才会抛出异常 std::bad\_alloc。如果出现了这个异常，那就意味着内存耗尽，或者有其他原因导致内存分配失败。所以，按照 C++ 标准，如果想检查 new 是否成功，则应该捕捉异常：

```
try
{
    int* pStr = new string[SIZE];
    ... // processing codes
}
catch (const bad_alloc& e)
{
    return -1;
}
```

但是市面上还存在着一些古老编译器的踪迹，这些编译器并不支持这个标准。同时，在这个标准制定之前已经存在的很多代码，如果因为标准的改变而变得漏洞百出，肯定会引起很多人抗议。C++ 标准化委员会并不想遗弃这些 Test-for-NULL 的代码，所以他

们提供了 operator new 的另一种可选形式—— nothrow，用以提供传统的 Failure-yields-NULL 行为。

其实现原理如下所示：

```
void * operator new(size_t cb, const std::nothrow_t&) throw()
{
    char *p;
    try
    {
        p = new char[cb];
    }
    catch (std::bad_alloc& e)
    {
        p = 0;
    }
    return p;
}
```

<new> 文件中也声明了 nothrow new 的重载版本，其声明方式如下所示：

```
namespace std
{
    struct nothrow_t
    {
        // ...
    };
    extern const nothrow_t nothrow;
}

// new and delete
void *operator new(std::size_t, std::nothrow_t const &) throw();
void operator delete(void *, std::nothrow_t const &) throw();

// array new and delete
void *operator new[](std::size_t, std::nothrow_t const &) throw();
void operator delete[](void *, std::nothrow_t const &) throw();
```

如果采用不抛出异常的 new 形式，本建议开头的代码片段就应该改写为以下形式：

```
int* pStr = new(std::nothrow) string[SIZE];
if(pStr==NULL)
{
    ... // 错误处理代码
}
```

根据建议 29 可知，编译器在表达式 new (std::nothrow) ClassName 中一共完成了两项任务。首先，operator new 的 nothrow 版本被调用来为一个 ClassName object 分配对象内存。假如这个分配失败，operator new 返回 null 指针；假如内存分配成功，ClassName 的构造函

数则被调用，而在此刻，对象的构造函数就能做任何它想做的事了。如果此时它也需要 new 一些内存，但是没有使用 nothrow new 形式，那么，虽然在 "new (std::nothrow) ClassName" 中调用的 operator new 不会抛出异常，但其构造函数却无意中办了件错事。假如它真的这样做了，exception 就会像被普通的 operator new 抛出的异常一样在系统里传播。所以使用 nothrow new 只能保证 operator new 不会抛出异常，无法保证 "new (std::nothrow) ClassName" 这样的表达式不会抛出 exception。所以，慎用 nothrow new。

最后还需要说明一个比较特殊但是确实存在的问题：在 Visual C++ 6.0 中目前 operator new、operator new(std::nothrow) 和 STL 之间不兼容、不匹配，而且不能完全被修复。如果在非 MFC 项目中使用 Visual C++ 6.0 中的 STL，其即装即用的行为可能导致 STL 在内存不足的情况下让应用程序崩溃。对于基于 MFC 的项目，STL 是否能够幸免于难，完全取决于你使用的 STL 针对 operator new 的异常处理。这一点，在 James Hebben 的文章《不要让内存分配失败导致您的旧版 STL 应用程序崩溃》中进行了详细的介绍，如果你在使用古老的 Visual C++ 6.0 编译器，而且对这个问题充满兴趣，请 Google 之。

#### 请记住：

当使用 new 申请一块内存失败时，抛出异常 std::bad\_alloc 是 C++ 标准中规定的标准行为，所以推荐使用 try{ p = new int[SIZE]; } catch( std::bad\_alloc ) { ... } 的处理方式。但是在一些老旧的编译器中，却不支持该标准，它会返回 NULL，此时具有 C 传统的 Test\_for\_NULL 代码形式便起了作用。所以，要针对不同的情形采取合理的处置方式。

### 建议 31：了解 new\_handler 的所作所为

在使用 operator new 申请内存失败后，编译器并不是不做任何的努力直接抛出 std::alloc 异常，在这之前，它会调用一个错误处理函数（这个函数被称为 new-handler），进行相应的处理。通常，一个好的 new-handler 函数的处理方式必须遵循以下策略之一：

- Make more memory available (使更大块内存有效)

operator new 会进行多次的内存分配尝试，这可能会使其下一次的内存分配尝试成功。其中的一个实现方法是在程序启动时分配一大块内存，然后在 new-handler 第一次被调用时释放它供程序使用。

- Install a different new-handler (装载另外的 new-handler)

程序中可以同时存在多个 new-handler，假如当前的 new-handler 不能获得更多的内存供 operator new 分配使用，但另一个 new-handler 却可以做到。在这种情形下，当前的 new-handler 则会通过调用 set\_new\_handler 在它自己的位置上安装另一个 new-handler。当 operator new 下一次调用 new-handler 时，它会调用最新安装的那个。

□ Deinstall the new-handler (卸载 new-handler)

换句话说，就是将空指针传给 `set_new_handler`，此时就没有了相应的 new-handler。当内存分配失败时，`operator new` 则会抛出一个异常。

□ Throw an exception (抛出异常)

抛出一个类型为 `bad_alloc` 或继承自 `bad_alloc` 的其他类型的异常。

□ Not return (无返回)

直接调用 `abort` 或 `exit` 结束应用程序。

以上的这些处理方式让我们在实现 new-handler functions 时拥有了更多的选择与自由。这些各式各样的 new-handler 函数是可以通过调用标准库函数 `set_new_handler` 进行特殊定制的，你可以按照自己的方式来对编译器的这一行为进行设定。这个函数同样也声明在 `<new>` 中：

```
namespace std
{
    typedef void (*new_handler)();
    new_handler set_new_handler(new_handler p) throw();
}
```

通过函数声明可以看到 `set_new_handler` 的形参是一个指向函数的指针，这个函数在 `operator new` 无法分配被请求的内存时调用。`set_new_handler` 的返回值是一个指向函数的指针，指向的是 `set_new_handler` 调用之前的异常处理函数。所以，可以按照以下方式使用 `set_new_handler` 函数：

```
//error-handling function
void MemErrorHandler()
{
    std::cerr << "Failed to allocate memory\n";
    std::abort();
}
//Application
const long long DATA_SIZE = 1024*1024*1024;
int main()
{
    std::set_new_handler(MemErrorHandler);
    std::cout << "Attempting to allocate 1 GB...";
    char *pDataBlock = NULL;
    try
    {
        pDataBlock = new char[DATA_SIZE];
    }
    catch(std::alloc& e)
    {
        ... //some processing codes
    }
}
```

```
    ... // other processing code
}
```

假如 operator new 分配空间的请求得不到满足，MemErrorHandler 函数将被调用，程序将按照函数中的设定处理方式运行。在标准 C++ 中，标准 set\_new\_handler 为用户类统一指定了错误处理函数 global new-handler。上述代码采用的就是 global new-handler 形式。

通过上述示例代码可以看出，new\_handler 必须有主动退出的功能，否则就会导致 operator new 内部死循环。因此 new\_handler 一般会采用如下形式，伪代码表示如下：

```
void MemErrorHandler()
{
    if( 有可能使得 operator new 成功 )
    {
        做有可能使得 operator new 成功的事
        return;
    }
    // 主动退出
    abort/exit 直接退出程序
    或 set_new_handler( 其他 newhandler );
    或 set_new_handler(0)
    或 throw bad_alloc() 或派生类
}
```

当然，我们可以根据被分配对象的不同，采用不同的方法对内存分配失败进行处理，实现对 class-specific new-handlers 的支持。为了实现这一行为，需要为每一个 class 提供专属的 set\_new\_handler 和 operator new 版本。假设要为 A class 设定特殊的内存分配失败处理方式，则需要在类 A 中声明一个 new\_handler 类型的静态成员（static member），并将其设置为 A class 的 new-handler 处理函数。所以就得到了下面的代码：

```
class A
{
public:
    static std::new_handler set_new_handler(std::new_handler p) throw();
    static void * operator new(std::size_t size) throw(std::bad_alloc);
    static void MemoryErrorHandler();
private:
    static std::new_handler m_curHandler;
};
// 静态类成员定义
std::new_handler A::m_curHandler = NULL;
```

C++ 标准中规定 set\_new\_handler 函数应该保存传递给它的函数指针，并返回前次调用时被保存的函数指针。上面 A 类中的 set\_new\_handler 也应该这么做：

```
std::new_handler A::set_new_handler(std::new_handler p) throw()
```

```

{
    std::new_handler oldHandler = m_curHandler;
    m_curHandler = p;
    return oldHandler;
}

```

接下来，我们自己定义该类的处理函数：

```

void MemoryErrorHandler()
{
    ... // processing code
}
void * operator new(std::size_t size) throw(std::bad_alloc)
{
    set_new_handler(MemoryErrorHandler);
    return ::operator new(size);
}

```

当然我们可以采用更好的设计方式（如类继承）来实现，这里就不赘述。如果读者感兴趣可以自己思考一下，或者求助于资源丰富的 Internet，它会提供详尽的参考资料。

#### 请记住：

了解 `new_handler` 的所作所为，并通过标准库函数 `set_new_handler` 对内存分配请求不能被满足的处理函数进行特殊定制。

## 建议 32：借助工具监测内存泄漏问题

内存管理确实是一个令众多 C/C++ 程序员感到费神又费力的问题，内存错误通常都具有隐蔽性，难以再现，而且其症状一般不能在相应的源代码中找到。C/C++ 应用程序的大部分缺陷和错误都和内存相关，预防、发现、消除代码中和内存相关的缺陷，成为 C/C++ 程序员编写、调试、维护代码时的重要任务。然而任何人都无法时刻高度谨慎，百密中难免会有一疏，一不小心就会发生内存问题。如果泄漏内存，则运行速度会逐渐变慢，并最终会停止运行；如果覆盖内存，则程序会变得非常脆弱，很容易受到恶意用户的攻击。因此，需要特别关注 C/C++ 编程的内存问题，特别是内存泄漏。幸运的是，现在有许多的技术和工具能够帮助我们验证内存泄漏是否存在，寻找到发生问题的位置。

内存泄漏一般指的是堆内存的泄漏。如果我们使用 `malloc` 函数或 `new` 操作符从堆中分配到一块内存，在使用完后，程序员必须负责调用相应的 `free` 或 `delete` 显式地释放该内存块，否则，这块内存就不能被再次使用，此时就出现了传说中的“内存泄漏”问题。如下面的代码片段所示：

```
void Function(size_t nSize)
{
    char* pChar= new char[nSize];
    if( !SetContent(pChar, nSize ) )
    {
        cout<<"Error: Fail To Set Content"<<endl;
        return;
    }
    ...//using pChar
    delete pChar;
}
```

程序在入口处分配内存，在出口处释放内存，但是这里忽视了代码片段中的 return；如果函数 SetContent() 失败，指针 pChar 指向的内存就不会被释放，会发生内存泄漏。这是一种常见的内存泄漏情形。

检测内存泄漏的关键是要能截获对分配内存和释放内存的函数的调用。通过截获的这两个函数，我们就能跟踪每一块内存的生命周期。每当成功分配一块内存时，就把它的指针加入一个全局的内存链中；每当释放一块内存时，再把它的指针从内存链中删除。这样，当程序运行结束的时候，内存链中剩余的指针就会指向那些没有被释放的内存。这就是检测内存泄漏的基本原理<sup>②</sup>。

检测内存泄漏的常用方法有如下几种：

#### □ MS C-Runtime Library 内建的检测功能

使用 MFC 开发的应用程序时，会在 Debug 模式下编译执行，程序运行结束后，Visual C++ 会输出内存的使用情况，如果发生了内存泄漏，在 Debug 窗口中会输出所有发生泄漏的内存块的信息，如下所示：

这是因为在编译过程中，IDE 自动加入了内存泄漏的检测代码。MFC 在程序执行过程中维护了一个内存链，以便跟踪每一块内存的生命周期。在程序退出的时候，`dbgheap.c` 文件中的 `extern "C" _CRTIMP int __cdecl _CrtDumpMemoryLeaks(void)` 函数被调用，遍历当前的内存链，如果发现存在没有被释放的内存，则打印出内存泄露的信息。

一般，大家都误以为这些内存泄漏的检测功能是由 MFC 提供的，其实不然。这是 VC++ 的 C 运行库 (CRT) 提供的功能，MFC 只是封装和利用了 MS C-Runtime Library 的

<sup>④</sup> 详细的算法可以参见 Steve Maguire 的《Writing Solid Code》。

Debug Function 而已。所以，在编写非 MFC 程序时我们也可以利用 MS C-Runtime Library 的 Debug Function 加入内存泄漏的检测功能。

要在非 MFC 程序中打开内存泄漏的检测功能非常容易，只须在程序的入口处添加以下代码：

```
_CrtSetDbgFlag( _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG)
                | _CRTDBG_LEAK_CHECK_DF );
```

这样，在程序运行结束时，如果还有内存块没有释放，它们的信息就会被打印到 Debug 窗口里，如下面的代码片段所示：

```
#include <crtdbg.h>

#ifndef _DEBUG
#define new new(_NORMAL_BLOCK, __FILE__, __LINE__)
#endif
void EnableMemLeakCheck()
{
    _CrtSetDbgFlag( _CrtSetDbgFlag(_CRTDBG_REPORT_FLAG)
                    | _CRTDBG_LEAK_CHECK_DF );
}

int main()
{
    EnableMemLeakCheck();
    _CrtSetBreakAlloc(53);
    int* pLeak = new int[10];

    return 0;
}
```

在 Debug 模式下，程序退出时，内存块 pLeak 因为没有显式地释放，发生了内存泄漏，泄漏信息被打印出来：

```
Detected memory leaks!
Dumping objects ->
main.cpp(26) : {53} normal block at 0x002E1508, 40 bytes long.
Data: <REDACTED> CD CD
Object dump complete.
```

请读者思考一下，\_CrtSetBreakAlloc(53) 起到的是什么作用？

目前这种方式只支持 MS 系统开发环境。当然，如果开发系统环境是 Linux，也可以根据 MS C-Runtime Library 内建检测功能的实现方式开发出自己的 Linux C-Runtime Library 内建检测版本。

#### □ 外挂式的检测工具

如果开发的是一个大型程序，MS C-Runtime Library 提供的检测功能便显得有点笨拙了。

此时，我们可以采用外挂式的检测工具 BoundsChecker 或 Insure++。

BoundsChecker 采用的是一种被称为 Code Injection 的技术，来截获对分配内存和释放内存的函数的调用的。简单地说，当程序开始运行时，BoundsChecker 的 DLL 被自动载入进程的地址空间中，然后它会修改进程中对内存分配和释放的函数调用，让这些调用首先转入它的代码，然后再执行原来的代码。BoundsChecker 在做这些动作时，无须修改被调试程序的源代码或工程配置文件，这使得使用它非常简便、直接。而 Insure++ 则是利用其专利技术（源码插装和运行时指针跟踪）来发现大量的内存操作错误，准确报告错误的源代码行和执行轨迹。

如果开发环境是 Linux，MS C-Runtime Library 内建检测功能就会彻底失效，BoundsChecker 或 Insure++ 也无能为力。这时，外挂式的检测工具 Rational Purify 或 Valgrind 便派上了用场。

Rational Purify 主要是针对软件开发过程中难以发现的内存错误、运行时错误。它可以在软件开发过程中自动地发现错误，准确地定位错误，并提供完备的错误信息，从而减少调试时间。同时它也是市场上唯一支持多种平台的相关工具，并且可以和很多主流开发工具集成。Purify 可以检查应用的每一个模块，甚至可以查出复杂的多线程或进程应用中的错误。另外，它不仅可以检查 C/C++，还可以对 Java 或 .NET 中的内存泄漏问题给出报告。

在 Linux 系统中，使用 Purify 非常简单，只须重新编译程序：

```
purify g++ -g main.cpp -o LeakDetector
```

运行编译生成的可执行文件 LeakDetector，就可以定位出内存泄漏的具体位置。

除了 Rational Purify，Valgrind 也是 Linux 系统下开发应用程序时用于调试内存问题的有效工具。它尤其擅长发现内存管理的问题，检查发现程序运行时的内存泄漏。

至于上述这些外挂式检测工具的具体使用方法就不赘述了。

根据应用程序的具体情况，合理采用上述方法和工具，可以有效防止和查找代码中的内存泄漏问题，并且能和开发人员日常编码无缝结合，有效提高开发效率，增强应用程序鲁棒性。

---

#### 请记住：

内存泄露是一个大问题，但是可以通过一定方法或借助于专业的检测工具，来查找并发现这些问题，有效地提升程序员的开发效率。

---

### 建议 33：小心翼翼地重载 operator new/ operator delete

虽然 C++ 标准库已经为我们提供了 new 与 delete 操作符的标准实现，但是由于缺乏对具体对象的具体分析，系统默认提供的分配器在时间和空间两方面都存在着一些问题：分配器速度较慢，而且在分配小型对象时空间浪费比较严重，特别是在一些对效率或内存有较大

限制的特殊应用中。比如说在嵌入式的系统中，由于内存限制，频繁地进行不定大小的内存动态分配很可能会引起严重问题，甚至出现堆破碎的风险；再比如在游戏设计中，效率绝对是一个必须要考虑的问题，而标准 new 与 delete 操作符的实现却存在着天生的效率缺陷。此时，我们可以求助于 new 与 delete 操作符的重载，它们给程序带来更灵活的内存分配控制。除了改善效率，重载 new 与 delete 还可能存在以下两点原因：

- 检测代码中的内存错误。
- 获得内存使用的统计数据。

相对于其他的操作符，operator new 具有一定的特殊性，在多个方面上与它们大不相同。首先，对于用户自定义类型，如果不重载，其他操作符是无法使用的，而 operator new 则不然，即使不重载，亦可用于用户自定义类型。其次，在参数方面，重载其他操作符时参数的个数必须是固定的，而 operator new 的参数个数却可以是任意的，只需要保证第一个参数为 size\_t 类型，返回类型为 void \* 类型即可。所以 operator new 的重载会给我们一种错觉：它更像是一个函数重载，而不是一个操作符重载。

关于 operator new 重载函数的形式，在 C++ 标准中有如下规定：

分配函数应当是一个类的成员函数或者是全局函数；如果一个分配函数被放于非全局名空间中，或者是在全局名空间被声明为静态，那这个程序就是格式错误的。⊕

也就是说，重载的 operator new 必须是类成员函数或全局函数，而不可以是某一名空间之内的函数或是全局静态函数。此外，还要多加注意的是，重载 operator new 时需要兼容默认的 operator new 的错误处理方式，并且要满足 C++ 的标准规定：当要求的内存大小为 0 byte 时也应该返回有效的内存地址。

所以，全局的 operator new 重载应该不改变原有签名，而是直接无缝替换系统原有版本，如下所示：

```
void * operator new(size_t size)
{
    if(size == 0)
        size = 1;
    void *res;
    for(;;)
    {
        //allocate memory block
        res = heap_alloc(size);
        //if successful allocation, return pointer to memory
        if(res)
            break;
    }
}
```

---

⊕ An allocation function shall be a class member function or a global function; a program is ill-formed if an allocation function is declared in a namespace scope other than global scope or declared static in global scope.

```

//call installed new handler
if (!CallNewHandler(size))
    break;
//new handler was successful -- try to allocate again
}
return res;
}

```

如果是用这种方式进行的重载，再使用时就不需要包含 new 头文件了。“性能优化”时通常采用这种方式。

如果重载了一个 operator new，记得一定要在相同的范围内重载 operator delete。因为你分配出来的内存只有你自己才知道应该如何释放。如果你偷懒或者是忘记了，编译器就会求助于默认的 operator delete，用默认方式释放内存。虽然程序编译可以通过，但是这将导致惨重的代价。所以，你必须时刻记得在写下 operator new 的同时写下 operator delete。相对于 operator new，重载 operator delete 要简单许多，如下所示：

```

void operator delete(void* p)
{
    if(p==NULL)
        return;
    free(p);
}

```

唯一要注意的一点就是，须遵循 C++ 标准中要求删除一个 NULL 指针是安全的这一规定。在全局空间中重载 void \* operator new(size\_t size) 函数将会改变所有默认的 operator new 的行为方式，所以一定要小心使用。

如果使用不同的参数类型重载 operator new/delete，则请采用如下函数声明形式：

```

// 返回的指针必须能被普通的 ::operator delete(void*) 释放
void* operator new(size_t size, const char* file, int line);
// 析构函数抛异常时被调用
void operator delete(void* p, const char* file, int line);

```

调用时采用以下方式：

```
string* pStr = new (__FILE, __LINE__) string;
```

这样就能跟踪内存分配的具体位置，定位这个动作发生在哪个文件的哪一行代码中了。在“检测内存错误”和“统计内存使用数据”时通常会用这种方式重载。

此外，我们还可以为 operator new 的重载使用参数默认值，甚至是不定参数。其原则和普通函数重载一样。

但是在使用全局重载时应该慎之又慎，因为这样做非常具有侵略性：这会让使用你编写的库的人没有选择的余地；同时，如果两个 lib 中都对 operator new 进行了重载，在使用时会出现这样的错误：duplicated symbol link error。这是多么令人恼火的一件事啊。

与全局 ::operator new() 不同，具体类的 operator new 与 delete 的影响面要小得多，它只影响本 class 及其派生类。为某个 class 重载 operator new 时必须将其定义为类的静态函数。因为 operator new 是在类的具体对象被构建出来之前调用的，在调用 operator new 的时候 this 指针尚未诞生，因此重载的 operator new 必须是 static 的：

```
class B
{
public:
    static void * operator new(size_t size);
    static void operator delete(void *p);
    // other members
};

void *B::operator new(size_t size)
{
    ...
}

void B::operator delete(void *p)
{
    ...
}
```

当然，同全局 operator new 重载一样，在类中重载成员 operator new 也可以添加额外的参数，并且可以使用默认值。另外，成员 operator new 也是可以继承的。但类中的 operator delete 也必须声明为静态函数。因为调用 operator delete 时，对象已经被析构，this 指针业已灰飞烟灭。

虽然为单独的 class 重载成员 operator new/ delete 是可行的，但不推荐使用。因为既然对它们进行了重载，说明它的内存分配策略已被进行了精心的特殊定制，从类似 `ClassName * p = new ClassName` 形式的代码中我们根本不能获得此信息。而且，我们有更加简单明了的 Factory 方案可以使用：

```
static ClassName* ClassName::CreateObject();
```

清晰明确优于模糊不清 (Explicit is better than implicit)，对此我深信不疑。

关于 operator new/operator delete 的重载，还有一个必须小心的问题，那就是在内存分配机制中必须要考虑对象数组内存分配这一点。C++ 将对象数组的内存分配看作是一个不同于单个对象内存分配的单独操作。对于多数的 C++ 实现，因为需要额外存储对象数量，`new[]` 操作符中的个数参数会是数组的大小加上存储对象数目的一些字节。所以，如果希望改变对象数组的分配方式，同样需要重载 `new[]` 和 `delete[]` 操作符。

### 请记住：

通过重载 operator new 和 operator delete 的方法，可以自由地采用不同的分配策略，从不同的内存池中分配不同的类对象。但是是否选择重载 operator new/delete 一定要深思熟虑。

## 建议 34：用智能指针管理通过 new 创建的对象

前面的建议中我们不厌其烦的一再重复：内存泄漏是一个很大很大的问题！为了应对这个问题，已经有许多技术被研究出来，比如 Garbage Collection（垃圾回收）、Smart Pointer（智能指针）等。Garbage Collection 技术一直颇受注目，并且在 Java 中已经发展成熟，成为内存管理的一大利器，但它在 C++ 语言中的发展却不顺利，C++ 为了追求运行速度，20 年来态度坚决地将其排除在标准之外。真不知 C++ 通过加大开发难度来换取执行速度的做法究竟是利还是弊。为了稍许平复因为没有 Garbage Collection 而引发的 C++ 程序员的怨气，C++ 对 Smart Pointer 技术采取了不同的态度，它选择对这一技术的支持，并在 STL 中包含了支持 Smart Pointer 技术的 class，赐予了 C/C++ 程序员们一件管理内存的神器。

Smart Pointer 是 Stroustrup 博士所推崇的 RAII (Resource Acquisition In Initialization) 的最好体现。该方法使用一个指针类来代表对资源的管理逻辑，并将指向资源的句柄（指针或引用）通过构造函数传递给该类。当离开当前范围（scope）时，该对象的析构函数一定会被调用，所以嵌在析构函数中的资源回收的代码也总是会被执行。这种方法的好处在于，由于将资源回收的逻辑通过特定的类从原代码中剥离出来，自动正确地销毁动态分配的对象，这会让思路变得更加清晰，同时确保内存不发生泄露。

它的一种通用实现技术是使用引用计数（Reference Count）。引用计数智能指针，是一种生命期受管的对象，其内部有一个引用计数器。当内部引用计数为零时，这些对象会自动销毁自身的智能指针类。每次创建类的新对象时，会初始化指针并将引用计数置为 1；当对象作为另一对象的副本而创建时，它会调用拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数；如果引用计数减至 0，则删除对象，并增加右操作数所指对象的引用计数；调用析构函数时，构造函数减少引用计数，直到计数为 0，释放对象空间。

Smart Pointer 具有非常强大的能力，谨慎而明智的选择能给我们带来极大的便利。前面已经说到 STL 中包含了支持 Smart Pointer 技术的 class，它就是智能指针：auto\_ptr。要使用 auto\_ptr，首先要包含 memory 头文件：

```
#include <memory>
```

auto\_ptr 可以指向一个以 new 建立的对象，当 auto\_ptr 的生命周期结束时，其所指向的对象之资源也会被自动释放，且不必显式地调用 delete，而对对象指针的操作依旧如故。例如：

```
class A
{
public:
    A(){}
    ~A(){}
    void Hello()
```

```

    {
        std::cout<<"Hello Smart Pointer";
    }
};

int main()
{
    std::auto_ptr<A> pA(new A());
    pA->Hello();
    return 0;
}

```

当然，也可以建立一个未指向任何对象的 auto\_ptr，例如：

```
std::auto_ptr<int> iPtr;
```

它就像空指针，未指向任何对象，所以也就不能进行操作，但是可以通过 get() 函数来判断它是否指向对象的地址：

```

if(iPtr.get() == 0) // 不指向任何对象
{
    iPtr.reset(new int(2011)); // 指向一个对象
}

```

auto\_ptr 还可以使用另一个 auto\_ptr 来建立，但是需要十分小心的是，这会造成所有权的转移，例如：

```

auto_ptr< string> sPtr1 (new string("Smart Pointer"));
auto_ptr< string> sPtr2 (sPtr1);
if( !sPtr1->empty() )
    cout<<*sPtr1<< endl;

```

当使用 sPtr1 来建立 sPtr2 时，sPtr1 不再对所指向对象的资源释放负责，而是将接力棒传递到了 sPtr2 的手里，sPtr1 丧失了使用 string 类成员函数的权利，所以在判断 sPtr1->empty() 时程序会崩溃。

auto\_ptr 的资源维护动作是以 inline 的方式来完成的，在编译时代码会被扩展开来，所以使用它并不会牺牲效率。虽然 auto\_ptr 指针是一个 RAII 对象，能够给我们带来很多便利，但是它的缺点同样不可小觑：

- ❑ auto\_ptr 对象不可作为 STL 容器的元素，所以二者带来的便利不能同时拥有。这一重大缺陷让 STL 的忠实拥趸们愤怒不已。
- ❑ auto\_ptr 缺少对动态配置而来的数组的支持，如果用它来管理这些数组，结果是可怕的、不可预期的。
- ❑ auto\_ptr 在被复制的时候会发生所有权转移。

Smart Pointer 作为 C++ 垃圾回收机制的核心，必须足够强大、具有工业强度，并且保证

安全性。可是 STL 中的 auto\_ptr 却像是扶不起的阿斗，不堪大用。在这样的情况下，C++ 标准委员会自然需要考虑引入新的智能指针。其中由 C++ 标准委员会库工作组发起的 Boost 组织开发的 Boost 系列智能指针最为著名。除此之外，还有 Loki 库提供的 SmartPtr、ATL 提供的 CComPtr 和 CComQIPtr。一个好消息是，就在 2011 年的 9 月刚刚获得通过的 C++ 新标准 C++ 11 中废弃了 auto\_ptr 指针，取而代之的是两个新的指针类：shared\_ptr 和 unique\_ptr。shared\_ptr 只是单纯的引用计数指针，unique\_ptr 是用来取代 auto\_ptr 的。unique\_ptr 提供了 auto\_ptr 的大部分特性，唯一的例外是 auto\_ptr 的不安全、隐性的左值搬移；而 unique\_ptr 可以存放在 C++0x 提出的那些能察觉搬移动作的容器之中。

在 Boost 中的智能指针共有五种：scoped\_ptr、scoped\_array、shared\_ptr、shared\_array、weak\_ptr，其中最有用的就是 shared\_ptr，它采取了引用计数，并且是线程安全的，同时支持扩展，推荐在大多数情况下使用。

boost::shared\_ptr 支持 STL 容器：

```
typedef boost::shared_ptr<string> CStringPtr;
std::vector< CStringPtr > strVec;
strVec.push_back( CStringPtr(new string("Hello")) );
```

当 vector 被销毁时，其元素——智能指针对象才会被销毁，除非这个对象被其他的智能指针引用，如下面的代码片段所示：

```
typedef boost::shared_ptr<string> CStringPtr;
std::vector< CStringPtr > strVec;
strVec.push_back( CStringPtr(new string("Hello")) );
strVec.push_back( CStringPtr(new string("Smart")) );
strVec.push_back( CStringPtr(new string("Pointer")) );

CStringPtr strPtr = strVec[0];
strVec.clear(); // strVec 清空，但是保留了 strPtr 引用的 strVec[0]
cout<<*strPtr<<endl; // strVec[0] 依然有效
```

Boost 智能指针同样支持数组，boost::scoped\_array 和 boost::shared\_array 对象指向的是动态配置的数组。

Boost 的智能指针虽然增强了安全性，处理了潜在的危险，但是我们在使用时还是应该遵守一定的规则，以确保代码更加鲁棒。

#### 规则 1：Smart\_ptr<T> 不同于 T\*

Smart\_ptr<T> 的真实身份其实是一个对象，一个管理动态配置对象的对象，而 T\* 是指向 T 类型对象的一个指针，所以不能盲目地将一个 T\* 和一个智能指针类型 Smart\_ptr<T> 相互转换。

- 在创建一个智能指针的时候需要明确写出 Smart\_ptr<T> tPtr<new T>。
- 禁止将 T\* 赋值给一个智能指针。

□ 不能采用 tPtr = NULL 的方式将 tPtr 置空，应该使用智能指针类的成员函数。

#### 规则 2：不要使用临时的 share\_ptr 对象

如下所示：

```
class A;
bool IsAllReady();
void ProcessObject(boost::shared_ptr< A> pA, bool isReady);
ProcessObject(boost::shared_ptr(new A), IsAllReady());
```

调用 ProcessObject 函数之前，C++ 编译器必须完成三件事：

- (1) 执行 "new A"。
- (2) 调用 boost::shared\_ptr 的构造函数。
- (3) 调用函数 IsAllReady()。

因为函数参数求值顺序的不确定性，如果调用 IsAllReady() 发生在另外两个过程中间，而它又正好出现了异常，那么 new A 得到的内存返回的指针就会丢失，进而发生内存泄露，因为返回的指针没有被存入我们期望能阻止资源泄漏的 boost::shared\_ptr 上。避免出现这种方式的方式就是不要使用临时的 share\_ptr 对象，改用一个局部变量来实现，在一个独立的语句中将通过 new 创建出来的对象存入智能指针中：

```
boost::shared_ptr<A> pA(new A)
ProcessObject(pA, IsAllReady());
```

如果疏忽了这一点，当异常发生时，可能会引起微妙的资源泄漏。

#### 请记住：

时刻谨记 RAI 原则，使用智能指针协助我们管理动态配置的内存能给我们带来极大的便利，但是需要我们谨慎而明智地做出选择。

## 建议 35：使用内存池技术提高内存申请效率与性能

Doug Lea 曾有言曰：“自 1960 年以来，动态内存分配已经成为大多计算机系统的重要部分。”<sup>①</sup>

动态内存管理确实是件让人头疼的事儿，然而在实际的编程实践中，又不可避免地要大量用到堆上的内存。而这些通过 malloc 或 new 进行的内存分配却有着一些天生的缺陷：一方面，利用默认的内存管理函数在堆上分配和释放内存会有一些额外的开销，需要花费很多时间；另一方面，也是更糟糕的，随着时间的流逝，内存将形成碎片，一个应用程序的运行会越来越慢。

<sup>①</sup> Dynamic memory allocation has been a fundamental part of most computer systems since roughly 1960... 摘自 Doug Lea 所写文章《A Memory Allocator》，详见 <http://gee.cs.oswego.edu/dl/html/malloc.html>。

当程序中需要对相同大小的对象频繁申请内存时，常会采用内存池（Memory Pool）技术来提高内存申请效率。经典的内存池技术，是一种用于分配大量大小相同的小对象的技术。通过该技术可以极大地加快内存分配 / 释放过程。内存池技术通过批量申请内存，降低了内存申请次数，从而节省了时间。对于大批量的小对象而言，使用内存池技术整体申请内存，减少了内存碎片的产生，对性能提升的帮助也是很显著的。

内存池技术的基本原理通过这个“池”字就进行了很好的自我阐释：应用程序可以通过系统的内存分配调用预先一次性申请适当大小的内存块（Block），并会将它分成较小的块（Smaller Chunks），之后每次应用程序会从先前已经分配的块（chunks）中得到相应的内存空间，对象分配和释放的操作都可以通过这个“池”来完成。只有当“池”的剩余空间太小，不能满足应用程序需要时，应用程序才会再调用系统的内存分配函数对其进行动态扩展。

经典的内存池实现原理如下：

```
class MemPool
{
public:
    MemPool(int nItemSize, int nMemBlockSize = 2048)
        : m_nItemSize(nItemSize),
          m_nMemBlockSize(nMemBlockSize),
          m_pMemBlockHeader(NULL),
          m_pFreeNodeHeader(NULL)
    {
    }
    ~MemPool();
    void* Alloc();
    void Free();
private:
    const int m_nMemBlockSize;
    const int m_nItemSize;

    struct _FreeNode
    {
        _FreeNode* pPrev;
        BYTE data[m_nItemSize - sizeof(_FreeNode*)];
    };

    struct _MemBlock
    {
        _MemBlock* pPrev;
        _FreeNode data[m_nMemBlockSize/m_nItemSize];
    };

    _MemBlock* m_pMemBlockHeader;
    _FreeNode* m_pFreeNodeHeader;
};
```

其中 MemPool 涉及两个常量：m\_nMemBlockSize、m\_nItemSize，还有两个指针变量 m\_pMemBlockHeader、m\_pFreeNodeHeader。指针变量 m\_pMemBlockHeader 是用来把所有申请的内存块（MemBlock）串成一个链表。m\_pFreeNodeHeader 变量则是把所有自由的内存结点（FreeNode）串成一个链表。内存块在申请之初就被划分为了多个内存结点，每个结点的大小为 ItemSize（对象的大小），共计 MemBlockSize/ItemSize 个。然后，这些内存结点会被串成链表。每次分配的时候从链表中取一个给用户，不够时继续向系统申请大块内存。在释放内存时，只须把要释放的结点添加到自由内存链表 m\_pFreeNodeHeader 中即可。在 MemPool 对象析构时，可完成对内存的最终释放。

Boost 库同样对该技术提供了较好的支持：

□ pool (#include <boost/pool/pool.hpp>)

boost::pool 用于快速分配同样大小的小块内存。如果无法分配，返回 0，如下所示。

```
boost::pool<> p(sizeof(double)); // 指定每次分配块的大小
if(p!=NULL)
{
    double* const d = (double*)p.malloc(); // 为 d 分配内存
    pA.free(d); // 将内存还给 pool
}
```

pool 的析构函数会释放 pool 占用的内存。

□ object\_pool (#include <boost/pool/object\_pool.hpp>)

object\_pool 和 pool 的区别在于：pool 指定每次分配的块的大小，object\_pool 指定分配的对象的类型，如下所示：

```
boost::object_pool<A> p;
```

用 A \* pA = p.malloc() 只会分配内存而不会调用构造函数，如果要调用构造函数应该使用 A \* const t = p.construct();

□ singleton\_pool (#include <boost/pool.singleton\_pool.hpp>)

singleton\_pool 和 object\_pool 一样，不过它可以定义多个 pool 类型的 object，给它们都分配同样大的内存块，另外 singleton\_pool 提供静态方法分配内存，且不用定义对象，如下所示：

```
struct PoolTag{};
typedef boost::singleton_pool<PoolTag,sizeof(int)> User_pool;
int * const t = User_pool::malloc();

my_pool::purge_memory(); // 用完后释放内存
```

□ pool\_allocator (#include <boost/pool/pool\_alloc.hpp>)

pool\_allocator 基于 singleton\_pool 实现，提供 allocator，可用于 STL 等：

```
std::vector<int, pool_allocator<int>> v;
v.push_back(13);
boost::singleton_pool<sizeof(int)>::release_memory(); // 显式调用释放内存
```

由此可见，Boost 确实是一个值得称赞、更值得使用的库，它能为我们提供极大的便利。当需要使用内存池技术时，请考虑 Boost。

---

**请记住：**

当你需要频繁地分配相同大小的对象，而又苦恼于默认的内存管理函数带来的问题时，内存池技术将是灵丹妙药，它能提高内存操作效率，以及应用程序的鲁棒性。

---

# 第4章 重中之重的类

正如 C++ 的乳名“C with class”所透露出的信息那样，类（class）在 C++ 世界中绝对是一个最重要的概念。精巧的类机制使得程序员可以脱离底层的枝枝蔓蔓，集中心智在对象层次上以完成设计。

所以，要学好 C++，类是必须了解熟悉的内容。

## 建议 36：明晰 class 与 struct 之间的区别

C++ 最初被称作为“C with class”，足见 class 在 C++ 语言中的地位。在 C++ 中，我们可以采用 class 自定义用户数据类型，不过，它还存在一个胞弟——struct，它俩似乎有着太多相同的特点与功能，因此很多人对二者的区别都不是很了解。

因为 C++ 之父在设计 C++ 语言时就考虑到要向下兼容 C，所以，C++ 中的很多东西都能在 C 语言中找到踪迹。因此，要谈 struct 就应该从 C 谈起。在 C 中 struct 具有如下定义：

struct 是一种自定义的数据类型。它是由若干“成员”组成的。每一个成员可以是一个基本数据类型或是一个用户自定义类型。其一般的定义形式为：

```
struct 结构名
{
    成员表列
};
```

既然 struct 是一种数据类型，那么就肯定不能定义函数。所以在面向过程的 C 中，struct 不能包含任何的函数，如下所示：

```
struct Rectangle
{
    int length;
    int width;
};
```

如果出现下面这样的结构体定义，编译器将报错：

```
struct Rectangle
{
    int length;
    int width;
    int GetSize(){ return length*width; }
};
```

面向过程的编程认为，数据和数据操作应该是分开的。然而当 struct 进入面向对象的 C++ 新时代时，其特性也有了新发展，比如上述在 C 中出现错误的函数，在 C++ 标准中却得到了允许，因为面向对象的编程观点认为，数据和数据的操作是一个整体，不应该分开。这就是 struct 在 C 与 C++ 两个时代的差异性。C++ 中 struct 被看成是一个对象，它可以包含函数，可以拥有构造函数、析构函数，同样拥有继承等能力。

如此一来，很多人不免疑惑：那 C++ 中的 struct 和 class 不就一样了么？是的，它们使用起来差别真的并不大。其主要差别有以下几点：

#### □ 关于使用大括号初始化

class 和 struct 如果定义了构造函数，就都不能用大括号进行初始化了；如果没有定义，struct 可以用大括号初始化，而 class 只有在所有成员变量全是 public 的情况下，才可以用大括号进行初始化。来看一段代码：

```
struct SA
{
    int a;
    int b;
};

SA data1 = {2, 3}; // OK

struct SB
{
    int a;
    int b;
    SB(int x, int y) : a(x), b(y){}
    ~SB(){}
};

SB data2 = {2, 3}; // ERROR

class CA
{
public:
    int a;
    int b;
};

CA data3 = {2, 3}; // OK

class CB
{
public:
    int a;
    int b;
    CB(int x, int y) : a(x), b(y){}
    ~CB(){}
};

CB data4 = {2, 3}; // ERROR

class CC
{
public:
```

```

CC(int x, int y) : a(x), b(y) {}
~CC() {}

private:
    int a;
    int b;
};

CC data5 = {2, 3}; //ERROR

```

在 VS2010 下编译上述代码会提示出现以下错误：

```
error C2552: "****": 不能用初始值设定项列表初始化非聚合
```

关于这种初始化的不同，在新的标准 C++ 11 中已不复存在了。因为新标准制定了统一的初始化语法，如下面的代码片段所示<sup>⊖</sup>：

```

// 类初始化
C c{0,0}; // 相当于 C++0X 中的：C c(0,0);
// 数组初始化
int* a = new int[3]{1, 2, 0};
// 成员变量初始化
class X
{
    int a[4];
public:
    X(): a{1,2,3,4} {}
};

// vector 容器初始化
vector<string> vs = {"first", "second", "third"};
// map 容器初始化
map singers =
{ {"Lady Gaga", "+1 (212) 555-7890"}, 
  {"Beyonce Knowles", "+1 (212) 555-0987"} };

```

虽然这种大括号初始化方式在新标准中得到了统一，但是现在它也仅仅是标准。在新标准全面执行之前，这种不同仍旧存在，这一问题也仍值得我们注意。

#### □ 关于默认访问权限

class 中默认的成员访问权限是 private 的，而 struct 中则是 public 的。看看下面这段代码：

```

struct SA
{
    int a;
    int b;
};

class CA

```

<sup>⊖</sup> 代码片段摘自前 C++ 标准委员会的 Danny Kalev 所写的文章《The Biggest Changes in C++11 (and Why You Should Care)》。

```

{
    int a;
    int b;
};

SA data1 = {2, 3};
CA data3 = {2, 3};
cout<<data1.a<<data3.a<<endl;

```

在 VS2010 中编译上述代码，会出现以下错误提示：

```
error C2248: "CA::data3": 无法访问 private 成员 (在 "CA" 类中声明)
```

#### □ 关于继承方式

class 继承默认是 private 继承，而 struct 默认是 public 继承，代码如下所示：

```

struct SA
{
    int a;
    int b;
};

struct D1 : SA
{
    int c;
};

class CA
{
public:
    int a;
    int b;
};

class D2 : CA
{
public:
    int c;
};

D1 a;
D2 b;
cout<<a.a<<endl;
cout<<b.a<<endl; // ERROR

```

在 class 的地盘上，struct 有了狗拿耗子多管闲事的嫌疑。但是考虑到“对 C 兼容”这一承诺，struct 就被保留了下来，并做了一些扩展，使其继承于 C 但又适合于面向对象。有一点你必须明确：在 C++ 中 struct 已经被扩展了，它已经不再是 C 时代的那个 struct 了。

其实，struct 与 class 之间最大的区别在于思想上。C 语言的编程单位是函数 (function)，语句 (statements) 是程序的基本单元。而 C++ 语言的编程单位是类 (class)。从 C 到 C++，

程序设计由以过程设计（模块化编程，modular programming）为中心向以数据组织（数据隐藏，data-hiding principle）为中心转移了。而从 struct 到 class 的改变就是这种编程思想变化的最好体现。

#### 请记住：

class 作为 C++ 中最重要的角色，与其胞兄 struct 在使用上有着一些细微的不同。我们在使用过程中要对这些不同有所了解，更重要的是要通过 struct 与 class 之间的不同体会程序设计思想上的变化。

### 建议 37：了解 C++ 悄悄做的那些事

所有的类都有着一个类似的中枢骨干，人送外号“Big Three”：

一个或多个构造函数 + 一个析构函数 + 一个拷贝赋值运算符

它们控制着类的基本操作：新对象的创建与初始化、为对象赋予一个新值，以及类的消亡清理。难道就没有一个类超出“三界”，打破这一规律？

答案是没有，在类的世界里，没有例外。即使像下面这样去完成一个空类 CEmpty 的定义：

```
class CEmpty { };
```

虽然你自己没有去声明 Big Three，但是编译器会悄悄地为你声明一个它自己的版本。所以当你写下空类 CEmpty 的定义时，其本质和写出下面一堆代码是一样的：

```
class CEmpty
{
public:
    CEmpty();
    CEmpty(const CEmpty&);
    ~CEmpty();
    CEmpty& operator=(const CEmpty& rhs);
};
```

这是多么奇妙，同时又是多么可怕啊！C++ 编译器竟然做了这么多的事儿。不过这些函数只有在它们被需要的时候才会生成，如下所示：

```
void Function()
{
    CEmpty e1;           // default constructor;
    CEmpty e2(e1);      // copy constructor
    e2 = e1;            // copy assignment operator
}
```

当然不要忘记了在离开 Function 函数时默默调用的析构函数 ~CEmpty()。

需要注意的是，编译器版本的拷贝构造函数和拷贝赋值运算符，只是简单地将原对象的每一个 Non-static 数据成员拷贝到目标对象中了，简单粗暴，因而容易出现问题。这一点将在建议 40 详细阐述。

而在 C++ 11 中，标准引入了两个指示符 delete 和 default，以便让程序员自行决定是否需要编译器为我们偷偷生成这些函数。delete 意为告诉编译器不自动产生这个函数，default 告诉编译器产生一个默认的。比如下面的代码片段：

```
//C++11 only
class A
{
    A()=default;                      // 需要生成
    virtual ~A()=default;              // 需要生成
    A & operator=( const A& ) = delete; // 禁止生成
    A( const A& ) = delete;          // 禁止生成
};
```

除了添加“主干”函数，对于空类 CEmpty，C++ 编译器还多做了一件事：为空类隐含地加一个字节。按照一般逻辑，CEmpty 类中不包含任何数据，是名副其实的空类，所以它的大小也应该是 0。得出这样的结论貌似有很强大的逻辑支持，可事实呢？我们来看一段代码：

```
#include<iostream>
int main()
{
    std::cout<<"sizeof(CEmpty) = "<<sizeof(CEmpty)<<"\n";
    return 0;
}
```

程序在 32 位系统中执行的结果输出如下：

```
sizeof(CEmpty) = 1
```

为什么会出现这样的结果呢？这是多不符合我们的逻辑！

空类大小不为 0 的原因还要从类的实例化说起。所谓类的实例化就是在内存中划分一块空间分配给一个具体的对象，每个实例在内存中都会被赋予一个独一无二的地址，空类也不例外。为了达到这个目的，编译器会悄悄地给一个空类隐含地加一个字节，这样空类在实例化后就能在内存得到独一无二的地址了。所以，空类的大小也就由 0 变为 1 了。

#### 请记住：

对于类，编译器会在悄悄地完成很多事：隐式产生一个类的默认构造函数，拷贝构造函数，拷贝赋值运算符和析构函数。而对于特殊的空类，为了能够实现它的实例化，编译器还会“强制”使其大小由 0 变成 1。

## 建议 38：首选初始化列表实现类成员的初始化

在 C++ 语言中，位列 Big Three 之首的构造函数被赋予了一定的功能：对类成员变量完成初始化赋值操作。这一步在类的生命过程中十分重要。打个比方，如果说应用程序从操作系统申请获得内存就如同地产商从政府手中拍得一块土地，那么类成员的初始化就是建筑商在这块竞得的土地上建起一栋未经装修的公寓。杂草丛生的土地经过建设具备了商品房的基本形态，而空空如野的 raw 内存经过初始化就被赋予了对象的生命气息。

类成员的初始化可采用如下两种形式来完成：在构造函数体中赋值完成和用初始化类成员列表完成，我们分别来看看。首先假设类声明如下：

```
class CStudent
{
public:
    CStudent(string name, int age, int id);
    ~CStudent();
private:
    string m_name;
    int    m_age;
    int    m_ID;
};
```

下面是在构造函数体中赋值完成。

```
CStudent::CStudent(string name, int age, int id)
{
    m_name = name;
    m_age  = age;
    m_ID   = id;
}
```

下面是用初始化类成员列表来完成。

```
CStudent::CStudent(string name, int age, int id)
: m_name(name)
, m_age(age)
, m_ID(id)
{ }
```

从功能上来讲，两种方法都可以。但是二者实现的细节稍有不同，第一种在构造函数体内实现的 “=” 操作本质是赋值（Assign）操作，而第二种才是真正的初始化（Initialization）。虽然一般情况下这两种方式都适用，但有些情况却只能使用第二种，比如：

□ const 成员变量只能用成员初始化列表来完成初始化，而不能在构造函数内被赋值。

来看一个代码片段，如下所示：

```
class A
```

```

{
public:
    A(int data = 0);
    ~A();
private:
    int m_data;
    const int DATA_TYPE;
};

```

如果采用最为熟悉的构造函数内赋值的方式初始化，其代码如下：

```

A::A(int data)
{
    m_data = data;
    DATA_TYPE = 0;
}

```

此代码在 VS2010 下编译时，会报错 “error C2758: A::DATA\_TYPE”：必须在构造函数基成员初始值设定项列表中初始化。DATA\_TYPE 是 const 的，是常量，按理来说不能被赋值，所以在这里这种初始化方式会宣告失败，那就只能采用初始化类成员列表的方式了，代码如下所示：

```

A::A(int data):DATA_TYPE(0) // OK
{
    m_data = data; // 最好也放入初始化成员列表
}

```

□ 如果类 B 中含有 A 类型的成员变量，而类 A 中又禁止了赋值操作，此时要想顺利地完成 B 中成员变量的初始化，就必须采用初始化列表方式。先看下面这段代码：

```

class A
{
private:
    A operator=( const A& rhs);
};

class B
{
public:
    B();
    ~B();
private:
    A m_A;
};

```

此时要在 B 的构造函数中调用 A 的赋值函数已经不可能，这时如果采用构造函数内赋值的方式对成员变量 m\_A 进行初始化，显然会力不从心，代码如下所示：

```
B::B()
{
    // error C2248: "A::operator =" : 无法访问 private 成员
    m_A = A();
}
```

解决的办法就是采用初始化列表方式，代码如下：

```
B::B():m_A(A()){ }
```

即使没有禁用赋值操作，还是不推荐采用构造函数体内的赋值初始化方式。因为这种方式存在着两个问题。第一，比起初始化列表，此方式效率偏低；第二，留有错误隐患。

假设类 B 中含有一个 A 类型的成员变量：

```
class A{ };

class B
{
public:
    B();
    ~B();
private:
    A m_A;
};
```

再用构造函数赋值方式，即：

```
B::B()
{
    m_A = A();
}
```

在这个过程中，会产生临时对象，临时对象的构造析构会造成无谓的效率损耗，而初始化列表方式就避免了产生临时对象所带来的问题。其实这就是赋值与初始化的区别。关于它们在效率上的差异，下面实验获得的数据绝对是最有利的证据：

```
int main()
{
    clock_t startTime = 0, endTime = 0;
    startTime = clock();
    for(int i=0; i<5000; i++)
    {
        CStudent student("Li Lei", 21, 2011001);
    }
    endTime = clock();
    int lastTime = endTime - startTime;
    return 0;
}
```

在 Intel Core(TM)2 Quad CPU Q8400, 2.67GHz 的机器上，我分别采用两种初始化方式进行了多次实验。采用赋值初始化方式时，平均的执行时间为 47ms，而初始化列表方式耗时为 31ms，性能提升了约 34%。

另外，之所以说它留有隐患，可能引发问题，是因为 C++ 会悄悄为我们加上默认的赋值函数，可对于千变万化的类而言，默认的赋值操作很可能存在问题。如果你没有注意，既没有重写，也没有禁止，那它极有可能在其他类的构造函数中引发问题。

对于初始化列表，也有一个问题需要说明，那就是初始化的顺序。与构造函数中的赋值方式不同，初始化列表中成员变量出现的顺序并不是真正初始化的顺序，初始化的顺序取决于成员变量在类中的声明顺序。例如：

```
class C
{
public:
    int m_num;
    std::string m_str;
    double* m_p;
    C():m_str("Hello"),m_p(NULL),m_num(0) { }
};
```

其初始化顺序是：m\_num → m\_str → m\_p。

需要注意的是，只有保证成员变量声明的顺序与初始化列表顺序一致才能真正保证其效率。

虽然初始化列表具有如此之多的优势，但赋值初始化并不是一无是处，如果遇到了以下的情况，采用赋值初始化也许更加便利：

```
class D
{
public:
    D(int data);
    ~D() {}
private:
    int m_a;
    int m_b;
    int m_c;
    int m_d;
    int m_e;
};

//Initiation_List Version
D::D(int data)
:m_a(data)
,m_b(data)
,m_c(data)
,m_d(data)
```

```

, m_e(data) {}

//Assignment Version
D::D(int data)
{
    m_a = m_b = m_c = m_d = m_e = data;
}

```

#### 请记住：

在 class 中成员变量初始化是一个不可或缺的步骤；在初始化成员变量时，出于对方法通用性及高效性的考虑，强烈推荐采用成员变量初始化列表的方式初始化变量，并且要保成员变量声明的顺序与初始化列表的顺序一致。

### 建议 39：明智地拒绝对象的复制操作

通过建议 37，我们了解到了 C++ 语言机制背地里为我们做的一些事情。但是，有时候这些事并不是我们想要的，比如，我们并不想使用编译器为我们产生的函数，也不想让使用我们所写代码的人使用这些函数。那么，我们应该明确地去拒绝（比如我们想禁止某些对象的复制时），当断则断，以免其乱！

假设你手头有一个项目是设计一套服务于某高校的学生缴费系统，很自然我们会想到选用一个类来表示学生信息：

```

class CStudent
{
public:
    CStudent(std::string name = "", int age = 0, int id = 0, int fund = 0);
    CStudent(const CStudent& other);
    ~CStudent();
    CStudent operator=( const CStudent & rhs );
private:
    std::string m_sName;
    int      m_nAge;
    int      m_nStudentID;
    int      m_nFund;
};

```

因为每一个学生都是独一无二的个体，绝对不会存在所有信息一模一样的情况。所以在这么重要的缴费系统中对 CStudent 对象进行复制就违反常理了。因而上述类声明中的拷贝构造函数、赋值函数都变得毫无意义了。有人企图通过注释的方式来告诉代码使用者：不能使用拷贝构造函数和赋值函数。虽然这是一种方法，但绝对不是一种好的处理机制，如果使用者根本不在意你的注释呢？所以最好的方法就是让 CStudent 对象的复制操作编译通不过，代

码如下所示：

```
CStudent s1("Li Lei",21, 2007209, 3960);
CStudent s2("Han Meimei",20, 2007210, 3960);
CStudent s3(s1); // compile error!!!
CStudent s4 = s2; // compile error!!!
```

通常而言，如果你不希望一个 class 支持某种操作，最简单的方法就是什么也不做，因为编译器会对你使用根本没有声明过的函数这一行为给予最直接的回击：ERROR。但是，对于复制操作运算它却无能为力，因为编译器不能一方面为你悄悄地声明它们，另一方面却又告诉你它的声明是无效的，那是自己打自己的嘴巴，愚蠢至极！

那我们应该采取什么样的方法来解决这一棘手的问题呢？类的访问控制（Access Control）为我们指明了前进的方向。所有的编译器生成的函数都是 public 的，即公有的，如果我们将它声明为 private 的，一方面阻止了编译器越俎代庖式的声明与定义，另一方面又能禁止对复制操作函数的显式调用，如下所示：

```
class CStudent
{
public:
    ...
private:
    CStudent(const CStudent& other);
    CStudent operator=( const CStudent & rhs);
    ...
};
```

但是，这种解决方案并不完美，如果成员函数和友元函数要对其访问，仍然可以畅通无阻而且合理合法，因为 C++ 的语言机制赋予了它们访问调用 private 成员函数的权利。不过天无绝人之路，假如我们只声明不定义，那么成员函数、友元函数对它们的调用也就会被编译器明令禁止：声明而不定义成员函数是合法的，但是，使用未定义成员函数的任何尝试将导致链接失败。这样就实现了类复制的完全禁止，用户代码中的复制尝试将在编译时标记为错误，而成员函数和友元函数中的复制尝试将在链接时出现错误。

上面介绍的这种技术在你熟悉的 std::iostream 类中已经得到了很好的应用，诸如 ios\_base、basic\_ios 和 sentry，都是采用这样的方式不允许复制操作的，感兴趣的话你可以去标准库源码中一探究竟。

说到拒绝类拷贝赋值操作这一点，还不得不提起令人称赞的 Boost 库。因为 Boost 库为我们提供了另一种解决方式。这种方式更加完美，因为它可以将链接错误提前到编译时，毕竟早一点发现错误比晚发现要更好。

它没有对类似 CStudent 的类声明 private 的拷贝构造函数和拷贝赋值运算符，而是特意声明了一个不可复制的类 boost::noncopyable。这个类本身非常简单，如下所示：

```

namespace noncopyable_ // protection from unintended ADL
{
    class noncopyable
    {
        protected:
            noncopyable() {}
            ~noncopyable() {}
        private: // emphasize the following members are private
            noncopyable( const noncopyable& );
            const noncopyable& operator=( const noncopyable& );
    };
}

```

为了禁止拷贝 CStudent 对象，我们只需要让其继承自 boost:: noncopyable，如下所示：

```

class CStudent: private boost:: noncopyable
{
...
};

```

当调用拷贝构造或赋值操作符对 CStudent 对象进行复制时，编译器将试图生成一个拷贝构造函数和一个拷贝赋值运算符，而它调用这些函数时会不可避免地调用基类的对应函数，可是在基类中这些操作是 private 的，所以这样的操作会被编译器拒绝。

仔细体会这段代码，还会发现使用 boost:: noncopyable 时应该注意以下地方：

- 从 noncopyable 继承时不能是 public 的。
- 多重继承有时会使空基类 noncopyable 优化失效，所以这不适用于多重继承的情形。

---

#### 请记住：

在某些需要禁止对象复制操作的情形下，可以将这个类相应的拷贝构造函数、赋值操作符 operator= 声明为 private，并且不要给出实现。或者采用更简单的方法：使用 boost:: noncopyable 作为基类。

---

## 建议 40：小心，自定义拷贝函数

在你声明一个类时，编译器会偷偷地为你声明类体系中的“Big Three”，其中的拷贝构造函数（Copy Constructor）和赋值运算符（Assignment Operator）都是用于对象拷贝的，所以将它们统称为拷贝函数（Copying Functions）。编译器在生成拷贝函数时会对所有的类一视同仁，不会特殊情况特殊处理，它只是简单地将原对象的每一个 Non-static 数据成员拷贝到目标对象中，这就是我们所说的浅拷贝。这个过程简单粗暴，如果类中有动态配置的内存，对象中包含资源，问题就会随之而产生。所以，如果类中具有动态配置的内存，请自行实现

拷贝函数，并告诉编译器你拒绝了它的默认实现。

首先，我们介绍一下浅拷贝和深拷贝可能会产生的问题。

浅拷贝是成员数据之间的一一赋值，又称为 Bitwise Copy。但是可能会有这样的情况：如果对象包含资源（堆资源或文件），当进行浅拷贝时，两个资源指针会指向同一资源，这就会造成两个对象访问同一资源，于是问题就出现了。比如下面的代码片段：

```
class CString
{
public:
    CString();
    CString(const char* data);

private:
    char* m_pData;
    int   m_nSize;
};

CString strSrc("Hello everyone!");
CString strDst(strSrc);
```

当我们写下上面这样的代码时，就会出现如图 4-1 所示的情况。

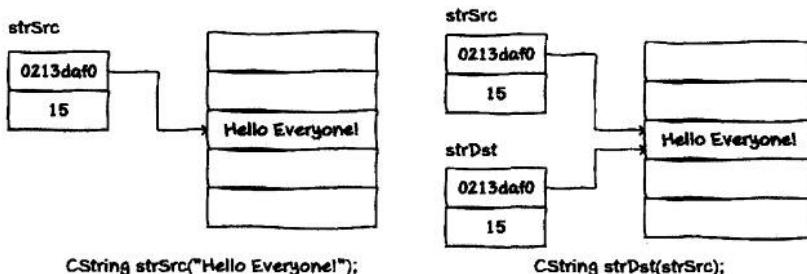


图 4-1 浅拷贝示意图

很明显，浅拷贝的行为类似于 `memcpy`。于是出现了如下问题：在释放资源的时候会产生资源归属不清的情况，这将导致程序运行出错，这是一个绝对危险的 Bug。

深拷贝就是用来解决这样的问题的，它会把资源也赋值一次，使对象拥有不同的资源，但其资源的内容是一样的。以堆资源为例，就是再开辟一片堆内存，把原来的内容拷贝过来，如图 4-2 所示。

`strDst` 与 `strSrc` 中的指针 `m_pData` 不再一样，它们分别指向了不同的内存块，但是这两个指针所指向的内存块中的内容却是一样的。

编译器为我们生成的拷贝函数都是浅拷贝版本。当类内没有动态配置的资源时，它会正常工作，而且工作得相当不错；但是如果出现了动态配置的资源，我们就不得不依靠自己的智慧与力量来动手消灭问题了。

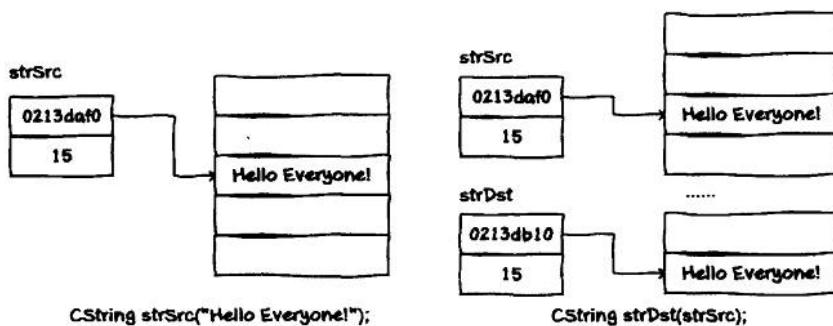


图 4-2 深拷贝示意图

自定义拷贝函数是一种良好的编程风格，它可以阻止编译器形成默认的拷贝函数。但是在拒绝编译器默认版本的同时，你也要注意将拷贝类中所有的数据成员拷贝全了。如果你忘记拷贝某一个数据成员，你的代码并不完善，但编译器为了“报复”，是不会告诉你相关情况的，它对此依旧会表示同意。假如我们在为 CString 自定义拷贝函数时忽略了数据成员 `m_nSize`，代码如下所示：

```

CString::CString(const CString& rhs)
{
    m_pData = new char[strlen(rhs.m_pData)+1];
    strcpy(m_pData, rhs.m_pData);
}
CString& CString::operator=(const CString& rhs)
{
    if(this == &rhs) return *this;
    if(m_pData!=NULL) delete m_pData;
    m_pData = new char[strlen(rhs.m_pData)+1];
    strcpy(m_pData, rhs.m_pData);
    return *this;
}

```

这种实现方式大部分编译器既不会报错，也不会通过警告提示你忘记拷贝某个数据成员。这种错误一般发生在两个情形之中：

(1) 对原有类进行了新设计，添加了新的数据成员，但是没有对该类的拷贝函数做适时的更新。

(2) 有继承发生时，忘记对基类部分的数据进行拷贝了。例如，我们在 CString 的基础之上派生出了带颜色的字符串类 CColorString，代码如下所示：

```

class CColorString : public CString
{
public:
    CColorString(const CColorString& rhs)

```

```

:m_dColor(rhs.m_dColor) {}

CCоВorString& operator=(const CCоВorString& rhs)
{
    if(this == &rhs) return *this;
    m_dColor = rhs.m_dColor;
    return *this;
}
private:
    COLOR m_dColor;
};

```

上述的拷贝函数貌似对所有数据成员都进行了相应的拷贝，但是它却忽略了基类部分。派生类的拷贝函数必须调用对应的基类函数，所以当你打算自己为一个派生类自定义拷贝函数时，必须注意同时拷贝其基类部分。上述 CCоВorString 正确的定义方式应该是：

```

class CCоВorString : public CString
{
public:
    CCоВorString(const CCоВorString& rhs)
        :CString(rhs), m_dColor(rhs.m_dColor){}
    CCоВorString& operator=(const CCоВorString& rhs)
    {
        if(this == &rhs) return *this;
        CString::operator=(rhs);
        m_dColor = rhs.m_dColor;
        return *this;
    }
private:
    COLOR m_dColor;
};

```

---

#### 请记住：

如果类内部出现了动态配置的资源，我们就不得不自定义实现其拷贝函数了。在自定义拷贝函数时，应该保证拷贝一个对象的 All Parts：所有数据成员及所有的基类部分。

---

### 建议 41：谨防因构造函数抛出异常而引发的问题

正如我们所看到的构造函数声明一样，构造函数是没有返回值的，所以我们不能依靠返回值来判断构造对象成功与否。为了解决这个问题，C++ 之父给了我们“标准”的解决方法：抛出一个异常。然而，在 C++ 中，对象构造过程中抛出的异常绝对是一个比较棘手、难以处理的问题。

虽然对象在构造过程中出现了异常，但它已经被赋予了部分生命。面对这么一个半死不

活的对象，我们该如何处理呢？

熟悉 Delphi 语言的程序员一定知道：Delphi 在调用构造函数时如果产生异常，一定会先调用析构函数，将分配的资源自动回收，然后再抛出异常。因为在 Delphi 语言的世界里任何对象都有死亡的权利，即使它是一个半死不活的东西。

然而，C++ 程序员在这件事上只有羡慕的份儿：如果一个 C++ 对象在出生的过程中出现问题，那它就不能称之为对象。既然它不是一个对象，析构也就无从谈起了。来看一段代码：

```
class CObject
{
public:
    CObject()
    {
        m_pSubObject = new CSubObject(); // ①
        .....
    }
    ~CObject()
    {
        .....
        delete m_pSubObject;
    }
private:
    CSubObject* m_pSubObject;
}
```

如果在代码①处发生了异常，则不会调用析构函数，因此在构造函数中①之前所创建的对象 m\_pSubObject 就得不到释放，这会造成内存泄露。所以抛出异常后的对象清理工作需要由程序员来负责了。回想一下自己的程序中是否也存在类似的问题，如果回答是 Yes，那你可就要注意了，这样的 C++ 代码是不安全的！

如果在 C++ 的构造函数里创建了其他东西，你就必须考虑构造函数发生异常的情况。在 C++ 构造函数中，正确的异常处理方法应该是这样的：发生异常时先将已经创建的东西释放掉，然后再重新抛出异常给上层调用代码去处理。因此，前面构造函数的“安全版本”就变成了这样：

```
CObject::CObject()
{
    m_pSubObject = new CSubObject();
    try
    {
        .....
    }
    catch(...)
    {
        delete m_pSubObject; // 清理对象
        throw; // 抛出异常，交给上层调用
    }
}
```

如果你讨厌 try...catch... 形式的代码繁琐，智能指针也是一个很不错的解决方法，它巧妙地利用 C++ 退出作用域时自动释放变量的机制来清理其维护的对象，代码如下所示：

```
class CObject
{
public:
    CObject()
    {
        m_pSubObject = new CSubObject();
        ..... // 尽管去产生异常吧！！！
    }
    ~CObject()
    {
        ..... // 对象的清理工作交由 std::auto_ptr 自动处理
    }
private:
    std::auto_ptr<CSubObject> m_pSubObject;
}
```

但是你不要忘记建议 34 提到的 std::auto\_ptr 的一些缺陷。

当然，我们还可以采用一些“非常规”的方式进行处理，方法是：在类中增加 Init() 函数和 Release() 函数，用于负责资源的分配与清理。在构造函数中调用 Init() 函数来完成对象的创建，然后通过 Init() 函数返回值判断对象的构建是否成功；如果失败，则调用释放函数 Release()。如下所示：

```
class CObject
{
public:
    CObject()
    {
        m_pSubObject = NULL;
        if(!Init())
            Release();
    }
    ~CObject()
    {
        Release();
    }
    bool Init()
    {
        try
        {
            m_pSubObject = new CSubObject();
            ... // other data's initiations
        }
        catch(...)
        {
            return false;
        }
    }
}
```

```

    }
    return true;
}
void Release()
{
    if(m_pSubObject==NULL) return;
    delete m_pSubObject;
    m_pSubObject = NULL;
}
private:
    CSubObject* m_pSubObject;
    ... // other member data
}

```

上述方式的核心思想就是通过类的设计来避免在构造函数中抛出异常，进而避免因为异常而引发的问题。

对象的部分构造是很常见的，异常的发生点也完全是随机的，这是我们不得不面对的事实。所以程序员要谨慎处理这种情况。

---

#### 请记住：

构造函数抛出异常会引起对象的部分构造，因为不能自动调用析构函数，在异常发生之前分配的资源将得不到及时的清理，进而造成内存泄露问题。所以，如果对象中涉及了资源分配，一定要对构造之中可能抛出的异常做谨慎而细致的处理。

---

## 建议 42：多态基类的析构函数应该为虚

在我们的项目中，常会声明并使用基类来实现应用程序的设计。比如，在各公司中，不同的员工会有不同的工作职能，像架构师、程序员、测试人员等，他们的工作职能就各不相同，但因为他们存在一定的共同点，所以我们会选择为他们设置一个基类，代码如下所示：

```

class CEmployee
{
public:
    CEmployee();
    ~CEmployee();
    .....
};

```

又因为这几种工作之间存在差异，所以还要为不同的工种建立特有的派生类，代码如下所示：

```

class CFrameworker : public CEmployee
{

```

```

public:
    CFrameworker();
    ~CFrameworker();
    .....
};

class CProgrammer : public CEmployee
{
public:
    CProgrammer();
    ~CProgrammer();
    .....
};

class CTester : public CEmployee
{
public:
    CTester();
    ~CTester();
    .....
};

```

当我们使用公共继承时，就会在基类与派生类之间创建一种 Is-A 关系，不管是架构师 (Frameworker)、程序员 (Programmer)，还是测试人员 (Tester)，他们都是职员 (Employee)。同时我们又习惯于用 CEmployee 来代表一个职员对象，而不管他的具体工作细节。因此，基类 CEmployee 的指针和引用实际上可以指向任何一个派生的对象：

```
CEmployee* pEmployee = CreateEmployee(type);
```

CreateEmployee(type) 返回的是一个建立在堆上的对象，为了避免内存泄露的发生，我们必须在合适的时机将这个对象完全删除，如下所示：

```
CEmployee* pEmployee = CreateEmployee(type);
..... // use it do what you want to
delete pEmployee; // 问题出现
```

在上述代码片段中，使用完 pEmployee 对象后，我们没有忘记及时地采用 delete pEmployee 将该块内存释放掉，并将其交还给操作系统。但是，这里却存在着一个很严重的问题：CreateEmployee() 会返回一个指向派生类对象的指针，这个对象也是通过一个基类指针删除的，可这个基类的析构函数却是非虚的。问题就在于此，C++ 中指出：当一个派生类对象通过使用一个基类指针删除，且这个基类有一个非虚的析构函数时，C++ 将不会调用整个析构链，结果会是未定义的。所以在这种情况下，只是调用了基类 CEmployee 的析构函数，具体对象的派生部分并没有被销毁，这就是传说中的“部分析构”问题。

解决这个问题的方法很简单：将基类的析构函数设置为虚。因为虚函数的最大目的就是允许派生类定制实现。所以，用基类指针删除一个派生类对象时，C++ 会正确地调用整个析

构链，执行正确的行为，以销毁整个对象，代码如下所示：

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

CEmployee* pEmployee = CreateEmployee(type);
..... // use it do what you want to
delete pEmployee; // OK
```

在实际使用虚析构函数的过程中，一般要遵守以下规则：当类中包含至少一个虚函数时，才将该类的析构函数声明为虚。因为一个类要作为多态基类使用时，它一定会包含一个需要派生定制的虚函数。相反，如果一个类不包含虚函数，那就预示着这个类不能作为多态基类使用；如果你非要去挑战规则，非要将它的析构函数声明为虚，那我不得不说这确实是个坏主意。

因为虚函数的实现要求对象携带着额外的信息，这一信息被称为虚函数表指针 vptr (Virtual Table Pointer)，这些信息用于在运行时确定该对象应该调用哪一个虚函数。在虚函数机制的实现中，vptr 指向一个被称为虚函数表 vtbl (virtual table) 的函数指针数组，每一个包含虚函数的类都会关联到虚函数表上。当一个对象调用虚函数时，可通过下面的步骤来确定实际被调用的函数：找到对象的 vptr 指向的 vtbl，然后在 vtbl 中寻找合适的函数指针。正是由于虚函数表指针 vptr 的加入，导致了该类对象的大小增加了，同时还使得在 C++ 和其他语言（比如 C 和 FORTRAN）中同样的对象不再具有相同的结构，从而失去了一定的可移植性。

同样，如果一个类的析构函数非虚，那你就要顶住诱惑，绝不能继承它，即使它是“出身名门”。比如标准库中的 string、complex，以及 STL 容器。如果贸然使用，后果同样不可预知，如下所示：

```
class UserDefinedString: public std::string
{
    ...
};
```

所以在有的时候你要明智地拒绝标准库的诱惑，不要去继承那些为我们带来巨大方便，但是具有非虚析构的类。

#### 请记住：

多态基类的析构函数应该是 virtual 的，也必须是 virtual 的，因为只有这样，虚函数机

制才会保证派生类对象的彻底释放；如果一个类有一个虚函数，那么它就该有一个虚析构函数；如果一个类不被设计为基类，那么这个类的析构就应该拒绝为虚。

---

## 建议 43：绝不让构造函数为虚

在建议 42 中，指出应将多态基类的析构函数设为虚。那么构造函数呢？它可不可以设为虚呢？

C++ 之父 Bjarne Stroustrup 在《The C++ Programming Language》里给出了答案，让我们看看他是怎么说的：

为构造对象，构造函数必须要事先知道对象的确切类型。所以，构造函数不能为虚。更进一步来讲，构造函数不是普通函数，特别是它会与内存打交道，而普通函数却不会。因此，你不可能获得指向构造函数的指针。<sup>Θ</sup>

这话有些深奥，让人难以理解。下面我试着用简单的语言解释一下其中的道理：

首先，我们还是要重复一下虚函数的工作机制：虚函数（Virtual Function）的多态机制是通过一张虚函数表（Virtual Table）来实现的。在这张表中，存放着一个类的虚函数的地址，这张表解决了继承、覆盖的问题。如果一个类具有虚函数，那么在它的对象空间中就会存在一张这样的表，表中存放的就是我们应该调用的函数地址。所以，当我们用父类的指针来操作一个子类的时候，这张虚函数表就显得尤为重要了，它就像一个地图一样，指明了实际应该调用的函数。由此我们可以看出，只有确定了对象的存在，这张表才有存在的可能。

构造函数又是干嘛的呢？这一点我们应该已经很清晰。无论对象构建在哪里，其中的两步都不可避免：首先，分配一块内存；然后，调用构造函数。假设构造函数是虚函数，那么就需要通过虚函数表来调用应该调用的函数。但此时我们面对的是一块 raw 内存，它是一块不毛之地，到哪里去找虚函数表呢？构造函数要做的事情之一是初始化它的虚函数表。调用一个还不存在的东西，从一开始就注定了它失败的命运。所以构造函数不能为虚函数。

话说到了这里，道理已经讲得比较清晰了，但是貌似还缺少一个实例。我们就把构造函数设成虚，看看编译器会怎么处理，如下所示：

```
class Base
{
public:
    virtual Base(){}  
}
```

<sup>Θ</sup> To construct an object, a constructor needs the exact type of the object it is to create. Consequently, a constructor cannot be virtual. Furthermore, a constructor is not quite an ordinary function. In particular, it interacts with memory management in ways ordinary member functions don't. Consequently, you cannot have a pointer to a constructor.

```

    virtual ~Base(){}
};

class Derived : public Base
{
public:
    virtual Derived(){}
    virtual ~Derived(){}
};

int main()
{
    Derived d;
    return 0;
}

```

在 VC2010 中，IDE 给出了错误提示：

```

1>----- 已启动生成：项目：VirtualCtor，配置：Debug Win32 -----
1>main.cpp
1>main.cpp(4): error C2633: "Base" : "inline" 是构造函数的唯一合法存储类
1>main.cpp(11): error C2633: "Derived" : "inline" 是构造函数的唯一合法存储类
===== 生成：成功 0 个，失败 1 个，最新 0 个，跳过 0 个 =====

```

在 Linux Gcc 编译器下，同样会给出了错误信息，如下所示：

```

main.cpp:4: 错误：构造函数不能被声明为虚函数
main.cpp:11: 错误：构造函数不能被声明为虚函数

```

如此看来，将构造函数声明为虚函数，编译器也是不同意的，这就基本上杜绝了我们误用的可能性。

#### 请记住：

在构造函数调用返回之前，虚函数表尚未建立，不能支持虚函数机制，所以构造函数不允许设为虚。

### 建议 44：避免在构造 / 析构函数中调用虚函数

编译器不允许构造函数为虚，那我们退而求其次，在构造函数中显式地调用虚函数总该可以了吧？是否行得通，我们还是用实例去验证问题，如下所示：

```

class Base
{
public:
    Base()
    {

```

```

        cout << "Base constructor\n";
        Init();
    }
    virtual void Init()
    {
        cout << "Base::Init " << endl;
    }
};

class Derived : public Base
{
public:
    Derived() : Base()
    {
        cout << "Derived constructor" << endl;
    }
    virtual void Init()
    {
        cout << "Derived::Init " << endl;
    }
};

int main()
{
    Derived d;
    return 0;
}

```

上述的代码片段顺利地通过了编译链接，得到运行结果：

```

Base constructor
Base::Init
Derived constructor

```

构造一个 Derived 类对象，为什么会调用 Base::Init 函数，而不是我们所期望的 Derived::Init 呢？这似乎有违于虚函数的工作机制啊！在基类的构造过程中，似乎没有找到正确的虚函数 Init。

这是为什么呢？其原因与建议 43 中为什么构造函数不能为虚有几分相似。

在派生类被正确地构造出来之前，调用派生类的虚成员函数是没有意义的，在基类构造器运行的时候派生类的数据成员还没有被初始化。派生类的正确构造必须以基类的正确构造为前提。为了保证在构造期间只能调用已经被正确构造的类函数，C++ 编译器首先会建立两个虚函数表 vtbl，一个是 Base 的，另一个则属于 Derived。随后 C++ 运行库会在构造序列期间调整虚函数指针 vptr，使它指向适当的 vtbl。如果在基类的构造过程中对虚函数的调用传递到了派生类上，派生类对象自然会通过 vptr 找到对应的 vtbl，进而执行正确的函数。但不幸的是，其时这些数据成员尚未被初始化。如果这时允许多态行为的发生，即通过父类的构

造函数调用了子类的虚函数，而这个虚函数要访问属于子类的数据成员时就有可能出错，导致悲剧的发生。所以它也就只能退而求其次，调用基类的虚函数 Init 了。

那么析构函数呢？在对象析构期间，它存在与上面类似的逻辑：执行哪个虚函数取决于它被绑定到了哪里。析构顺序遵从的是从继承类到基类，一旦派生类数据析构，在基类析构函数中调用的虚函数就没了多态的能力，C++ 仅仅会将它作为一个基类对象来进行处理。

关于这一点，在 C++ 标准规范中也有详细说明，如下所示：

成员函数，包括虚成员函数，都可以在构造、析构的过程中被调用。当一个虚函数被构造函数（包括成员变量的初始化函数）或者析构函数直接或间接地调用时，调用对象就是正在构造或者析构的那个对象。其调用的函数是定义于自身类或者其基类的函数，而不是其派生类或者最底层派生类的其他基类的重写函数。<sup>⊖</sup>

如果你在构造 / 析构中调用了虚函数，且应用可以正常执行，貌似没什么严重问题，但这只能说明你足够幸运，正如上面的示例代码一样。只要是稍稍有所修改，潜在的危险性就会大大增加，如下所示：

```
class Base
{
public:
    Base()
    {
        cout << "Base constructor\n";
        Init();
    }
    virtual void Init() = 0;
};
```

在这种情况下，因为函数 Init 是基类 Base 中的纯虚函数，所以当应用程序调用到这里时，悲剧就发生了，纯虚函数调用失败，程序崩掉。

所以，不要在构造 / 析构函数中调用虚函数。因为这种调用不会如你所愿，既无法让你获得梦想的多态，还会带来了一系列令人头疼的问题。如果你以前是一个 Java 或 C# 程序员，那么这一点你应该更加注意，这正是 C++ 与其他语言的重大区别之一。

### 请记住：

如果在构造函数或析构函数中调用了一个类的虚函数，那它们就变成普通函数了，失去

<sup>⊖</sup> Member functions, including virtual functions, can be called during construction or destruction. When a virtual function is called directly or indirectly from a constructor (including from the mem-initializer for a data member) or from a destructor, and the object to which the call applies is the object under construction or destruction, the function called is the one defined in the constructor or destructor's own class or in one of its bases, but not a function overriding it in a class derived from the constructor or destructor's class, or overriding it in one of the other base classes of the most derived object. —— C++ Standard ANSI ISO 2003 12.7.3

了多态的能力。换句话说就是，对象不能在生与死的过程中让自己表现出“多态”。

---

## 建议 45：默认参数在构造函数中给你带来的喜与悲

具有默认参数的函数会给我们带来巨大的便利，这一点我们已经不用再去证明了。当默认参数遇到构造函数时，便利依旧。

假设我们需要设计一个文字类 CText，它的成员变量包括文字内容 m\_Content、字体颜色 m\_Color、字体 m\_Font、字体大小 m\_Size。为了满足不同的情形，我们为它设计了多样化的构造函数，如下所示：

```
class CText
{
public:
    CText(string str)
        :m_Content(str)
        ,m_Color(defaltColor)
        ,m_Font(defaultFont)
        ,m_Size(defaultSize){ }
    CText(string str, COLOR color )
        :m_Content(str)
        ,m_Color(color)
        ,m_Font(defaultFont)
        ,m_Size(defaultSize){ }
    CText(string str, COLOR color, FONT font)
        :m_Content(str)
        ,m_Color(color)
        ,m_Font(font)
        ,m_Size(defaultSize){ }
    CText(string str, COLOR color, FONT font, SIZE size)
        :m_Content(str)
        ,m_Color(color)
        ,m_Font(font)
        ,m_Size(size){ }

private:
    string m_Content;
    COLOR  m_Color;
    FONT   m_Font;
    SIZE   m_Size;
};
```

虽然上面的这段代码实现了所需的功能，但其中存在着大量冗余的代码，严重影响了代码的整洁性。如果采用了默认参数，这种不适的感觉会立刻烟消云散、无影无踪，如下所示：

```
class CText
{
```

```

public:
    CText(std::string str, COLOR color = defaultColor, FONT font = defaultFont,
SIZE size = defaultSize)
        :m_Content(str)
        ,m_Color(color)
        ,m_Font(font)
        ,m_Size(size){ }
private:
    string m_Content;
    COLOR m_Color;
    FONT m_Font;
    SIZE m_Size;
};

```

正是因为有了默认参数，构造函数的个数才由四个减少为一个，同时功能上没有任何的损失，这提高了函数的易用性。你照样可以按照以下方式来调用构造函数：

```

CText text1("Hello Beijing");
CText text2("Hello Beijing", RED);
CText text3("Hello Beijing", RED, NewRoman);
CText text4("Hello Beijing", RED, NewRoman, 48);

```

但是如果稍有不慎，默认参数同样会给你带来一些麻烦：不合理地使用默认参数，将会导致重载函数的二义性，如下所示：

```

class CTimer
{
public:
    CTimer(string name, int hour=0, int min=0, int sec=0)
        : m_sName(name), m_nHour(hour), m_nMin(min)
        , m_nSecond(sec){}
    CTimer(string name)
    {
        m_nHour = m_nMin = m_nSecond = 0;
        m_sName = name;
    }
    ~CTimer(){}
private:
    string m_sName;
    int m_nHour;
    int m_nMin;
    int m_nSecond;
};

```

悲剧发生了！编译器会直截了当地给你抛出错误信息：

```

error C2668: "CTimer::CTimer" : 对重载函数的调用不明确，可能是 "CTimer::CTimer(string)" 或 "CTimer::CTimer(string,int,int,int)"

```

权利与义务是对等的，在C++的世界里这又一次得到了体现：在享受某种技术所带来便

利的同时，你必须面对可能出现的错误，并为此担负责任！

### 请记住：

合理地使用默认参数可以有效地减少构造函数中的代码冗余，让代码简洁而有力。但是如果不够小心和谨慎，它也会带来构造函数的歧义，增加你的调试时间。

## 建议 46：区分 Overloading、Overriding 及 Hiding 之间的差异

在 OO 的世界中存在着三个十分容易混淆的概念：重载 (Overloading)、重写 (Overriding)、隐藏 (Hiding)。它们彼此相似，容易让 C/C++ 程序员们一头雾水。接下来，我们对这三个概念一一分析。

### □ 重载

重载是指同一作用域的不同函数使用相同的函数名，但是函数的参数个数或类型不同。简单地讲，就是不同的函数使用同一标识符，并且这些函数位于同一作用域。重载在 C 中就已经存在了，正如你所熟悉的 abs 函数一样，代码如下所示：

```
double abs(double);
int abs(int);
abs(1); //call abs(int);
abs(1.0); //call abs(double);
```

这两个全局函数 abs(int) 和 abs(double) 互为重载，都用了标识符 abs（在 C++ 中函数名字是由函数声明中的标识符和其形参的类型组合而成的），并且位于同一作用域。

也许大家与我一样，真正认识重载是在进入类的时代后。就像下面的代码片段那样，重载函数就在一个类空间里具有相同名字、不同参数的一些函数。比如打印机类 Printer 中的 Print 函数：

```
class Printer
{
public:
    void Print(int data);
    void Print(float data);
    void Print(const char* pStr, size_t size);
    ... // other code
};
```

但是，如果将 Printer 作为基类，派生出继承类 StringPrinter：

```
class StringPrinter : public Printer
{
public:
    void Print(const string& str);
    ... // other code
```

```
};
```

派生类 StringPrinter 中的 Print 函数并不是基类 Printer 中 Print 函数的重载兄弟，因为它们分属于不同的作用域。所以当你写下如下代码时，编译器会毫不留情地警示：

```
StringPrinter myPrinter;
myPrinter.Print(2011); // 编译报错
```

这是因为在派生类的作用域中没有找到 Print(int) 这样的函数定义，基类 Printer 中的 Print 被派生类中的 Print (const string&) 掩盖了，于是就出现了“参数不匹配”的错误提示。如果想让它们兄弟四个构成重载，只需将基类中的 Print 函数声明引入到派生类的作用域中，如下所示：

```
class StringPrinter : public Printer
{
public:
    using Printer::Print;
    void Print(const string& str);
    ... // other code
};
```

#### □ 重写

重写是指在派生类中对基类中的虚函数重新实现，即函数名和参数都一样，只是函数的实现体不一样。重写是我们十分熟悉的一个操作，它与虚函数的实现息息相关。这里涉及两个关键要素：派生类和虚函数。换句话说，派生类对基类中的操作进行个性化定制就是重写。比如下面的代码片段：

```
class CWorker
{
public:
    CWorker(){}
    ~CWorker(){}
    virtual void WorkingLog()
    {   std::cout<<"Working..."<<std::endl; }
};

class CDriver : public CWorker
{
public:
    CDriver () {}
    ~ CDriver () {}
    virtual void WorkingLog()
    {   std::cout<<"Driveling..."<<std::endl; }
};
```

但是重写有几个需要注意的问题：

(1) 函数的重写与访问层级 (public、private、protected) 无关，如下所示：

```
class CDriver : public CWorker
```

```

{
public:
    CDriver () {}
    ~ CDriver () {}

private:
    virtual void WorkingLog()
    {   std::cout<<"Driveling..."<<std::endl; }
}

```

虽然派生类中的 WorkingLog 与基类的访问层级不同，但还是成功地实现了对该函数的特殊定制。上述代码片段中对于 WorkingLog 函数访问层级的设置仅仅是一个示例，不具任何实际参考意义。一般情况下，派生类改写的函数应该和基类对应的函数有相同的访问层级，但不排除某些特殊情况的存在。

### (2) const 可能会使虚成员函数的重写失效。

常量成员函数与一般的成员函数在函数签名中是不同的，其常量属性是函数签名中的一部分。就像下面的代码那样，基类中的 WorkingLog 函数并不具有常量属性，但其派生类中却多出了 const，如下所示：

```

class CDriver : public CWorker
{
public:
    CDriver () {}
    ~ CDriver () {}

    virtual void WorkingLog() const
    {   std::cout<<"Driveling..."<<std::endl; }
};

```

因为具有不同的函数签名，所以派生类中的 WorkingLog 函数并没有重写基类中的 WorkingLog 函数。

### (3) 重写函数必须和原函数具有相同的返回类型。

因为函数的返回类型不是函数签名的一部分，所以若派生类重写了基类类型中对应的函数，那么它们必须有相同的返回类型。如果返回值不同，编译器会抛出“重写虚函数返回类型有差异”的错误警示，如下所示：

```

class CDriver : public CWorker
{
public:
    CDriver () {}
    ~ CDriver () {}

    virtual bool WorkingLog()
    {
        std::cout<<"I am a teacher"<<std::endl;
    }
};

```

此规则存在一种例外情形，称为“协变返回值类型”。

### □ 隐藏

隐藏理解起来比较直观，就是指派生类中的函数屏蔽基类中具有相同名字的非虚函数。所以它的两个重要要素就是派生类和非虚函数。

在调用一个类的成员函数时，编译器会沿着类的继承链逐级地向上查找函数的定义，如果找到就停止。如果一个派生类和一个基类有一个同名函数，由于派生类在继承链中处于下层，编译器则最终会选择派生类中的函数。如此一来，基类的同名成员函数就会屏蔽隐藏，编译器的函数查找也就到达不了基类中。

还是采用前面的 StringPrinter 类中的 Print 函数来说明这一问题，代码如下所示：

```
class Printer
{
public:
    void Print(int data);
    void Print(float data);
    void Print(const char* pStr, size_t size);
    ... // other code
};

class StringPrinter : public Printer
{
public:
    bool Print(int data);
    void Print(const string& str);
    ... // other code
};
```

当编译器在继承链中查找到 Print 函数时，派生类中的 Print 函数阻止了它向上寻找，隐藏了基类中的 Print。

上面洋洋洒洒地讲了一通，也许会使大家有些晕。这三个概念始终都在作用域、函数名称、参数、返回值、有无 virtual 修饰等这几点上绕来绕去。下面对这三个概念做一下对比，也许一张简单的表格会让大家对它们的理解变得更清晰（如表 4-1 所示）。

表 4-1 重载、重写、隐藏的对比

	作用域	有无 virtual	函数名	参数类型	返回值类型
重载	相同	可有可无	相同	不同	可同可不同
重写	不同	有	相同	相同	相同（协变）
隐藏	不同	可有可无	相同	可同可不同	可同可不同

请记住：

重载、重写、隐藏三个概念虽然在 OO 中有着不同的作用，但它们却极为相似，所以极易混淆。要正确理解它们之间的差异，需将注意力集中在作用域、有无 virtual、函数名、参数类型、返回值类型五大方面。

## 建议 47：重载 operator= 的标准三步走

在 C 语言里，赋值运算是应用最为广泛的操作之一。到了 C++ 时代，C++ 之父 Bjarne Stroustrup 希望将此操作继承并发扬光大，让所有的自定义数据类型都具备类似于内置数据类型的赋值运算能力，那么这就要对赋值运算符（=）进行重载了。然而这个赋值运算符的重载很是让人迷惑，经常会出现问题。

### □ 不要让编译器帮你重载赋值运算符

赋值运算符的重载并不是必须由程序员来完成的，有时候编译器也会帮你完成。这一点在建议 37 中已经详细讲过。但是由编译器完成的赋值运算符重载函数只是简单地将原对象位于 stack 中的域逐位地拷贝、赋值给了新对象，就像对待 POD 对象那样。所以如果对象存在于 heap 域上的话，这种没有区分深拷贝和浅拷贝的重载方式，就会产生问题。就像下面的代码片段所示的那样：

```
class CString
{
public:
    CString():pChar(NULL) {}
    CString(char* data)
    {
        pChar = new char[strlen(data)+1];
        strcpy(data, pChar);
    }
    ~CString()
    {
        if(pChar!=NULL)
        {
            delete[] pChar;
            pChar = NULL;
        }
    }
private:
    char* pChar;
};
```

我们没有为类 CString 重载赋值运算符，但是编译器为我们偷偷地合成了一个，其功能类似于下面的函数：

```
CString& operator=(const CString& rhs)
{
    pChar = rhs.pChar;
    return *this;
}
```

当我们按照以往的经验写下下面的代码片段时，问题发生了：

```
CString greetWords("Hello World");
CString otherWords;
otherWords = greetWords;
```

正如图 4-3 所示的那样，赋值后，两个对象的成员指针指向了同一块内存，在对象超出作用域时，两个对象先后析构，于是同一块内存被先后释放了两次。这就是编译器采用的浅拷贝所引发的后遗症。

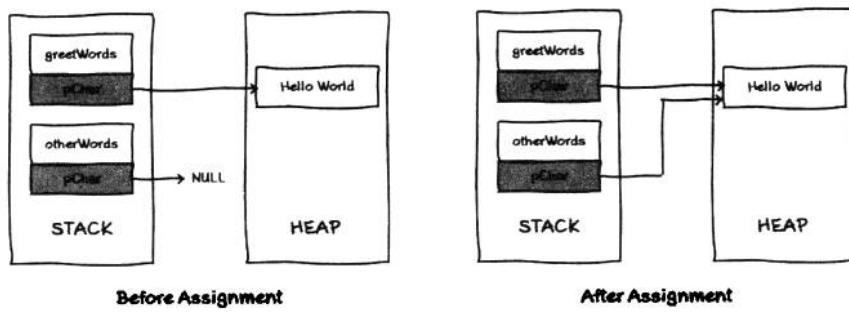


图 4-3 浅拷贝赋值操作示意图

此时要做的就是重写赋值运算符重载函数，拒绝编译器提供的默认版本。在重载时，需要采用深拷贝（如图 4-4 所示），代码如下所示：

```
CString& operator=(const CString& rhs)
{
    if(pChar!=NULL) delete[] pChar;
    pChar = new char[strlen(rhs.pChar)]+1;
    strcpy(pChar, rhs.pChar);
    return *this;
}
```

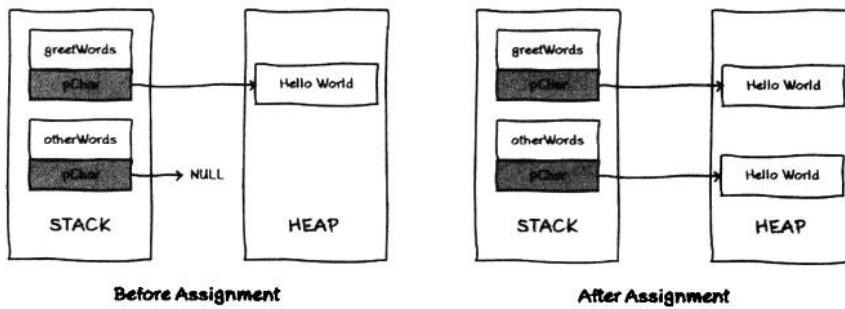


图 4-4 深拷贝赋值示意图

当然，以上重载函数并非最终版本，这里只是为了说明深拷贝 / 浅拷贝问题。请勿模仿，否则后果自负。

### □ 一定要检查自赋值

自赋值虽然看起来有些愚蠢，没什么实际意义，但却是 C++ 语法中允许的操作之一。就像下面的代码：

```
class CString { };
CString str1("Hello C++");
str1 = str1;
```

特别是在引用这一语法进入 C++ 的世界之后，自赋值似乎披上了一件合理的外衣，变得更加隐蔽。就像 strA = strB，当 strA 是 strB 的引用时，自赋值就理所当然地发生了。

可能有人要问为什么要检查是否是自赋值，最直接的原因就是效率，特别是在具有内存分配操作时。如果函数能在一开始就发现其是自赋值，那么它就可以直接返回，从而节省了大量的工作：

```
CString& CString::operator=(const CString& str)
{
    if(*this == str)
        return *this;

    if(pChar!=NULL)
        delete[] pChar;
    pChar = new char[strlen(str.pChar)+1];
    strcpy(pChar, str.pChar);
    return *this;
}
```

如果以为自赋值检查的意义仅仅在于效率，那就大错特错了。自赋值检查还有一个更加重要的作用。试想一下，如果没有自检这一步骤，当自赋值发生时会出现什么问题呢？因为目标对象的 pChar 不为空，所以它会将目标对象的 pChar 指向的内存首先释放。又因为是自赋值，原对象 pChar 与目标对象 pChar 指向的是同一地址，所以原对象指针指向的数据将被删除。这就导致后面基于其字符串内容而进行的内存空间申请、数据复制等全部变成了无源之水、无根之草。如图 4-5 所示。

关于自赋值检查，还有一点必须说明：上述代码自赋值检查的成立要基于一个条件，那就是 operator== 存在。所以在重载 operator= 时应该首先重载 operator==。

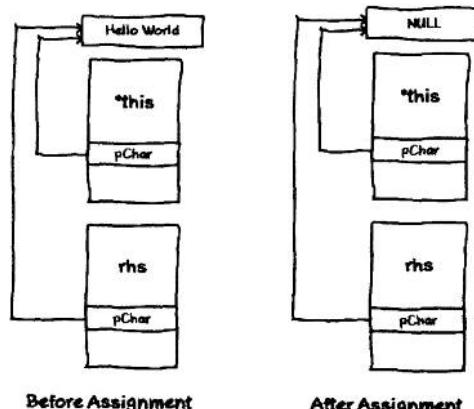


图 4-5 自赋值操作

### □ 赋值运算符重载需返回 \*this 的引用

很多 C++ 新手会将赋值运算符重载函数的返回值设为空，并将所有的数据都逐一进行了复制，而且还注意了深拷贝 / 浅拷贝的问题，按理说返回 void 应该没什么问题，不过这其中却忽略了我们的一个使用习惯：链式赋值！

在 C++ 中，内置数据类型是支持链式赋值的，即  $a = b = c$ 。所以，对于用户自定数据类型来说，当然它们也会支持链式赋值了，如下所示：

```
class CString { };
CString str1("Hello C++");
CString str2, str3;
str3 = str2 = str1;
```

因为赋值运算符具有右结合性，即会首先将等号右边的对象赋给等号左边，因此上面的链式赋值就翻译成了：

```
str3.operator=(str2.operator=(str1));
```

要使上面的这条语句正常执行，operator= 的返回值类型必须能被它自己当成一个输入参数接受。也就是说，返回值必须是一个该类的对象或引用：

```
CString CString::operator=(const CString& str);
CString& CString::operator=(const CString& str);
```

是对象还是引用？这似乎不再是一个问题！引用之于对象的优点在于效率！如果返回对象，就不可避免地会带来临时对象的构造、析构这些不必要的操作过程。而返回引用恰恰可以将这些不必要的操作降到最低。

除了效率之外，还有一个原因使你不得不选择返回引用。考虑下面的链式赋值语句：

```
class CString { };
CString str1("Hello C++");
CString str2("Hello World");
CString str3;
(str3 = str1) = str2;
```

这样的代码是完全合法的，括弧只是确保首先执行最左边的赋值而已。不过，此时如果返回的是临时对象，编译器会毫不犹豫地停止工作，因为编译器不允许使用临时对象调用成员函数。所以，为了能够更加灵活地使用赋值运算符，选择返回引用绝对是明智之举。

### □ 赋值运算符重载函数不能被继承

赋值运算符运算是不能被继承的，即便写出类似下面的代码：

```
class CString
{
public:
    CString();
```

```

CString() (char* data);
~CString();
CString& operator=(const CString& rhs);
private:
    char* pChar;
};
class ColorString : public CString
{
public:
    ColorString();
    ~ColorString();
private:
    DWORD m_dColor;
};

```

虽然 ColorString 类继承自 CString 类，但是 CString 类中的 operator= 函数并没有按照继承体系向下传递。这是为什么呢？是否还记得建议 37 中提及的编译器悄悄为我们做的那些事儿？答案就在那里！虽然我们没有显式地去为派生类声明赋值运算符重载函数，但是编译器悄悄地为我们完成了这些。所以，这个“隐形”的 operator= 函数就将基类中的 operator= 函数隐藏了。

在派生类的赋值运算符重载中，有一个十分常见的错误：忘记对基类成员变量进行赋值。就像上面代码所示的 ColorString 类，有人会写出这样的赋值运算符重载函数：

```

ColorString & ColorString::operator=(const ColorString& rhs)
{
    if(this == &rhs) return *this;
    m_dColor = rhs.m_dColor;
    return *this;
}

```

这样的赋值方式完全是不负责任的，它只会将派生类 ColorString 的颜色信息进行正确赋值，对于基类中的 pChar 则不会处理。所以这是一个不完整的赋值运算符重载。

派生类中的赋值运算符重载函数应该负责对该对象中的所有成员数据一一进行赋值，当然也包括其中的基类部分。在对基类进行赋值时，最简单的方式就是直接在派生类赋值函数中调用基类赋值运算符重载函数，即

```

ColorString & ColorString::operator=(const ColorString& rhs)
{
    if(this == &rhs) return *this;
    CString::operator=(rhs);
    m_dColor = rhs.m_dColor;
    return *this;
}

```

综上所述，我们可以得到这么一条黄金定律：如果需要给类的数据成员动态分配空间，则必须实现赋值运算符。此时，赋值运算符的重载形式可以表示为以下形式：

```

ClassName& ClassName::operator=(const ClassName& rhs)
{
    自赋值检查
    释放原有空间 & 申请新空间 & 数据复制
    返回 *this
}

```

如果是派生类，请不要忘记对基类成员变量的赋值进行处理。

### 请记住：

在重载赋值运算符时，应该时刻注意以下几条：

- 区分该类应该采用深拷贝还是浅拷贝。
- 返回的是左值 \*this 的引用。
- 遵守赋值运算符重载过程中的标准三步走。
- 如果是派生类，注意基类成员变量的赋值运算处理。

## 建议 48：运算符重载，是成员函数还是友元函数

运算符重载就是赋予已有的运算符多重含义，它是实现多态性的有效手段之一。在 C++ 中我们可以通过重载运算符来对运算符进行特殊的定制，使它能够用于特定类的对象，以执行特定的功能，增强 C++ 语言的扩充能力。

运算符重载与普通函数相比，其功能更加直观，使用也更加方便，就像如下的 Add 函数与 + 运算符重载函数：

```

class Complex{}
Complex Complex ::Add(const Complex& c1,const Complex& c2) {}
Complex Complex ::operator +(const Complex &c) {}

```

上面通过重载标准运算符，为用户提供了具有直观意义的接口，使程序表达更加简洁，增强了代码可读性，减少了使用错误。

说到运算符重载，肯定会有人想到运算符重载的四项基本原则（也许你并不这么称呼它们）：

- 不可臆造运算符。
- 运算符原有操作数的个数、优先级和结合性不能改变。
- 操作数中至少一个是自定义类型。
- 保持重载运算符的自然含义。

一般说来，运算符的重载可采用如下两种形式：

- 成员函数形式。

□ 友元函数形式。

这两种形式都可访问类中的私有成员。不过，这两种形式中，我们该采用哪一种呢？这确实是一个问题。

假设我们有一个复数类 Complex，要将其运算符重载为类的成员函数，如下所示：

```
class Complex
{
public:
    Complex(int real = 0,int imag = 0)
        :m_Real(real),m_Imag(imag){}
    ~Complex(){}
    Complex operator +(const Complex &c);
    Complex operator -(const Complex &c);
private:
    int m_Real;
    int m_Imag;
};

inline Complex Complex::operator +(const Complex &c)
{
    return Complex(m_Real + c.m_Real, m_Imag + c.m_Imag);
}
inline Complex Complex::operator -(const Complex &c)
{
    return Complex(m_Real - c.m_Real, m_Imag - c.m_Imag);
}
```

如果在应用程序中出现了类似如下的代码片段：

```
Complex c1(10,20);
Complex c2(20,10);
Complex sum = c1 + c2;
Complex sub = c1 - c2;
```

编译器会将其解释为：

```
sum = c1.operator+(c2);
sub = c1.operator-(c2);
```

由此可见，采用成员函数形式重载运算符的格式应如下所示：

```
<类名> operator <运算符> (<参数表>) {.....}
```

对于双目运算符，参数仅有一个；对单目运算符，若重载为成员函数，不能再显式地声明参数。这是因为，重载为成员函数时，已经隐含了一个参数，它就是 this 指针。

如果重载为友元函数呢？将上面的例子进行改造，得到下面的代码片段：

```
class Complex
{
```

```

public:
    Complex(int real = 0, int imag = 0):m_Real(real),m_Imag(imag){}
    ~Complex() {}
    friend Complex operator +(const Complex &c1,
const Complex &c2);
    friend Complex operator -(const Complex &c1,
const Complex &c2);
private:
    int m_Real;
    int m_Imag;
};

Complex operator +(const Complex &c1, const Complex &c2)
{
    return Complex(c1.m_Real + c2.m_Real,
                   c1.m_Imag + c2.m_Imag);
}
Complex operator -(const Complex &c1, const Complex &c2)
{
    return Complex(c1.m_Real - c2.m_Real,
                   c1.m_Imag - c2.m_Imag);
}

```

采用友元函数形式的运算符重载函数，其定义的格式如下所示：

```
friend <类型说明符> operator <运算符> (<参数表>) {.....}
```

当重载友元函数时，将不存在隐含的参数 this 指针。这样，对于双目运算符来说，友元函数有两个参数；对于单目运算符来说，友元函数只有一个参数。

两种实现方案除了上述的声明形式有所差异外，还存在着一个很难发现的区别：如果运算符被重载为友元函数，那么它就会获得一种特殊的属性，能够接受左参数和右参数的隐式转换；如果是成员函数版的重载则只允许右参数的隐式转换。还是以实例来进行说明，假设定义了一个字符串类 CString，如下所示：

```

class CString
{
public:
    CString(char* str);
    ...
private:
    char* m_pStr;
};

```

因为 CString 的构造函数参数为一个 char，所以如果采用友元形式的 operator+(const CString&, const CString&)，那么 char+CString 和 CString+char 都能正常工作；而如果采用的是成员函数形式的 CString::operator+(const CString& rhs)，则只能接受 CString+char，如果执行的是 char+CString 形式则会编译出错。事实上在我们的使用习惯中 char+CString 和

`CString+char` 都是可以被接受的，只是此时相较于成员函数的实现形式，友元函数似乎领先了半个身位。需要注意的是，隐式转换由于临时变量的增加往往效率不高。如果应用对效率要求较高，建议选择定义多个运算符的友元重载版本，如下所示：

```
CString& operator+(const CString&, const CString&);
CString& operator+(const char*, const CString&);
CString& operator+(const CString&, const char*);
```

到了这里，两种方式之间该如何选择似乎有了一个结果。一般说来，建议遵守这么一个不成文的规定：对双目运算符，最好将其重载为友元函数，因为这样更方便些；而对于单目运算符，则最好重载为成员函数。

但是，一定注意上面的限制性修饰词“一般说来”，也就是说这条规则并不是绝对的，有特殊情况存在。有些双目运算符是不能重载为友元函数的，比如赋值运算符`=`；同时有些单目运算符只能重载为成员函数，例如函数调用运算符`()`、下标运算符`[]`和指针`->`等，因为这些运算符在语义上与`this`都有太多的关联：`=`表示的是“将自身赋值为...”，`[]`表示的是“自身的第`n`个元素”。如果它们被重载为友元函数，会出现语义上的不一致。这可是运算符重载应该避免的问题。

还有一个需要特别说明的运算符是输出运算符`<<`。因为`<<`的第一个操作数一定是`ostream`类型，所以`<<`只能重载为友元函数，如下所示：

```
friend ostream& operator<<(ostream &os ,const Complex &c);
ostream& operator<<(ostream &os ,const Complex &c)
{
    os<<c.m_Real<<"+"<<c.m_Img<<"i"<<endl;
    return os;
}
```

---

#### 请记住：

运算符重载是 C++ 多态的重要实现手段之一。一般而言，对于双目运算符，最好将其重载为友元函数；而对于单目运算符，则最好重载为成员函数。但是一定记得其中的例外情况。

---

## 建议 49：有些运算符应该成对实现

熟悉 C# 的读者一定知道 C# 中存在着三对比较运算符：

- == 与 !=
- > 与 <
- >= 与 <=

对于这三对比较运算符，C# 要求必须成对重载。如果重载了 == 也必须重载 !=，否则编译器就会产生错误，并罢工。然而在 C++ 的世界中，编译器并没有刻意地强制这么做。举个例子，如果你对复数类 Complex 重载了 == 运算符而没有去理会与之对应的 !=，只要你不调用到 != 运算符，代码就不会出现什么问题，如下所示：

```
class Complex
{
public:
    Complex(int real = 0, int imag = 0)
        :m_Real(real), m_Img(imag){}
    ~Complex(){}
    friend bool operator ==(const Complex &c1, const Complex &c2);
private:
    int m_Real;
    int m_Img;
};

Complex c1, c2;
if(c1==c2) ...
if(!(c1==c2)) ...
```

但是，如果你习惯性地将 if( !(c1==c2) ) 写成了 if( c1!=c2 )，那悲剧就发生了，C++ 编译器会告诉你：

没有找到接受“Complex”类型的左操作数的运算符

所以，为了使用习惯的统一，在进行比较运算符重载时，建议成对实现：如果重载了 operator==，请不要遗漏 operator!=；如果重载了 operator<，请不要遗漏 operator>；如果重载了 operator<=，请不要遗漏 operator>=。

除了上述三对比较运算符外，还有双目算术运算符。如果定义了 A op B，那么也应该提供运算符的赋值形式：A op= B（op 可能是 +（加）、-（减）、\*（乘）、/（除）等）。A op= B 和 A = A op B 具有相同的含义。

通过对 op 赋值形式的重载，一方面可以让运算符的使用更加符合习惯；另一方面，A op= B 可以避免 A 的多次运算，效率上更高。为了减少代码重复，提高效率，op 和 op= 实现的标准方法就是用 op= 来定义 op，如下所示：

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

T operator op(const T& lhs, const T& rhs)
{
    T temp(lhs);
```

```

    return temp op= rhs;
}

```

当然，这个标准操作并不绝对，还要具体情况具体分析，比如，对于复数而言，用 `operator*=` 实现 `operator* =` 会更方便、有利！

#### 请记住：

为了更好地适应使用习惯，很多运算符重载时最好成对实现，比如 `==` 与 `!=`、`<` 与 `>`、`<=` 与 `>=`、`+` 与 `+=`、`-` 与 `-=`、`*` 与 `*=`、`/` 与 `/=`。

## 建议 50：特殊的自增自减运算符重载

`++`、`--` 在 C++ 中是比较特殊的运算符，因为二者都有着两种不同的运算形式：前缀和后缀。虽然两种形式都只有一个参数，但意义上却不尽相同，就像建议 12 所讲的那样。所以，当你决定要重载这两个运算符时，也要针对前缀与后缀的不同来分别实现。

运算符重载本质上只不过是一个函数而已，如果一个函数具有相同的名字、相同的参数，编译器就会分不清它们。所以在面对自增 `++`、自减 `--` 运算符重载时，需要用另外的方式对它们的前缀和后缀形式加以区分。方法很简单，给其中的一个多加上一个标识参数，这个参数在运算中没有任何用途，只是从语法上来区分函数名，如下所示：

```

T& operator++();           // ++ 前缀
const T operator++(int);   // ++ 后缀
T& operator--();           // -- 前缀
const T operator--(int);   // -- 后缀

```

细心的你也许会发现，后缀形式返回的是一个 `const` 对象。这样做其实隐藏着极为深刻的意义。运算符重载有一个重要的原则，即不应该破坏它原有的自然语义和使用习惯。对于内置类型 `int`，是不允许出现如下代码的：

```

int i = 0;
i++++; // ERROR!!!

```

`i++++` 在语法上是错误的，所以为了保持一致，重载之后的后缀形式也应该遵循相应的语法习惯。

假设有一个自定义整数类 `UserInt`，其自增后缀操作的返回值不是 `const` 类型，如下所示：

```

class UserInt
{
public:
    UserInt( int num=0 ) : m_num(num) {}
    UserInt operator++(int)

```

```

    {
        UserInt temp(m_num);
        m_num++;
        return temp;
    }
private:
    int m_num;
};

```

在写下如下的代码时：

```
UserInt data(0);
data++++++; // + 3
```

虽然代码可以执行，但它与内置类型 int 的使用习惯不一致，所得的结果是 data.m\_num==1，也不是我们想要的 3，这是因为后缀返回的是一个临时对象，从第二次调用开始，执行加 1 操作的对象已不再是 data 本体。这种违反我们直觉的操作应该是被禁止的，所以它的返回值被赋予了 const 属性。根据 const 的特性可知，如果试图改变它的值，编译就会报错，代码如下所示。

```

const UserInt operator++(int)
{
    UserInt temp(m_num);
    m_num++;
    return temp;
}
UserInt data(0);
data++++; // ERROR!!!

```

所以，后缀操作重载时返回值应该为一个 const 对象。

除此之外，还要谨记“优先使用前缀操作”，至于原因，无非是建议 12 所介绍的效率问题。

#### 请记住：

`++`（自增）、`--`（自减）是 C++ 中比较特殊的两个运算符，它们的前缀操作与后缀操作重载声明太过相似，难以区分，所以 C++ 语法规规定：对后缀操作增加一个标识参数，以示区分。此外，为了保持使用习惯上的一致以及表达意义上的直接，后缀操作会返回 const 对象。

### 建议 51：不要重载 operator&&、operator|| 以及 operator,

说起短路求值法（Short-Circuit Evaluation）读者可能会感到些许陌生，但是提到它在布尔表达式中的应用相信你一定清楚：在 C/C++ 中，如果在布尔表达式中使用了短路求值法，

那么一旦确定了布尔表达式的真假，即使还有部分表达式没有被测试，它也会停止运算，最典型的就是 `&&`（与）运算和 `||`（或）运算。就像下面的代码片段：

```
class CStudent { ... };
CStudent* pStudent = NULL;
...
// && 运算
if( pStudent!=NULL && pStudent->GetAge()<20 )
...
// || 运算
if( pStudent==NULL || pStudent->GetAge()>=20 )
...
```

相信上面的代码形式大家都很熟悉。在 `&&` 运算示例中，如果 `pStudent!=NULL` 不成立，`pStudent->GetAge()<20` 就不会再被运算，表达式返回 `false`；换句话说，只有在 `pStudent!=NULL` 的前提下，`pStudent->GetAge()<20` 才会被计算求值。`||` 运算也是类似的，如果 `pStudent==NULL` 成立，`pStudent->GetAge()>=20` 的求值计算就绝不会被提上日程。`&&` 和 `||` 的这一特性，与其他的运算表达式有着很大的不同，究其原因是因为编译器在实现该运算的过程中采用了短路求值法。

C++ 允许我们根据用户自定义的数据类型来重载 `&&` 和 `||` 运算符。重载方法和其他的运算符重载一样，既可以重载为全局函数，也可以重载为某个类的成员函数，如下所示：

```
// expression1 && expression2
expression1.operator&&(expression2)      // 类成员函数
operator&&(expression1, expression2)    // 全局函数
```

但是在重载的实现中，我们却很难遵守原来的短路求值规则，因为我们是通过重载函数的形式来实现运算符重载的。在调用函数时，应当首先确定所有的参数，所以在调用函数 `operator&&` 和 `operator||` 时，参数 `expression1` 和 `expression2` 会被首先计算，而且是必须执行。另外，由于 C++ 规范中没有规定 `expression1` 和 `expression2` 这两个函数参数的计算顺序，所以它们中哪一个表达式会首先被计算，完全不在我们的掌控之中。

为了不改变 `&&` 和 `||` 的短路求值特性，使其遵循一贯的使用习惯，请不要重载 `&&` 和 `||`。

逗号运算符也比较特殊，正如建议 6 中所讲的那样，一个包含逗号的表达式首先会计算逗号左边的表达式，然后计算逗号右边的表达式，整个表达式的结果是逗号右边表达式的值。在重载中，需要模仿这样的行为特性。但是，不幸的是很难模仿成功。逗号运算符的重载不外乎以下两种形式：

```
expression1.operator,(expression2) // 类成员函数
operator,(expression1, expression2) // 全局函数
```

在重载过程中，逗号两边的表达式 `expression1` 和 `expression2` 都是作为函数参数来处理的。你不能保证左边的表达式先于右边的表达式计算，因为函数参数的计算顺序是你没办法

控制的。可见，逗号运算符重载函数很难遵循原有的行为特性。

不要将“可以重载”作为重载某一运算符的理由，重载是为了提高代码的可读性、可维护性，所以在缺少一个较好理由的时候，坚决不要去重载某个运算符，因为贸然重载会破坏运算符原有的行为特性，所带来的麻烦可能远超其便利。`&&`、`||` 和逗号运算符就是其中的典型代表。

#### 请记住：

`“&&”（与运算）、“||”（或运算）和“,”（逗号运算符）都具有较为特殊的行为特性，重载会改变运算符的这些特性，进而影响我们原有的习惯，所以请不要去重载这三个可以重载的运算符。`

## 建议 52：合理地使用 `inline` 函数来提高效率

函数是一种高级的抽象，它的引入可以让我们只关心函数的功能和使用，而不必操心函数的具体实现方式。但是，函数在带来便利的同时，也会带来效率降低的问题。调用函数之前需要保护现场并暂存执行的地址，转回后则要恢复现场，并按原来保存的地址继续执行。因此，函数调用会有一定的时间和空间方面的开销。特别是对于一些函数体代码不大但又被频繁调用的函数来讲，解决其效率问题更为重要。

为了解决这一个问题，C 语言引入了“宏”这一武器。但是正如建议 25 所讲的那样，宏有很多缺点，所以 C++ 的创始人成功地引入了 `inline`（内联）函数来解决这一问题。内联函数具有不同于一般函数的调用方式：它会像宏一样在调用函数处用内联函数体的代码进行替换，而不用像一般的函数那样保存现场、跳来跳去。这样就节省了调用开销，提高运行效率。

内联函数具有与宏定义相同的代码效率，但在其他方面却要优于宏定义。因为内联函数还遵循函数的类型和作用域规则，所以它能像一般的函数那样进行调用与调试。

在大多数 C++ 程序中，内联是一个编译时行为。因为编译器只有首先了解了函数的大致情况后，才能够在函数调用处用函数体代替之，因此内联函数一般情况下都应该定义在头文件中。内联函数的定义方法很简单，一般来说分为以下两种方式。

#### □ 显式方式：在函数定义之前添加 `inline` 关键字

在标准库中我们经常能发现这样的内联函数声明方式，比如文件 `<algorithm>` 中的标准 `max` 模板定义：

```
template<class _Ty> inline
const _Ty& (max)(const _Ty& _Left, const _Ty& _Right)
{ // return larger of _Left and _Right }
```

```

    return (_DEBUG_LT(_Left, _Right) ? _Right : _Left);
}

```

一般情况下，我们习惯于“声明于.h、定义在.cpp”，所以 `inline` 通常会出现在 `.cpp` 文件的函数体附近。但是，模板是这一规则的例外，就像上面的 `max` 模板定义所示，它的 `inline` 是出现在头文件里的。

需要提醒的是，内联函数只有和函数体声明放在一起时 `inline` 关键字才具有效力。所以，如果想这样声明一个内联函数：

```
inline void InlineFunction(int para);
```

那么在编译器看来，它与普通的函数没有两样。所以，`inline` 应该时刻保持在函数体的周围，永不分离，如下所示：

```

inline void InlineFunction(int para)
{
    ... // processing codes
}

```

这样它才具有了比一般函数更快的执行能力。

#### □ 隐式方式：将函数定义于类的内部

C++ 标准规定：如果在类内部定义了函数体的函数，则默认其为内联函数，而不管是否有 `inline` 关键字。所以，下面的 `GetAge` 和 `SetAge` 函数虽然没有 `inline` 修饰，但它们同样具有内联的效果：

```

class Person
{
public:
    int GetAge() const { return m_nAge; }
    void SetAge(int age) { m_nAge = age; }
    ...
private:
    int m_nAge;
    ...
};

```

这种方式在定义类私有成员变量的存取函数时应用得很广泛，具有较高的效率。

当然，内联函数也有一定的局限性。`inline` 是对编译器的一次内联请求，编译器有权利拒绝它，坚持有所为有所不为。当编译器遇到内联函数时，就会针对函数体的上下文进行优化，以确定是否执行内联（注意区分本句话中的两个“内联”的不同：前者指的是用 `inline` 修饰，后者是指代码替代动作）。如果编译器认为当前的函数过于复杂、函数体过大，或者这个函数是虚函数，就会拒绝将其内联，而将其视为普通函数来处理。所以，一个给定的函数是否得到内联，很大程度上取决于你正在使用的编译器。

内联也是有开销的，不假思索地盲目使用内联是非常不明智的做法。在使用内联函数时，应该注意如下几点：

- (1) 内联函数的定义必须出现在内联函数第一次被调用之前。所以，它一般会置于头文件中。
- (2) 在内联函数内不允许用循环语句和开关语句，函数不能过于复杂。递归函数就是一个反例。
- (3) 依据经验，内联函数只适合于只有 1~5 行的小函数。
- (4) 对于内存空间有限的机器而言，慎用内联。过分地使用内联会造成函数代码的过度膨胀，会占用太多空间。
- (5) 不要对构造 / 析构函数进行内联。不要被你的眼睛欺骗，尽管构造 / 析构函数看似很短、很简单。至于为什么，请参考《Effective C++》第 2 版中的条款 33。
- (6) 大多开发环境不支持内联调试，所以为了调试方便，不要将内联优化放在调试阶段之前。
- (7) 在优化面前，既要坚持 80-20 原则，又要遵守“何乐而不为”的标准（有限的优化也是优化，小小的改变可获得一定的提升，何乐而不为）。

---

#### 请记住：

仅仅对执行代码少（一般量化标准 1~5 行）、调用频率高的程序进行内联。同时注意内联所带来的代码膨胀，谨慎使用内联。

---

## 建议 53：慎用私有继承

在类的继承体系中，我们最熟悉且应用也最为广泛的就是公有继承（public inheritance）。在 C++ 中，公有继承被视为“is-a”关系。公有继承使用 public 关键字，就像下面的代码：

```
class Animal
{
public:
    void Eat(){ std::cout<<"Eating..."<<std::endl; }
};

class Tiger : public Animal
{
public:
    bool IsKing() { return true; }
};
```

上面的继承关系很好地表达了“Tiger 是一种 Animal”的含义，所以 Tiger 继承了 AnimalEat 的能力。但是如果将关键字 public 换成 private，重新组织上面的代码：

```

class Animal
{
public:
    void Eat(){ std::cout<<"Eating..."<<std::endl; }
};

class Tiger : private Animal
{
public:
    bool IsKing() { return true; }
};

```

经过私有继承，Tiger 对象就不能再像原来那样去调用基类 Animal 中的 Eat 函数了。编译器不能将派生类对象转型为基类对象，所以也就不能调用基类中的成员函数。很明显，私有继承所表达的含义改变了，已不再是 is-a 的关系。

那么私有继承表达的含义是什么呢？

私有继承会使基类的所有东西（包括所有的成员变量与成员函数）在派生类中变成 private（私有）的，也就是说基类的全部在派生类中都只能作为实现细节，而不能成为接口。所以私有继承意味着“只有 implementation（实现）应该被继承，interface（接口）应该被忽略”，也就意味着是“is-implemented-in-terms-of（根据……而实现）”的内在关系。如果类 D 私有继承自类 B，那只能是因为类 B 中的某些特性可以实现类 D，而不是因为在类 B 对象和类 D 对象之间有什么概念上的关系。所以，私有继承只是一种实现技术，而不是设计。

如此说来，私有继承和组合（Composition）确实有几分相似。借用著名的 Car Has-A Engine 的例子：

```

class Engine
{
public:
    Engine(int numCylinders);
    void Start(); //Starts this Engine
};

class Car
{
public:
    //Initializes this Car with 8 cylinders
    Car() : m_engine(8) { }
    //Move this Car by starting its Engine
    void Move()
    { m_engine.Start(); }
private:
    Engine m_engine; //Car has-a Engine
};

```

它可以使用私有继承来进行表达：

```

class Car : private Engine
{
    //Car has-a Engine
public:
    //Initializes this Car with 8 cylinders
    Car() : Engine(8) { }
    //Move this Car by starting its Engine
    void Move()
    { Engine::Start(); }
};

```

在大多数情况下，组合是值得推荐的。因为通常我们不需要访问其他类太多的内部细节，但是私有继承却给了我们这样的能力，并且要我们承担昂贵的维护成本。所以说采用组合方式在概念上更加容易理解。那么，请尽量使用组合，必要时才使用私有继承。

不过什么时机才可称为“必要时”呢？“必要时”主要包括但不是仅仅包括以下两种情况：

- 当派生类需要访问基类保护成员时

如果我们出于某种考虑，不想别人去调用类 Engine 中的 Start 函数，所以将其声明为 protect，如下所示：

```

class Engine
{
public:
    Engine(int numCylinders);
protected:
    void Start();    // Starts this Engine
};

```

此时依旧采用常规的组合方式，编译器就会拒绝为你工作，给出这样的提示：

```
error C2248: "Engine::start" : 无法访问 protected 成员
```

遇到这种情况，我们只好求助于私有继承，Engine 和 Car 之间根本就不存在概念上的“is\_a”的关系，而是通过 Engine 的 Start 来启动 Car。私有继承可以帮助你完成目标：既能正确地表达概念，又能使你获得在一个类中调用另外一个类保护函数的超凡能力。

- 需要重定义继承来的虚函数（inherited virtual function）时

假设我们已经有了一个功能完善的时钟类 CTimer，如下所示：

```

class CTimer
{
public:
    explicit CTimer(int frequency);
    virtual void onTick() const;
    //...
};

```

我们能够根据需要，对这个时钟类设定 Tick 频率，每次 Tick 时，它都会调用一个 virtual 函数 OnTick。接下来，如果我们想借助这个时钟类 CTimer 实现游戏类 CGame 的

Tick，那么重写虚函数 OnTick 就在所难免了。因为 CGame is a CTimer 在概念上明显不成立，所以公用继承应该被否定并抛弃，私有继承便理所当然地进入了我们的考虑范畴，如下所示：

```
class CGame : private Timer
{
public:
    virtual void onTick() const
    {
        Timer::onTick();
        ... // other processing codes
    }
};
```

在上述的情形下，因为存在虚函数和保护成员，这使得私有继承成为了表达类之间“根据……而实现”关系的唯一有效途径。私有继承成为了一种设计策略，因为只有继承才能访问保护成员，也只有继承才能使虚函数可以重新被定义。而组合就显得力不从心了！

所以，在大多情况下，尽量使用组合，避免使用私有继承。而当私有继承成为你可以使用的最合适的设计策略时，就要毫不犹豫地、谨慎小心地去使用它。

#### 请记住：

私有继承意味着“只有 implementation 应该被继承，interface 应该被忽略”，代表着是“is-implemented-in-terms-of”的内在关系。通常情况下，这种关系可以采用组合的方式来实现，并提倡优先使用组合的方案。但是如果存在虚函数和保护成员，就会使组合方案失效，那么请勇敢地使用私有继承。

## 建议 54：抵制 MI 的糖衣炮弹

MI（多重继承）多年来一直是一个敏感而富有争议的话题。它将 OO 世界划分成了两个阵营：以 C++、Eiffel 为代表的力挺派和以 Object C、Smalltalk 为主力的反对派。拥护者认为它更能自然地塑造世界，将其视为避免笨拙地混合继承的利器；而反对者认为它打开了潘多拉魔盒，不仅迟缓低效，而且引入了多处混淆，同时还会带来单继承所不存在的复杂性问题。

MI 到底存在哪些利与弊呢？我们采用“沙发床”的经典例子来进行说明。假设有两个类——沙发 CSofa 和床 CBed，沙发是可以坐在上面的，而床可以睡在上面，它们功能不同，代码如下所示：

```
class CSofa
{
public:
```

```

void SitOn(); // 坐在上面
...
};

class CBed
{
public:
    void SleepOn(); // 睡在上面
    ...
};

```

但是当沙发和床的结合体——沙发床出现时，MI 似乎是最自然的表达方式，如下所示：

```

class CSofaBed : public CSofa, public CBed
{
public:
    ... // 其他接口
};

```

这里将 MI 的美妙之处展示得淋漓尽致。但是，必须提醒你，不要因为这些而忘乎所以，要抵制住这些糖衣炮弹的诱惑，因为在这背后隐藏着巨大的危机。仅仅在上面的 CSofa 类和 CBed 类中各加一个同名的成员函数就可以让你知晓危机是什么，如下所示：

```

class CSofa
{
public:
    void SitOn();
    virtual void Clean();
    ...
};

class CBed
{
public:
    void SleepOn();
    virtual void Clean();
    ...
};

```

因为两个类中存在着同名函数，所以调用这个函数会造成一个模棱两可的局面。解决之道就是明确地指出所调用的是哪个，如下所示：

```

CSofaBed* pSofabed = new CSofaBed();
pSofabed->Clean();           // ERROR: 对 Clean 的访问不明确
pSofabed->CBed::Clean();    // 调用 CBed::Clean
pSofabed->CSofa::Clean();   // 调用 CSofa::Clean

```

虽然采用明确调用的方式程序能够运行，但是这么明确的调用会直接破坏掉该函数原本引以为傲的虚拟特性。当然，这也并不是没有解决方法，有经验的程序员会采用增加一对辅

助类的方式来巧妙地绕过这个问题，如下所示：

```
class AuxSofa : public CSofa
{
public:
    virtual void CleanSofa() = 0;
    virtual void Clean() { CleanSofa(); }
};

class AuxBed : public CBed
{
public:
    virtual void CleanBed() = 0;
    virtual void Clean() { CleanBed(); }
};

class CSofaBed : public AuxSofa,
                 public AuxBed
{
public:
    virtual void CleanSofa();
    virtual void CleanBed();
    ... // 其他接口
};
```

因为两个辅助类 AuxSofa 和 AuxBed 中都含有纯虚函数，所以在实体类 CSofaBed 中必须对那些纯虚函数进行定义。在使用的时候采用以下方式，可以重新获得函数的虚拟特性：

```
CSofaBed* pSofabed = new CSofaBed();
CBed* pBed = pSofabed; // cast
CSofa* pSofa = pSofabed; // cast
pBed->Clean();
pSofa->Clean();
```

以上只能说是 MI 所带来问题的冰山一角，它带来的最大问题是所谓的“钻石型继承结构（DOD）”。有人将 DOD 称为 Diamond of Death（死亡之钻），足见问题之严重性。要想构成钻石型继承，我们只需在前面的例子中添加一个家具类 CFurniture，如下所示：

```
class CFurniture
{
public:
    virtual void Placing(int where); // 放置在什么地方
    virtual void Clean() = 0;
    ...
};
```

因为沙发和床都是家具中的一种，它们与家具之间存在着 is\_a 的关系，所以采用公有继承的方式实现类 CSofa 和 CBed，如下所示：

```
class CSofa : public CFurniture { };
```

```
class CBed : public CFurniture {};
```

再加上 CSofaBed 类，DOD 搭建完毕，如图 4-6 所示。

因为 CSofaBed 类继承自类 CSofa 和 CBed，它们各有一份 CFurniture 数据，所以在 CSofaBed 类对象空间中存在着两份 CFurniture，这显然是多余的。要想避免之，就必须让 CSofa 和 CBed 都将 CFurniture 声明为虚基类（virtual base class）。虚继承虽然能解决两份 CFurniture 的问题，但是它同样会引起程序空间与执行时间成本的增加，因为其实现是通过“对象指针”来完成的。

除了虚继承，DOD 还会加剧同名函数所引发的调用不确定性。如果上述例子中，CSofa 没有对函数 Placing 进行 overriding，而 CBed 类对它进行了特殊定制，那么此时，CSofaBed 类对象调用的该是哪一个 Placing 呢？如下所示：

```
CSofaBed* pSofaBed = new CSofaBed;
pSofaBed->Placing(w);
```

答案是：如果 CFurniture 是 CSofa 和 CBed 的非虚基类，那么编译器会因为二义性而罢工；如果是虚基类，那么会直接调用 CBed::Placing(w)，因为 CBed 对 Placing 的重定义优先级要高于虚基类中的函数定义。

鉴于 MI 有如此高的复杂性，并且随之而来的是设计的高技巧、维护的高难度。所以，坚决抵制 MI 的糖衣炮弹，尽量少使用 MI。

#### 请记住：

MI 意味着设计的高复杂性、维护的高难度性，所以不要为了塑造世界时的那点自然而给自己带来这么多不必要的麻烦。经受住 MI 糖衣炮弹的诱惑，尽量少使用 MI。

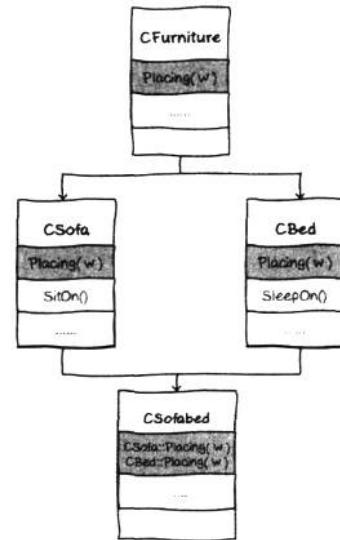


图 4-6 DOD 示例

## 建议 55：提防对象切片

多态是 C++ 的重要特征之一，它将接口 How 与具体实现 What 成功地进行了分离，有效地提高了代码的组织性和可读性。多态的实现方式之一就是晚绑定，而晚绑定的实现需要虚函数机制和继承机制的支持：关键字 virtual 告诉编译器它应该实行晚绑定，编译器给每个包含虚函数的类创建虚函数表 vtable，以便存放类的虚函数地址；编译器会通过虚函数指针 vptr 在 vtable 中查找函数地址，并在运行时确定对象的类型和合适的调用函数。由此可见继

承机制与虚函数机制是实现晚绑定多态的重要元素。

但是需要注意的是，多态的实现必须依靠指向同一类族的指针或引用。否则，就可能出现著名的对象切片（Object Slicing）问题。就像下面的这个例子：

```
class Bird
{
public:
    Bird(const string& name) : m_name(name) {}
    virtual string Feature() const
    {
        return m_name + " can fly.";
    }
protected:
    string m_name;
};

class Parrot : public Bird
{
public:
    Parrot(const string& name, const string& food)
        : Bird(name), m_food(food) {}
    virtual string Feature() const
    {
        return (m_name + " can fly and likes to eat " + m_food);
    }
private:
    string m_food;
};

void DescribeBird(Bird bird)
{
    cout << bird.Feature() << endl;
}

int main()
{
    Bird bird1("Crow");
    DescribeBird(bird1); // 正常执行
    Parrot bird2("Polly", "millet");
    DescribeBird(bird2); // 出乎意料
    return 0;
}
```

程序执行，输出的结果是：

```
Crow can fly.  
Polly can fly.
```

而不是我们期待的：

```
Crow can fly.  
Polly can fly and likes to eat millet.
```

究其原因，就是发生了对象切片。引用《Thinking in C++》中的一段话：

如果你强制向上转换一个对象，而非对象的指针或引用，那么很可能会发生令你吃惊的事儿：对象被切片了，保留下的是一个子对象，这个子对象对应于你的转换目标类型。<sup>⊖</sup>

C++ 内存模型规定，如果出现继承结构，内存分布一定是先基类部分的数据，后派生类部分的数据。在派生类向基类映射的过程中，派生类对象中基类部分的数据会被强行“切”掉。这里的“切”字之所以加上引号，是因为这个字在某种程度上会引起误解，基类部分的数据被切掉后其实并没有被丢弃，而是将其当成了一个基类对象。在上述示例代码中，Parrot 类对象 bird2 被

切片后，会经由拷贝构造在栈上将其生成为一个新的 Bird 对象，供函数 DescribeBird 使用。Parrot 类对象 bird2 在切片前后发生的变化如图 4-7 所示。

由此可见，对象切片通常发生在派生类对象被赋值到基类对象的时候。由于派生类在继承基类时，通常会增加一些变量或函数，所以派生类对象的大小要比基类对象大，在赋值时，子类对象会产生多余部分，所以会发生切片现象。

如果采用对象指针或引用，上述问题都会成为“浮云”，多态会按照我们的期望走上正轨，如下所示：

```
// 引用版  
void DescribeBird(Bird& bird)  
{  
    cout<<bird.Feature()<<endl;  
}  
// 指针版  
void DescribeBird(Bird* bird)  
{  
    cout<<bird->Feature()<<endl;  
}
```

当类中没有虚拟机制时，也有可能会发生对象切片，只不过在这种情况下编译器会提示出错信息，如下所示：

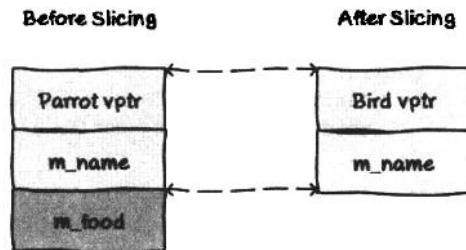


图 4-7 对象切片的前后对比图

<sup>⊖</sup> If you upcast to an object instead of a pointer or reference, something will happen that may surprise you: the object is “sliced” until all that remains is the subobject that corresponds to the destination type of your cast.

```

class Base
{
public:
    Base():m_data(0){}
    void Set(int num){ m_data = num; }
    int Get(){ return m_data; }
private:
    int m_data;
};

class Derive: public Base
{
public:
    Derive():Base(){}
    void Print(){ cout<<Get(); }
};

Derive d;
Base b = d;
b.Print(); // Error

```

此时产生对象切片的原因是 bitwise 的拷贝构造将派生类的部分信息全部抛弃了，对象 b 中根本就没有了关于派生类 Print 的任何信息。不过此时的对象切片很容易发现，且解决方法简单，故不赘述。

#### 请记住：

多态的实现必须依靠指向同一类族的指针或引用。否则，就可能出现著名的对象切片（Object Slicing）问题。所以，在既有继承又有虚函数的情况下，一定要提防对象切片问题。

## 建议 56：在正确的场合使用恰当的特性

C++ 语言之所以被称为“a better C”，很大一部分原因是在 C++ 中增加了一些它所特有的高级特性：虚函数、虚基类、多重继承、RTTI 等。这些高级特性的引入确实为我们的编程带来了便利，但是这些看上去简单的特性却可能导致效率低下。所以要使用它们时，必须先了解这样做的代价，以便在正确的场合使用恰当的特性。这也是对一个合格 C++ 程序员的基本要求。

这些语言特性的实现细节很大程度上取决于编译器所选择的实现方法。这些方法可能会对对象的大小、成员函数的执行效率带来显著的影响，所以我们应该对这些特性有一个基本的了解，知道编译器在后台为我们做了哪些事。

#### □ 虚函数

虚函数机制的实现是通过虚函数表和指向虚函数表的指针来完成的。关键字 virtual 告诉

编译器该函数应该实行晚绑定，编译器对每个包含虚函数的类创建虚函数表 VTable，以放置类的虚函数地址。编译器秘密放置了指向虚函数表的指针 VPtr，当多态调用时，它会使用 VPtr 在 VTable 表中查找要执行的函数地址。在如下代码片段中，类 D 派生自类 B，并且具有一些虚函数：

```
class B
{
public:
    B();
    virtual ~B();
    virtual void Fun1();
    virtual int Fun2(int para);
    void Fun3();

    ...
};

class D : public B
{
public:
    D();
    virtual ~D();
    virtual void Fun1();
    virtual void Fun4();

    ...
};
```

基类 B 和派生类 D 虚函数表的组织形式如图 4-8 所示。

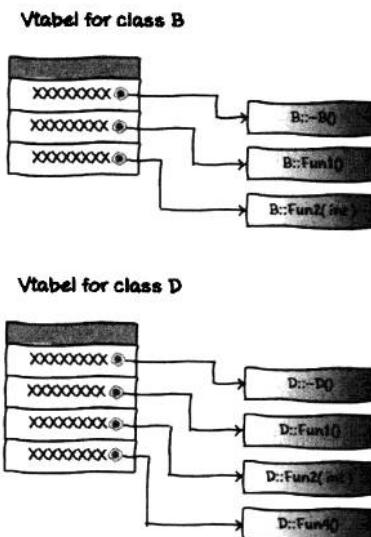


图 4-8 类 B 和 D 的虚函数表

很明显，类 D 继承自类 B，它不仅对基类原有的构造函数和函数 Fun1 重新进行了定义，而且新添加了虚函数 Fun4。

有了虚函数表还只是完成了虚函数机制一半的工作，我们还需要通过一定地方法来实现对象与虚函数表的对应。这就是前面所说的指向虚函数表的指针 VPtr，这个指针存放在对象空间中，如图 4-9 所示。

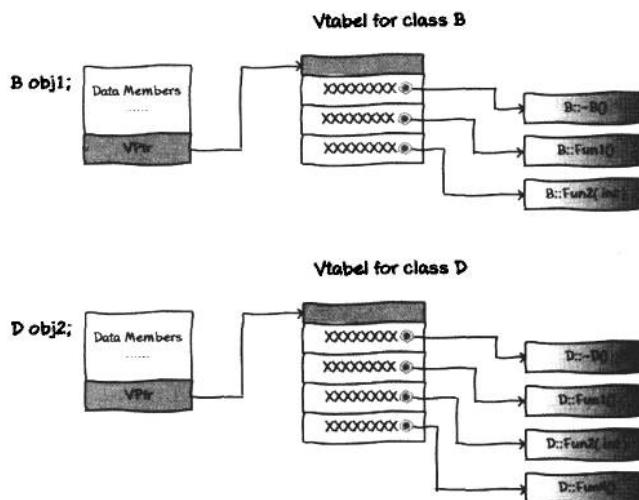


图 4-9 VPtr 与 VTable 的对应

虽然虚函数的实现有些复杂，增加了虚函数表、虚函数指针，但它依然是 C++ 众多运行时多态特性中开销最小、最常用的机制。也许你会因此而感到诧异，因为对于具有虚函数的类来说，不仅要为其对象分配多余的空间来存放 VPtr，还要满足存放虚函数表而带来的空间需求，怎么会是开销最小的呢？这是因为在程序中每个类只有一个虚函数表的拷贝，所以它占用的空间不是很大。

在时间方面，虚函数的调用开销包括两次整型加法的开销和一次指针间接引用的开销。其动态绑定的实现步骤如下：

- (1) 根据对象的虚指针 VPtr 找到该对象对应的虚函数表 VTable，所需的开销仅是一次偏移量调整（整形加法）加上一次指针间接运算。
- (2) 在 VTable 中找到被调用函数的对应指针。这个步骤实质上也是一次整型加法运算。
- (3) 调用 (2) 中得到的指针所指向的函数。

所以步骤 (1) 和 (2) 的开销与 (3) 中函数的复杂调用开销（保存现场→传递参数→传递返回值→恢复现场）相比是微不足道的。

所以，在绝大部分应用中，虚函数机制所带来的空间和时间成本是我们能够负担得起的，它通常并不会成为性能上的瓶颈。但还是必须注意：在对性能要求比较苛刻的场合，要

慎用虚函数。

#### □ 多重继承

对于多重继承来说，对象内部会有多个 VPtr，所以这就使偏移量计算变得复杂了，而且会使对象占用的空间和运行时开销都变大。如下面的代码片段所示：

```
class A { ... };
class B { ... };
class C : public A, public B { ... };
```

如果 A 和 B 中都具有虚函数，那么 C 类对象的内存布局就会像图 4-10 所示一样。

一个对象之中有三个 VPtr，所以需要更多的空间去存储这几个虚函数指针，同时函数偏移量的计算也要三选其一，计算因此也就变复杂了。所以，与虚函数相比，多重继承对性能造成的影响要稍大些。

#### □ 虚基类

虚基类与多重继承的情况类似。因为虚基类就是为了多重继承而产生的，关于这一点在建议 54 中已经详细阐述。作为一种支持多继承的语言，虚基类有时是保证类层次结构正确一致的一种必不可少的手段。在虚基类的实现过程中经常会使用指向虚基类的指针来避免基类的重复。所以对象内部除了插入基类虚函数指针外，还要插入指向虚基类的指针。与多重继承相比，对象中的指针更多。我们还是以建议 54 中的钻石型继承结构（DOD）为例来说明，其内存排列情况会变成如图 4-11 所示的形式。

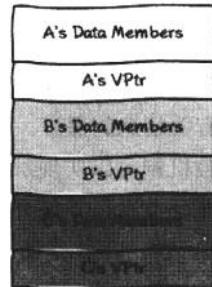


图 4-10 C 类对象的内存布局

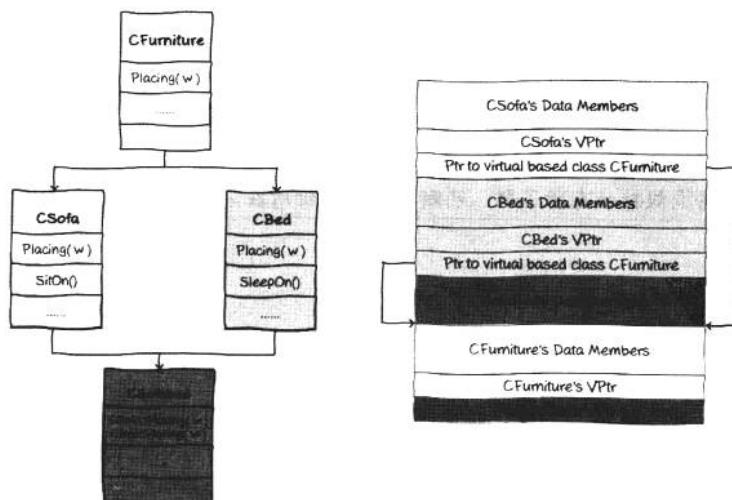


图 4-11 DOD 内存分布示例

上述的内存排列情况仅仅是“示例”，是概念层次上的。至于编译器的实现完全取决于编译器本身，不同的编译器之间也会存在差异。不过这个示例却可以很清晰地表明：虚基类确实导致了对象中指针数量的增加。因此对象会更加膨胀，函数偏移量的计算也更加复杂。所以，在对性能要求比较高的应用中要慎重考虑它的使用。

在 DOD 中，如果虚基类没有数据成员，那么就可以毫不客气地抛弃虚基类这一特性，消除因为虚基类而带来的开销。没有数据成员就不会带来建议 54 中所讲的问题。

#### □ 运行时类型检测 (RTTI)

运行时类型检测 (Run-Time Type Identification, RTTI) 是我们在程序运行时得到对象和类有关信息的保证。一般情况下，我们并不需要知道一个类的确切类型，因为 C++ 提供的虚函数机制可以实现那种类型的正确行为。但是有些时候，确定一个匿名的多态指针指向对象的准确类型也是很有用的。

典型的 RTTI 是通过在 VTable 中放一个额外的指针来实现的。这个指针指向一个专门用于描述该特定类型的 type\_info 结构。每一个类型仅有一份 type\_info 结构的拷贝。我们一般通过 typeid() 表达式获取对象的类型信息，typeid() 带有一个参数，它可以是一个对象引用也可以是一个指针，它返回一个 type\_info 类型常量对象的引用。在使用中，我们可以应用运算符 “==” 和 “!=” 来比较这些对象，也可用 name() 来获得该类型的名称。

这种基于虚函数的 RTTI 是一个很方便也很有用的特点，它已出现在几乎所有的类库中，其主要代价就是存储 type\_info 类型信息所带来的存储空间的增加和获得该信息时取址操作带来的时间耗费。

由此可见，上述高级特性并不完美，总会在设计上有所取舍。但是通常而言这些特性无论是空间上的开销还是时间上的开销一般都可以接受。在一些特殊情况下，比如存储布局需要和传统的 C 结构兼容或在对性能要求极高的应用中，则要慎重考虑。在不正确的场合使用它们必然会引起逻辑、行为和性能上的问题。

---

#### 请记住：

对 C++ 的一些高级特性有所了解，明晰其主要代价所在，在实际应用中根据需求有所取舍。

---

## 建议 57：将数据成员声明为 private

数据是任何编程语言都离不开的话题，无论是在面向过程的 C 中还是在面向对象的 C++ 中。不过，它们在对待数据的态度上却是截然不同的：在 C 的世界里，数据只是配角，它是服务于过程设计的；而到了 C++ 的时代，数据摇身一变成了程序设计的中心，扮演着一号角色。最具代表性的例子就是结构与类：结构可以看作是一堆缺乏封装的内存位，访问直接；而类则有牢固可靠的封装屏障和良好定义的接口。

所以从 C 到 C++，程序设计是由以过程设计向以数据组织为中心转移的过程，这是一种编程思想的转变。我们要尽量地将数据成员进行封装，给它配置上安全屏障，禁止非法访问。C++ 封装数据最简单的方法就是用关键字 public、private、protected 修饰数据成员，赋予它们不同的访问权限。大多数情况下，建议将数据成员声明为 private。

下面我们来看看为什么数据成员不应该声明为 public。

其一，实现数据成员的访问控制。

public 代表着最高的访问权限，如果将一个数据成员声明为 public，那么每一个人都可以读写访问它，这是失去了数据保护与封装的意义。如果数据成员声明为 public，那么这个类与结构体无异。用 private 修饰成员变量，可让数据成员隐藏起来，通过数据成员的 Getter 和 Setter 来实现访问控制，这样就可以更加精确地控制成员的可读写性。就像下面的代码片段：

```
class AccessLevelΘ
{
public:
    int GetReadOnly() const { return m_nReadOnly; }
    void SetReadWrite(int value) { m_nReadWrite = value; }
    int GetReadWrite() const { return m_nReadWrite; }
    void SetWriteOnly(int value) { m_nWriteOnly = value; }
private:
    int m_nNoAccess; // 不可读写
    int m_nReadOnly; // 只可读
    int m_nReadWrite; // 可读可写
    int m_nWriteOnly; // 只可写
};
```

将数据成员声明为 private，通过函数去获得和设置它的值，可以实现禁止访问、只读访问和读写访问。这样就无法违背类设计者的设计意图，必须遵循数据的访问设置进行访问，于是保护了数据成员的安全。

其二，在将来时态下设计程序，为之后的各种实现提供了弹性。

将数据成员隐藏于功能性的接口中，其实是为我们预留了以后改变实现决策的权利，并为之后的各种实现提供了弹性，我们可以在将来用一种更好的实现方式替换现有的实现，这就是封装的力量。封装度越高，系统赋予我们修改它的能力就越强。

如果数据成员被声明为 public，就赋予了客户使用它的权利，从某种意义上来说是在鼓励这种行为。当你将来想对该类的实现重新进行设计时，会发现这是一件不可能完成的事，因为太多的客户代码直接访问了对象的数据成员，如果发生改变，太多的客户代码会受影响，甚至遭受破坏。假设类 B 包含一个 public 数据成员 m\_Num，随后改变了类 B 的实现，

<sup>Θ</sup> 此段代码是在 Scott Meyers 的版本上做了修改，在此对原作者表示感谢！

消除了数据成员 m\_Num。想想那将会有多少代码会被破坏呢？答案就是：所有使用了 m\_Num 的客户代码。而如果用 private 来修饰数据成员，通过函数访问的数据成员、客户代码对类实现的变化基本免疫。其实，可以自由地对类的实现进行改变，只要保证原有接口的存在就行。

其三，保持语法的一致性。

笔者习惯于说“最后，但并非是最不重要的（Last, but not the least）”，这里也写下这句话，因为这一条很重要。如果我们将数据成员声明为 private，那就意味着客户访问对象的唯一方法就是通过成员函数。坚持这一原则，我们就不必在访问类成员方式的问题上再纠结了，这可以节省我们的宝贵时间。

除了 public 外，为什么也反对 protected 数据成员呢？

因为 protected 对数据的保护不够彻底，它的派生类是可以访问 protected 数据成员的。所以这会在小范围内引起 public 所带来的几个问题：protected 数据成员不能实现派生类对数据成员的访问控制、类实现的变化会破坏派生类的代码和语法一致性。也就是说，不能将数据成员声明为 public 的理由也完全适用于 protected，只不过 public 的作用范围比 protected 大那么一点点，protected 的封装效果也不好。

#### 请记住：

将数据成员声明为 private 是具有相当充分的理由的：(1) 实现数据成员的访问控制；(2) 在将来时态下设计程序，为之后的各种实现提供弹性；(3) 保持语法的一致性。

所以，请将数据成员声明为 private。

## 建议 58：明晰对象构造与析构的顺序

对象的构造与析构掌管着对象的“生杀大权”，所以对于对象的构造与析构也有着比较严格的先后执行顺序，特别是在继承关系中以及存在成员类对象时。

首先，看一下继承关系中的构造与析构顺序。还是以那个老套的沙发床为例，如下所示：

```
class CSofa
{
public:
    CSofa(){ cout<<"Sofa's ctor"<<endl; }
    virtual ~CSofa(){ cout<<"Sofa's dtor"<<endl; }
};

class CBed
{
public:
```

```

CBed(){ cout<<"Bed's ctor"<<endl; }
virtual ~CBed(){ cout<<"Bed's dtor"<<endl; }
};

class CSofaBed : public CSofa, public CBed
{
public:
    CSofaBed(){ cout<<"SofaBed's ctor"<<endl; }
    virtual ~CSofaBed(){ cout<<"SofaBed's dtor"<<endl; }
};

int main()
{
    CSofaBed sofaBed;
    return 0;
}

```

代码运行结果如下：

```

Sofa's ctor
Bed's ctor
SofaBed's ctor
SofaBed's dtor
Bed's dtor
Sofa's dtor

```

通过这段程序，我们可以得到如下的结论：对象的构造都是从类的最根处开始的，由深及浅，先基类后子类，层层构造，这个顺序不能改变。如果含有多个基类，那么就按照声明顺序（class CSofaBed : public CSofa, public CBed）由前及后执行（先 CSofa 后 CBed）。析构函数则严格按照构造的逆序执行。

当一个类存在成员类对象时，构造与析构顺序也比较容易混淆。还是写段代码来进行验证：

```

class CSofa { ... };
class CBed { ... };
class CHome
{
public:
    CHome(){ cout<<"Home's ctor"<<endl; }
    virtual ~CHome(){ cout<<"Home's dtor"<<endl; }

private:
    CSofa m_sofa;
    CBed m_bed;
};

```

家中有沙发又有床是很普遍的，就像上面的 CHome 类，它包含一个沙发类 CSofa 对象和一个床类 CBed 对象。结果很清晰。

```
Sofa's ctor
Bed's ctor
Home's ctor
Home's dtor
Bed's dtor
Sofa's dtor
```

由此可见，成员对象构造函数的调用顺序与成员对象的声明顺序严格一致，析构顺序是构造顺序的严格逆序。这是因为类的声明是绝对唯一的，而类的构造函数可以有多个，所以按照声明才会使析构函数得到唯一的逆序。

如果再复杂一点，让继承体系和成员对象相遇在一个类的声明里，那又会怎么样呢？如果我们将成员对象看作是普通的成员变量，也许就会得到这样的结论：首先调用基类的构造函数，然后调用成员对象的构造函数。我们的推断并没错，事实确实也是如此。如果你有兴趣，可以设计一个这样的例子程序，亲自验证。

#### 请记住：

在继承体系中，对象的构造都是从类的最根处开始的，由深及浅，先基类后子类，层层构造。如果含有多个基类，那么就按照声明顺序，由前及后执行。

当类中出现成员对象时，成员对象构造函数的调用顺序与成员对象的声明顺序严格一致。

如果继承遇到成员对象，基类构造函数依然会被首先调用。

### 建议 59：明了如何在主调函数启动前调用函数

某些应用程序需要在调用主程序 main 之前开始启动功能，例如 Logger（日志记录），该函数必须在调用实际的程序之前开始工作。也许有人会因此而感到疑惑，这怎么可能实现呢？因为“main() 是程序执行的入口”就像一条真理一样深深地印在了我们的脑海里。一边是操作系统从 main() 开始执行，一边是要在 main() 之前调用一些程序功能函数，这个矛盾能否解决呢？

答案是可以解决。最简单的实现方式是调用一个全局对象的构造函数。因为从概念上说，全局对象是在程序开始前已经完成了构造，而在程序执行之后才会实施析构。就像下面的代码片段：

```
void ActivateLog()
{
    std::cout<<"Log Starting..." << std::endl;
}
void Logging()
{
    std::cout<<"Logging..." << std::endl;
```

```

}

class Logger
{
public:
    Logger()
    {
        ActivateLog();
    }
~Logger()
{
    std::cout<<"Log Ending..." << std::endl;
}
};

Logger log; // 全局变量

int main()
{
    std::cout<<"Main() start" << std::endl;
    //... application code
    Logging();
    return int(&(std::cout<<"Main() end" << std::endl));
}

```

程序执行结果：

```

Log Starting...
Main() start
Logging...
Main() end
Log Ending...

```

很明显，程序执行的结果告诉我们：全局对象 log 在 main() 开始之前完成了构造，在构造过程中，log 触发了函数 ActivateLog()，启动了日志记录功能。当 main() 开始时，日志功能将正常执行。main() 函数返回后，全局对象 log 析构，将内存空间交还给了操作系统。

这项技术也被广泛地用在了标准模板库（Standard Template Library，STL）里。例如，iostream 的对象 cout 和 cin 都是在 main() 之前被构建的。那么 C++ 如何确保全局对象在 main() 之前被构建呢？

在 C 语言中，全局变量的初始化发生在所有代码执行之前，属于编译期初始化。对于内置类型变量，无须调用构造和析构函数进行资源的申请与释放！但是到了 C++ 时代，由于类的引入，全局对象必须在 main() 函数之前调用构造函数来完成对象的构建，在 main() 结束之后调用析构函数释放资源，于是问题变得不再像 C 时代那么简单。这时就需要一种机制或办法来帮助我们安置构造 / 析构函数的调用操作。在大多数的实现方式里，核心会运行专门的启动代码，启动代码会在启动 main() 之前完成所有的初始化工作。这个所谓的启动代码就是 C Runtime 函数库的 Startup 代码。

在应用程序执行时，系统会先调用 Startup，完成函数库初始化、进程信息设立、I/O stream 产生，以及对 static 对象的初始化等动作。然后 Startup 调用 main() 函数，把控制权交给 main() 函数。main() 函数执行完毕，控制权又交回给 Startup。到了这里，你肯定会茅塞顿开，对“main() 是程序执行的入口”也会有更加深刻的理解。

---

**请记住：**

如果想在主程序 main 启动之前调用某些函数，调用全局对象的构造函数绝对是一个很不错的方法。

---

# 第5章 用好模板，向着 GP 开进

代码复用一直是我们程序员苦苦追求的重要目标之一。现代编程语言为我们提供了三种主要的途径：结构化、面向对象和泛型。“20世纪70年代，结构化的程序设计方法风靡一时；到了80年代，面向对象独霸天下；而从90年代中期开始，泛型编程则开始大行其道<sup>⊖</sup>。”泛型编程允许程序员在使用强类型程序设计语言编写代码时定义一些可变部分。

在泛型编程的思想里，大部分基本算法被抽象，被泛化，独立于与之对应的数据结构，用于以相同或相近的方式处理各种不同情形。而在C++语言中支持这一思想的技术就是模板。模板已经成为C++泛型编程中不可缺少的一部分，成为C++程序员的绝佳武器。

接下来就让我们更加深刻地去了解、认识这一武器。

## 建议 60：审慎地在动、静多态之间选择

多态（Polymorphism）的字面含义是：具有多种不同的形态。在程序设计中，它被广泛地认为拥有“一种将不同的行为和单个泛化记号相关联的能力”。说到多态，我们不由自主地会想到两个关键词：虚函数机制和模板。

在静态类型语言中，多态是动静结合的产物，它将静态类型的安全性和动态类型的灵活性融为一体，达到了“动静兼济总相宜”的境界。它的实现方式有两种：一种是利用OOP中的子类型多态（Subtyping Polymorphism），一种是利用GP（泛型编程）中的参数多态（Parametric Polymorphism）。在C++语言中，它们的具体实现方式就是上面所提的虚函数机制和模板。从实现机制上看，二者的不同之处在于何时将一个变量与其实际类型所定义的行为进行绑定：虚函数机制配合继承机制，生效于运行期，属于晚绑定（Late Binding），是动多态（Dynamic Polymorphism）；而模板将不同的行为和单个泛化记号相关联发生在编译期，属于早绑定（Early Binding），被称为静多态（Static Polymorphism）。所以，我们要区分它们之间的不同，根据应用场景，审慎选择，择适者而用。

### □ 动多态

动多态的技术基础是继承机制和虚函数，它在继承体系之间通过虚函数来表达共同的接口。这一点我们已经说过了多遍，直接看代码片段：

```
class CShape // 图形类——抽象接口类
{
```

<sup>⊖</sup> 改编自文章《C++ 模板：过犹不及》。

```

public:
    virtual void Draw() = 0;
    ...
};

class CRect : public CShape // 四方形类——具体类
{
public:
    virtual void Draw()
    {
        // To do
    }
};

class CTriangle : public CShape // 三角形类——具体类
{
public:
    virtual void Draw()
    {
        // To do
    }
};

```

在上面的代码中，我们定义了一个图形类 CShape 作为抽象接口类，并通过继承这个类定义了两个实体类：四边形类 CRect 和三角形类 CTriangle。抽象接口类中声明了方法 Draw，实体类中则针对不同的情况进行特殊的定制实现。客户程序可以通过指向基类的指针或引用来操纵具体的对象，虚函数机制则保证了对应方法的正确调用，如下所示：

```
CShape* pShape1 = new CRect;
pShape1->Draw(); // CRect's Draw
```



```
CShape* pShape2 = new CTriangle;
pShape2->Draw(); // CTriangle's Draw
```



由此可见，动态适用于类层次结构，需要虚函数机制的支持。

#### □ 静多态

静多态的技术基础是模板，通过“彼�单独定义但支持共同操作的具体类”来表达共性。也就是说，类之间可以毫无联系，但是它们必须具有相同的操作。再看下面这个例子，我们定义了三个类：CBird、CCloud、CAirplane。很明显，鸟、云、飞机三者之间根本没法建立起 Is-A 或 Has-A 的逻辑关系，它们是毫无瓜葛的三种实体：

```

class CBird // 鸟
{
public:
    ...
private:
    CVector3D m_vPos;
```

```

};

class CCloud // 云
{
public:
    ...
private:
    CVector3D m_vPos;
};

class CAirplane // 飞机
{
public:
    ...
private:
    CVector3D m_vPos;
};

```

但是它们有一个相同的行为——在湛蓝的天空中“翱翔”：Fly，它们的位置信息m\_vPos可以被改变。所以，它们就有了“相同的操作”，具备了用模板实现多态的逻辑：

```

template <typename CObject>
void Fly(const CObject& obj)
{
    Move Object's Position...
}

```

注意，CCObject是模板参数，而非公共基类。我们按照下面的方式进行调用：

```

CBird bird;
Fly< CBird >(bird);
CCloud cloud;
Fly< CCloud >(cloud);
CAirplane plane;
Fly< CAirplane >(plane);

```

在编译时，编译器会按照客户代码进行编译，并得到不同的函数，静态多态成功实现：

// 运行结果：(多姿多彩的飞行)



与动态多态相比，静态多态始终在和参数“较劲儿”，它适用于所有的类，与虚函数无关。

从应用形式上看，静态多态是发散式的，让相同的实现代码应用于不同的场合；动态多态是收敛式的，让不同的实现代码应用于相同的场合。从思维方式上看，前者是泛型式编程风格，它看重的是算法的普适性；后者是对象式编程风格，它看重的是接口与实现的分离度。

除了这些高深语义上的差异，它们主要还有以下几方面的不同：

- 动多态的函数需要通过指针或引用传参，而静多态则可以传值、传值针、传引用等，“适应性”更强。
- 在性能上，静多态优于动多态，因为静多态无间接访问的迂回代码，它是单刀直入的。
- 因为实现多态的先后顺序不同，所以如果出现错误，它们抛出错误的时刻也不一样：动多态会在运行时报错，而静多态则在编译时报错。

尽管二者有着这么多的区别，但它们却有一个共同的目的：在保证必要的类型安全的前提下，突破编译期间过于严苛的类型限制，追求“动静兼济总相宜”的至高境界。在一些高级应用中，我们可能需要结合使用这两种多态机制，以实现对象操作的优雅、安全和高效。

#### 请记住：

动静两种多态各有侧重点：静态绑定，重在算法的普适性上，好让相同的实现代码应用于不同的场合；而动态绑定，重在接口与实现的分离度上，好让不同的实现代码应用于相同的情形。

根据应用的具体情形，合理地运用基于类继承、虚函数的动多态和基于模板的静多态，增强程序的简洁性、灵活性、可维护性、可重用性和可扩展性。

## 建议 61：将模板的声明和定义放置在同一个头文件里

在 C++ 中，一个类通常会对应着两个文件：一个是头文件 Header File，用于保存类的声明；另一个是定义文件 Definition File，用于保存程序的实现，如图 5-1 所示。

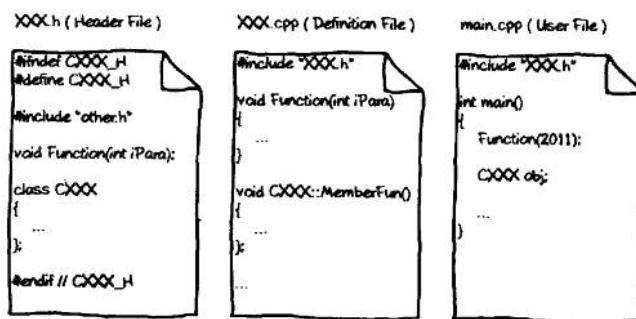


图 5-1 类文件结构

在 .h 文件中声明函数和类，然后将它们的定义放置在一个单独的 .cpp 文件中，这似乎已经成为了 C++ 程序员们所奉行的一贯准则或标准。这是因为分离编译模式 (Separate Compilation Model) 允许在一个翻译单元 (Translation Unit) 中定义 (define) 函数、类型、

类对象等，然后在另一个翻译单元中引用它们。编译器（Compiler）处理完所有的翻译单元后，链接器（Linker）接下来会处理所有指向 `extern` 符号的引用，从而生成单一可执行文件。这使得 C++ 代码编写看起来称心而优雅。

但是当这条准则遇到模板时，这种习惯性做法将变得不再有用。由于模板类型不是一种实类型，它必须等到类型绑定后才能确定最终类型，所以在实例化一个模板时，必须要能够让编译器“看到”在哪里使用了模板，而且必须要看到模板确切的定义，而不仅仅是它的声明，否则将不能正常而顺利地产生编译代码。原因很简单，编译器无法预先知道 `class T` 的实参是什么。所以，标准会要求模板的实例化与定义体放到同一翻译单元中。

解决这一问题的具体方法有三种：

第一种方法（同时也是最好的办法），是将模板的声明和定义都放置在同一个 .h 文件中，如下面的代码片段所示：

```
// Temp.h - 头文件

template<class T> // 函数模板
T MAX(const T& a, const T& b)
{
    return(a > b ? a : b);
}

template <class T> // 类模板
class List_item
{
public:
    List_item( T value, List_item *item_to_link_to = NULL ){}
    T GetValue() const { return m_value; }
    void SetValue(const T& et){ m_value = et; }
    void SetNext(List_item *link) { m_pNext = link; }
    List_item* GetNext(){ return m_pNext; }

private:
    T          m_value;
    List_item *m_pNext;
};
```

这是一种极为通用的模板定义方式，在 STL 头文件中我们能很轻松地找到类似上述代码片段的模板定义代码。但是，声明与定义放在一起，这在某种程度上破坏了 C++ 编程的优雅性。另一方面因为函数定义出现在了头文件中，所以上述文件中的函数同时会被当成内联函数，这也会引起代码膨胀的隐忧。

第二种方法是按照旧有的习惯性做法来处理：声明是声明，实现是实现，二者相互分离，但是在需要包含头文件的地方做一些改变。下面的代码片段在 `Temp.h` 文件中声明了一个函数模板和一个类模板：

```
// Temp.h

template<class T>
T MAX(const T& a, const T& b);

template <class T>
class List_item
{
public:
    List_item( T value, List_item *item = NULL );
    T GetValue() const;
    void SetValue(const T& et);
    void SetNext(List_item *link);
    List_item* GetNext();
private:
    T           m_value;
    List_item *m_pNext;
};


```

按照旧有的习惯，模板的实现依旧出现在了 Temp.cpp 文件中，如下所示：

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

template <class T>
List_item<T>::List_item( T value, List_item *item)
:m_value(value),m_pNext(item){}

template <class T>
T List_item<T>::GetValue() const { return m_value; }

template <class T>
void List_item<T>::SetValue(const T& et){ m_value = et; }

template <class T>
void List_item<T>::SetNext(List_item *link) { m_pNext = link; }

template <class T>
List_item<T>* List_item<T>::GetNext(){ return m_pNext; }
```

但是，在使用模板时，必须用 #include “Temp.cpp” 替换掉 #include “Temp.h”：

```
#include "Temp.cpp" // 不再是熟悉的头文件 Temp.h
int main()
{
    int maxInt = MAX(5,9);
    List_item<int> intList(5);
    return 0;
}
```

但是这样的做法“据说”并没有得到所有编译器的支持，不过 VC++ 和 Gcc 对此都支持

的（测试版本为 VC++2010、GCC 4.3.2）。另一方面，这种做法会引起重复编译，因而也会导致编译性能降低。

那么有没有那种“声明实现相分离，并且包含的还是头文件”的解决方法呢？有，这就是第三种方法：使用关键字“`export`”！

可以在.h文件中，声明模板类和模板函数，在.cpp文件中，使用关键字`export`来定义具体的模板类对象和模板函数。待其他用户代码文件中包含声明的头文件后，就可以使用这些对象和函数了。就像下面这样：

```
// output.h - 声明头文件
template<class T> void output(const T& t);

// out.cpp - 定义代码文件
#include <output.h>
export template<class T>
void output (const T& t) {std::cout<<t;}

//main.cpp: 用户代码文件
#include "output.h"
int main()
{
    output(4); // 使用 output()
    output("Hello");
    return 0;
}
```

这种方式十分符合本建议开始时所介绍的那种文件结构形式。但是，这里还有一个不得不说的问题：并非所有的编译器都支持`export`关键字。Comeau C/C++ 和 Intel 7.x 编译器支持此关键字，但号称“百分百支持 ISO”的VC++ 和 GCC 却对此视而不见。在最新的 VS 2010 中，`export`关键字被标蓝，这表示 IDE 认识它；但是不要高兴得太早，当按下 F7 进行编译时，它会抛出警告：

```
warning C4237: 目前还不支持“export”关键字，但已保留该关键字供将来使用。
```

至于为什么不支持，是因为它的性能太次。在模板编译过程中，编译器会像.NET 和 Java 那样，为模板实体生成一个“中间伪代码（IPC, intermediate pseudo-code）”，使其他翻译单元在实例化时可找到定义体；而在遇到实例化时，则根据指定的实参再将此 IPC 重新编译，从而达到“分离编译”的目的。这个过程受到了几乎所有知名编译器供应商的强烈抵制。所以，请不要使用`export`关键字，至少在高效的`export`出现之前不要使用。至于将来会怎么样，那就等到将来再讨论吧。

---

#### 请记住：

函数模板、类模板不同于一般的函数、类，它们不能像一般的方式那样进行声明与定

义，标准要求模板的实例化与定义体必须放在同一翻译单元中。实现这一目标有三种方法，但是最优策略还是：将模板的声明和定义都放置在同一个.h文件中，虽然在某种程度上这破坏了代码的优雅性。

## 建议 62：用模板替代参数化的宏函数

模板被看作是一种参数化的类或函数，这一点与参数化的宏函数有着异曲同工之妙。特别是在遇到如下的情况时。

两个数字比较大小是我们比较常用的计算，在此计算过程中，由于C++语言是强类型语言，所以会要求我们针对不同类型的参数定义不同类型的函数，用个专业的词来说就是针对不同的参数类型一一进行重载，如下所示：

```
int min( int a, int b){ return a < b ? a : b; }
float min(float a, float b){ return a<b ? a : b; }
double min( double a, double b ) { return a < b ? a : b; }
int max(int a, int b){ return a>b ? a : b; }
float max(float a, float b){ return a > b ? a : b; }
double max(double a, double b){ return a > b? a : b; }
```

这绝对是一个大工程量的体力活，我们不仅要为int、float、double、char等内置数据类型实现对应的函数，甚至还要针对许许多多的自定义类型进行重载。同时，还会出现大量的重复性代码。这都是我们所不能忍受的。

那有什么简便高效的解决方法吗？

这时我们可以尝试用预处理器的宏扩展设施来解决这个问题，于是就有了如下“参数化的宏函数”版本：

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

在C语言中，我们经常会这么做，而且这样的宏函数也非常好用。但是在C++语言中，它们并不像在C中那样受欢迎。因为C++对于这类可以抽象的算法提供了更好的办法，那就是模板，如下所示：

```
template <class T>
const T & min(const T & t1, const T & t2) { return t1>t2?t2:t1; }

template <class T>
const T & max(const T & t1, const T & t2) { return t1<t2?t2:t1; }
```

有了这样的模板，我们又获得了自由，可以像在C语言中使用min/max宏一样来使用这两个函数模板了。

抛弃 C 语言中参数化的宏函数而选择模板，并不是我们 C++ 程序员喜新厌旧，而是与模板相比，参数化的宏函数有着两个致命缺点：

其一，缺乏类型检查，所以它天生就是不安全的。当不同类型的参数进行比较时，有可能会引起错误。即使参数类型相同，也可能会导致代码与意愿南辕北辙，假如写出下面的代码片段：

```
int a = 10, b = 10;
int c = min(a++, b--); // a = ? b = ? c = ?
```

a 和 b 的值将不再是我们所期望的 11 和 9，分别变成了 11 和 8，而 c 的值更是出乎意料地变成了 9。这与我们的预想相差甚大。

其二，有可能在不该进行宏替换的时候进行了替换，违背了作者的意图。假如，我们在某一个类中定义了一个与宏函数同名的函数：

```
class Demo
{
public:
    float min(float x, float y);
    int max(int x, int y);
    //other code
}
```

此时，宏定义会将 min() 函数和 max() 函数的声明进行代码替换，这会引发编译错误。

所以，我们需要像建议 25 所说的那样，尽量抛弃宏！针对那些类似 min/max 的可以抽象的算法，我们就用更加安全的模板来代替参数化的宏函数。请时刻记住：模板是实现代码复用的一种工具，它可以实现类型参数化，达到让代码真正复用的目的。

---

#### 请记住：

模板相较于参数化的宏函数具有相当的优势，所以用模板去代替参数化的宏函数。

---

## 建议 63：区分函数模板与模板函数、类模板与模板类

在使用模板时，我们会频繁地遇到这么几个术语：

**函数模板、模板函数、类模板、模板类**

关于这几个术语，因为它们太过相似，所以我们经常会误用。接下来，我们的任务就是彻底辨清这几个术语，以避免概念上的混淆和使用上的错误。

在现代汉语中，我们一般习惯于将词的重点放在后面，这也影响了 C++ 术语的汉化，上述这几个术语就是证明。接下来一一分析之。

### 函数模板 VS. 模板函数

函数模板的重点在于“模板”两个字，前面的“函数”只是一个修饰词。其表示的是一个专门用来生产函数的模板。而模板函数重点在“函数”，表示的是用模板所生成的函数。

函数模板的一般定义形式为：

```
template<class 数据类型参数标识符>
返回类型标识符 函数名(数据类型参数标识符 形参)
{
    //.....
}
```

将函数模板的模板参数实例化后会生成具体的函数，此函数就是模板函数。由函数模板所生成的模板函数的一般形式为：

```
函数名<数据类型参数标识符>(数据类型参数标识符 形参)
```

这里的数据类型参数标识符所对应的是对象实际需要的数据类型。

就像下面的这个例子，我们定义了一个模板 Function，它可以根据不同的参数来生成特定的函数：

```
template <typename T>
void Function(const T & a)
{
    ...
}
```

在运用时，我们可以根据不同的参数类型生产出多样的模板函数，如：Function<int>、Function <double>、Function <UserFefinedClass\*> 等。

### 类模板 VS. 模板类

类模板是为类定义的一种模式，它使类中的一些数据成员和成员函数的参数或返回值可以取任意的数据类型。类模板的一般定义形式为：

```
template<class 数据类型参数标识符>
class 类名
{
    //.....
}
```

在类定义中，凡是采用标准数据类型的数据成员、成员函数的参数前面都要加上类型标识符，在返回类型前也要进行同样的处理。如果类中的成员函数要在类的声明之外定义，则它必须是模板函数。其定义形式为：

```
template<class 数据类型参数标识符>
数据类型参数标识符 类名<数据类型参数标识符>::函数名(数据类型参数标识符 形参1, ..., 数据类型参数标识符 形参n)
```

```
{
    函数体
}
```

将类模板的模板参数实例化后生成的具体类，就是模板类。模板类的一般形式为：

类名 < 数据类型参数标识符 > 对象名 1, 对象名 2, ..., 对象名 n;

例如，我们定义了一个类模板：

```
template <class T>
class List_item
{
public:
    List_item( T value, List_item *item_to_link_to = NULL ) {}
    T GetValue() const { return m_value; }
    void SetValue(const T& et){ m_value = et; }
    void SetNext(List_item *link) { m_pNext = link; }
    List_item* GetNext(){ return m_pNext; }

private:
    T             m_value;
    List_item *m_pNext;
};
```

对应的模板类就可能是：

```
List_item<int> list1(5);
List_item<float> list2(5.0f);
List_item< UserFefinedClass > list2(UserFefinedObj);
```

由此可见，函数模板和类模板处于实例化之前，而模板函数或模板类则在实例化之后，用户可使用函数模板和类模板来构造模板函数或模板类。函数模板和类模板是工具，是模子，而模板函数或模板类是对象，是产品。

**请记住：**

函数模板和模板函数、类模板和模板类它们是面貌相似的孪生兄弟，要区分它们，就要深谙中国语言的博大与精深：语言的重心在后面，前面的词是作为形容词使用的。

## 建议 64：区分继承与模板

inheritance 继承是面向对象的重要概念，而 templates 模板则是解决部分问题的最佳方式。继承和模板看似毫不相干的两个概念，其实也有一个十分微妙的共同点：它们面对的都是“很多类”。

继承描述的是多种类型具有相同的性质，特别是公开继承，比如猫是一种动物，老虎也

是一种动物，它们都会“叫”。这样的描述如果转化为 C++ 语言，就是如下的代码片段：

```
class CAnimal
{
public:
    virtual void Howl() = 0;
};

class CCat : public CAnimal
{
public:
    virtual void Howl()
    {
        cout<<" 喵 ~~ 喵 ~~" << endl;
    }
};

class CTiger : public CAnimal
{
public:
    virtual void Howl()
    {
        cout<<" 啊呜 ~~" << endl;
    }
};
```

这是最常见、也是最理所应当的一种处理方式。那么用模板是否也可以处理呢？如果仅仅是针对“叫”这一种共同的行为，也许可以行得通。我们尝试着写出如下的代码：

```
class CCat
{
public:
    void Howl()
    {
        cout<<" 喵 ~~ 喵 ~~" << endl;
    }
};

class CTiger
{
public:
    void Howl()
    {
        cout<<" 啊呜 ~~" << endl;
    }
};

template<class CAnimal>
void AnimalHowl(const CAnimal& obj)
{
    obj.Howl();
}
```

不出所料，编译通过，运行正确。

也许有人已经注意到了上文中的“仅仅是”三个字。之所以这么说，是因为模板针对的是不同类型所具有的相同类型的操作。如果在猫与虎的例子中，它们具有了其他的属性，比如年龄与性别。上述的 CAnimal 类就应该被写成如下的形式：

```
enum GENDER
{
    GENDER_MALE = 0,
    GENDER_FEMALE,
};

class CAnimal
{
public:
    virtual void Howl() = 0;
    .....

private:
    unsigned char m_nAge;
    GENDER m_eGender;
};
```

如果此时还想尝试着用模板去解决，难度会陡增，这绝非明智之举。因为模板没有较好的办法来处理类中的这些属性。即使没有这些干扰因素，用函数模板 AnimalHowl 来解决这一问题仍然也是漏洞百出，很不合理，因为不同动物的 Howl 这一行为根本就不同；在上面的函数模板中，我们只是“巧妙”地借用了它们同样的接口 Howl() 来实现这一行为表面层次上的相同。所以，我们要为这些不同的类去实现各种各样的 Howl()。既然要为所有的类都实现 Howl() 这一行为，那函数模板 AnimalHowl 也就失去了它存在的最大意义。

模板的长处在于处理不同类型间“千篇一律”的操作。相较于类继承，这些类不必具有什么相同的性质。我们还是以那个传统而又熟悉的 Stack 来举例：

```
template<typename T>
class CStack
{
public:
    CStack() : m_pTop(NULL) { };
    ~CStack() { ... }
    void Push(const T &value) { ... }
    T Pop() { ... }
    inline bool Empty() const { return m_pTop == NULL; }
private:
    CStack(const CStack &rhs);
    CStack operator=(const CStack &rhs);

private:
    struct Node
    {
```

```

    T data;
    Node *next;
    Node(const T& value, Node *nextNode)
        : data(value), next(nextNode){ }
    };
    Node *m_pTop;
};

```

实现了这样的类模板，我们就可以从容地处理不同类型数据的 FILO (First In Last Out, 先进后出) 行为了，不管是 char、int、float、double 等内置数据类型，还是像 CCat、CTiger 这样的用户自定义类型都可以。

不过，上面这个问题可以用继承来解决么？我们继续尝试：

```

class CStack
{
public:
    virtual bool Empty() const = 0;
private:
    CStack(const CStack &rhs);
    CStack operator=(const CStack &rhs);
};

class CIntStack : public CStack
{
public:
    CIntStack() : m_pTop(NULL){ };
    ~CIntStack(){ ... }
    void Push(int value){... }
    int Pop() { ... }
    inline bool Empty() const{ return m_pTop == NULL; }

private:
    struct Node
    {
        int data;
        Node *next;
        Node(const int value, Node *nextNode)
            : data(value), next(nextNode){ }
    };
    Node *m_pTop;
};

class CCatStack : public CStack
{
public:
    CCatStack() : m_pTop(NULL){ };
    ~CCatStack(){ ... }
    void Push(const CCat& value){ ... }
    CCat Pop() { ... }
}

```

```
inline bool Empty() const{ return m_pTop == NULL; }

private:
    struct Node
    {
        CCat data;
        Node *next;
        Node(const CCat& value, Node *nextNode)
            : data(value), next(nextNode){ }
    };
    Node *m_pTop;
};
```

上述设计虽然勉强可以工作，但是这样的设计确实是不折不扣、地地道道、纯纯粹粹的失败设计：我们不仅要针对每种类型分别实现一模一样的操作，费时费力，而且这还会给我们带来大量重复性的代码，降低了代码的简洁性。所以不能仅仅因为有相同的行为就贸然地采用继承。

由此可见，虽然继承和模板都针对“很多类”，但是其中还是有很大的不同的。我们要具体情况具体讨论，因地制宜，采用合理的方法，优化我们的设计。

---

#### 请记住：

继承和模板具有很微妙的共同点，有时有些问题两种方式都能达到目的。但是我们必须知道：能达到目的的设计并非一定是好设计。所以需要我们就事论事，因地制宜，优化设计。

---

# 第 6 章 让神秘的异常处理不再神秘

任何程序都会出现异常。对于那些可以预料的错误，在程序设计时，我们应当制定相应的预防策略，以便防止异常发生后造成严重的后果。一个好的应用程序，不仅要保证在用户正确操作时，运行正常、正确，更应该具有一定的容错能力，在应用环境出现意外或用户操作不当时，也应有合理的反应。异常处理对于编写健壮的软件来说非常重要，是否有完善的异常处理机制也是评价某一程序设计语言优劣的一个重要标准。

在 C++ 中，异常处理仿佛被披上了一层神秘的面纱，让人费解而又好奇。然而，其思想却是那么的简单且朴素：在底层发生的问题，逐级上报，直到某一级有能力处理这一问题为止，这就跟现实社会中我们处理事情一样。

默念着异常处理的朴素思想，揭开隐藏在异常处理中的秘密，让神秘的异常处理不再神秘。

## 建议 65：使用 exception 来处理错误

在编写计算机代码的过程中，错误绝对是不可避免的。那么遇到异常我们该怎么处理呢？也许，有人会使用 return value（返回值），甚至是直接中断程序的运行。这也是我们最为熟悉的处理方式，特别是对于那些从 C 时代走过来的人来说。

### □ 中断程序运行

这种方式一般会使用标准 C 库提供的 abort() 或 exit() 两个函数来强行终止程序的运行，简单粗暴。例如在下面的代码片段所示的例子中，当除数为零时，停止运行并给出提示信息：

```
double Divide(double x, double y)
{
    if(y==0)
    {
        cout<<"Error: Dividing Zero.\n";
        exit(1);
    }
    return x/y;
}
int main()
{
    double result = 0.0;
    result = Divide(2011,0);
    ...
    return 0;
}
```

这种方式适用于一般的小型应用程序。如果出现异常，程序立马停止运行，无条件释放资源。采用这样的处理方式时，一旦出现异常，我们必须一切都从头开始。

对于中大型程序来说这绝对是不可接受和允许的。因为在大中型程序中，函数之间有着明确的分工和复杂的调用关系，出现错误的程序往往处于函数调用链的低层。如果采用上述简单粗暴的处理方式，就没有机会把调用链中上层函数已经完成的一些工作做善后处理了。例如，如果上层函数已经申请了堆对象，那么堆对象就不能正常释放，就会造成内存泄露。

#### □ 返回值 / 错误码方式

这种方式用函数的返回值来标志函数是否执行成功。就像我们无比熟悉的 main 函数，正常退出时返回 0，否则返回其他值。这种方式继承自 C 语言，它的好处就是简单方便，而且具有较高的效率。但是它也并非完美无瑕、无懈可击的，它仍然存在如下问题：

(1) 影响代码可读性：如果每个函数都有这样的返回值，为了保证程序的正确运行，我们必须对每个函数都进行正确性验证，并且在调用函数的时候对其返回值逐一进行检查，这样程序代码中很大一部分就可能要花费在错误处理上了。

(2) 隐藏潜在的问题：采用这样的方式处理异常时，它不会强制处理错误，而且在很多情况下即使不进行异常处理，程序仍然能够运行，但运行结果是不可预知的。

除了这两种异常处理方式外，我们还可以使用 assert（断言）宏、errno 全局变量、goto 出错跳转，或者 setjmp/longjmp 等方式。但是在这些旧时代的方法中，无论选择哪一种方式来处理异常都不可能达到“代码稳健鲁棒”这一目标。另外，除了要达到保证软件正确性这一初级目标外，我们还希望软件自身具备一定的容错能力，当它在面对千奇百怪的错误时能做出合理的反应，提高程序的健壮性。

事实上，C++ 设计者已经为我们考虑到了这一点，所以为其建立了比较完善的异常处理机制。C++ 异常处理机制是一个用来有效地处理运行错误的非常强大且灵活的工具，相较于传统的方法，它提供了更多的弹性、安全性和稳固性，克服了传统方法所带来的问题。

C++ 异常处理的基本思想是简化程序的错误代码，为程序的健壮性提供一个标准检测机制：若底层发生问题，则逐级上报，直到有能力处理此异常为止。也就是说在应用程序中，若某个函数发现了错误并引发异常，这个函数就将沿着函数调用链将该异常向上级调用者依次传递，请求调用者捕获该异常并处理该错误。如果调用者不能处理该错误，就继续向上级调用者传递，直到异常被捕获、错误被处理为止。如果程序最终没有相应的代码处理该异常，那么该异常最后会被 C++ 系统所接受，C++ 系统就简单地终止程序的运行，如图 6-1 所示。

那么，异常到底能给我们带来什么呢？

#### (1) 增强程序的健壮性。

如果系统某个地方 throw 异常，程序员必须在某个地方 catch 它并进行处理：处理错误或重新 throw。这种处理是强制的，如不处理，程序就会崩溃。如果采用错误码来处理错误，错误的返回值是可以被忽略的，不过这个错误可能会导致系统的运行处于无定义状态，在某

一时刻也许会导致系统崩溃。同时，异常还是处理构造函数失败的唯一方式。

### (2) 使代码变得更简洁优美，更易维护。

异常处理是在某个地方 `throw` 异常，然后又在某个地方 `catch`。在多层调用关系中，异常可以从内层嵌套中直接跳出，将错误处理集中化，这样就可以使出错处理程序与“一般”的代码分离开来，让代码更简洁而灵活。如果采用返回错误码的方式，必须在每个调用处处理返回值，这会增加许多重复的代码。就像《Refactoring：Improving the Design of Existing Code》(《重构：改善既有代码的设计》)一书中所讲的那样：

Unix 和基于 C 的系统通常会用一个返回码来标示程序成功还是失败。Java 有一个更好的处理方式，那就是异常。之所以异常更好，是因为它能够将错误处理过程从通常的处理过程中剥离出来。这使得程序更加简单易懂，我希望你也能够认同“可理解性近于美德”这句话。⊕

虽然书中说的是 Java，但这一条对于 C++ 来说也绝对适用。

### (3) 错误信息更灵活、丰富。

异常被设计成了一个特殊的对象，具有继承的特征。所以我们可以通过继承实现多样的异常类，让异常保存更多更详细的错误信息，而不是像错误码那样简单而含糊。

但是异常也并非十全十美的。作为 C++ 的一个高级特性，异常处理会带来一定的开销。据有关统计表明：使用异常，运行期间在不发生异常的情况下，性能可能会下降 5% ~ 14%。其次，虽然异常处理能将异常处理代码与一般的普通代码分离开来，增强了代码可读性，但是同时它也破坏了程序的结构性，增加了代码管理和调试的难度。因为函数返回点可能在意料之外，光凭查看代码是很难评估程序控制流的。再次，要想轻松编写正确的异常安全代码，需要大量的支撑机制配合，所以使用异常就意味着要付出更多的代价。

综上所述，为了增强代码的健壮性，用异常代替错误码，采用异常来处理 C++ 错误，抛弃旧时代，迎接新未来！

---

### 请记住：

C++ 从 C 中继承了一些古老的错误处理方式，尽管在某些地方某些时刻这些方式还依然

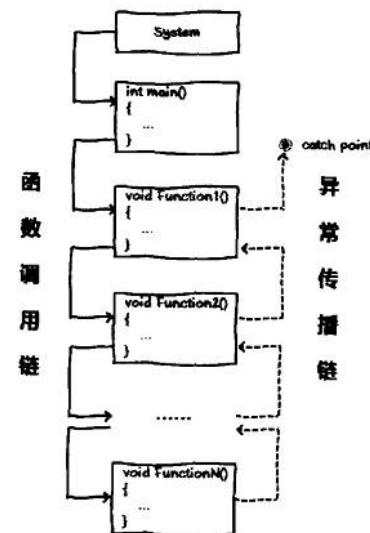


图 6-1 函数调用与异常传播示意图

⊕ Unix and C-based systems traditionally use a return code to signal success or failure of a routine. Java has a better way: exceptions. Exceptions are better because they clearly separate normal processing from error processing. This makes programs easier to understand, and as I hope you now believe, understandability is next to godliness.

有效，但它们已不是C++时代的最优选择。C++的异常处理能够克服传统方式的一些缺点，增强代码健壮性，所以用异常代替错误码。

## 建议 66：传值 throw 异常，传引用 catch 异常

从语法的角度来看，函数参数声明与catch子句中的参数声明具有很大的相似性，它们的参数都可以是值类型（value）、引用类型（reference）和指针类型（pointer）中的任何一种。正如下面的代码片段所示：

```
// 函数参数声明
void Function(CString str);    // by value
void Function(CString& str);   // by reference
void Function(CString* pStr);  // by pointer

// 异常参数声明
try
{
    .....
}
catch(exception value_e) // by value
{
}
catch(exception& reference_e) // by reference
{
}
catch(exception* pointer_e) // by pointer
{}
```

既然有这三种方式，我们应该采用哪一种呢？为什么要采用那一种呢？这就是接下来需要我们探讨的问题。

首先讨论一下通过指针捕获异常（catch by pointer）的方式。从指针的特点来看，通过指针捕获异常在理论上应该是效率最高的。这一点在函数参数的传递上已经得到了很好的证明。通过指针捕获异常时，异常可能存在以下几种抛出方式。

(1) 抛出局部对象指针，代码如下所示：

```
try
{
    exception e;
    ...
    throw &e; // throw a pointer
}
catch(exception* pointer_e) // by pointer
```

```
{
...
}
```

抛出异常，跳出 try 块后，局部对象被编译器销毁，抛出的指针成了名副其实的野指针。野指针的危险性不用多言！所以这种方式不可取。

(2) 抛出全局对象或静态对象的指针，代码如下所示：

```
exception g_Except;

try
{
...
    throw &g_Except; // throw a pointer
}
catch(exception* pointer_e) // by pointer
{
...
}
```

这种方式能够保证程序的正确性，不会生成野指针。但是，这需要程序员付出额外的努力：要为所有的异常类型声明一个全局（或静态）变量，保证进入 catch 块后，其所获得的指针是正确且有效的。可见这种方式也不甚合理！

(3) 抛出指向动态分配内存的指针，代码如下所示：

```
try
{
    exception* pExcept = new exception();
...
    throw pExcept; // throw a pointer
}
catch(exception* pointer_e) // by pointer
{
...
}
```

这种方式看似更合理，它既不会像抛出局部对象指针那样出现程序错误，也不用像抛出全局对象指针那样需要程序员付出额外的辛苦。但是如果真的抛出指针，难免会涉及让我们爱恨交加的内存管理问题。当成功地抛出指针时，内存释放的重担也就随之传递到了异常捕获的地方，我们要手动负责内存的清理，更加可恶的是如果遇到如下情况：

```
try
{
    Function(); // throw a pointer
}
catch(exception* pointer_e) // by pointer
{
```

```
    ...
}
```

因为我们看不到 Function() 函数的具体实现，无法确定它所抛出的指针是指向全局（或静态）对象的，还是指向动态内存空间的。所以我们会左右为难，不知道该不该执行清理操作。

看起来通过指针捕获异常是我们应当避免采用的操作方式。

那么传值呢？

来看下面的例子：

```
try
{
    exception e
    ...
    throw e; // throw a value
}
catch(exception value_e) // by value
{
    ...
}
```

此时，我们无须考虑异常对象的清理，因为编译器本身会处理这一复杂的过程。但是它又有效率上的缺陷：按值传递过来的异常对象 value\_e 的作用域在 catch 块内；当抛出 e 时，编译器会调用复制构造函数，复制局部变量到临时对象 1 中；在通过 E value\_e 传递进 catch 块时，复制构造函数再次启动，复制临时对象 1 到临时对象 2 中。由此可见，catch by value 因为临时对象的构造与析构，会引起效率的降低。这与函数参数传值极为相似，这种方式也应抛弃。

如果认为我们抛弃 catch by value 这一方式仅仅是为了效率，那就大错特错了。因为在 C++ 中异常也是对象，也可以继承派生，呈现多态。传值会不经考虑地截断数据，这种粗暴的处理方式会影响异常对象的多态性。

最终，历史的重任落到了引用（reference）的身上：

```
try
{
    exception e
    ...
    throw e; // throw a value
}
catch(exception& reference_e) // by reference
{
    ...
}
```

**catch by reference** 虽然在效率上不及指针，但是与指针相比它不涉及对象清理的问题，同时相较于按值传递又减少了一次异常对象的复制构造与析构，而且没有对象的切割问题，能够保证异常对象的多态性。

下面对上述几种方式进行全面盘点和总结（如表 6-1 所示）。

表 6-1 异常传递方式对比

	按值传递	引用传递	指针传递
捕获语法	catch (exception e)	catch (exception* e)	catch(exception& e)
抛出方式	throw exception(); exception e; throw e; throw global_e;	throw exception(); exception e; throw e; throw global_e;	throw new exception(); throw & global_e;
效率	低（三次构造析构）	中（两次构造析构）	高（一次构造析构）
异常对象销毁时机	局部对象离开作用域时自动销毁，临时变量在 catch 块执行完后自动销毁		在 catch 块执行完后，需要动态的销毁
安全性能	可能发生对象切片，安全性能较低	相对较好	可能发生内存泄露，甚至是程序崩溃，安全性能较低
综合测评	59（差）	85（良好）	61（一般）

由此可见，“**throw by value, catch by reference**”是我们在处理异常时应该采用的标准形式。当然，为了让代码更加安全健壮，我们通常会将 **const** 关键字应用到这里：

```
try
{
    ...
    throw exception(); // throw a value
}
catch(const exception& reference_e) // by const reference
{
    ...
}
```

#### 请记住：

虽然传值与传指针都有一定的优势，但是其劣势也同样明显。所以建议采用综合性能最好的传引用方式：**throw by value, catch by reference**，即

```
try{
    ...
    throw exception(); // throw a value
}
catch(const exception& reference_e) { // by const reference
    ...
}
```

## 建议 67：用“throw;”来重新抛出异常

当异常被 catch 时，控制权便到了异常处理块中；它通过异常对象得到异常信息，并进行相应的异常处理。但是不要以为有了异常处理，我们就万无一失了。这个 catch 块并不一定有能力解决这个问题。如果它处理不了，那该怎么办呢？C++ 语法规定：如果当前的异常处理程序块不能够及时地处理这个异常，那就必须重新抛出异常（exception rethrow），把这个异常交给上一层函数的异常处理模块去解决。就像在现实工作中，我们遇到了处理不了的问题，就将它报告给上层领导，由他们来决定如何处理一样。

按照这个思路，写出了如下的代码片段：

```
int main()
{
    try
    {
        try
        {
            throw 2012;
        }
        catch(int value)
        {
            // respond (partially) to exception
            cout << "I am not able to stop 2012" << endl;
            // translate this responsibility to others
            throw value;
        }
    }
    catch(...)
    {
        cout << "God help you! Believe in God!" << endl;
    }

    return 0;
}
```

在第一个 catch 块内，异常处理代码不能够解决异常 2012 所带来的问题，所以就原封不动的将 value 重新抛出了。这一切看起来是那么地合情合理。

但是不要忘记异常对象也具有继承的能力，如果在异常传递的过程中，派生类型的对象通过基类型参被 catch，然后再被按照基类的形式 rethrow 至上层，那肯定就会出现“对象切片”的问题，这会导致异常对象信息的部分丢失。就像接下来的例子那样：

```
class exception_1 : public exception { ... };

try
{
    try
```

```

{
    ... // do other things
    throw exception_1();
}
catch(const exception& e)
{
    cout<<"I can not handle this exception"<<endl;
    throw e;
}
}
catch(...)
{
    ... // handling exception
    cout<<"I can handle this exception"<<endl;
}

```

当异常处理块对该异常表示无能为力时，我们抛出了静态形参类型对象 e，而异常的实际类型却是 exception\_1。抛出的对象 e 不再是原来的那个对象 exception\_1()（临时对象），而是当前捕获对象的一个拷贝，新的异常对象在不知不觉中被强行转换为了基类对象，异常对象也就被迫丢失了部分信息。

除了上述差异外，以下两种方式在效率方面也有所不同：

```

// 方式1
catch (const exception& w) // 捕获exception异常
{
    ... // 处理异常
    throw; // 重新抛出异常
}
// 方式2
catch (const exception& w) // 捕获exception异常
{
    ... // 处理异常
    throw w; // 重新抛出异常
}

```

第一个 catch 块中重新抛出的是当前捕获的异常，而第二个 catch 块中重新抛出的是当前捕获异常的一个新的拷贝。因为拷贝动作带来了异常对象的构造与析构，所以第二种方式还会带来额外的系统开销，其效率更低。

为了避免出现这些问题，在重新抛出异常时，不要采用上述的形式。请使用异常重新抛出的标准语法形式<sup>⊖</sup>：

```

try
{
    // ...

```

---

<sup>⊖</sup> 请见 C++ Standard(ISO/IEC) 的 15.1 “Throwing an exception”，P300

```

}
catch (...) // catch all exceptions
{
    ... // respond (partially) to exception
    throw; //pass the exception to some other handler
}

```

虽然重新抛出时，我们没有指定具体的异常，但仍然将一个异常对象沿链向上进行了传递，并且被抛出的异常依旧是原来的异常对象，不会受到 catch 形参的限制。所以，当 catch 形参是基类异常类型时，重新抛出的异常的实际类型既可以是该基类类型，也可以是该基类的派生类型。

对于异常的重新抛出，还有两点需要特别注意：

- (1) 重新抛出的异常对象只能出现在 catch 块或 catch 调用的函数中。
- (2) 如果在处理代码不执行时碰到“throw ;”语句，将会调用 terminate 函数。

#### 请记住：

在重新抛出异常对象时，请注意它的特殊语法形式：

```

try
{
    ...
}
catch (...) // catch all exceptions
{
    ... // respond (partially) to exception
    throw; //pass the exception to some other handler
}

```

## 建议 68：了解异常捕获与函数参数传递之间的差异

在建议 66 中，我们讨论了异常抛出与捕获时应该采用的方法。当我们写下异常捕获语句时，细心的读者可能会发现：catch 子句中的参数声明与我们所熟悉的函数参数的声明很相似：

```

class exception { ... }; //一个具体的异常类
//值
catch (exception e) ...
void Function1(exception e);
//引用
catch (exception& e) ...
void Function2(exception& e);
//const 引用
catch (const exception& e) ...
void Function3(const exception& e);

```

```
// 指针
catch (Widget *pe) ...
void Function4(exception* pe);
//const 指针
catch (const exception* pe) ...
void Function5(const exception* pe);
```

虽然异常 catch 子句和函数参数的传递方式相似，可以是传值、传引用、传值针三种方式中的任何一种，但是这并不能掩盖其背后的巨大差异。差异在什么地方呢？下面我们通过以下几个方面来一一进行分析。

#### □ 控制权

一般来说，函数调用时，控制权最后会返回给调用者，而当异常抛出的时候，控制权就不再返回了，如图 6-2 所示。

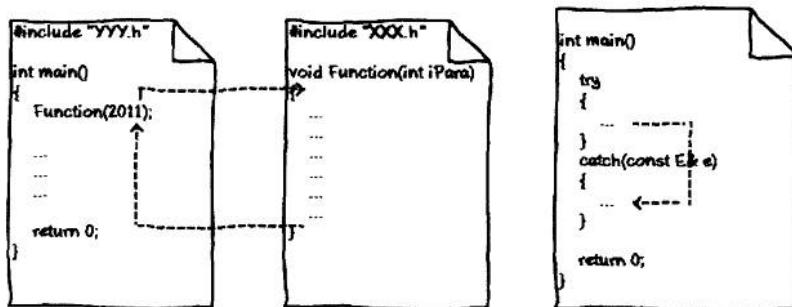


图 6-2 控制权转移示意图

调用 Function() 函数时，控制权会由 main 函数转移到 Function() 中，并执行 Function 函数；当函数执行完毕，顺利返回时，控制权会再次回到 main 函数中；而当异常抛出时，控制权就会和这异常一起进入到 catch 块内，永不返回。

#### □ 对象拷贝次数

异常对象在传递过程中总是被拷贝的，无论是传值，还是传引用。比如下面的示例代码片段：

```
try
{
    ... // do other things
    throw exception();
}
catch (const exception& e)
{
    ... // handling exception
}
```

exception() 生成的是一个临时对象 tempObj1，当抛出异常时，编译器会偷偷地调用拷贝

构造函数，生成临时对象 tempObj2；因为当控制权离开 try 块时，tempObj1 也就走到了生命的尽头，它会被编译器强行析构，从计算世界里消失。于是传递异常信息的重担就落到了临时对象 tempObj2 的肩上。tempObj2 会被 catch 截获，然后进行相应的处理。而对象作为参数传递给函数时却不需要额外的拷贝行为，如下所示：

```
void Function1(const exception& e){ ... }

exception e;
Function(e);
```

在上面的函数调用过程中，异常对象 exception 的构造函数只是调用了一次，Function 函数操纵的就是这一对象本体。与异常传递相比，其拷贝动作减少了，所以相较而言，参数传递更具效率。

#### □ 类型转换

对象作为异常被抛出与作为参数传递给函数相比，前者的类型转换比后者要少。异常与 catch 异常说明符匹配的规则要比参数传递时实参和形参类型的匹配规则更严格。当 catch 语句针对异常进行类型匹配时，只有以下两种类型的转换可能会发生：

(1) 允许从一种类型转换到无类型指针。所以，以下两种 catch 方式具有异曲同工之妙：前者可以捕获任何异常类型的指针，而后者则可以捕获任何异常类型的值。

```
try{
    ... // do other things
}
catch(const void* e){ // catch all exception
    ... // handle this exception
}

try{
    ... // do other things
}
catch(...){ // catch all exception
    ... // handle this exception
}
```

(2) 允许基于继承的类型转换，如下所示：

```
class exception_1 : public exception { };
class exception_2 : public exception { };
class exception_1_1 : public exception_1 { };

try
{
    ... // do other things
    throw exception_1();
}
```

```
catch(const exception& e)
{
    ... // handle this exception
}
```

用于捕获基类类型的 catch 语句可以捕获这个类体系派生出来的任何类型的异常。在上面的例子中无论抛出的是基类 exception 类型的异常，还是派生类 exception\_1、exception\_2 类型的异常，甚至是派生类的派生类类型 exception\_1 的异常，catch 语句都能将其捕获。

而在函数参数中允许的标准算术转换、类型定义的转换在 catch 类型匹配的时候都是不允许的。

#### □ 类型匹配

catch 子句进行异常类型匹配的顺序与函数类型匹配具有很大的差异，它们采取的是两种截然不同的匹配策略，前者是“最先匹配”，后者是“最优匹配”。换句话说就是，前者的匹配依赖于它们在源代码中出现的顺序，第一个类型匹配成功的 catch 将被执行。而当一个对象调用一个函数时，被选择的函数是函数类型匹配最佳的那个，而不用考虑其所在的位置。

所以，在写异常 catch 语句时应该倍加小心，须对其顺序做慎重考虑：派生类的异常捕获要放到基类异常捕获之前，catch(...) 要放到所有类型异常捕获之后，千万不要写出这样的代码：

```
class exception_1 { };
class exception_2 { };

try
{
    ... // do other things, may throw exception
}
catch(...)
{
    ... // handle this exception
}
catch(const exception_1& e)
{
    ... // handle this exception
}
catch(const exception_2& e)
{
    ... // handle this exception
}
```

上面的代码中最先匹配的是 catch(...), 所以针对 exception\_1 和 exception\_2 的异常处理块永远不会执行到。要纠正此问题，我们要做的仅仅是调整一下它们的顺序：

```
try
{
```

```

    ... // do other things, may throw exception
}
catch(const exception_1& e)
{
    ... // handle this exception
}
catch(const exception_2& e)
{
    ... // handle this exception
}
catch(...)
{
    ... // handle this exception
}

```

#### 请记住：

异常与函数参数的传递存在着一定的相似性，但是不要将它们混为一谈。它们在很多方面都存在着一定的差异，比如控制权、对象拷贝的次数、异常类型转换及异常类型匹配等地方。

## 建议 69：熟悉异常处理的代价

异常处理的引入使我们的代码更加健壮，为我们的工作带来了便利。但是，天下没有免费的午餐，在享受便利的同时，我们同样要为异常处理付出一定的代价。为了能够成功地实现异常处理，编译器必须要引入一些额外的数据结构和相应的处理机制，而这些工作都是需要时间和空间的。

首先，异常处理机制需要记录额外的数据。针对每一个 C++ 函数，编译器需要记录相应的函数调用链，以便在函数调用栈中逐级向上寻找匹配的 catch 块；同时，还需要记录 try 块表<sup>Θ</sup>和栈回退表，以便完成异常捕获和栈回退动作，确保在异常被抛出、捕获并处理后，所有生命期已结束的对象都会被正确的析构，它们所占用的空间也会被正确地回收，如图 6-3 所示。

这些代价一般说来是不可避免的，即使代码中没有出现 try、throw、catch 关键字。

除了记录这些数据所带来的空间开销外，在程序执行的过程中，如果上述数据结构没有及时更新，那么它们也是没有意义的，所以我们需要消耗 CPU 时间来对这些数据进行更新。

另外，异常抛出与捕获也会带来相应的代价，而且这也是问题的关键所在。当出现异常时，C++ 异常处理器需要检查发生异常的位置是否在当前函数的某个 try 块之内，如果在当

<sup>Θ</sup> 记录与之对应的 catch 块及 catch 块所处理的异常类型的表。

前函数中的某个 try 块内，那么就需要找出与该 try 块配套的 catch 块的相关信息。如果不在该函数内，则需要沿着函数调用链逆序查找。与此同时，栈回退操作也要沿着异常处理链逐层向上进行。虽然这里所带来的开销相对较大，但是由于异常发生的几率相对较小，所以它对程序总体运行效率的影响也是有限的。

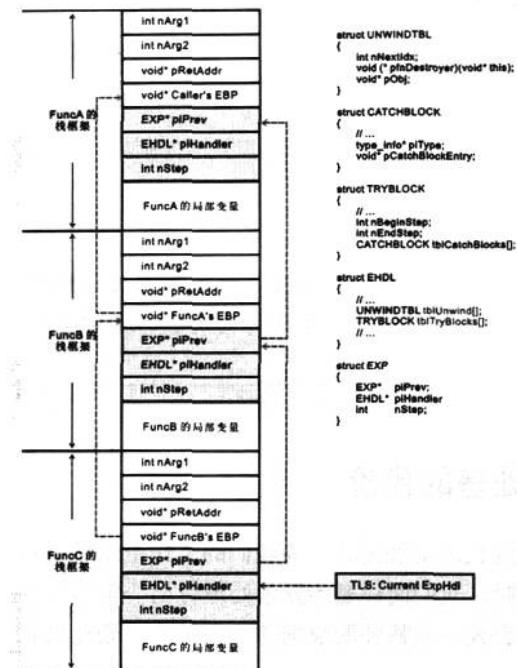


图 6-3 C++ 函数调用栈框架示例<sup>⊖</sup>

一般来说，相较于不支持异常的程序，支持异常的程序运行速度相对稍慢，程序的尺寸也稍大。以下面的代码片段为例：

```

// version 1: 某用 C++ 异常处理
#include <exception>
int main()
{
    int* pData = NULL;
    for(int i=0; i<1024; i++)
    {
        pData = new int[1024];
        delete pData;
    }
    return 0;
}

```

<sup>⊖</sup> 此图引自白杨老师的文章《C++ 异常机制的实现方式和开销分析》，详见 <http://baiy.cn>。

```

}

// version 2: 启用 C++ 异常处理
#include <exception>
int main()
{
    int* pData = NULL;
    for(int i=0; i<1024; i++)
    {
        try
        {
            pData = new int[1024];
        }
        catch(const std::bad_alloc& e)
        {
            return -1;
        }
        delete pData;
    }
    return 0;
}

```

VS2010 允许我们决定是否选择支持异常。我针对上述两个版本在 VS2010 编程环境中进行了试验，得到了如表 6-2 所示的结果<sup>⊖</sup>：

表 6-2 采用异常处理对程序的影响

	禁用 C++ 异常	启用 C++ 异常
程序执行时间（秒）	0.01018026	0.01037468
生成可执行代码大小（字节）	6656	7168

由此可以看出，如果在程序中没有用到 try、throw、catch，没有使用异常处理，在程序编译期间选择去掉对异常处理的支持也是一种优化方式。但是这里有一个前提，就是编译器允许我们去控制是否支持异常处理。

在实际应用中，异常处理带来的主要代价并非上述运行时的代价。不要惊讶，这是真的。它所带来的代价如下所示<sup>⊕</sup>：

首先，如果在现有函数中添加了 throw 语句，那么必须检查所有的调用点。所有的调用点至少得有基本的异常安全保护，否则会导致异常永远捕获不到。

其次，如果使用异常，代码评估难度会陡然增加，仅凭查看代码是很难明确程序的控制流的，函数返回点很可能出现在我们意料之外的地方，这会导致代码的管理和调试难度加大。

再次，异常安全要求同时采用 RAII 和其他不同的编程实践。如果想轻松编写正确的异常安全代码，需要有大量的支撑机制配合。允许使用异常会驱使我们不断地为此付出代价：

⊖ 程序执行时间是在 Debug 模式下获得的。

⊕ 摘自《google style C++ code》。

使生成的二进制文件体积变大，延长了编译时间，还可能增加地址空间压力。

最后，还要考虑与原有代码的整合问题。对于现有代码来说，引入异常会牵连到所有的相关代码。如果新项目允许异常向外扩散，在跟以前未使用异常的代码整合时那将是个麻烦。

正是鉴于这几方面的原因，Google 说，我们不使用 C++ 异常。

综上所述，C++ 异常也有“硬币的两面性”，它是利与弊的结合体。在清楚其带来的好处的同时，还需要了解 C++ 异常所带来的诸多代价。建议综合考虑各种因素，对代价成本做一个合理的评估，对异常利弊做一个审慎的权衡，并且根据具体项目的具体情况，谨慎使用异常处理。

#### 请记住：

异常处理在为我们带来便利的同时，也会带来时间和空间上的开销，使程序效率降低，体积增大，同时会加大代码调试和管理的成本。所以在应用 C++ 异常时要综合考虑所有的因素，三思之后再对 C++ 异常说 YES 或 NO。

## 建议 70：尽量保证异常安全

关于 C++ 中异常的争论在任何一个 C++ 技术论坛都不会少。确实，异常的引入会带来便利，但同时也需要付出一定的代价，关于这一点已经在建议 69 中探讨过。异常的复杂性使其成为一个难以用好的语言特性。但是如果确定选择或被迫使用这一特性，那么请尽量保证异常安全！

在 Herb Sutter 的《Exceptional C++》中，他将异常的安全等级划分成了三个层次。

#### □ 基本保证（The Basic Guarantee）

基本保证要求异常被抛出后，程序中剩下的所有东西都处于合法状态，没有对象或数据结构的破坏，不会发生资源泄露现象，确保出现异常时程序处于未知但有效<sup>⊖</sup>的状态，保证最基本的安全性。然而，程序的精确状态可能是不可预期的。

#### □ 强保证（The Strong Guarantee）

强保证，在某些地方又被叫做强力保证，它要确保操作的事务性满足“提交或回退语义”。即要么成功，程序处于目标状态，要么不发生改变，保持原有状态。换句话说，强保证的程序有且仅有两种可能的状态：按照预期成功执行了函数，或者继续保持函数被调用时的状态。

#### □ 不抛出保证（The Nothrow Guarantee）

顾名思义，不抛出保证就是允诺在任何情况下都不可能引发任何异常，这是层次最高的

<sup>⊖</sup> 所谓有效，即对象的不变式检查全部通过。

异常安全保证。

很明显，以上三种异常等级提供的安全性是依次加强的。从异常安全的观点来看，最理想的状况就是让我们的代码具有不抛出保证。然而这对于大多数函数而言，是很难达到的。通常，保证异常安全的一般性规则就是，尽量提供可达到的最强保证。

接下来，我们用一个例子来说明如何保证异常安全，并写出异常安全的代码。

首先从异常情况下的资源管理开始。假设有如下这样一个类 CClass，因为这个类应用于多线程应用中，所以我们需要为其设置互斥锁，以避免多个线程同时对数据进行操作，如下所示：

```
class CSubClass{};
class CMutex{}; // 互斥锁

class CClass
{
public:
    CClass();
    ~CClass();
    void DoSomething();
private:
    CSubClass* m_pData;
    CMutex m_mutex;
};

void CClass::DoSomething()
{
    Lock(m_mutex);
    ... // do something
    delete m_pData;
    m_pData = new CSubClass;
    Unlock(m_mutex);
}
```

这样的代码设计看上去优雅而实用。但是从异常安全的角度来看却是漏洞百出：

(1) 如果 new CSubClass 抛出异常，m\_mutex 就永远不会被 Unlock (解锁)，产生资源泄露现象。

(2) 如果 new CSubClass 抛出异常，m\_pData 就会成为野指针，指向一块已不存在的内存空间。这有多么危险，我不想再做强调！

也就是说上面的代码连最基本的“基本保证”都达不到。为了规避这些问题，我们采用 C++ 异常机制对上述函数进行重写：

```
void CClass::DoSomething()
{
    Lock(m_mutex);
    try
```

```

    {
        ... // do something
        delete m_pData;
        m_pData = new CSubClass;
    }
    catch(const std::exception& e)
    {
        Unlock(m_mutex);
        throw;
    }
    Unlock(m_mutex);
}

```

try...catch 的方式固然能够写出异常安全的代码，但是这样做带来的不仅仅是代码的混乱，还有效率的降低，而这正是很多人抨击 C++ 异常的理由。既然不推荐这种方式，那么又有什么更好的解决方案吗？答案是：使用对象来管理资源，也就是 C++ 之父推荐的 RAI。

首先，我们需要设计一个辅助类 Clock：

```

class CLock
{
public:
    explicit CLock(CMutex& lock)
        : m_lock(lock){ Lock(m_lock); }
    ~CLock(){ Unlock(m_lock); }
private:
    CMutex& m_lock;
};

```

利用这个辅助类重新设计函数 DoSomething()：

```

void CClass::DoSomething()
{
    CLock lock(m_mutex);
    ... // do something
    delete m_pData;
    m_pData = new CSubClass;
}

```

大功告成！写一个类管理一种资源，一劳永逸，不用不厌其烦地 catch 来 catch 去了。也许有人会感到惊讶，一个简单的资源管理类竟然会使我们的代码变得如此清晰而优雅，而且提供了异常安全的基本保证，更加不可思议的是此异常安全的代码中竟然看不到任何的 try、throw 和 catch，但是事实确实如此！

不过，不要高兴得太早，使用对象管理资源仅仅是良好设计的开始，而不是终点。我们的目标绝不该只停留在最低层次的“基本保证”上。

对于一个方法来说，在常规的执行过程中，我们可能会需要多次修改对象的状态，在方法执行的过程中，对象是可能处于非法状态的，如果此时发生异常，对象将变得无效，这就

需要实现强保证了。实现强保证有一种通常的设计策略，即在可能失败的过程中计算出对象的目标状态，但是不修改对象，在决不会失败的过程中，把对象替换到目标状态上。这种策略一般被称之为“Copy & Swap”。

经过多年的实践，我发现这类强保证的实现也有一定的技法，即将每一个对象中的全部数据放入到一个单独的实现对象中，然后将一个指向实现对象的指针交给原对象。这通常被称为“pimpl idiom”。

在下面的代码片段中我们通过智能指针对资源进行了管理，实现了基本保证：

```
class CSubClass_1{ ... };
class CSubClass_2{ ... };

class CClass
{
public:
    CClass();
    ~CClass();
    void DoSomething();

private:
    std::auto_ptr<CSubClass_1> m_pData1;
    std::auto_ptr<CSubClass_2> m_pData2;
}

void CClass::DoSomething()
{
    ... // do something
    m_pData1.reset(new CSubClass_1);
    m_pData2.reset(new CSubClass_2);
}
```

接下来我们采用 pimpl 技法，对该类进行改造升级，以期实现异常安全的强保证，如下所示：

```
class CClass
{
    struct PMImpl
    {
        std::auto_ptr<CSubClass_1> m_pData1;
        std::auto_ptr<CSubClass_2> m_pData2;
    };

public:
    CClass();
    ~CClass();
    void DoSomething();

private:
    std::auto_ptr<PMImpl> m_pImpl;
}
```

```

void CClass::DoSomething()
{
    ... // do something
    std::auto_ptr<PMImpl> pTemp(new PMImpl);
    pTemp->m_pData1.reset(new CSubClass_1);
    pTemp->m_pData2.reset(new CSubClass_2);

    std::swap(m_pImpl,pTemp);
}

```

在强保证版本的代码中，我们使用了一个局部变量 pTemp。首先将计算出的目标状态放在 pTemp 中，然后再安全地进入目标状态。如果计算目标状态失败，则原状态不做任何改变。但是需要注意的是，在通常情况下，它不能保证全部函数都是强力异常安全的。如果在函数中调用了低异常保证层次的函数，该函数的异常保证层次也会随之降低，这就是木桶理论在异常层次中的体现。

另外一个需要注意的问题就是效率。在 Copy & Swap 的过程中需要改变一个对象的数据拷贝，然后在一个不会抛出异常的操作中将被改变的数据和原始数据进行交换。这不可避免地增加了时间和空间上的开销。

最后我们要说的是理想状态的顶级异常安全保证：不抛出保证。之所以用到了“理想”二字，是因为通常我们并不需要这么强的安全保证，并且难以实现这么强的安全保证。但是有三类函数是例外的：析构函数、释放函数和 swap 函数。因为这三个函数是实现“基本保证”和“强保证”的基石：析构和释放函数不抛出保证是 RAII 技术有效的基本保证；swap 不抛出保证是为了“在决不失败的过程中，把对象替换到目标状态中”。

综上所述，如果采用了异常机制，请尽量保证异常安全：努力实现强保证，至少实现基本保证。常用的方法包括：

- 使用超强的 RAII，保证在产生异常时，资源会自动回收，实现基本保证。
- 使用 pimpl 帮助实现 RAII，并把逻辑操作分派到各个成员内部当中，使之在发生异常时保持一致性，通过 Copy & Swap 实现强力保证。
- 为析构函数、释放类函数和 swap 函数提供最高层次的不抛出保证。

借用下面这段话结尾：

四十年前，到处都是 goto 的代码被尊为最佳实践，现在我们为书写结构化控制流程而奋斗；二十年前，全局可访问数据被尊为最佳实践，现在我们为封装数据而奋斗；十年以前，写函数时不必考虑异常的影响被尊为最佳实践，现在我们为写异常安全的代码而奋斗。

时光在流逝。我们生活着。我们学习着。

——C++ 戮言

---

**请记住：**

如果决定采用异常机制，请尽量保证异常安全：努力实现强保证，至少实现基本保证。合理应用一些技法，优化我们的程序设计。

要写好一个异常安全的模块，请将它们牢记于心：

RAII、pimpl、数据一致性、swap。

---

# 第 7 章 用好 STL 这个大轮子

纵观整个软件领域，数十年来都在为了一个目标而奋斗——可复用性（reusability）。从最早的面向过程的函数库，到面向对象的程序设计思想，再到后来的各种组件技术，以及风靡一时的设计模式，无一例外，STL 亦然。

STL 背后蕴涵的是一种新的程序设计思想——泛型化设计（Generic Programming）。早在 1971 年，David R. Musse 就开始在计算机几何领域发展并倡导这种设计观念，尽管那时没有任何编程语言支持。

当时间来到 1987 年，在贝尔实验室工作的 Alexander Stepanov 开始首次采用 C++ 语言进行泛型软件库的研究。但相当遗憾的是，当时的 C++ 语言还没有引入模板这一重要语法，所以他别无选择的采用了继承机制来实现。

又过了 5 年，Alexander Stepanov 再次将泛型化算法作为他研究工作的重点，此时的他多了一个重要的合作伙伴 Meng Lee。经过长时间的努力，STL 的雏形——一个包含有大量数据结构和算法部件的庞大运行库出现在了世人面前。

后来，随着 C++ 标准的不断改进，STL 也在不断地进行相应的演化。1994 年 2 月，这一庞大计划正式地被 ANSI/ISO C++ 标准所接纳，STL 成为 C++ 标准中不可或缺的一大部件。

软件领域有一个著名的描述软件重用的谚语：不要重复造轮子（Don't reinvent the wheel）。而本章中所讲的 STL 无疑是 C++ 领域中那个最大、最重要的轮子。所以我们有必要了解这个大轮子，用好这个大轮子。

## 建议 71：尽量熟悉 C++ 标准库

“设计更多的库来扩充功能要好过设计更多的语法”，C++ 之父 Bjarne Stroustrup 先生不止一次地表达过这样的观点。在 C++ 中，库的地位是非常高的。因为库让我们实现了代码的可复用性（reusability）。现实中，C++ 的库门类繁多，它所解决的问题也是千奇百态的。不过，在这些库中，C++ 标准库扮演的角色最为重要，注意，没有之一。

标准库为 C++ 程序员们提供了 C++ 程序的基本设施，或者说是一个可扩展的基础性框架，高度体现了软件的可复用性。标准库的发布对 C++ 程序风格的影响非常巨大，以致 C++ 之父还专门为此撰写了一篇名为《Learning Standard C++ as a New Language（把标准 C++ 当做一门新语言来学习）》的文章。经过这么多年的实践，C++ 标准库被证明是具有工业级别强度的佳作，无愧于“标准”二字。我们可以应用标准库中的各种现有数据结构、算法开发

我们自己的程序，这等于是站在巨人的肩膀上进行开发，从中获得了极大的便利；同时我们也可以通过继承现有类，编制符合接口规范的容器、算法、迭代子，从而对C++标准库进行合理的扩展。

C++标准库主要包含了如下几大组件（如图7-1所示）。

#### □ C标准函数库

C标准函数库基本保持了与原有C语言程序库的兼容性，这是在履行C++之父当初“对C兼容”的承诺。当然，C标准函数库也有少许的变化。这一点在建议23中有所涉及。

#### □ 输入/输出(input/output)

输入/输出部分的基础是原有标准库中的*iostream*部分，但是标准库中将其模板化了，从而提供对C++程序输入输出的基本支持。它在功能上保持了与原有*iostream*的兼容性，增加了异常处理的机制，并支持国际化(internationalization)。

#### □ 字符串(string)

字符串部分用来处理文本。它提供了足够丰富的功能。事实上，文本是一个*string*对象，它可以被看作是一个字符序列，字符类型可能是*char*，也可能是*wchar\_t*。

#### □ 容器(containers)

容器部分涵盖了许多数据结构，比如*list*(链表)、*vector*(向量)、*queue*(队列)、*stack*(堆栈)等。这些数据结构为我们的工作提供了巨大的便利。

#### □ 算法(algorithms)

算法部分包含了大约70个通用算法，用于操控各种容器，同时也可操控内建数组。比如，*find*用于在容器中查找等于某个特定值的元素，*for\_each*用于将某个函数应用到容器中的各个元素上，*sort*用于对容器中的元素排序。所有这些操作都是在保证执行效率的前提下进行的。

#### □ 迭代器(iterators)

如果没有迭代器的撮合，容器和算法便无法结合得如此完美，它起着一种黏合剂的作用。每个容器都有自己的迭代器，只有容器才知道如何访问自己的元素。

#### □ 国际化(internationalization)

国际化部分扮演着消除文化和地域差异的角色，采用*locale*和*facet*可以为程序提供众多国际化支持，包括对各种字符集的支持，日期和时间的表示，数值和货币的处理等，以满足不同地区人们的不同使用习惯。

#### □ 数值(numerics)

数值部分包含了一些数学运算功能，提供了复数运算的支持。

#### □ 语言支持(language support)

语言支持部分包含了一些标准类型的定义，以及其他语言特性的定义。

#### □ 诊断(diagnostics)

诊断部分提供了用于程序诊断和报错的功能，包含了异常处理(exception handling)、

断言 (assertions)、错误代码 (error number codes) 三种方式。

#### □ 通用工具 (general utilities)

通用工具为 C++ 标准库的其他部分提供支持，当然也可以在自己的程序中调用相应功能。比如，动态内存管理工具、日期 / 时间处理工具。

由此可以看出，C++ 标准库确实提供了很多对我们有帮助的东西。不过，虽然 C++ 标准库很大、非常大、相当大 (C++ 标准中关于它的描述就有 300 多页)，但 C++ 标准库并非提供了一切，有些方面它也是有欠缺的，比如图形用户接口，这也是 C++ 标准库多年来的一大遗憾。

在 C++ 标准库提供的组件中，因为字符串、容器、算法、迭代器四部分采用了模板技术，一般被统称为 STL (Standard Template Library，即标准模板库)。当然它还有另一种解释：STepanov & Lee <sup>⊖</sup>，即两个 STL 史上的著名人物：前者是 Alexander Stepanov，STL 的创始人；后者是 Meng Lee，STL 得以推行的功臣。

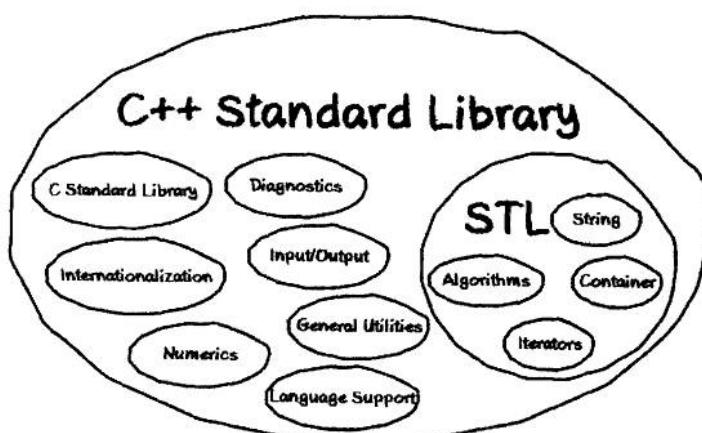


图 7-1 C++ 标准库组成

STL 是一个具有工业强度的、高效的 C++ 程序库。它被容纳于 C++ 标准程序库中，是 ANSI/ISO C++ 标准中最新也是极具革命性的一部分。该库包含了诸多在计算机科学领域里常用的基本数据结构和基本算法。从逻辑层次来看，STL 中体现了泛型化程序设计的思想 (generic programming)，且引入了诸多新的名词，比如需求 (requirements)、概念 (concept)、模型 (model)、容器 (container)、算法 (algorithm)、迭代器 (iterator) 等。从实现层次来看，整个 STL 是以一种类型参数化 (type parameterized) 的方式实现的，模板构成了整个 STL 的基石。在 STL 背后蕴含着的是泛型化程序的设计 (GP) 思想，在这种思想

<sup>⊖</sup> 这一提法源自 1995 年 3 月，Dr.Dobb's Journal 的特约记者、著名技术书籍作家 Al Stevens 对 Alexander Stepanov 的一篇专访。

里，大部分基本算法被抽象，被泛化，独立于与之对应的数据结构，以相同或相近的方式处理各种不同情形。

ANSI/ISO C++ 文件中的 STL 是一个仅被描述在纸上的标准，对于诸多的 C++ 编译器而言，需要用 C++ 编写的 STL 代码来实现标准中所描述的内容。STL 有不同的实现版本，例如 SGI STL、STLport、HP STL、Rouge Wave STL 等。

在 C++ 标准中，STL 被组织成了下面 13 个头文件：`<algorithm>`、`<deque>`、`<functional>`、`<iterator>`、`<vector>`、`<list>`、`<map>`、`<memory>`、`<numeric>`、`<queue>`、`<set>`、`<stack>` 和 `<utility>`。

到此为止，我们熟悉了 C++ 标准库的大体框架与内容。不难得出这样一个结论：C++ 标准库是一个巨大的资源。所以，应该尽量去了解它。如果你是一个奉行“实用至上”的开发者，应该尽量了解 C++ 标准库组件使用中的细枝末节，在应用开发过程中，尽情应用它所提供的组件，从而简化设计，避免重复劳动，节省大量开发时间，提高编程效率。如果你是一个信奉“技术为王”的技术爱好者，可以将它作为经典的教材，特别是 STL 部分，学习其中的设计思想与技巧，并为你所用，提高技术水平。当然，亦可以兼而有之！

总之，一句话，应该尽量熟悉 C++ 标准库！

最后，推荐一本书，Nicolai M. Josuttis 著的《The C++ Standard Library: A Tutorial and Reference》，它可以成为学习 C++ 标准库的重要工具书。

---

#### 请记住：

C++ 标准库是一个非常有用的资源，需要尽量熟悉它，知道它的各个组件，让其为你的学习、实践而服务。

---

## 建议 72：熟悉 STL 中的有关术语

STL 为程序世界引入了很多的术语，也许在阅读标准模板库（STL）文献或文档的时候你就遇到过它们。不要对这些术语视而不见，熟悉这些 STL 术语是更好地使用 STL 的一大前提。

### □ 容器（Container）

容器是一个对象，它将对象作为元素来存储。通常情况下，它是作为类模板来实现的，其成员函数包括遍历元素、存储元素和删除元素。STL 标准容器一般分为两大类：序列容器和关联容器，关于这一点在以后的建议中会详细阐述。其中 `std::list` 和 `std::vector` 是我们应用最为广泛的两种典型的容器类。

### □ 泛型（Genericity）

泛型就是通用，或者说是类型独立。最好的例证就是上面所讲的容器类。STL 中对于容

器类的定义是非常宽松的，因为它适用于字符串、数组、结构体，或者是对象。一个真正的容器是不局限于某一种或某些特定的数据类型的。相反，它可以存储任何内置类型或用户自定义类型。而这样的容器就被认为是通用的，是泛型的。泛型被看作是 STL 的最重要的特征之一。

除此之外，为了简化编写函数对象的过程，STL 还给出了函数对象的标准基类：`std::unary_function` 和 `std::binary_function`。两者都声明在头文件 `<functional>` 中。正如名字的字面含义，`unary_function` 被用作是接受一个参数函数的对象基类，而 `binary_function` 是接受两个参数函数的对象基类。这些基类的定义如下：

```
template < class Arg, class Res >
struct unary_function
{
    typedef Arg argument_type;
    typedef Res result_type;
};

template < class Arg, class Arg2, class Res >
struct binary_function
{
    typedef Arg first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Res result_type;
};
```

这些模板并不提供任何实质性的功能。它们只是确保其派生函数对象的参数和返回值有统一的类型名称。在下面的例子中，`is_vowel` 继承自 `unary_function`，接受一个参数：

```
template < class T >
class is_vowel: public unary_function< T, bool >
{
public:
    bool operator ()(T t) const
    {
        if ( (t=='a') || (t=='e') || (t=='i')
            || (t=='o') || (t=='u') )
            return true;
        return false;
    }
};
```

因为函数对象是通用编程中的一个重要部分。在设计实现过程中，坚持标准规范将会省去很多麻烦。

#### □ 算法 (Algorithm)

STL 的算法种类繁多，可以满足我们大多数的需求；算法就是对一个对象序列所采取的某些操作，例如 `std::sort()` 排序、`std::copy()` 复制和 `std::remove()` 删除。STL 中的算法都是以

函数模板的形式来实现的，这些函数的参数都是对象迭代器。

#### □ 适配器 (Adaptor)

适配器是一个非常特殊的对象，它的作用就是使函数转化为函数对象，或者是将多参数的函数对象转化为少参数的函数对象。

STL中定义了多种类型的序列适配器，主要有以下三类：

#### bind

```
bind1st // 通过绑定第一个参数，使二元的函数对象转化为一元的函数对象
bind2nd // 通过绑定第二个参数，使二元的函数对象转化为一元的函数对象
not1    // 对一元的函数对象取反
not2    // 对二元的函数对象取反
```

#### ptr\_fun

`ptr_fun`是指将现有的函数转换为Functor（函数指针）的功能。在STL中提供这个功能的Functor是`pointer_to_unary_function`和`pointer_to_binary_function`这两个类，这两个类分别对应一元和二元两种函数。

#### mem\_fun

`mem_fun`提供的功能是将某个类中的成员函数转变为Functor。

函数适配器可以插入到一个现有的类或函数中来改变它的行为。例如，将一个特殊的适配器插入到`std::sort()`算法中，就可以控制排序是降序还是升序了：

```
vector<int> lv;
...
sort(lv.begin(), lv.end(), not2(less<int>()) );
```

#### □ O(h)

$O(h)$ 是一个表示算法性能的特殊符号，在STL规范当中用于表示标准库算法和容器操作的最低性能极限。 $O(h)$ 可以帮助评估一个算法或某个特定类型容器中某个操作的效率。`std::find()`算法遍历了序列中的元素，在最坏的情况下，需要遍历序列中的所有元素，其性能可以表示为：

```
T(n) = O(n). /* 线性复杂度 */
```

#### □ 迭代器 (Iterator)

迭代器是一种可以当做通用指针来使用的对象。迭代器可以用于元素遍历、元素添加和元素删除。STL定义了5种主要的迭代器：

(1) 输入迭代器和输出迭代器 (input iterators and output iterators)：前者允许迭代器前行，并提供只读访问；而后者也允许迭代器前行，但提供只写访问。

(2) 前向迭代器 (forward iterators)：它支持读取和写入权限，但只允许一个方向上的遍历。

(3) 双向迭代器 (bidirectional iterators)：它允许用户在两个方向遍历序列。

(4) 随机访问迭代器 (random access iterators)：它支持迭代器的随机跳跃，以及“指针算术”操作，例如：

```
string::iterator it = s.begin();
char c = *(it+5); /* 将 s 的第 6 个字符赋值给 c */
```

请注意，上述迭代器列表并不具有继承关系，它只是描述了迭代器的种类和接口。下面的迭代器类是上面类的超集。例如，双向迭代器不仅提供了前向迭代器的所有功能，还包括了一些附加功能。

当然这并不是 STL 引入的全部术语，但绝对是其中最为重要的术语。掌握熟悉这些术语，可以加深、加快我们对 STL 的了解，让 STL 应用之路更加顺畅便捷。

---

#### 请记住：

STL 的引入带来了很多新生的术语，要了解这些术语，以帮助我们更好、更快、更深的理解 STL，应用 STL。

---

### 建议 73：删除指针的容器时避免资源泄漏

STL 中的容器是非常优秀的，这一点毋庸置疑。它有很多的优点，其中最让我们叹为观止的就是它的对象自销毁能力。换句话说，当容器自己被销毁时它会自动销毁其容纳的每个对象，让 C++ 程序员不再为容器用完之后的清除工作而劳心费神。STL 容器的这一能力着实为我们解决了一个大麻烦，但是也不能对它的这一能力太过放心，因为稍有不慎，内存泄露就有可能重出江湖。如下所示：

```
void DoSomething()
{
    vector<CTestObj*> vec_objs;
    for (int i=0; i=OBJECTS_COUNT; i++)
    {
        vec_objs.push_back(new CTestObj);
    }
    ... // use vec_objs to do something
}
```

当函数执行完毕，STL 容器 vec\_objs 超出作用域会被销毁，同时容器中包含的每个元素也会被销毁。但不幸的是，销毁的元素只是指针，`delete` 并没有作用于 `new` 得到的对象，所以它的析构函数也就永不会被调用！于是内存泄漏由此而生。出现这种情况是因为只有代码编写者才知道这样的指针是否应该被删除，删除 `new` 的对象是程序员自己的责任。于是，我们得到了如下的版本：

```

void DoSomething()
{
    vector<CTestObj*> vec_objs;
    for (int i = 0; i = OBJECTS_COUNT; i++)
    {
        vec_objs.push_back(new CTestObj);
    }
    ... // use vec_objs to do something

    vector<CTestObj*>::iterator it = vec_objs.begin();
    for (; it != vec_objs.end(); ++it)
    {
        delete *it;
    }
}

```

如果要求不是很苛刻，这个版本完全可以正常工作。不过，它还存在一个问题：这段代码不是异常安全的。如果在“use vec\_objs to do something”过程中抛出了一个异常，资源泄漏将再次不期而至。解决这一问题的方法也很简单，那就是：使用智能指针的容器来代替普通指针的容器。Boost 库中的 shared\_ptr 绝对是一个不错的选择。利用 shared\_ptr 对上述示例代码进行改造，可得到如下代码片段：

```

void DoSomething()
{
    typedef boost::shared_ptr<CTestObj> SmartTestPtr;
    vector<SmartTestPtr> vec_objs;
    for (int i = 0; i = OBJECTS_COUNT; i++)
    {
        vec_objs.push_back( SmartTestPtr(new CTestObj));
    }
    ... // use vec_objs to do something
}

```

这个版本的代码实现了异常安全，即使在“use vec\_objs to do something”过程中抛出异常，资源也会被安全释放，不会发生资源泄漏。

需要注意的是，建立 auto\_ptr 的容器是非常可怕，非常危险的，是我们应该必须避免的行为。在建议 75 中会对这一问题进行详细讨论。

#### 请记住：

STL 容器虽然智能，但尚不能担当删除它们所包含指针的这一责任。所以，在要删除指针的容器时须避免资源泄漏：或者在容器销毁前手动删除容器中的每个指针，或者使用智能引用计数指针对象（比如 Boost 的 shared\_ptr）来代替普通指针。

## 建议 74：选择合适的 STL 容器

STL 包含多方面的内容，其中容器（Container）被众多 C++ 程序员认为是 STL 中最主要的组成部分。因此，STL 容器受到了广大 C++ 程序员的喜爱与推崇，正如 Scott Meyers 所讲的那样“STL 不是 just good（刚刚好），而是 really good（非常好）”。

相较于数组，容器更强大且更灵活。C++ 为我们提供了多种多样的容器，具体可以分成如下几种：

- 标准 STL 序列容器：vector、string、deque 和 list。
- 标准 STL 关联容器：set、multiset、map 和 multimap。
- 非标准序列容器：slist（单向链表）和 rope（重型字符串）。
- 非标准关联容器：hash\_set、hash\_multiset、hash\_map 和 hash\_multimap。
- 标准非 STL 容器：数组、bitset、valarray、stack、queue 和 priority\_queue。

面对这么多的容器类型，如何选择合适的类型成为摆在我们面前的一个重要问题。针对这个问题，C/C++ 标准给出了比较权威的使用建议：

向量、链表和队列向程序员提供了不同的复杂度取舍，我们应该据此来选择使用它们中的哪一个。向量是一种序列类型，默认使用它。当需要频繁地在序列中插入或者删除元素时，应该选择链表。当大多数的插入或删除发生在序列的开始或结尾时，队列是我们最该选择的数据结构。<sup>⊖</sup>

也就是说，vector 是我们默认使用的序列类型，应最先考虑；如果要很频繁地对序列中部进行插入和删除操作时，优先选用 list；当有大量的插入和删除动作发生在序列的头部或尾部时，deque 就成了最优选择。上述建议是基于容器的特点而制定的。然为美中不足的是，它只是涉及其中的三种容器：vector、list 和 deque。

### □ 向量（vector）

人送外号“会自动增长的数组”，相较于普通意义上的数组，它的最大优势就是能够动态改变自身大小。其实现的内部数据结构是数组。类在创建时，会同时创建一个定长数组，随着数据不断被写入，当数组被填满时，它会重新开辟一块更大的内存区，并把原有的数据复制到新的内存区中，从而抛弃原有的内存，如此反复。由于它的每个数据的大小相同，并且会无间隔地排列在内存中，所以它可以用常数时间来访问和修改任意元素。在序列尾部进行插入（push\_back）和删除（pop\_back）操作时，则具有常数时间的复杂度。但是在容器前

<sup>⊖</sup> vector, list, and deque offer the programmer different complexity trade-offs and should be used accordingly. vector is the type of sequence that should be used by default. list should be used when there are frequent insertions and deletions from the middle of the sequence. deque is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.— C++ 23.1.1.2 P468 ISO/IEC 14882:2003(E)

端和中间进行数据操作时对性能的影响比较大，这是因为这些操作必然会引起数据块的移动。

#### □ 表（list）

其实现的数据结构是双向链表，它可以进行双向遍历，其访问时间与到两端的距离成正比；链表是没有自动增长能力的，需要重新开辟内存。list 提供了 push\_back、push\_front、pop\_back、pop\_front 四个方法来对 list 两端的数据进行增加和删除操作；在 list 中的某个位置上进行插入和删除操作时，会花费常数时间，因为它不必进行数据块的移动，只需将指向前后的指针做一个调整。

#### □ 双端队列（deque）

它基本上与向量相同，唯一的不同是，其在序列头部的插入和删除操作也具有常量时间的复杂度。

一般说来，这三种标准 STL 容器是我们应用得最为广泛的容器类型，可以满足绝大多数的应用需求。但是 STL 给我们的选择远非这三种，我们应该了解得更多，如表 7-1 所示。

表 7-1 容器类型对比

类型	主要特点描述
slist	应用单向链表实现。不可双向遍历，只能从前到后地遍历。其他的特性同 list 相似
stack	适配器，它可以将任意类型的序列容器转换为一个堆栈，一般使用 deque 作为支持的序列容器。元素只能后进先出（LIFO），不能遍历整个 stack
queue	适配器，它可以将任意类型的序列容器转换为一个队列，一般使用 deque 作为支持的序列容器。元素只能先进先出（FIFO），不能遍历整个 queue
priority_queue	适配器，它可以将任意类型的序列容器转换为一个优先级队列，一般使用 vector 作为底层存储方式。只能访问第一个元素，不能遍历整个 priority_queue。第一个元素始终是优先级最高的一个元素
set	键和值相等。键唯一。元素默认按升序排列
multiset	键可以不唯一。其他特点与 set 相同
hash_set	元素按照所用的 hash 函数分派，它能提供更快的搜索速度。其他特点与 set 相同
hash_multiset	键可以不唯一。其他特点与 hash_set 相同
map	键唯一。元素默认按键的升序排列
multimap	键可以不唯一。其他特点与 map 相同
hash_map	元素按照所用的 hash 函数分派，它能提供更快的搜索速度。其他特点与 map 相同
hash_multimap	键可以不唯一。其他特点与 hash_map 相同

我们可根据不同容器的不同特点，结合应用情形选择合适的容器。在选择时，我们可以参照如图 7-2 所示的流程进行筛选。

除了上述方式以外，我们也可以将 Scott Meyers 曾经总结过的“容器选择时需要考虑的 11 个问题<sup>⊖</sup>”作为筛选标准：

⊖ Scott Meyers 的著作《Effective STL》中对此有所建议，详情可参阅该书。

(1) 需要在容器的任意位置插入一个新元素吗？如果是，则要使用序列容器，关联容器做不到。

(2) 是否关心元素在容器中的顺序？如果不关心，hash 容器就是可行的选择。否则，避免使用 hash 容器。

(3) 是否必须使用标准 C++ 中的容器？如果是的话，我们就可以将选择范围集中在标准 STL 序列容器和关联容器两大类上。

(4) 需要哪一类的迭代器？如果必须是随机访问迭代器，在技术上就只能限于 vector、deque 和 string，但也可以考虑 rope。如果需要双向迭代器，slist 和 hash 容器就要被排除在外。

(5) 当插入或者删除数据时，是否非常在意容器内现有元素的移动？如果是，就必须放弃连续内存容器。

(6) 容器中的数据的内存布局是否需要兼容 C？如果是，就只剩唯一的选择了，即 vector。

(7) 对元素的查找速度有较高的要求吗？如果是，着重考虑 hash 容器、排序的 vector 和标准的关联容器。

(8) 容器的底层使用了引用计数，对此介意吗？如果是，就得避开 string 和 rope，因为二者的很多实现是基于引用计数的。

(9) 需要插入和删除的事务性语义吗？也就是说，需要有可靠的回退（roll back）插入和删除的能力吗？如果是，就需要使用基于节点的容器。如果需要多元素插入的事务性语义，就应该选择 list，因为 list 是唯一提供多元素插入事务性语义的标准容器。

(10) 要把迭代器、指针和引用的

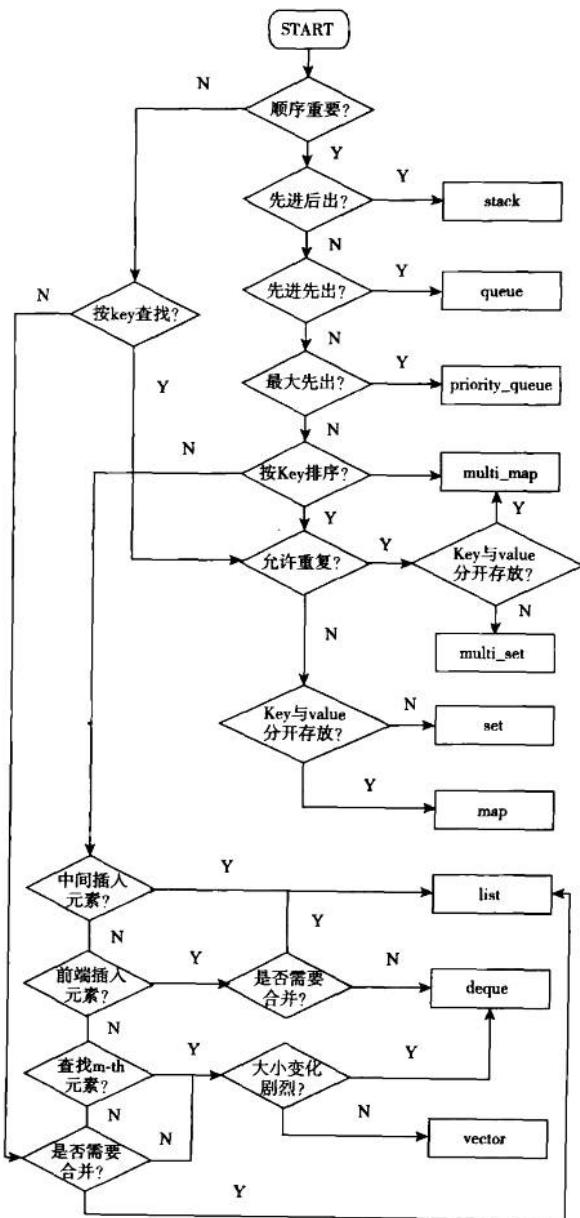


图 7-2 STL 容器选择流程图

失效次数减到最少吗？如果是，就应该使用基于节点的容器，因为在这些容器上进行插入和删除操作不会使迭代器、指针和引用失效。一般来说，在连续内存容器上进行插入和删除操作会使所有指向容器的迭代器、指针和引用失效。

(11) 你需要的序列容器是不是具有以下特征：a) 可以使用随机访问迭代器；b) 只要没有删除操作而且插入只发生在容器结尾，那么指针和引用的数据就不会失效。如果遇到这种情况，`deque` 就是理想的容器。当插入只在容器结尾发生时，`deque` 的迭代器也可能会失效，`deque` 是唯一一个“在迭代器失效时不会使它的指针和引用失效”的标准 STL 容器。

当然，这 11 个问题不是全部，但是经过这 11 条标准的筛选，候选的容器范围肯定已经非常小了。

#### 请记住：

容器（Container）是 STL 为我们带来的最大福利，面对多样的容器，需要慎重选择，考虑具体的应用环境，选择最为合适的容器。

### 建议 75：不要在 STL 容器中存储 auto\_ptr 对象

`auto_ptr` 是 C++ 标准中提供的智能指针，它是一个 RAII 对象，它在初始化时获得资源，析构时自动释放资源。基于 `auto_ptr` 的这一优点，很多 C++ 程序员妄图将 `auto_ptr` 的容器（COAPs，Container Of `auto_ptrs`）作为解决容器资源泄漏问题的一个简单、直接、高效的解决方案，所以，他们会写出如下代码片段：

```
class CTestObj{ };
bool ObjCompare(const auto_ptr<CTestObj>& lhs,
                const auto_ptr<CTestObj>& rhs ) { ... }

vector< auto_ptr<CTestObj> > ObjVector;
// 为 ObjVector 赋值
...
// 操作 1
auto_ptr<CTestObj> pTemp = ObjVector[0];
// 操作 2
sort(ObjVector.begin(), ObjVector.end(), ObjCompare);
```

一切看起来都是那么地合情合理，但是大部分的编译器都会拒绝。无情的事实告诉我们：这样是行不通的。这个问题非常严重，以致惊动了标准化委员会，他们对此采取了禁用 COAPs 的严厉措施以避免这类问题的发生。喜欢刨根问底的读者也许会发问：为什么要禁止 COAPs 呢？这恐怕还要从 STL 容器和 `auto_ptr` 对象说起。

C++ 标准中规定：STL 容器元素必须能够进行拷贝构造和赋值操作；也就是说，STL 元

素对象可以进行安全的赋值操作，可以将一个对象拷贝到另一个对象上，从而获得两个独立的、逻辑上相同的拷贝。而 auto\_ptr 对象恰恰不能满足这一条件，拷贝一个 auto\_ptr 将会改变它的值，这是不安全的。在示例代码中，当 pTemp 完成赋值时，原来的指针 ObjVector[0] 就变成了 NULL。任何对该元素的操作企图都将导致应用程序在运行时崩溃，即使像操作 2 那样没有进行显式地拷贝或赋值操作，灾难同样难以避免。诸如 sort()、swap() 之类的算法中会创建一个或多个容器元素的临时拷贝，这些“隐身”的临时拷贝会使 STL 容器中的对象变成无效对象，从而破坏了原本正确有效的元素数据。

除了上述可能发生的灾难外，还有一个因素阻止我们使用 COAPs，那就是 COAPs 是不可移植的。虽然 C++ 标准禁止它们，但还是有些厂商不按照套路行动，它们推出了允许 COAPs 的 STL 平台。这就直接导致使用了 COAPs 的代码在各平台间所进行的移植测试以失败告终。

幸好在新的 C++ 标准 11 中，智能指针 auto\_ptr 被标准委员会的委员们抛弃到历史的垃圾堆里了。这个扶不起的阿斗的历史使命也将结束。

所以，永不使用 auto\_ptr 的容器 COAPs，即使 STL 平台允许。当然，一个 auto\_ptr 并不能代表所有的智能指针；禁止 COAPs 也并不是表示禁止所有智能指针的容器，只不过 auto\_ptr 不是可以用的智能指针。我们完全可以选择其他拷贝安全的智能指针来作为容器元素，比如 boost::shared\_ptr。

#### 请记住：

禁止使用 COAPs，但不是禁止使用智能指针的容器！禁止的原因有两条，其一：auto\_ptr 拷贝操作不安全，会使原指针对象变 NULL；其二：严重影响代码的可移植性！

## 建议 76：熟悉删除 STL 容器中元素的惯用法

在应用中，我们通常不可避免地要对容器中的某些特定元素进行删除操作。这看起来并不是什么困难的问题。我们先写一个循环来迭代容器中的元素，如果迭代元素是要删除的元素，则删除之。任务简单，代码也简单：

```
vector<int> intContainer;
...
for (vector<int>::iterator it = intContainer.begin();
     it != intContainer.end();
     ++it )
{
    if( (*it) == 25 ) intContainer.erase(it);
}
```

写此代码原意是将 vector 中值为 25 的元素删除，相信很多程序员首先会想到使用这个方法。但不幸的是，这样做是错误的，这么做会带来诡秘的未定义行为。因为当容器的一个元素被删时，指向那个元素的所有迭代器都已经失效了（关于这一点会在建议 77 中详细讲述）。当 intContainer.erase(it) 返回时，it 已经失效。在 for 循环中对于失效的 it 执行自增操作，这是一件多么不靠谱的事儿啊！

既然这样行不通，那么我们可以求助于 STL 提供的 remove 算法。借助 remove 算法来实现作者的意图。所以就出现了如下的代码：

```
vector<int> intContainer;
...
size_t before_size = vector_int.size();
remove(intContainer.begin(), intContainer.end(), 25);
size_t after_size = vector_int.size();
```

调试运行，结果却出乎我们的意料，before\_size 竟然和 after\_size 一样大。也就是说，remove 没有改变容器中元素的个数。到底是哪里出了问题？

这是 STL 新手最容易出现的错误！其实，跟踪一下程序运行的过程，我们会发现在 remove 操作的前后，vector\_int 内的数据竟然出现了诡异的变化，如图 7-3 所示。

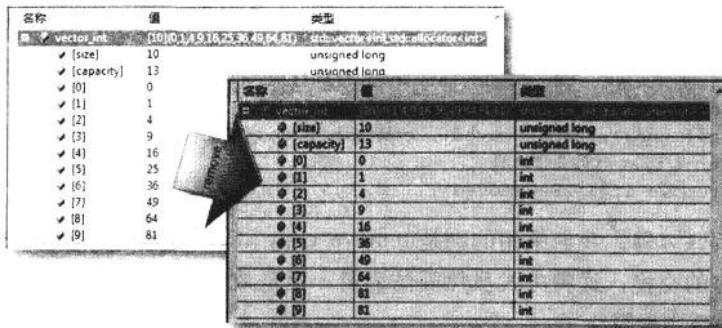


图 7-3 vector 在使用 remove 算法前后的对比

remove 算法并没有按照我们所期望的那样执行。仔细分析 STL 中的 remove 算法，会发现：remove 只会将不应该删除的元素前移，然后返回一个迭代器，该迭代器指向的是那个应该删除的元素。仅此而已！所以如果要真正删除这一元素，在调用 remove 之后还必须调用 erase，这就是 STL 容器元素删除的“erase-remove”的惯用法：

```
intContainer.erase(
    remove(intContainer.begin(), intContainer.end(), 25),
    intContainer.end() );
```

erase-remove 的惯用法适用于连续内存容器，比如 vector、deque 和 string（如果将

`string` 也看作是一种特殊的容器的话), 它同样也适合 `list`, 但是, 这并不是我们推荐的方法, 因为使用 `list` 的成员函数 `remove` 会更高效, 代码如下:

```
list<int> list_int;
...
list_int.remove(25);
```

所以当选用的容器是 `list` 时, `remove` 成员函数是去除特定值元素的最佳方法。

如果是标准关联容器呢? 标准关联容器没有 `remove` 成员函数, 使用 STL 算法的 `remove` 函数时编译通不过<sup>Θ</sup>。所以上述 `remove` 形式对于标准关联容器并不适用! 在这种情况下, 解决办法就是调用 `erase`:

```
map<int, int> mapContainer;
...
mapContainer.erase(25);
```

对于标准关联容器, 这样的元素删除方式简单而高效, 时间复杂度为  $O(\log n)$ , 即对数时间。

如果我们的需求不再是删除某一个具有特定值的元素, 而是删除具备某一条件的一些元素呢? 比如, 有一个 `Container<int>` 类型的容器, 我们要删除容器中所有小于 25 的元素。针对那些可以使用 `remove` 的容器, 我们只需要将 `remove` 替换成 `remove_if` 即可:

```
bool Is2BeRemoved(int value);
// container 如果是 vector、deque、string，或者 list
container.erase(remove_if(container.begin(),
    container.end(), Is2BeRemoved), container.end());
// container 是 list 时的最佳选择
container.remove_if(Is2BeRemoved);
```

对于标准关联容器, 我们采用的则是 `remove_copy_if & swap` 组合的解决方式。但是由于涉及容器内容的交换, 这种方式虽然简洁容易, 但是效率相对较低。如果对此不能忍受, 那你只能自己动手实现算法了。

#### 请记住:

- 删除容器中具有特定值的元素

如果容器是 `vector`、`string` 或 `deque`, 使用 `erase-remove` 的惯用法。如果容器是 `list`, 使用 `list::remove`。如果容器是标准关联容器, 使用它的 `erase` 成员函数。

- 删除容器中满足某些条件的所有元素

如果容器是 `vector`、`string` 或 `deque`, 使用 `erase-remove_if` 惯用法。如果容器是 `list`, 使用 `list::remove_if`。如果容器是标准关联容器, 使用 `remove_copy_if & swap` 组合算法, 或者

<sup>Θ</sup> 至于为什么 `remove` 算法不适于标准关联容器, 感兴趣的读者可以 Google 之。

自己写一个遍历删除算法。

---

## 建议 77：小心迭代器的失效

迭代器 (iterator) 是 STL 中的一个重要部分，正是因为它的存在，让 STL 中另外的两大主角：容器和算法完美地结合在了一起，它是二者联系在一起的桥梁。由于迭代器与指针有着太多的共同点，并且有着相似的操作方式（解除引用 (operator\*()) 和自增操作 (operator++())），致使很多人将二者当成了一样东西。这种想法是绝对不正确的。迭代器是一个对象，其内存大小为 12 (`sizeof(vector<int>::iterator) = 12`，实验 IDE：VC2010，32 位操作系统）。相较于指针，多出来的 8 个字节干什么了呢？如果有人对这一问题感兴趣，可参看编译器中 iterator 的实现代码。

在前面的建议中，我们已不止一次地说起过“迭代器失效”的问题。还是来看建议 76 中的那段示例代码：

```
vector<int> intContainer;
...
for (vector<int>::iterator it = intContainer.begin();
     it != intContainer.end();
     ++it)
{
    if (*it == 25) intContainer.erase(it);
}
```

当 vector 容器中的一个元素被删除时，指向那个元素的迭代器 it 就失效了。而在 for 循环中我们还要对 it 进行自增操作。如果对于一个已经失效的迭代器进行操作，就像吃了失效的药物一样，我们将面对不堪设想的严重后果。

STL 标准容器都有可能会出现迭代器失效的情况，而引起迭代器失效的最主要操作就是插入、删除。那这些迭代器为什么会失效呢？且看接下来的详细分解：

对于序列容器（如 vector 和 deque），插入和删除操作可能会使容器的部分或全部迭代器失效。因为 vector 和 deque 必须使用连续分配的内存来存储元素，向容器中添加一个元素可能会导致后面邻接的内存没有可用的空闲空间而引起存储空间的重新分配。一旦这种情况发生，容器中所有的迭代器就会全部失效。而删除当前的 iterator 则会使后面所有元素的 iterator 都失效。这是因为删除一个元素会导致后面所有的元素都向前移动一个位置。幸运的是，针对序列容器的插入，很少同时会涉及迭代器的操作，而针对删除操作 STL 为我们准备了 `erase-remove` 惯用法，如下所示：

```
bool ShouldDelete(const ContainerElement& value)
{ ... }
```

```
Container.erase(remove_if(Container.begin(), Container.end(), ShouldDelete));
```

关于元素删除的更详细介绍可参见建议 76。

对于关联容器（如 map、set、multimap 和 multiset），删除当前的 iterator，仅仅会使当前的 iterator 失效，其影响范围不像序列容器那样大。这是因为关联容器存储元素所需的内存不是连续的。比如 map 之类的容器是使用红黑树来实现的，在插入、删除一个结点时不会对其他的结点造成影响。所以在删除元素时，只要在 erase 后递增当前的 iterator，使当前的 iterator 指向下一个有效结点即可，代码如下所示：

```
for (iter = Container.begin(); iter != Container.end();)
{
    //...; // do something
    if (ShouldDelete(*iter))
        Container.erase(iter++);
    else
        ++iter;
}
```

对于特殊的 list，它使用了不连续分配的内存，并且它的 erase 方法也会返回下一个有效的 iterator，因此上面两种方法都可以使用。

#### 请记住：

在对容器进行元素增删操作时，要时刻提防“迭代器失效”的情况发生，不仅要明白为何序列容器与关联容器在这一方面存在不同，更应该熟知如何去避免这一错误的发生。

## 建议 78：尽量使用 vector 和 string 代替动态分配数组

数组，在我们的应用程序中经常出现，它在我们的代码中扮演着十分重要的角色。对于编译时大小确定的数组，我们通常采用以下方式来实现：

```
const int MatrixLineCnt = 3;
const int MatrixRowCnt = 3;
float Matrix[MatrixLineCnt][MatrixRowCnt] =
{
    { {0.1f, 1.2f, 0.8f},
      {0.5f, 3.5f, 4.8f},
      {0.7f, 1.8f, 2.9f} };
```

如果数组大小在编译时不确定，上述方式就不再适用。此时，我们就要通过 new 来进行内存的动态分配：

```
char* szName = new char[length];
```

```
...
delete[] szName;
```

使用 new，可是一件必须担负责任的事儿（这一点在第 4 章已经详细介绍过）：必须确保这块内存完成历史使命后会使用 delete 来完成它的释放操作，确保使用了 delete/delete[] 的正确形式，确保这块内存被删除且仅被删除一次。

如果我们的精力被这些细枝末节纠缠着，那确实是一件令人极度恼火的事儿。不过，STL 的出现为我们带来了福音：使用 vector 和 string 不仅能让我们摆脱这些繁缛的小问题，还有助于我们写出安全性更好、伸缩性更强的代码。

具体的说，相较于内建数组，vector 和 string 具有以下几方面的优点。

#### □ 它们能够自动管理内存

vector 和 string 消除了手动释放内存所带来的麻烦，它们可以管理自己的内存：当元素添加到 vector 和 string 中时它们的内存会自动增长；当一个 vector 或 string 销毁时，它的析构函数会自动销毁容器中的元素，回收存放那些元素的内存。这一切都不需要我们付出额外的精力。

#### □ 它们提供了丰富的接口

vector 和 string 绝对是功能完善、质量上乘的容器，它们为我们提供了丰富的接口，可以满足我们大多数的需求。因为 vector 和 string 这两个容器不仅提供了像 begin、end 和 size 这样的成员函数，更有 iterator、reverse\_iterator 或 value\_type 那样的 typedef，在应用 STL 算法时我们可以更加得心应手。

#### □ 与 C 的内存模型兼容

vector 和 string 中的数据可以很简单地传递给 C 语言 API，所以有关整合遗留代码这一问题，对于我们来说毫无压力。这一点会在建议 79 中详细阐述。

#### □ 集众人智慧之大成

STL 是经过了千锤百炼的，而且还在经受着千锤百炼。它经过了非常多次的优化，其品质是绝对有保证的。所以在安全和效率等方面，vector 和 string 都有着不错的表现。

综上所述，当我们准备写下类似“new T[...]”这样的代码时，请尽量使用 vector 或 string 来代替。

#### 请记住：

如果在使用动态分配数组，需要做的工作可能比想象中的要多。要减轻这些负担，就使用 vector 或 string 来代替。

最后，不要忘记感谢一下 vector 和 string，它们确实减轻了我们的负担。

## 建议 79：掌握 vector 和 string 与 C 语言 API 的通信方式

在建议 78 中，我们推荐使用 vector 和 string 代替动态分配的数组，其中的一个优点就是在保证自身优点的同时，与 C 内存模型兼容，它可以将本身的数据很简单地传递给 C 语言的 API。那么具体实现方式是怎么样的呢？这是我们应该掌握的。

一般而言，使用 vector:: operator[] 和 string::c\_str 是 STL 实现与 C 语言 API 通信的最佳方式。

vector 的存储区是连续的，所以我们可以直接通过第一个元素的地址获取 vector 数据的指针。正如下面的代码片段所示：

```
vector<int> intContainer;
...
int* pData = NULL;
pData = &(intContainer[0]);
```

当然，除了这种方式，我们还可以通过 begin()、front() 来获得：

```
pData2 = &(intContainer.begin()[0]);
// 或者 pData = &(*intContainer.begin());
pData = &(intContainer.front());
// 或者 pData = &((&intContainer.front())[0]);
```

如果获取的是第 n 个元素，只需要将上面的 0 换作 n 即可：

```
// 方式1
pData = &(intContainer[n]);
// 方式2
pData = &(intContainer.begin()[n]);
// 方式3
pData = &((&intContainer.front())[n]);
```

当然，一定得注意了，不要犯数组越界的错误。

对于 string，不要采用类似 vector 的方法，因为并不是所有的 string 实现采用的都是连续内存。幸好 string 提供了一个成员函数 c\_str，它返回一个以“\0”结束的 C 风格的字符串：

```
string str = "hello 2011";
const char* sz = str.c_str();
```

细心的读者也许会从 string 的成员函数列表中发现另一个新大陆：string::data。这是一个有些争议的函数，因为有些资料中认为这个 data 函数返回的虽然也是一个字符串，但是与 c\_str 相比，它返回的字符串没有结束符 “\0”。而在 VC2010 编译器所带的实现代码中，它却与 c\_str 一模一样，如下所示：

```
// 文件 xstring Line 1505
const _Elem *data() const
```

```
// return pointer to nonmutable array
return (c_str());
}
```

所以在VC2010编译环境中，如果想将string类型转换为C风格的char\*类型，string::data也是一个选择。至于在其他的开发环境中，我不做保证！

#### 请记住：

为了代码的复用，我们不得不应用或整合一些旧时代的代码，这并不是我们摒弃STL容器，特别是vector和string的理由。因为二者不仅能够而且善于与C语言的API通信交流。使用vector::operator[]和string::c\_str是实现STL容器与C语言API通信的最佳方式。

## 建议80：多用算法调用，少用手写循环

一提起容器元素的遍历操作，我们首先想到的就是使用for或while实现的循环：

```
for (iter = Container.begin(); iter != Container.end();)
{
    ... // do something
}

while(iter != Container.end())
{
    ... // do something
    ++iter;
}
```

这是很自然很淳朴的想法。但是当我们开始动手写这样的循环时，C++之父Bjarne Stroustrup的谆谆教导就会在耳边响起：多用算法，少用循环<sup>Θ</sup>。

用算法调用代替手工编写的循环，具有以下几方面的优点：

□ 效率更高

算法通常会比程序员产生的循环更高效。这体现在三个方面：

首先，STL算法是由实现标准容器的那帮专家实现的，基于他们对容器的了解，他们所写的算法的效率绝对不会比你我所实现的版本低，他们采用了库的使用者无法采用的方式来优化容器遍历。

其次，所有的STL算法使用的计算机科学比一般的C++程序员能拿得出来的算法都要复杂——有时候会复杂得多。

最后，使用函数调用可以减少不必要的函数调用，正如下面的代码片段所示：

<sup>Θ</sup> 源自《The C++ programming Language (Special 3<sup>rd</sup> Edition)》，Addison-Wesley出版，出版时间是2000年。

```

list<ContainerElement> Container;
list<ContainerElement>::iterator iter;

// 手写循环版
for (iter = Container.begin(); iter != Container.end();)
{
    iter->DoSomething();
}

// 算法调用版
for_each(Container.begin(),
          Container.end(),
          mem_fun_ref(&ContainerElement::DoSomething));

```

在手写循环版中，每次循环都会调用一次 list 成员函数 end，而算法调用版则可以避免这种不必要的重复调用。

#### □ 不易出错

手写循环比调用算法更容易产生错误，比如在循环中使用了无效迭代器。如果确定使用手写循环，那么就必须时刻关注它们是否被不正确的操纵了或变得无效了。而算法早已经历了无数的测试，也对一些常见的错误进行了调试，所以不必纠缠于一般的错误并为此烦恼不已。

还有什么理由舍近而求远，热衷于那些容易出错的手写循环呢？这真的是冒着不必要的危险，担着不必要的麻烦，艰难地穿行在由迭代器引起的荆棘丛中！

将迭代器扔给算法吧！让它们去考虑操纵迭代器时的各种诡异行为吧！

#### □ 可维护性更好

算法通常使代码比相应的显式循环表达力更强、更干净、更直观。而原始的 for 和 while 循环却无法透露出循环体的目的，要想获得这些内在的语义信息，必须阅读循环体。

“最好的软件应该是那些最清晰的、最容易懂的、容易增强和维护、适用于新环境的软件”，这是 Scott Meyers 说的，也是软件界高度认同的，这在 STL 算法中得到了极大地体现。我们可以从算法的名字中得到其功能的暗示，比如 insert 是插入、find 是查找、sort 是排序、for\_each 是遍历。而手写的循环却不能做到这一点。STL 使用高层次的术语取代了低层次的词汇，软件的抽象层次也在这样的改变中得到了提升，并因此而更容易实现、增强和维护。

虽然算法相较于手写循环有一定的优势，但也并不是说我们得完全将手写循环抛进历史的垃圾堆里。手写循环在一些情形下还是值得采用的，特别是一些简单的循环：

```

vector<int> intContainer;
for(size_t i=0; i<10; i++)
    intContainer.push_back(i*i);

```

如果在这种情况下颇费心机地去使用算法，给人的感觉就是杀鸡用牛刀、大才被小用，算法反而不如手写循环简单明了。

所以，尽量用函数调用代替手写循环，但绝不抛弃手写循环！

---

**请记住：**

相较于手写循环，STL 算法在效率、正确性、可维护性三方面具有一定的优势，并且 STL 算法丰富至极，涉及面也较为广泛，所以很多本来需要我们用手写循环来实现的任务可以通过 STL 算法调用来实现。所以我特别推荐采用 STL 算法。多用算法调用，少用手写循环。

---





## 第二部分

# 编码习惯和规范篇

### 本部分内容

- 第 8 章 让程序正确执行
- 第 9 章 提高代码的可读性
- 第 10 章 让代码运行得再快些
- 第 11 章 零碎但重要的其他建议

# 第 8 章 让程序正确执行

前辈 Brian Kernighan 曾经教导我们：

Make it right before you make it faster.  
Keep it right when you make it faster.  
Make it clear before you make it faster.

可见，程序的正确执行是我们的第一需求，也是最基本的需求。

如何让我们的程序正确执行，少崩溃少 Bug 呢？这是一个很老套的话题。要真正解决这一问题，正如卖油翁的那句“我亦无他，唯手熟尔”，也就是说需要日积月累的经验沉积。当然，本章的一些建议也会给大会一些帮助与启迪。

## 建议 81：避免无意中的内部数据裸露

相比于 C, C++ 的优点之一就是数据的封装。通过封装对内部数据进行了保护，只留下了一些合理的操作接口，避免了用户对内部数据的误操作。然而，如果不是足够小心，让接口传回了一个内部数据的句柄，那么我们所有的美好愿望都将化作泡影。

首先解释一下这里所说的“句柄”。我们可以将其理解为指向内部数据的句柄，它们可以是指针，也可以是引用。总之，得到它们与得到内部数据几无差异。这样的情况是怎么发生的呢？还是来看下面的示例代码：

```
class CString
{
public:
    CString(const char* sz);
    ~CString();
    char& operator[](int index) const;
    operator char*() const;
private:
    char* m_szData;
};

const CString words("Shit!! What a shame! ");
```

我们设计了一个字符串类 CString。因为球队在比赛中 0:4 的惨败让我愤怒地写下了上面的 words（“Shit，太丢人了”）。它被声明为一个 const 对象。之所以用 const 进行修饰，是希望 words 不会被改变。

然而，这个美好的愿望却被两个函数打破了，破得粉碎：

```
inline CString::operator char*() const
{
    return m_szData;
}
inline char& CString::operator[](int index) const
{
    return m_szData[index];
}
```

虽然这两个函数都加上了 `const`（为什么要加上 `const` 关键字呢？请仔细考虑一下，另外在建议 82 中也将会详细论述），但它们的设计还是存在错误：将内部数据 `m_szData` 的 handles 传到对象以外的世界中去了。虽然 `private` 限制了我们对 `m_szData` 的直接操作，但是这两个函数却让那些不怀好意的人们暗自窃喜。正如下面的代码片段所示，有人利用这两个接口对 `const` 的对象内容进行了修改：

```
char* szData = words;
strcpy(szData, "Smile to this failure, just a match");
char char_1 = words[0];
char_1 = 'O';
```

虽然他是好心好意劝我（字符串被他修改成了“微笑面对这场失利，仅仅一场比赛而已”），但是我还是不想让他获得“直接面对私有数据”的权利。这样的设计会让我们对数据封装的努力变得毫无意义。所以，我们必须做出改变，优化我们的函数设计。

按照 Scott Meyers 的提示，我们可以采取以下两种方式。

#### □ 去除函数的 `const` 属性

代码如下所示：

```
class CString
{
public:
    CString(const char* sz);
    ~CString();
    char& operator[](int index);
    operator char*();
private:
    char* m_szData;
};
```

此时如果还像原来那样操作 `const CString` 对象 `words`，编译器就会直接拒绝：

```
error C2440: “初始化”：无法从“const CString”转换为“char *”
error C2678: 二进制“[”：没有找到接受“const CString”类型的左操作数的运算符（或没有可接受
的转换）
```

但是像 `const CString` 对象转变成 `char*` 这样的需求也并不过分，所以这种“宁可错杀

三千，不可漏网一个”的做法需要谨慎对待。

#### □ 改变函数实现

对于第一个函数，传出原对象数据拷贝的句柄，`const` 属性保持不变：

```
CString::operator char*() const
{
    char* copy = new char[strlen(m_szData)+1];
    strcpy(copy, m_szData);
    return copy;
}
```

但是在这个实现中，我们动态申请了内存，所以不仅效率相对较低，而且还要注意不要产生内存泄露。

对于第二个函数，直接传回一个 `char`，同样保持 `const` 属性：

```
char& CString::operator[](int index) const
{
    return m_szData[index];
}
```

函数返回的不再是引用，而是 `m_szData[index]` 的一个拷贝。幸好，`char` 是一个内置数据类型，所以“引不引用”对于效率来说几乎毫无影响。

对于第一个函数的设计，我们还是有不少遗憾的。有没有两全其美的办法，既能满足我们的通用需求又可以保证不错的效率呢？

答案是：鱼与熊掌，可以兼得！所采取的策略就是：传回一个指向 `const char` 的指针，代码如下所示。

```
CString::operator const char*() const
{
    return m_szData;
}
```

---

#### 请记住：

对于 `const` 成员函数，不要返回内部数据的句柄，因为它会破坏封装性，违反抽象性，造成内部数据无意中的裸露，这会出现很多“不可思议”的情形，比如 `const` 对象的非常量性。

---

## 建议 82：积极使用 `const` 为函数保驾护航

`const` 源自英文 `constant`，即恒定不变。`const` 在 C++ 中绝对是一个神通广大的角色，被 `const` 修饰的东西都会受到强制保护，以预防其有意外的变动，从而提高程序健壮性。正如

在建议 25 中讲述的那样，`const` 变量之于宏具有不少的优点，然而这并不足以体现它的强大威力。当 `const` 遇到函数时，它的最大潜能才得到了最完美的体现。

`const` 的真正威力强在哪？一般说来体现在以下几个方面：

#### □ 修饰函数形式的参数

函数参数根据用途可分成两类：输出型参数和输入型参数。前者是将参数作为输出，此时不论它是什么数据类型，也不论它采用的是“指针传递”还是“引用传递”，都不能加 `const` 修饰，否则该参数将失去输出功能。所以，`const` 只能修饰输入参数。

如果输入参数采用的是“指针传递”，那么加上 `const` 修饰可以防止意外地改动该指针，起到保护作用。比如字符串拷贝函数：

```
void StringCopy(char *strDestination, const char *strSource);
```

其中 `strSource` 是输入参数，`strDestination` 是输出参数。给 `strSource` 加上 `const` 修饰后，如果函数体内的语句试图改动 `strSource` 的内容，编译器将指出错误。

如果输入参数采用的是“引用传递”呢？对于非内置数据类型，引用传递省去了临时对象的构造、复制和析构，与单纯的“值传递”相比其效率会有所提升。但是，它也会带来一定的副作用，比如，引用传递有可能改变参数。看下面的这个例子：

```
class A { ... };
void Function(A& para){ ... }

A object;
Function(object);
```

在函数 `Function` 中，`object` 作为输入参数是不该被改变的，但是如果改变了又会如何呢？通过实验我们知道，事实上，我们是可以改变 `object` 的，而且毫无惩罚。这时，`const` 就该派上用场了：

```
void Function(const A& para){ ... }
```

此时，在 `Function` 内部若有任何修改 `object` 的企图都会被禁止。

这里有一点需要强调一下，对于内置数据类型的输入参数，不要将“值传递”的方式改为“`const` 引用传递”，不要将 `void Function(const int x)` 写成 `void Function(const int &x)`。因为这样做的结果是既达不到提高效率的目的，又降低了函数的可读性。

#### □ 修饰函数返回值

一般情况下，如果函数返回值采用的是“值传递方式”，那么加 `const` 修饰是没有任何价值的。因为函数会把返回值复制到外部临时的存储单元中，用 `const` 修饰一个临时对象是多么滑稽的主意啊！但是也有例外，比如某些用户自定义类型中的二目操作符重载就是例外，因为此时大多会涉及新对象的产生。就像下面的复数类 `CComplex` 加法操作符 + 的重载：

```

class CComplex
{
    ...
private:
    float m_real;
    float m_imagin;

    friend CComplex operator+( const CComplex& lhs,
                               const CComplex& rhs );
};

CComplex operator+( const CComplex& lhs,
                     const CComplex& rhs )
{
    CComplex result( lhs.m_real + rhs.m_real,
                     lhs.m_imagin + rhs.m_imagin );
    return result;
}

```

按照这样的无 `const` 函数实现，当我们写下如下“毫无意义”的代码片段时，编译器会默不作声，放任纵容此种行为：

```

CComplex a(5.0f, 3.56f);
CComplex b(3.14f, 2.0f);
CComplex c;
(a + b) = c;

```

可是，这样的行为在 C++ 的世界中是不被接受的。如何杜绝这样的操作呢？可用 `const` 对返回的 `CComplex` 进行修饰：

```

const CComplex operator+( const CComplex& lhs,
                           const CComplex& rhs ){ ... }

```

这样的状况并非只发生在“返回值为对象”时，有时引用类型的返回值同样需要用 `const` 修饰，只不过这样的情形相对较少。这其中最主要的一个情形就是以只读方式获取类的非公有成员变量：

```

class Student
{
public:
    Student(string name);
    ...
    string& GetName(){ return m_szName; }
private:
    string m_szName;
};

```

我们对 `Student` 类的私有数据成员 `m_szName` 设置了 `Get` 函数，既保证代码的使用者能够获取 `Student` 对象的名字，又保证不能对其进行越权操作。然而当看到如下的代码时，你

会发现愿望总是美好的，现实却总是残酷的：

```
Student LiLei("Li Lei");
...
LiLei.GetName() = "Han MeiMei";
```

原本是男同学 Li Lei 却被人改成了可爱的女生 Han MeiMei，多么可恶的恶作剧啊！为了杜绝这样的行为，要为那些函数的返回值加上 const：

```
class Student
{
public:
    ...
    const string& GetName() { return m_szName; }
private:
    ...
};
```

如果函数返回值是一个指针，在加上 const 之后，函数返回值（即指针）的内容就不能被修改了，该返回值只能被赋给加上了 const 修饰的同类型指针。例如：

```
const char * GetString(void) { ... }
const char *str = GetString();
```

#### □ 修饰成员函数

用 const 修饰成员函数的目的也是提高程序的健壮性。const 成员函数不允许对数据成员进行任何修改，一旦这么做了，编译器就会抗议罢工，指出错误，如下所示：

```
class Student
{
public:
    ...
    void PrintDetail() const;
private:
    ...
};

void Student::PrintDetail() const
{
    SetName("Hello"); // ERROR1
    m_szName = "Hello"; // ERROR2
    ... // print
}
```

总体说来，关于 const 成员函数，须遵循以下几个规则：

- (1) const 对象只能访问 const 成员函数，而非 const 对象可以访问任意的成员函数。
- (2) const 对象的成员是不可修改的，然而 const 对象通过指针维护的对象却是可以修改的。

(3) const 成员函数不可以修改对象的数据，不管对象是否具有 const 性质。

#### 请记住：

const 的强大作用体现在对函数的修饰上：参数、返回值、成员函数本身，灵活使用 const 会使我们的代码更健壮、更安全！所以，积极大胆地使用 const，为函数保驾护航，这将给你带来很大的益处，尽量使用 const (Use const whenever you need)。

## 建议 83：不要返回局部变量的引用

局部变量和引用是我们非常熟悉的两个概念。引用，即别名，它必须指向一个有效对象；而局部变量，顾名思义，是在局部范围内有效的对象，一旦超出其生命周期，该对象就会灰飞烟灭。

不过，当这两者遇到一起，又会演绎怎样的故事呢？让我们拭目以待。

假设我们需要一个复数类 CComplex，并且支持复数的四则运算。这对于我们来说就是小菜一碟，写代码也是轻车熟路、顺手拈来：

```
class CComplex
{
public:
    CComplex(float real = 0, float imagin = 0)
        : m_real(real), m_imagin(imagin) { }
    ~CComplex() { }

private:
    float m_real;
    float m_imagin;

    friend const CComplex& operator+( const CComplex& lhs, const CComplex& rhs );
    friend const CComplex& operator-( const CComplex& lhs, const CComplex& rhs );
    friend const CComplex& operator*( const CComplex& lhs, const CComplex& rhs );
    friend const CComplex& operator/( const CComplex& lhs, const CComplex& rhs );
};

inline const CComplex& operator+( const CComplex& lhs, const CComplex& rhs )
{
    CComplex result( lhs.m_real + rhs.m_real, lhs.m_imagin + rhs.m_imagin );
    return result;
}
...

// 客户代码
CComplex a(1.02f,3.21f);
CComplex b(7.10f,5.44f);
```

```
CComplex c = a + b;
const CComplex& d = a + b;
```

写下上面的应用代码，在VC++中编译运行时，你会发现结果正在意料之中， $c = d = 8.12 + 8.65i$ 。可是，不要洋洋得意，如果你敏锐而细心，会发现IDE抛出了一个警告：

```
warning C4172: 返回局部变量或临时变量的地址
```

也就是说函数的实现并不完美。在函数operator+调用的过程中，都发生了什么呢？首先构造了一个局部变量result，接着生成result的别名，并作为函数的返回值抛出，最后局部变量超出生命周期，被销毁。而此时result的别名还存在，所以警告就产生了。

在C++标准中，临时对象的引用是没有定义的，所以结果也是未知的。之所以在VC++中得到了正确的结果，只能说明你运气较好，微软的工程师们对此作了特殊处理（说明：作者仅在VS2008和VS2010中进行了测试）。

问题的根源似乎找到了。主要原因就是返回的对象析构的时机有点早。也许有人会问：如果我们使用new出来的对象会怎样呢？代码如下所示：

```
inline const CComplex& operator+( const CComplex& lhs, const CComplex& rhs )
{
    CComplex *result = new CComplex( lhs.m_real + rhs.m_real, lhs.m_imagin + rhs.
        m_imagin );
    return *result;
}
```

根据前面的建议可知，new和delete应该是配对存在的。所以，在调用完上述函数后，为了避免内存泄露，我们还必须手动删除所分配的内存：

```
CComplex a(1,1);
CComplex b(1,4);

const CComplex& c = a + b;
delete &c;
```

这样方式有两个方面的缺点：

(1) operator+函数只申请内存，易造成内存泄露，特别是在链式操作时，增加了用户使用的负担。

(2) 内存的申请与删除不在同一模块中，将一个函数功能硬生生地分到了两个层次的函数中，这影响了模块功能的完整性与单一性，破坏了函数的内聚性。

所以，返回new出来的对象引用也是不可取的。

### 请记住：

局部变量的引用是一件不太靠谱的事儿，所以尽量避免让函数返回局部变量的引用。同时也不要返回new生成对象的引用，因为这样会让代码层次混乱，让使用者苦不堪言。

## 建议 84：切忌过度使用传引用代替传对象

相较于传对象，传引用的优点我们已经熟记于心：它减少了临时对象的构造与析构，所以更具效率。但是我们不能因此而患上了效率强迫症，完全用传引用代替传对象。也就是要把握一个度，切忌过度盲目地使用传引用代替传对象。

引用，是一个已有对象的别名。传引用的前提是存在一个已有的对象。函数中产生对象有两种基本方式：以 CComplex x; 方式在栈中生成或以 new CComplex(); 形式在堆上获得。代码如下所示：

```
class CComplex
{
    ...
    friend const CComplex operator+( const CComplex& lhs, const CComplex& rhs );
};

// 以 CComplex x; 方式在栈中生成
const CComplex operator+( const CComplex& lhs, const CComplex& rhs )
{
    CComplex result( lhs.m_real + rhs.m_real, lhs.m_imagin + rhs.m_imagin );
    return result;
}

// 以 new CComplex(); 形式在堆上获得
const CComplex operator+( const CComplex& lhs, const CComplex& rhs )
{
    CComplex* pResult = new CComplex( lhs.m_real + rhs.m_real, lhs.m_imagin + rhs.
        m_imagin );
    return *pResult;
}
```

是不是有些似曾相识呢？它就是建议 83 中的示例代码。在这段代码的 operator+ 重载函数中，返回的值或为临时对象，或为堆对象。这两种方式都不能用传引用代替，一旦用传引用代替传对象，就又回到了建议 83 所讲述的问题上了。

既然这两条路都被堵死了，那么如果采用 single—static 的方式会怎样呢？明确地告诉你：此路亦不通。主要原因是它会引起一种隐藏极深的 Bug：

```
const CComplex& operator+( const CComplex& lhs,
                           const CComplex& rhs )
{
    static CComplex s_complex;
    ... // 为 s_complex 赋值
    return s_complex;
}

// 客户端代码
```

```
const CComplex& c = a + b;
const CComplex& f = d + e;
```

当执行完第二条语句时，对象 c 的值也会随之发生变化。所以当写下类似如下的代码时：

```
CComplex a, b, c, d;
if( (a+b) == (c+d) ) ...
```

就如同写下了 if( true )...，而与 a、b、c、d 的具体值毫无关联了。所以，这样的方式也是不甚合理的。

至此，我们企图用传引用代替传对象的努力都宣告失败。所以，不要过分地热衷于用传引用替换传对象。盲目采用传引用所带来的麻烦将远甚于它所带来的效率改善，丢了西瓜捡起芝麻的故事又将上演。

#### 请记住：

传引用有优点，但是也并非任何时候都有效。审慎地考察具体应用情形，审慎地使用传引用替代传对象。必须传回内部对象时，就传对象，勿传引用。

### 建议 85：了解指针参数传递内存中的玄机

参数扮演的角色是多样的，既可以作为输入，也可以作为输出，比如用指针参数传递函数内部动态申请的内存。这是一个比较容易出错的知识点，因此也就成为了各家 IT 公司面试时出现频率极高的笔试题目之一。先看如下示例代码：

```
void GetMemory1(char *pStr, int num)
{
    pStr = new char[num];
}

int main()
{
    char *strHello = NULL;
    GetMemory1(strHello, 100);
    strcpy(strHello, "Hello C++ Tips");
    delete[] strHello;
    return 0;
}
```

如果单从代码上看，貌似一切都很正常。但是当我们按下 F5 运行程序时，Bug 就出现了：

0xC0000005: 写入位置 0x00000000 时发生访问冲突

从错误中我们可以看出，GetMemory(strHello, 100) 并没有使 strHello 获得期望的内存，

strHello 依旧是 NULL；所以在向 strHello 写入数据时，程序崩溃。

我们分析一下参数的传递过程，以便从中找出问题的所在：

针对函数的每个参数，编译器都要为其生成一个临时副本，指针参数 pStr 的副本是 \_pStr，并且会使 \_pStr = pStr = 0x00000000。在函数体内的程序修改了 \_pStr 的内容，也就是改变了 \_pStr 指向的内存地址，而 pStr 却丝毫未变，仍旧是 0x00000000。所以函数 GetMemory 不能输出任何东西。不仅如此，每执行一次 GetMemory 就会造成一块内存的泄露。

如果非得要用指针参数去申请内存，那么就应该改用“指向指针的指针”。代码片段如下所示：

```
void GetMemory2(char **p, int num)
{
    *p = new char[num];
}
int main()
{
    char *strHello = NULL;
    GetMemory2(&strHello, 100);
    strcpy(strHello, "Hello C++ Tips");
    delete[] strHello;
    return 0;
}
```

为什么“指向指针的指针”就能实现预想的功能呢？这个理解起来有些难度，还得分析一下参数的传递过程：

实参 str 传入函数 GetMemory 时，传入的是存放指针 str 的地址 address\_str，编译器为其生成临时副本 \_address\_str，并且 \_address\_str = address\_str = 0x000f1850（可能是任意非 NULL 地址），因此在执行函数语句之前，\_address\_str 指向的也是 0x00000000。在函数内部，程序修改了 \_address\_str 所指向的内存地址，因为二者应指向相同的地址，所以 address\_str 所指向的地址也就会有相应的改变。那么采用指针的指针这种方式就可以实现动态内存的传递了。

当然更容易的方法还是用返回值来传递动态内存，代码如下所示：

```
char *GetMemory3(int num)
{
    char * p = new char[num];
    return p;
}
```

这里需要特别强调的是，不要用 return 语句返回指向“栈内存”的指针，因为该内存 在函数结束时会自动消亡。

现在将这个话题扩展开来，如果想在函数内部对传入参数的值进行修改，那么单纯地传入这个参数是绝对不行的，需要传入的必须是欲改变量的指针。举例来说，如果参数本身是

`int`, 那么传入的类型应该是 `int*`; 如果参数本身是 `float*`, 那么传入的类型应该是 `float**`。还是得拿出那个老套的数值交换 Swap 的例子来进一步说明, 代码如下所示:

```
// 错误版本, swap 前后值不发生变化
void Swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

// 正确版本(当然也可以用引用实现)
void Swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

所以准确地说, 在本节之初的那个错误示例中, 传的不是指针, 而是值, 只不过恰恰是一个指针值而已。

#### 请记住:

用指针参数传回一块动态申请的内存, 是很常见的一种需求。然而如果不甚小心, 就很容易造成严重错误: 程序崩溃 + 内存泄露! 解决之道就是用指针的指针来传递, 或者换种内存传递方式, 用返回值来传递。

## 建议 86: 不要将函数参数作为工作变量

工作变量, 顾名思义, 就是在函数实现中使用的变量。因为是在函数实现中, 所以会相对频繁地修改工作变量。在建议 85 中我们说到了函数指针参数传递动态申请的内存问题: 虽然函数参数在传入函数时, 编译器都会为其生成一个临时拷贝, 但是对拷贝的修改有时也会改变参数内容。所以如果将函数参数作为工作变量来使用, 那么函数参数很有可能会被改变。一旦错误地改变了参数内容, 那么结果将会是很危险的。

所以, 应该防止将函数参数作为工作变量。而对于那些必须改变的参数, 最好先用局部变量代替之, 最后再将该局部变量的内容赋给该参数。这样在一定程度上保护了数据的安全。

如下的函数实现方式不是我们推荐的:

```
void DataSum( unsigned int num, int *data, int *sum )
{
```

```

*sum = 0;

for (unsigned int count = 0; count < num; count++)
{
    *sum += data[count];
}
}

```

更好的方式应该是这样：

```

void DataSum( unsigned int num, int *data, int *sum )
{
    int sum_temp = 0;

    for (unsigned int count = 0; count < num; count++)
    {
        sum_temp += data[count];
    }

    *sum = sum_temp;
}

```

---

请记住：

一句话：为了数据安全，最好不要将函数参数作为工作变量。

---

## 建议 87：躲过 0 值比较的层层陷阱

这是众多 C++ 程序员都认为比较简单，但依旧容易犯错的问题。所以在这里我认为还是有必要提一提。

看下面的一段代码：

```

BYTE count = 100;

while( count >= 0 )
{
    ... // do something
    count--;
}

```

就这么简单的几行代码却出现了一个大问题。有人能发现它吗？

代码本意是让 while 循环体内的代码执行 100 次后终止，然而事与愿违，while 成了一个死循环，非暴力不能停止。看到这儿，也许有人会迅速地想起 BYTE 的那个定义：

```
typedef unsigned char BYTE
```

BYTE 的取值范围是 [0, 255]，其中的任何值都满足 while 循环的判断条件。所以要想实现原本设计的功能，count 的数据类型就必须改变一下，由 BYTE 变为 char。

这只是 0 值比较陷阱中的冰山一角，也是那最容易看到的一角。接着看下面的案例。Fay 是一名数学爱好者，他先后进行了 1000 次实验来验证圆周率，现在他要在这一堆实验数据中筛选出与理论值相等的那些实验编号，所以就有了以下的代码片段：

```
const float PI = 3.1415926f;
const int TEST_COUNT = 1000;

float TEST_DATA[TEST_COUNT] = ...; // Read from file
vector<int> Result;
for (int i = 0; i < TEST_COUNT; i++)
{
    float data = TEST_DATA[TEST_COUNT];
    if(data - PI == 0 )
        Result.push_back(i);
}
```

然而令 Fay 不解的是，Result 中竟然没有一个数。难道是这 1000 次实验的数据出了问题？不，问题还是出在了 0 值的比较上，即浮点 0 的比较。也许这时有人会想起，在计算机世界中根本不存在浮点零，所谓的浮点零只不过是一个很小很小、十分趋近于 0 的值。所以 if(data - PI == 0 ) 的判断是没有意义的。正确的做法是：

```
const float FLOAT_ZERO = 0.00000001;
...
for (int i = 0; i < TEST_COUNT; i++)
{
    float data = TEST_DATA[TEST_COUNT];
    if( abs(data-PI) <= FLOAT_ZERO )
        Result.push_back(i);
}
```

最后还要重复一个前面已经讲过的有关零值比较的问题：

```
if( student_count = 0 )
{
    cout<<" 现在还没有学生 "<<endl;
}
else
{
    cout<<" 开始上课 ..." <<endl;
}
```

当这样的系统交付学校使用时，即使一个学生也没有，课还是开始了。究其原因，还是与零值比较难脱干系。if( student\_count = 0 ) 中本该出现的比较操作符 == 被赋值操作符 = 替代了，所以这句话就变成了 if( 0 )，这样对应块内的程序将永远不会被执行。要避免这样的

零值比较问题，最好的方法是“细心 + 好习惯”，比如如下代码：

```
if( 0 == student_count )
...
```

如果不小心将 `==` 变成了 `=`，编译器就会因此提示错误：

```
error C2106: "=" : 左操作数必须为左值 (VS2010)
```

综上所述，零值有陷阱，比较须谨慎。

#### 请记住：

零值有陷阱，比较须谨慎。一般要特别关注三方面：(1) 0 在不在该类型数据的取值范围内？(2) 浮点数不存在绝对 0 值，所以浮点零值比较需特殊处理；(3) 区分比较操作符 `==` 与赋值操作符 `=`，切忌混淆。

### 建议 88：不要用 `reinterpret_cast` 去迷惑编译器

`reinterpret_cast`，简单地说就是保持二进制位不变，用另一种格式来重新解释，它就是 C/C++ 中最为暴力的类型转换，所实现的是一个类型到一个毫不相关、完全不同类型的映射。`reinterpret_cast` 仅仅重新解释了给出对象的比特模型，它是所有类型转换中最危险的。来看下面的代码片段：

```
class A { ... };
class B { ... };
void Function()
{
    A* pa = new A;
    void* pv = reinterpret_cast<A*>(pa);
    B* pb = (B*)pv;
    ...
}
```

如果类 A 和 B 的内存分布不清楚，贸然使用 `reinterpret_cast`，那么结果将会很恐怖。所以，使用 `reinterpret_cast` 一般会有一个假设前提条件：对于对象的内存分布，程序员知道的比编译器要多。只要程序员认为可行，编译器就放弃检查，按照程序员的吩咐去操作。所以，保证其正确的方法就是程序员足够明智。

一般的类型转换编译器会根据类型信息进行一些分析、处理，例如：

```
int i=2011;
double d=static_cast < double > (i);
```

变量从 `int` 转换到 `double` 的过程中，`static_cast` 需要将整数 2011 转换为双精度整数

2011，然后再为双精度整数 d 补足比特位，得到结果 2011.0。这是我们所熟悉的转换过程。

而 `reinterpret_cast` 的行为却很怪异：

```
int i = 2012;
double d=reinterpret_cast<double &>(i);
```

在转换过程中，`reinterpret_cast` 仅仅是复制 i 的比特位到 d，缺乏必要的分析，所以在进行计算以后 d 中包含了无用值。这严重违背了 C++ 的类型安全性原则。

此外，使用这个操作符的类型转换时，其转换结果几乎都是执行期定义的。因此，使用 `reinterpret_casts` 的代码很难移植。

当然，`reinterpret_casts` 除了赋予我们重新解释比特位的权利之外，并非毫无价值。它最普通的用途就是在函数指针类型之间进行转换。

例如，假设有一个函数指针数组：

```
//FuncPtr: 指向函数的指针，该函数没有参数，返回值为 void
typedef void(*FuncPtr)();
//funcPtrArray 是一个能容纳 10 个 FuncPtrs 指针的数组
FuncPtr funcPtrArray[10];
```

如果想在 `funcPtrArray` 中直接存入如下函数的指针，确实行不通。因为返回值的类型不一致：

```
int doSomething();
funcPtrArray[0] = &doSomething;//ERROR！类型不匹配
```

而 `reinterpret_cast` 可以让你梦想成真：

```
funcPtrArray[0] = reinterpret_cast<FuncPtr>(&doSomething);
```

它会告诉编译器以返回 `void` 的形式去看待函数 `doSomething`。

所以，不要用 `reinterpret_casts` 迷惑编译器，尽量避免使用 `reinterpret_casts`，除非是在其他转换都无效的非常情形下。记住 Henry Spencer 的这句话：欺骗编译器的人，最终将自食恶果。

**请记住：**

还是一句话：`reinterpret_casts` 是一个极为危险的家伙，尽量少去招惹它，除非必须。

## 建议 89：避免对动态对象指针使用 `static_cast`

`static_cast` 在用于类层次结构中基类和子类之间指针（或引用）的转换时有一个显著特点：进行上行转换（把子类的指针或引用转换成基类表示）是安全的；进行下行转换（把基

类指针或引用转换成子类表示)时,由于没有动态类型检查,所以是不安全的。

这短短几十个字的规则虽然记起来简单,但是应用起来确实有些难度,因为我们要时刻关注 `static_cast` 转换是安全的还是危险的,劳心费力!当遇到这种情形时,最应想到的应该是用 `dynamic_cast` 代替 `static_cast`。因为 `static_cast` 映射范围更宽,这种没有限制的映射伴随着更多的不安全,而 `dynamic_cast` 在处理基类和子类之间的转换时技高一筹:在类层次间进行上行转换时, `dynamic_cast` 和 `static_cast` 的效果是一样的;在进行下行转换时, `dynamic_cast` 具有类型检查的功能,比 `static_cast` 更安全。

比如下面的示例代码:

```
class B { ... };
class D : public B { ... };

void Function(B* pb)
{
    D* pd1 = dynamic_cast<D*>(pb);

    D* pd2 = static_cast<D*>(pb);
}
```

如果 `pb` 指向一个 `D` 类型的对象, `pd1` 和 `pd2` 是一样的,并且对这两个指针执行 `D` 类型的任何操作都是安全的。但是,如果 `pb` 指向的是一个 `B` 类型的对象,那么 `pd1` 将是一个指向该对象的指针,而对它进行 `D` 类型的操作则将是不安全的,因为它会无中生有地创造出 `D` 所特有的那部分信息,而 `pd2` 将是一个空指针。

除此之外,相比于 `static_cast`, `dynamic_cast` 的效率更低。所以,用 `dynamic_cast` 代替 `static_cast` 只能是在效率要求不是很高的场合下使用。

当然,这并不是最佳方案。最好的方法应该是分析调用链,理清类型信息流向,对代码进行重构或重新设计,尽量消除这种不好的向下转换。但是这需要深厚的经验积累,是一个可以意会而不可言传的至高境界,不是一般新手力所能及的。所以我们要做的就是,努力学习,抓紧实践,加快从新手到老手、到高手的转变。

#### 请记住:

在类层次结构中,用 `static_cast` 完成基类和子类指针(或引用)的下行转换是不安全的。所以尽量避免对动态对象指针使用 `static_cast`,可以用 `dynamic_cast` 来代替,或者优化设计,重构代码。

## 建议 90: 尽量少应用多态性数组

在函数设计中,我们经常会发现一些数组形式的参数,例如:

```
void Function(const int array[], int count) { ... }
```

函数参数中的 `const int array[]` 等同于 `const int array*`，在这种情况下，数组和指针有着异曲同工之妙。如果参数类型是内置类型，这里没有任何问题。但如果是用户自定义类型，这样的函数设计形式就会埋下错误隐患，如下所示：

```
class Base {};

void Function(const Base array[], int count)
{
    for(int i = 0; i < count; i++)
    {
        // using array[i] to do something
    }
}
```

当给函数传递一个含有 `Base` 对象的数组变量时，该函数不会出什么问题，代码如下所示：

```
Base array[12];
...
Function(array, 12);
```

因为在函数中 `array[0]` 等同于 `*( array )`，`array` 指向的是数组的起始地址，而 `array[i]` 也就等同于 `*( array + i*sizeof(Base) )`。通过这种方式，编译器会识别出这些连续的对象。

然而当函数 `Function` 涉及对象的多态时，Bug 就产生了：

```
class Derived : public Base{};
Derived array2[10];
...
Function(array2, 10);
```

类的多态特性是通过指针或引用来实现的，换句话说就是通过基类指针或引用来操作派生类以获得多样的行为。然而到了 `Function` 函数中，这种用基类指针实现派生类操作的企图将宣告失败。当我们把派生类对象数组传给 `Function` 函数时，编译器在分析对象的过程中会产生错误，它们在计算 `array[i]` 的起始位置时并不能智能地将 `Base` 替换为 `Derived`，即编译器依旧认为 `array[i]` 等同于 `*( array + i*sizeof(Base) )`，而不是 `*( array + i*sizeof(Derived) )`，它会将内存中的 `Derived` 对象当成 `Base` 对象来处理。如果出现这样的问题，编译器并不能及时地告诉你停止程序的执行，其输出结果将是未知且不愉快的。

基类指针和派生指针间的可替代性允许我们通过基类指针来操作派生类对象，这也是类多态特性的一个重要基础，而 C 语言时代的指针运算却与指针类型密切相关，于是二者之间的不可协调引发了上述问题。

现有的技术手段很难给出一个比较满意的解决方式，所以建议：尽量少地应用多态性数组。

请记住：

多态性数组一方面会涉及 C++ 时代的基类指针与派生类指针之间的替代问题，同时也会涉及 C 时代的指针运算，而且常会因为这二者之间的不协调引发隐蔽的 Bug。所以，谨慎地选择使用多态数组！

## 建议 91：不要强制去除变量的 const 属性

在 C++ 中，`const_cast<T*>(a)` 一般用于从一个类中去除以下这些属性：`const`、`volatile` 和 `_unaligned`。比如下面的这段代码：

```
class A { ... };
void Function()
{
    const A *pA = new A;      //const 对象
    //Compile ERROR. 不能将 const 对象指针赋值给非 const 对象
    A *pb = pa;

    // Fine. It is OK.
    A *pC = const_cast<A*>(pa);
    ...
}
```

`const_cast` 功力了得，就如同现实中的野蛮拆迁队，无论是常量指针、常量引用，还是常量对象，只要派它上场，拆除 `const` 保护将是轻而易举的事。通过 `const_cast`，常量指针可以被转化成非常量指针，并且仍然指向原来的对象；常量引用可以被转换成非常量引用，并且仍然指向原来的对象；常量对象也可以被转换成非常量对象，但会生成一个新的对象副本。

这样做虽然能够让我们的程序暂时编译通过，但是会带来非常严重的后遗症。`const` 本来给我们的变量加上了一层特殊的保护，如果使用 `const_cast` 强制地去除变量的 `const` 属性，不仅会使这层保护失效，而且有可能会导致程序崩溃。所以，先人“勿以恶小而为之”的教诲不仅适用于现实生活中，在 C++ 世界里一样也要遵守。

首先，看一个使 `const` 保护失效，而错误修改变量的例子：

```
class Student
{
public:
    Student(string name = "", int scorer = 0)
        :m_strName(name), m_nScorer(scorer){}
    ~Student() {}
    void SetScorer(int scorer){ m_nScorer = scorer; }
private:
    string m_strName;
```

```

    int m_nScorer;
};

void Function()
{
    const Student* pExample = new Student("三好学生", 90);
    // pExample->SetScorer(20);    // ERROR
    Student *pStudent = const_cast<Student *>(pExample);
    ... // do something
    pStudent ->SetScorer(20);    // Fine
    delete pStudent;
}

```

三好学生是有标准的，而且这个标准是不能随便修改的，特别是学习成绩一项（至少90分）。所以，我们将这个榜样设为 `const`，仅供大家学习参照。但是，如果有人为了一时的方便，应用 `const_cast` 强行去除了该对象的 `const` 属性，并在无意中修改了这个标准。那么，到时候会发现突然间人人都是三好学生了。

也许有人对这样的名头毫不在意，认为存在上面的这种问题还可以忍受，但是当你遇到下面这种情况时，你还会这么淡然吗？对于本身定义时为 `const` 的类型，应用 `const_cast` 去掉了 `const` 属性，然后在对这块内容进行 `write` 操作时，程序罢工：

```

const char* pStr = "Hello, 2012.";
char* sz = const_cast<char*>( pStr );
sz[0] = 'A';

```

虽然编译通过，但是当执行到第三条语句时，问题就出现了，系统不允许对其地址进行 `write` 操作。这其中的原因一般在于编译器在处理常量数据时，会将其存放在只读存储器（ROM）或具有写保护的随机访问存储器（RAM）中。如果试图对这种物理上的 `const` 对象进行 `write` 操作，其表现多为内存故障。

所以，不要使用 `const_cast` 强制地去除变量的 `const` 属性。

#### 请记住：

强制去除变量的 `const` 属性虽然可以带来暂时的便利，但这不仅增加了错误修改变量的几率，而且还可能会引发内存故障。所以，勿以恶小而为之！

# 第9章 提高代码的可读性

“任何傻瓜都可以写出让计算机理解的代码，而优秀的程序员则可以写出让人类明白的代码”。<sup>①</sup>

这是 Martin Fowler 说的一句话，换句话说就是：写出机器明白的代码容易，但写出人们容易理解的代码就比较难了。其实，这里所指的就是代码的可读性。代码可读性是计算机编程世界中一个普遍且重要的主题。在编程领域中，可读性指的是：人类读者对于源代码的功能意图、流程控制和操作运行是否容易把握<sup>②</sup>。

我们知道，代码是由人编写维护，由机器执行的。所以在对代码质量的评价体系中必须包含可读性和运行时效率两个维度。而这两者在某种程度上是矛盾的。在硬件资源相对有限的早期，我们希望编译后可执行程序的 size 更小，运行更快；而如今，在大多的应用需求中，机器资源不再是问题，不过，随着系统的规模逐渐庞大复杂起来，维护问题也就显得更加重要了，所以行业内对代码可读性的要求愈加强烈。于是就有了如下的代码片段<sup>③</sup>：

```
/**  
 * Code Readability  
 */  
if (readable()) {  
    be_happy();  
} else {  
    refactor();  
}
```

有研究<sup>④</sup>发现，一点点简单的可读性改造，也能让代码变得更为简短，并且大大缩短了看懂所需的时间。良好的代码可读性不仅可以使代码外观整洁、清晰，改善代码的描述，使其更富条理性，并且可以提高代码的可维护性，减少软件工程的开发维护成本。对于代码可读性的研究不仅仅局限在工程界中，在学术界它也是一个研究方向。各国的研究者们力求找到一个合适的数学模型来对代码的可读性进行评价。在 Raymond P.L. 提出的模型<sup>⑤</sup>中主要包

<sup>①</sup> “Any fool can write code that a computer can understand. Good programmers write code that humans can understand”. ——Martin Fowler

<sup>②</sup> 此乃维基百科的定义。

<sup>③</sup> 源自《Top 15+ Best Practices for Writing Super Readable Code》。

<sup>④</sup> James L. Elshoff , Michael Marcotty, *Improving computer program readability to aid modification*, Communications of the ACM, v.25 n.8, p.512-521, Aug 1982.

<sup>⑤</sup> Ray mond P.L.Buse,Westley Weirner *Learning a Metric for Code Readability*, IEEE Trans.Software Engzneering Vol.36 No.4, 546-558, July/Aug 2010.

含以下几个因素（如图 9-1 所示，其中 # 代替“…的个数”）。

Avg.	Max.	Feature Name
✓	✓	line length (# characters)
✓	✓	# identifiers
✓	✓	identifier length
✓	✓	indentation (preceding whitespace)
✓	✓	# keywords
✓	✓	# numbers
✓	✓	# comments
✓	✓	# periods
✓	✓	# commas
✓	✓	# spaces
✓	✓	# parenthesis
✓	✓	# arithmetic operators
✓	✓	# comparison operators
✓	✓	# assignments (=)
✓	✓	# branches (if)
✓	✓	# loops (for, while)
✓	✓	# blank lines
	✓	# occurrences of any single character
	✓	# occurrences of any single identifier

图 9-1 代码可读性的影响因素

在本章中，将针对上述主要因素，阐述提高 C++ 代码可读性的 9 个建议。

## 建议 92：尽量使代码版面整洁优雅

整洁优雅的代码犹如大师的书法作品，版面设计也是很受重视的。想要写出整洁优雅的好代码，需要注意以下几点：

### □ 避免代码过长

我们的眼睛其实是很挑剔的，在阅读时，每一行文字的长度会直接影响我们的视觉感受。也正是基于这个原因，报纸文章才会按照如图 9-2 所示的方式排版。

在图 9-2 中，避免了一行内容的文字数过长，让我们的视觉感受更舒适，这是报刊出版业遵守的一条准则。这条准则之于计算机代码编写同样成立：不要编写过长的代码，这是一个最佳实践。

如果存在较长的语句（一般而言大于 80 字符），那么就要将其分成多行内容来书写。在



图 9-2 报纸版面图

Θ 此图取自《Top 15+ Best Practices for Writing Super Readable Code》。

长表达式划分时，以低优先级操作符为界，该操作符放在新行之首，并将新行进行适当的缩进，使排版更为整齐，提高语句的可读性，如下所示。

```
bool error_flag = prcessing_task_id <= MAX_TASK_ID  
    && used_time_length < TIME_OUT_LENGTH  
    && return_number == TASK_FINISH_SUCEEFULLY;  
  
class Student  
{  
public:  
    Student(string name = "", int scorer = 0, int age = 0)  
        : m_strName(name),  
          m_nScorer(scorer),  
          m_nAge(age){}  
  
    ...  
private:  
    string m_strName;  
    int m_nScorer;  
    int m_nAge;  
};
```

如果在循环、判断等语句中有较长的表达式或语句，同样需要进行适应的划分，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，如下所示：

```

for( too_long_parament_initialization;
    too_long_parament_condition;
    too_long_parament_update)
{
    DoSomething();
}

if( prcessing_task_id <= MAX_TASK_ID
    && used_time_length < TIME_OUT_LENGTH
    && return_number == TASK_FINISH_SUCEFULLY )
{
    DoSomething();
}

```

如果函数或过程中的参数较长，也要进行适当的划分，如下所示：

```
for_each( Container.begin() ,  
          Container.end() ,  
          mem_fun_ref( &ContainerElement::DoSomething ) );
```

## □ 代码缩进和对齐

代码缩进可以让我们很容易地对程序进行分块。所以程序块要采用缩进风格来编写，缩进的空格数为 4 个。建议对齐与缩进只使用空格键，而不要采用 TAB 键。因为 TAB 键在不同的编辑器中所代表的空格数目是不一样的，使用 TAB 会造成程序布局不整齐。

程序块的分界符（“{”和“}”）应独占一行并且位于同一列，同时应与引用它们的语句左对齐，如下所示：

```
if ( ... ) { // 不好的版式风格
    ...
}

if ( ... ) // 推荐的版式风格
{
    ...
}
```

在函数体的开始、类的定义、结构的定义、枚举的定义及 if、for、do、while、switch、case 语句中的程序都要采用缩进方式。

#### □ 空行分隔段落

空行起着分隔程序段落的作用，合理地加入空行将使程序的布局更加清晰。

相对独立的程序块之间、变量说明之后必须加空行。比如每个类的声明之后、函数定义之后，以及同一个函数体内相对独立的代码块之间都应该加上空行以示分隔，如图 9-3 所示。

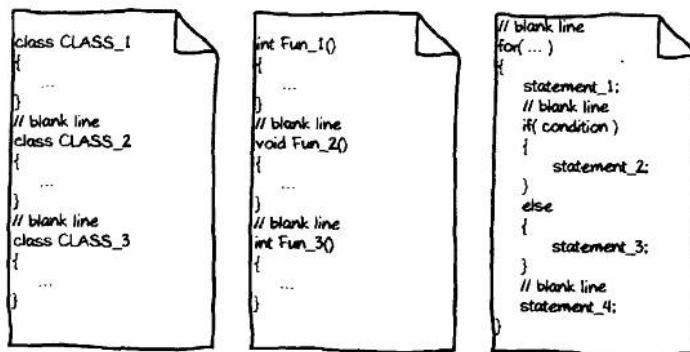


图 9-3 空行分隔段落示例

#### □ 使用空格

使用空格的目的是使代码更加清晰。当两个以上的关键字、变量、常量进行对等操作时，在它们之间的操作符的前面、后面加空格，如下所示：

```
for(int i=0;i<COUNT;i++)      // 不好的版式风格
for (int i=0; i<COUNT; i++) // 推荐的版式风格

void Function(int x,int y,int z); // 不好的版式风格
void Function(int x, int y, int z); // 推荐的版式风格
```

像 const、virtual、inline、case 等关键字之后至少要留一个空格，否则关键字就不再是关键字了，无法解析；而像 if、for、while、catch 等关键字之后应留一个空格再跟上左括号“(”，

以使关键字更加突出更加明显。

若是碰到逗号 “,”，只在其后加空格即可；如果分号 “;” 不是一行的结束符，在其后面也应加空格。

比较操作符，赋值操作符 “=” 和 “+=”、算术操作符 “+” 和 “%”、逻辑操作符 “&&” 和 “&”、位域操作符 “<<” 和 “^” 等双目操作符的前后应加空格。

“!”、“~”、“++”、“--”、“&”（地址运算符）等单目操作符的前后可以不加空格。

“->”、“.” 前后不加空格。

#### □ 语句行

不允许把多个短语句写在一行中，即一行只写一条语句。

if、for、do、while、case、switch、default 等语句自占一行，且 if、for、do、while 等语句的执行语句部分无论有多少内容都要加括号 {}。

关于美的讨论是一个仁者见仁智者见智的问题，在每个人的心中都有着属于他自己的一个定义与答案，对于代码之美亦然。但是上面这几条关于版面的建议也可以算是 C++ 程序员之间一个不成文的规定，是为了让版面优雅而达成的共识，所以最好还是遵守这些通用的规则吧！

#### 请记住：

注意代码的版面设计，让代码不仅“心”美“貌”也美。

## 建议 93：给函数和变量起一个“能说话”的名字

变量与函数是代码中最基本的两个元素，通过什么对它们进行区分呢？那就是名字，不同的变量或函数都有一个属于它们自己的、独一无二的名字。好的变量名和函数名可让阅读代码的人马上就知道该变量或函数的作用，很容易就能理解程序的大概结构和功能。可以说，好的名字是会说话的。

那么怎样的名字才能说话呢？先看一个程序员的恶作剧：

曾经有一个特别好吃的程序员，在他的代码作品中所有的变量名字都是用不同的食物来命名的，比如 Apple、Chocolate、Beef……然而有一天，他离开了这家公司，他的工作交接给了另一个程序员。这样的命名方式对接手者来说无疑是个悲剧，他看不出这些食物之间是一个什么样的关系，因此他花费了很长的时间对这些“食物”进行翻译，还为此专门制作了一个变量名称对应表。像这样的密码式命名方式无疑会增加代码的维护成本，削弱代码的可读性。

密码式的命名方式是不能说话的，它仅仅是一个名字而已，能说话的名字应该能告诉我们更多的信息。所以，不要天书秘文，应给函数、变量起一个能表意、会说话的名字。

在这么多年的开发实践中，我们已经形成了一些通用著名的命名规则，这其中最为流行的当属匈牙利命名法则。著名的 MS 公司就是采用的这套规则。匈牙利命名法的主要思想是“在变量和函数中加入前缀以增加人们对代码的理解”。

关于命名，至今也没有一个固定的规则，也许永远都不会有。所以，我们给出的只能是一个普遍规则，而不是绝对规则。

(1) 名称必须直观，可望文生义，不必解码。上面的食物变量就是最好的反面教材。

(2) 长度要符合“min\_length && max\_information”（最小名长度最大信息量）的原则，要用最少的字符表示最全的信息。

(3) 与整体风格保持一致。这里说的整体风格包括：所采用操作系统的代码风格、所采用开发工具的代码风格、应用程序原有的代码风格等。比如，在 Windows 和 Unix 中开发的应用程序会有不同的代码风格：Windows 中喜欢骆驼命名法，而 Unix 中则偏好下划线命名法，如下所示：

```
int studentCount = 0; // 骆驼命名法
int student_count = 0; // 下划线命名法
```

这两种方式没有优劣之分，没有明显“最好的”样式；所以我们需要做的就是尽量保持它们的一致性。

(4) 变量名称应该是一个“名词”，或者是“形容词 + 名词”；而函数名称应该是“动词 + 名词”的组合。

(5) 杜绝仅靠大小写来区分的名称标识符。

(6) 变量名之前附加前缀用来识别变量类型，具体前缀如表 9-1 所示。

表 9-1 变量附加前缀说明

类型	前缀	备注
short/int/long	n	带符号整数
unsigned short		
unsigned integer	u	无符号整数
unsigned long		
char/unsigned char	c	字符
float,double	f	浮点数
bool	b	布尔量
char*/unsigned char*	sz	程序中明确作为字符串使用
WORD	w	Windows 平台专用
DWORD	dw	Windows 平台专用
指针	p	一般 p 后面会跟随一个指示具体类型的前缀
指针的指针	pp	其他更多指针依此类推

(7) C++ 类或结构的成员变量附加前缀“m\_”；全局变量名称附加前缀“g\_”。

(8) 单字符变量只能用作循环变量。变量和参数是用小写字母开头的单词组合而成的，常量采用全部大写的字母表示。

```
for (int i=0; i<10; i++) ...
void GetMemory(char **p, int num);
const int MAX_SIZE = 10;
```

(9) 类名采用“C+首字母大写的单词”形式来命名。

```
class CStudent { ... };
```

其中(6)到(9)在Windows应用程序开发中较为常见。

除了对函数、变量命名之外，我们还会遇到源代码文件的命名问题。一般而言，每一个C++类必须使用一个独立的源文件进行书写；如果这个类对应一个.h文件和一个.cpp文件，那么这两个文件应该具有相同的名称（当然不包括扩展名），并且要求文件名称必须与类名相对应。比如：

```
类名: class CStudent
.h文件: Student.h
.cpp文件: Student.cpp
```

另外，不同的编译器对文件名大小写的敏感度是不同的；比如Student.cpp和student.cpp在VC++中是一样的，而在Gcc编译器中它们则表示不同的文件。

#### 请记住：

函数或变量的名称不仅仅是一个普通的标识符，给它一个会表意、能说话的名字，可以让代码更加易懂、可读。

## 建议 94 合理地添加注释

注释是程序的重要组成部分之一。好的注释应该是简明扼要地点明程序的突出特征，帮助别人更好更容易地理解程序、阅读程序。但是，注释也并非越多越好。凡事都讲究一个度，过犹不及的道理同样适合代码注释。

Phil Haack大师曾经说过：“一旦一行代码显示屏幕上，你也就成了这段代码的维护者”。所以，代码注释的服务对象不仅仅是那些将来维护你代码的开发人员，还包括你自己。你自己将是好的代码注释的第一个受益者。因此，就算是为了自己，也要做好代码注释。

在团队工作中，标准化的注释是我们所推荐的。当然，要写出标准化的注释可以使用注释规范和工具来完成。本文中的注释就是采用文档生成工具DOXYGEN进行处理的。下面分享几个添加注释的技巧：

□ 使用统一的注释方法为每个层次的代码块添加注释

(1) 针对每个源文件 .cpp、头文件 .h 文件进行注释，其中主要包括摘要信息、作者信息，以及最近的修改日期等，如下所示：

```
/**
 * @file      程序文件名称 (如 ByteBuffer.cpp)
 * @brief     程序文件的简要说明

 *           程序文件的详细说明 (简要与详细说明之间间隔一个空行)
 * @author    作者姓名
 * @version   版本编号 修订日期 修订者 修订内容
 */

注:
注释必须以 “/**” 开头
允许存在多个 @version 指示符描述版本修订历程 (每个 @version 指示符描述一个版本)
```

(2) 每个方法 (或者函数) 的注释信息主要包括用途、功能、参数和返回值等内容，如下所示：

```
/**
 * @brief     函数的简要说明

 *           函数的详细说明 (简要与详细说明之间间隔一个空行)
 * @param    参数名称 参数说明
 * @return   返回值说明
 * @retval   值 返回值具体单个值说明
 */

注:
注释必须以 “/**” 开头
注释必须置于函数定义原型之上
允许存在多个 @param 指示符描述多个参数 (每个 @param 指示符描述一个参数)
允许存在多个 @retval 指示符描述多个具体的返回值说明 (每个 retval 指示符描述一个返回值)
@param、@return、@retval 指示符均是可选项
```

(3) 变量的注释一般添加在每行代码的后面，以便对变量的用途进行说明，如下所示：

```
// 变量的说明
```

```
注:
注释必须以 “//” 开头
注释必须置于变量定义之后，与变量定义处于同一行
```

当然并非所有的变量声明都需要添加注释。如果变量能够自表意，注释就多余了。比如：

```
class CStudent
{
    ...
private:
```

```

    string m_szName;           // 名字
    unsigned char m_nAge;      // 年龄
    int m_nStudentID;         // 学号
};

```

(4) 对于用户自定义类型（包括结构、类等），如果其命名不是充分自注释的，必须加以注释。对于自定义类型的注释，其格式如下所示：

```

/*
 * @brief      类型的简要说明
 *
 *             类型的详细说明 (简要与详细说明之间间隔一个空行)
 */


```

注：  
注释必须以“/\*\*”开头  
注释必须置于类型定义之上  
对子元素的注释可参照对函数、变量的注释格式

(5) 如果有多个代码块，并且不同代码块是用于完成不同任务的，则需要在每个代码块前添加一个注释，向读者阐明这段代码的主要功能，如下所示：

```

Code comment 1 // 代码块1的主要功能
Code Block 1

```

```

Code comment 2 // 代码块2的主要功能
Code Block 2

```

```

Code comment 3 // 代码块3的主要功能
Code Block 3

```

注：  
注释必须以“//”开头  
不同的代码块之间以空格分开

#### □ 避免不必要的注释

在为代码添加注释时，要避免给出一些显而易见的注释。写下这些无用的注释不仅会浪费时间，还会转移读者对代码细节的理解，同时是对代码读者的不尊重，是在侮辱读者的智慧，比如以下代码：

```

if (a == 5)    // 如果a等于5
for( int i = 0; i<10; i++)  // i循环，从0到9
nCount = 0;   // 将计数值设为0

```

#### □ 掌握代码注释量的一个度

如果代码注释过少，则会影响代码读者对程序意图的理解；如果过多，又会增加读者的负担。所以对于代码的注释量也要掌握一个合适的度。一般情况下，源程序的有效注释量必

须达到 20%。

#### □ 边写代码加边注释

有些人不喜欢添加注释，并且用“我很忙，没时间”这样的理由作为借口。事实上，在编写代码的同时添加注释，不仅可以帮你更加清晰地理清思路，还能够保证注释与代码的一致性。

当然，有一些开发者觉得应该在写代码之前加注释，用于理清头绪。比如下面的示例代码片段<sup>⊖</sup>：

```
void ProcessOrder()
{
    // Make sure the products are available
    // Check that the customer is valid
    // Send the order to the store
    // Generate bill
}
```

这样的代码注释方式起到了流程图的作用，这也不无道理，全凭个人习惯。

#### □ 注释要简明而准确

在代码的功能、意图层次上进行注释，提供有用、额外的信息。注释的内容要清楚、明了，含义要准确，防止二义性。注释的目的是为了解释代码的目的、功能和采用的方法。

#### □ 注意带有标签的特有作用

在做 MFC 开发时，我们经常会出现 TODO 标签。这样的标签不是用于解释代码的，而是引起注意或传递信息的，是为了便于一个团队中程序员之间的沟通。

```
void DrawPicture(string pic_name)
{
    // to be continued...
}
```

代码注释其实是代码世界中一个相当多彩的部分，或艺术，或搞笑，或诗情画意，体现着程序员严谨之外的其他才艺。虽然有些人建议不要写这样的注释，但是如果在枯燥的代码阅读中能看到让你莞尔一笑的注释，这何尝不是一件令人高兴舒服的事儿呢？

#### 请记住：

为代码添加必要的注释，方便自己，方便他人。

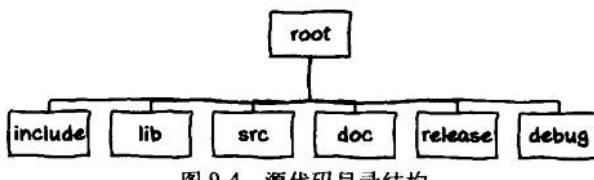
## 建议 95：为源代码设置一定的目录结构

为源代码设置一定的目录结构也是非常有必要的，这就好比将众多的文件进行了类别划

<sup>⊖</sup> 改写自文章《提高代码可读性的十大注释技巧分享》。

分，这样，代码阅读者（包括作者自己）才可以相对轻松地找到其所需要的文件。所以，从某种意义上来说，这也是提高代码可读性的一个手段。

一般情况下，源代码的目录结构设置如图 9-4 所示。



源代码目录结构的说明如表 9-2 所示。

表 9-2 源代码目录结构说明

名称	说明
include	存放程序使用的库程序的头文件
lib	存在应用程序所需的静态链接库
src	存放程序的源代码；src 还可以是多级目录
doc	存放源代码相关的说明文档
release	存放本程序编译的 release 版本的程序文件
debug	存放本程序编译的 debug 版本的程序文件

这样的划分只适用于较为简单的应用开发。如果程序较为复杂，还可以对该目录进行适当的扩充。比如，如果程序需要在 Windows 和 Linux 分别进行编译，则可以在根目录下添加 win32 和 linux 两个子目录，然后在这个目录下分别添加 release 和 debug 子目录。

#### 请记住：

如果一个软件所涉及的文件数目比较多，通常要将其进行划分，为其设置一定的目录结构，以便于维护。

### 建议 96：用有意义的标识代替 Magic Numbers

Magic Numbers，中文直译叫做“魔数”，一般是指没有明显的原因而出现在代码中的具体数字。比如下面的这段示例代码中的 12：

```

for(int i = 0; i < 12; i++)
{
    DoSomething(i);
}
  
```

12 的具体意义是什么呢？可能是一年中包含的 12 个月，可能是一个小组中的成员数，

也可能是某项活动的重复数量。所能确定的就是，如果单从上面的代码所提供的信息量来看，我们根本无法确定 12 所代表的含义。

数字本身的信息量相对有限，不能提供任何说明信息，并且它的可读性较差，在存在重复数据时，这样的使用方式还会增加维护的复杂度，比如，我们必须要区分代码中的此 12 和彼 12 是否为同一情形。

解决这一问题的最好途径就是用有意义的标识来替换这些魔数：在 C 中用宏，在 C++ 中用常量。下面就是通过这样的方式对上述“问题”代码进行了改造，如下所示：

```
const unsigned char MONTHS_PER_YEAR = 12;
const unsigned char MEMBER_COUNT = 12;
const unsigned char ACTIVITY_REPEAT_COUNT = 12;

for(int i = 0; i < MONTHS_PER_YEAR; i++)
{
    PrintDaysCount(i);
}

if( member_index < MEMBER_COUNT)
    AnswerQuestions();

while( repeat_count < ACTIVITY_REPEAT_COUNT)
{
    RepeatActivity();
    repeat_count++;
}
```

这样的名称不仅增加了信息量，提高代码的可读性，并且提供了唯一的维护点，使得维护相对简单。所以，避免在代码中出现诸如 12 和 3.1415926 这样的数字，用类似 MONTHS\_PER\_YEAR 和 PI 这样的标识符替换它们。

---

#### 请记住：

用宏或常量替代信息含量较低的 Magic Numbers，绝对是一个好习惯，这样可提高代码的可读性与可维护性。

---

## 建议 97：避免使用“聪明的技巧”

对于这个问题，大师们早有忠告，在 1972 年图灵奖的颁奖典礼上，Dijkstra 说过：“优秀的程序员很清楚自己的能力是有限的，所以他对待编程任务的态度是完全谦卑的，特别是，他们会像逃避瘟疫那样逃避‘聪明的技巧’”。

何谓聪明的技巧呢？

看下面的这个例子：假设有两个整型变量 `x` 和 `y`，我们要实现 `x` 和 `y` 这两个数的交换。这是一个再简单不过的问题了。但是当写出下面的代码时，如果没有注释，能知道它意欲何为吗？

```
// Swap 技巧版
x ^= y;
y ^= x;
x ^= y;
```

对于大多数的程序员来说，这确实是一个难题。实际上，它完成了 `x` 和 `y` 的交换，而且没有使用多余的变量。与我们的经典版本相比，上述方式节省了内存（因为省去了中介变量）：

```
int temp = x;
x = y;
y = temp;
```

经典版本之所以经典就是因为，即使没有注释，我们也能知道这三行代码的意图，可读性强，更容易理解。所以，我们宁肯放弃“聪明的技巧”，从而换取更强的可读性。

#### 请记住：

正如其名，“聪明的技巧”对于“技巧”要求较高，一般程序员难以琢磨明白，也就影响了代码的可读性和可维护性。所以，最好避免使用“聪明的技巧”。

## 建议 98：运算符重载时坚持其通用的含义

运算符是代码中必不可少的元素，运算符重载也是我们经常面对的一个问题。在重载时坚持运算符的通用含义，比如“+”是加，“-”是减，“\*”是乘，“/”是除。就像下面的复数的加法与乘法运算：

```
CComplex operator+( const CComplex& lhs,
                     const CComplex& rhs      )
{
    return CComplex( lhs.m_real + rhs.m_real,
                     lhs.m_imagin + rhs.m_imagin );
}
CComplex operator*( const CComplex& lhs,
                     const CComplex& rhs      )
{
    return CComplex( lhs.m_real * rhs.m_real
                     - lhs.m_imagin * rhs.m_imagin ,
                     lhs.m_real * rhs.m_imagin
                     + lhs.m_imagin * rhs.m_real );
}
```

虽然没有所谓的标准对其进行规范，坚持运算符的通用含义应该是所有程序员之间的一个默认协议。这默认协议看似平淡无奇，对程序开发没有什么具体的帮助，实际上，它在开发阶段就能够保证代码的正确性和可读性。假如我不遵循这一协议，坚持将“\*”重载为加法运算，将“+”重载为乘法运算，来看看会怎样，如下所示：

```
CComplex operator+( const CComplex& lhs,
                     const CComplex& rhs )
{
    return CComplex( lhs.m_real * rhs.m_real
                    - lhs.m_imagin * rhs.m_imagin ,
                    lhs.m_real * rhs.m_imagin
                    + lhs.m_imagin * rhs.m_real );
}
CComplex operator*( const CComplex& lhs,
                     const CComplex& rhs )
{
    return CComplex( lhs.m_real + rhs.m_real,
                    lhs.m_imagin + rhs.m_imagin );
}
```

对于这样的重载定义，语法和机器都表示同意和允许，但对于你代码的使用者和维护者而言，这绝对是一个大陷阱，而对于作者本人也并无任何的便利可言。代码使用者一定会按照以往的习惯用“+”代表加法，用“\*”代表乘法，他们会对已经发生的错误毫无察觉。

所以，杜绝这种“损人不利己”的行为，切忌随便地更改它们原有的含义。

**请记住：**

运算符重载时坚持该运算符的通用含义，拒绝损人不利己！

## 建议 99：避免嵌套过深与函数过长

嵌套过深和函数过长都会严重危害代码的可读性。就像下面的这段代码，代码里有众多的 if 判断语句：

```
if(condition_1)
{
    if(condition_1_1)
    {
        if(condition_1_1_1)
        ...
    }
    else if(condition_1_2)
    {
        if(condition_1_2_1)
```

```

    ...
}

else
{
    ...
}

}
else
{
    ...
}

```

多级判断的嵌套会增加我们阅读代码时的脑力消耗，要理解这样代码需要太多的上下文信息，我们要在自己的脑子里维护一个“栈”，为每一级的 if 匹配右括号，并按照功能将其分块。

函数过长也会引起这样的问题：函数本应该是一个容易理解的功能单元，如果函数功能过繁过长，那么在阅读代码时就需要更多的上下文信息来帮助理解了，这会使读者不堪重负。

所以嵌套切忌过深，函数也不要过长。

#### 请记住：

嵌套过深或函数过长都会增加代码的理解难度，增加代码阅读时的脑力消耗！所以应尽量避免之。

## 建议 100：养成好习惯，从现在做起

罗马不是一天建成的，好的代码习惯也不是只看几个规范就能养成的。不要过分地专注于规范的细枝末节，与其将时间放在 A 与 B 孰优孰劣的争论上，不如立即付诸实践，从现在做起，遵守代码规范，养成良好的编程习惯！

从今天起，争取做到以下几点：

建立自己的代码规范

在软件开发合作团队中，程序员一方面要依赖大量其他程序员完成的代码，另一方面又要提供大量的代码给其他人使用。由于 C++ 编码自由度很高，所以容易编写出风格迥异的代码。如果缺少一套编程规范，就难以对大家的风格做出合理的限定和统一了，这会增加代码开发和维护成本。因此，在每个团队中都需要建立一套编程规范。

对于崇尚个人英雄主义的独行侠，国际流行的代码规范应该是你的首选。国际流行的代码风格一般是经过千锤百炼，得到程序员们广泛支持的，是大家眼中公认的“好规范”。所以，选择国际流行的代码风格，会让你的工作得到更多人的肯定与支持。这既是作者对未来读者负责的最好体现，也有助于作者的编程习惯与国际的接轨。其中，Google 公司的 C++

代码规范就非常值得推荐。

□ 将规范融入编程过程

遵循代码规范，要求在写代码的同时就遵循命名规范、书写风格，并且注意代码的可读性、可维护性。而且这是在编写代码过程中就必须注意遵循的，减少事后补救，将规范融入编程过程之中。

□ 坚持不懈直到养成习惯

任何习惯的养成都需要长时间的坚持，不抛弃，不放弃，最后的胜利必定属于你！

---

**请记住：**

从现在开始，培养良好的编程习惯。

---

# 第 10 章 让代码运行得再快些

马丁路德金有一个梦想，我们程序员也有一个梦想，那就是希望自己写出的代码运行得更快。

C++ 语言本身的基因决定了其在执行速度方面的优势。在众多的语言中，Google 将 C++ 列为最复杂的那一个，同时也是运行性能最优良的那一个，可从 Google 工程师 Robert Hundt 在《Loop Recognition in C++/Java/Go/Scala》中给出的数据看出（如图 10-1 所示），C++ 在运行速度上是鹤立鸡群的。

然而这些并不可以成为我们放弃代码优化、放弃追求更快梦想的理由。我们可以让 C++ 代码运行得更快，就像下面的建议所说的那样。

Benchmark	Time [sec]	Factor
C++ Opt	23	1.0X
C++ Dbg	197	8.6X
Java 64-bit	134	5.8X
Java 32-bit	290	12.6X
Java 32-bit GC	106	4.6X
Java 32-bit SPEC GC	89	3.7X
Scala	82	3.6X
Scala low-level	67	2.9X
Scala low-level GC	58	2.5X
Go 6g	161	7.0X
Go Pro	126	5.5X

图 10-1 语言运行时间对比

## 建议 101：用移位实现乘除法运算

在大部分的 C/C++ 编译器中，用移位的方法比直接调用乘除法子程序生成代码的效率要高。所以很多时候我们可以通过移位来实现乘除运算，比如：

```
a=a*4;  
b=b/4;
```

可以将其改为：

```
a=a<<2;  
b=b>>2;
```

当然，这里的 b 有个限制，必须为无符号整数。

不要以为只有乘以或除以 2 的 n 次方才可以移位代替，实际上，只要是乘以或除以一个整数常量，均可以用移位的方法得到结果，如  $a = a * 9$  可以拆分成  $a = a * (8+1)$ ，即  $a = (a << 3) + a$ 。同理， $a = a * 7$  就可以拆分成  $a = a * (8-1)$ ，即  $a = (a << 3) - a$ 。

同理可得到除法。但是移位只对整数运算起作用，如果遇到浮点数的乘除运算，便不再推荐这条建议。

## 建议 102：优化循环，提高效率

据统计，在C++/C的循环语句中，for语句是使用频率最高的，while语句次之，do语句最少用。在多重循环中，CPU跨切循环层的次数会对循环的效率产生相对较大的影响，所以如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少CPU跨切循环层的次数，提高效率。

这是事实，有实验为证：

```
// 版本 1:
for (int i=0; i<8192; i++)
{
    for (int j=0; j<100; j++)
    {
        sum = sum + i + j;
    }
}

// 版本 2:
for (int i=0; i<100; i++)
{
    for (int j=0; j<8192; j++)
    {
        sum = sum + i + j;
    }
}
```

在VS2010上进行了100次试验，得到版本1所耗费的时间约为版本2的1.2倍~1.3倍。

如果循环体内存在逻辑判断，并且循环次数很大，宜将逻辑判断移到循环体的外面。如下所示：

```
// 修改前:
for (int j=0; j<8192; j++)
{
    if(condition)
    {
        DoSomeThings();
    }
    else
    {
        DoOtherThings();
    }
}
// 修改后:
if(condition)
{
    for (int j=0; j<8192; j++)
        DoSomeThings();
```

```

    }
else
{
    for (int j=0; j<8192; j++)
        DoOtherThings();
}

```

在一些特别简单的情形下，最好的方法是尽量分解这些小循环，比如下面的示例：

```

// 版本 1:
int nSquare[4] = {0};
for(int i = 1; i < 5; i++)
    nSquare[i-1] = i * i;

// 版本 2:
int nSquare[4] = {0};
nSquare[0] = 1; nSquare[1] = 4;
nSquare[2] = 9; nSquare[3] = 16;

```

其中，第二个版本会更高效，虽然书写起来比版本 1 稍稍麻烦一点（多几个字而已）。

### 建议 103：改造 switch 语句

switch 语句是我们经常见到的一个普通的编程技术，在具体的执行过程中可以转化成多种不同算法的代码，其中最常见的是跳转表和比较链 / 树。它们所生成的代码将按照顺序进行比较，如果发现匹配，程序就跳转到满足条件的语句上并执行。对于 case 的值，推荐按照它们发生的相对频率来排序，把最可能发生的情况放在第一位，最不可能的情况放在最后。比如下面的示例片段：

```

// 修改前:
int nDaysPerMonth;
...
switch ( nDaysPerMonth )
{
case 28:
case 29:
    DoSometing1();
    break;
case 30:
    DoSometing2();
    break;
case 31:
    DoSometing3();
    break;
default:
    break;
}

```

```

}
// 修改后:
int nDaysPerMonth;
...
switch ( nDaysPerMonth )
{
case 31:
    DoSomething3();
    break;
case 30:
    DoSomething2();
    break;
case 28:
case 29:
    DoSomething1();
    break;
default:
    break;
}

```

如果 switch 语句中的 case 标号很多，为了减少比较的次数，应当把这样的 switch 语句转为嵌套的 switch 语句。把发生频率高的 case 标号放在一个 switch 语句中，并且是嵌套 switch 语句的最外层，把发生频率相对低的 case 标号放在另一个 switch 语句中。

## 建议 104：精简函数参数

函数在调用时会建立堆栈来存储所需的参数值，因此函数的调用负担会随着参数列表的增长而增加。所以，参数的个数会影响进栈出栈的次数，当参数很多的时候，这样的操作就会花费很长的时间。因此，精简函数参数，减少参数个数可以提高函数调用的效率。

如果精简后的参数还是比较多，那么可以把参数列表封装进一个单独的类中，并且可以通过引用进行传递。这种方式将会节省一定的时间，对于那些执行时间较短但是调用频率较高的函数来说尤为有效。如下所示：

```

// version 1
void PrintStudentInfo_v1(int id, const std::string& name, int age, int score)
{
    ...
}

// version 2
class CStudent
{
public:
    CStudent();

```

```

int m_nStudentID;
std::string m_strName;
int m_nAge;
int m_nScore];
};

void PrintStudentInfo_v2(const CStudent& student)
{
    ...
}

```

## 建议 105：谨慎使用内嵌汇编

汇编语言与其他高级语言相比，更接近机器语言，效率更高，所以在应用程序中如果碰那些对时间要求苛刻的部分，可以采用汇编语言来重写，代码如下所示：

```

// 修改前（普通版）
int result = 25;
result *= 1024;

// 修改后（汇编版）
mov    dword ptr [result],25
mov    eax,dword ptr [result]
shl    eax,0Ah
mov    dword ptr [result],eax

```

采用汇编语言，可以在程序执行速度上有一定的提高。然而，在内嵌汇编之前要充分斟酌与权衡，不能想当然地去实施这个方法。

首先，开发和测试汇编代码是很辛苦的工作，它将花费更长的时间。当然，我们可以借助于反编译工具，并且可以让大多数编译器生成汇编代码，然后仔细地检查它，修改它，手动地提高代码的效率。

其次，因为汇编语言的可读性较差，这将使将来的代码维护变得非常困难。一方面，我们不能保证将来维护代码的程序员一定会对汇编有所了解；另一方面，如果想要把软件运行于其他平台上，相关的汇编代码还需要根据平台进行重写。

综上所述，内嵌汇编可以提升效率，但是需要谨慎使用，尤其是那些对编译器优化手法不甚了解的人。

建议 101~建议 105 的内容简短，对代码效率的提升也很细微，所以，我称之为“微建议”。除此之外，使用寄存器变量、使用增量减量操作替代加一减一操作等也都属于提高代码效率的“微建议”。

虽然以上这几个建议是“微建议”，但是确实都能提升应用的效率。不过，也不可否认，对于大多数的应用而言，这点效率的提升真的是微不足道的，可是既然效率能提高，代价又几乎可以忽略，那又何乐而不为呢？

---

**请记住：**

记住这几个提升代码效率的“微建议”，虽效果有限，但仍值得推荐。

---

## 建议 106：努力减少内存碎片

通常而言，应用程序是不受内存泄露影响的，但是如果应用程序运行了很长一段时间，频繁地分配和释放内存则会导致其性能逐渐下降，甚至有可能致使程序崩溃。这是为什么呢？因为经常性地动态分配和释放内存会造成堆碎片，尤其是应用程序分配的是很小的内存块时。支离破碎的堆空间可能有许多空闲块，但这些块既小又不连续，像一盘散沙，很难被再次使用。就像下面的堆空间（0 表示空闲内存块，1 表示使用中的内存块）：

```
100101010000101010110
```

上述堆是高度分散的。如果想要分配一个包含 5 个单位（即 5 个 0）的内存块，将是不可能的，尽管系统总共还有 12 个空闲的空间单位，但可用内存是不连续的。而下面的堆可用内存空间虽然少，却不是支离破碎的：

```
1111111111000000
```

我们能做些什么来避免上面所说的堆碎片呢？

首先，尽可能少地使用动态内存。在大多数情况下，可以使用静态或自动储存，或者使用 STL 容器，减少对动态内存的依赖。

其次，尽量分配和重新分配大块的内存块，降低内存碎片发生的几率。例如，不要为一个单一的对象分配内存，而是一次分配一个对象数组。当然，我们也可以求助于自定义的内存池。

---

**请记住：**

内存碎片会影响程序执行的效率，所以应尽量减少内存碎片：首先，尽可能少地使用动态内存；其次，在不得不使用动态内存时，尽量分配大块内存。

---

## 建议 107：正确地使用内联函数

内联（inline）函数既能够去除函数调用所带来的效率负担，又能够保留一般函数的优点。这在建议 52 中已经详细阐述过。不过，内联函数并不是万能良药，在某些情况下，它不仅不能像期望的那样提升性能，甚至会起反作用。偷鸡不成蚀把米的故事在 C/C++ 的世界中也会上演。因此在使用 inline 的时候应该慎之又慎。

在函数名称前加 `inline`，这仅仅是对编译器提出的建议，至于是采取还是忽略这个建议完全由编译器自行决定：

`inline` 关键字告诉编译器最好能采用内联方式来处理。但是，编译器可能会生成单独的函数实例和标准的调用链接，而不是将这个函数内联地插入到代码中。⊕

只有当函数非常短小的时候使用 `inline` 才能得到预想中的效果。对于编译器来说，函数内联与否取决于以下关键性的因素：

- 函数体的大小。
- 是否有局部对象被声明。
- 函数的复杂性（是否存在函数调用、递归等）。

如果编译器忽略了程序员的 `inline` 建议，拒绝这个内联函数，那么这个函数会被当成普通的函数来处理。

如果决定使用内联函数，那么还得面对两个令人头疼的问题：

第一，该如何进行维护？一个函数开始的时候可能满足了设定的内联标准，并以内联的形式出现，但是随着系统的扩展升级，函数体中增添了一些新的功能，函数变得复杂庞大了，这就使内联函数不再适合内联，此时需要把函数前面的 `inline` 去掉，并把函数体放到相应的源文件中，这会使维护的难度有所增加。

第二，不可避免的重新编译。当内联函数实现改变的时候，用户必须重新编译他们的代码以反映出这种改变。而对于非内联函数，用户仅仅需要重新连接。

所以，使用内联关键字 `inline` 是一个“经验活儿”，需要依靠积累的经验来进行判断。合理地使用内联，可以提高程序的性能，反之，会带来意想不到的副作用！

---

#### 请记住：

`inline` 关键字的正确使用需要以长期积累的经验为依托。只有用好内联，才能得到预想中的效率提升。

---

## 建议 108：用初始化取代赋值

在 C 语言中，变量的声明只允许放置在一个函数体的开头部分，然而在 C++ 中变量的声明则可以出现在程序的任何位置上。这种改变使得我们有机会通过用初始化取代赋值来达到提升性能的目的。先看下面的代码片段：

---

⊕ The `inline` keyword tells the compiler that inline expansion is preferred. However, the compiler can create a separate instance of the function (instantiate) and create standard calling linkages instead of inserting the code inline.—— MSDN

```

class CComplex
{
public:
    CComplex(float real = 0, float imagin = 0)
        : m_real(real), m_imagin(imagin) { }
    CComplex(const CComplex& rhs){ ... }
    ~CComplex(){ }
    CComplex operator +(const CComplex& rhs)
    {
        return CComplex(m_real + rhs.GetReal(),
                        m_imagin + rhs.GetImagin());
    }

    void SetValue(float real, float imagin)
    { m_real = real; m_imagin = imagin; }
    float GetReal() const {return m_real;}
    float GetImagin() const {return m_imagin;}
private:
    float m_real;
    float m_imagin;
};

// 修改前版本:
CComplex a, b, c;
a.SetValue(12.0f, 30.1f);
b.SetValue(20.0f, 11.1f);
c = a + b;

// 修改后版本:
CComplex a(12.0f, 30.1f), b(20.0f, 11.1f);
CComplex c( a + b );

```

修改前代码的代价为：3 次构造函数 + 2 次成员函数 + 1 次赋值函数 + 3 次析构函数；而修改后的代价则是：3 次构造函数 + 3 次析构函数。

由此可见，以用户初始化代替赋值，可以使效率得到较大的提升，因为这样可以避免一次赋值函数 `operator =` 的调用。因此，当我们在赋值和初始化之间进行选择时，初始化应该是首选。

这一点在建议 38 中也得到了很好的体现。

需要注意的是，对基本的内置数据类型而言，初始化和赋值之间是没有差异的，因为内置类型没有构造和析构过程。

#### 请记住：

用初始化取代赋值可以减少对赋值操作的执行，所以当我们在赋值和初始化之间进行选择时，初始化应该是首选。

## 建议 109：尽可能地减少临时对象

临时变量对于我们来说是隐形的，因为我们在代码中看不到它的存在，然而它却又真实存在于代码中。临时对象如同正常对象一样，有生也有灭，它会通过调用构造函数创建，通过调用析构函数释放。所以，我们应当尽量减少不必要的临时对象，降低临时对象对代码性能的损耗。

首先挑战一下大家的代码分析能力吧，阅读下面这段代码，试着找出其中的临时变量：

```
string FindAddr( list<Employee> l, string name)
{
    for( list<Employee>::const_iterator i = l.begin();
        i != l.end();
        i++ )
    {
        if( *i == name )
        {
            return (*i).addr;
        }
    }
    return "";
}
```

总共有几处存在临时变量呢？

答案是 5 处！有点不可思议吧，短短几行代码竟然出现了如此多的临时变量。接下来一一揭晓。

### 1&2: list<Employee> l, string name

这两个参数是通过传值来传递的，所以在调用过程中，会根据具体的实参来生成对应的拷贝，也就是临时对象，并传入函数以供使用。这里的临时对象不是必须的，我们可以采用传常量引用或指针的方式来避免。

### 3: i++

这还是那个老套的“前缀与后缀”问题，在后缀重载实现中，对象需自增，然后返回包含原值的临时对象，而前缀操作就可以避免临时对象的产生，详细内容参看建议 12。

### 4: if( \*i == name )

虽然我们没有看到类 Employee 的具体定义，但是有一点能确定：Employee 能够转化为 string 类型或具有一个构造函数 Employee(const string& name)，否则上述代码是不能执行的。无论是哪一种情况，都会产生临时对象。

### 5: return "";

空字符串 "" 会通过调用 string 构造函数转化成一个临时的 string 对象，以供返回。

下面对前面出现的这 5 个临时对象进行精简，得到优化本版。

```

string FindAddr( const list<Employee>& l,
                  const string& name )
{
    string addr;
    for( list<Employee>::const_iterator i = l.begin();
          i != l.end();
          ++i )
    {
        if( (*i).name == name )
        {
            addr = (*i).addr;
            break;
        }
    }
    return addr;
}

```

由此可以总结出，容易产生临时对象的几种主要情形如表 10-1 所示。

表 10-1 临时对象产生的主要情形

发生情形	避免方法
参数	采用传常量引用或指针取代传值
前缀与后缀	优先采用前缀操作
参数转换	尽量避免这种转换
返回值	遵循 single-entry/single-exit 原则，避免同一函数中存在多个 return 语句

#### 请记住：

临时对象不可避免，但是可以减少。尽可能地减少临时对象的出现，就是尽可能地减少临时对象所带来的性能损耗。建议按照以上临时对象经常出现的几种情形所对应的避免方法来检查、优化你的代码。

## 建议 110：最后再去优化代码

Premature optimization is the root of all evil!

——Donald Ervin Knuth <sup>Θ</sup>

不成熟的优化是万恶之源，这是前辈留给我们的谆谆教诲，也是我们在做代码优化时必须恪守的准则。

在进行代码优化之前，我们需要先回答以下几个问题：

Θ 唐纳德·克努特（Donald Ervin Knuth）是经典巨著《计算机程序设计的艺术》的作者。——编辑注

### (1) 算法是否正确？

这是在做代码优化前必须要搞清楚的一个问题。对一个错误的程序做优化，这是一件多么荒谬的事儿啊！

### (2) 如何在代码优化和可读性之间进行选择？

在默认情况下，我们首要关注的是代码的清晰情况与易读性。清晰的代码更容易理解，更容易重构，也更容易优化。而代码的优化只在必要的时候进行。同时，也要避免过度优化，杜绝代码“大杂烩”。

### (3) 该如何优化？

按照经典的“20-80 原则”，代码执行时间的 80% 花费在 20% 的代码上，所以我们在优化之前首先要找到代码效率的瓶颈所在，确定那 20% 的代码在哪里。这个工作可以借助编译器附带的代码分析（profiling）工具来完成（如果所使用的编译器具有这样的工具）。如果对这些瓶颈部分作一些特定的优化，其效果将最为明显有效。

### (4) 如何选择优化方向？

代码优化也要分层次进行，先算法，再数据结构，最后才是实现细节，这样才会最有效。如果对现存的所有代码实现细节的优化都不能让程序跑得更快，这时就该考虑一下是不是优化的方向出了什么问题。就像我们十分熟悉的排序，一般说来<sup>⊖</sup>，快速排序总比简单的冒泡要快，即使你的冒泡算法再怎么优化。

如果已经找到了上述问题的答案，那就开始优化之旅吧。

---

#### 请记住：

代码的优化首先要保证算法的正确性，并且不能以牺牲代码的可读性为代价；在优化过程中要先找到程序执行效率的瓶颈所在，然后遵循“先算法，再数据结构，最后实现细节”的规则，由粗到细来进行优化。

---

<sup>⊖</sup> 如果排序的目标数据存储环境是磁带，或者是其他不支持随机访问的存储介质，冒泡法就比快排更合适。（感谢钱林松老师的建议指正）。

# 第 11 章 零碎但重要的其他建议

好的编码习惯和规范远远不是代码的正确、可读及效率三方面就能完全覆盖的，它还包括很多的细枝末节，虽然这些细枝末节有些零碎，但是这并不是说它们就不重要，它们之中或促进编译，或利于调试，也在为使代码更优秀默默地做着贡献。

欲知详情，请仔细阅读、品味接下来的这 9 条建议。

## 建议 111：采用相对路径包含头文件

#include "XXX.h"，即文件包含命令，它的功能是把指定的文件插入它所在代码行的位置上，从而把指定的文件和当前的源程序文件连成一个源文件。在 C/C++ 的程序设计中，文件包含命令应用广泛。这句代码不仅告诉编译器我们的代码需要包含这个文件，还会告诉链接器这个文件在哪里。假如我们使用了第三方的库文件，但是没有告诉链接器它所在的位置，那么链接器肯定会向我们抱怨“找不到 XXX 库”。

告诉链接器文件位置的方式有两种：绝对路径和相对路径。

### □ 绝对路径

在 Windows 系统中，以盘符开头的路径叫做绝对路径，比如：

D:\project\include

而在 Linux 系统中，以 “/” 开头的路径叫做绝对路径，比如：

/home/project/include

---

**注意：“/”是 Unix 家族、Linux 所有目录的根目录。**

---

### □ 相对路径

一个“点”（“.”）代表的是当前目录所在的路径。对应 Windows 示例中的路径：

D:\project\include

两个“点”（“..”）代表的是相对于当前目录的上一层目录路径。对应 Windows 示例中的路径：

D:\project\

从上面的示例中，也许有人已经发现 Windows 系统的目录分隔符是 “\”，而 Linux 系

统的目录分隔符是“/”（需要注意的是，“/”在 Windows 系统中依然有效）。而对于“.”和“..”，它们的含义在两种系统上是相同的。

假如我们要在 D:\project\main.cpp 文件中包含头文件 D:\project\include\XXX.h，如果采用绝对路径，应该是：

```
#include "D:\project\include\XXX.h"
```

而采用相对路径，就变成了如下形式：

```
#include ".\include\XXX.h"
```

单单是从字符数量上来说，采用相对路径就已经占据了上风。除此之外，采用绝对路径来指定某个头文件的位置时，还存在着一些其他问题：

团队开发时，难以合作。不同的开发者可能将同一个库安装在不同的目录下，这会为团队开发带来不必要的麻烦。以我们比较熟悉的 boost 库为例，假如有人把它安装在了 D:\boost 目录下，在包含 boost 库头文件时采用了绝对路径，自己编译运行不会出现什么问题，但是当团队中的其他人下载了此代码，编译运行时就会报错“找不到 XXX.h”，这仅仅是因为其他人将 boost 库安装在了不同的目录下。

也许有人会说，我是个人开发，不会遇到这种麻烦，所以我可以放心地使用绝对路径了吧？答案依旧是不推荐。如果你安装了多个版本的 boost 库，且采用的是绝对路径，要想在不同的版本间做对比那就不得不一一修改那些路径信息了，而要是采用相对路径就没有这些麻烦。

所以，包含文件时推荐使用相对路径。项目设置 include 和 lib 路径时同理。

#### 请记住：

当你写 #include 语句时，推荐使用相对路径；此外，要注意使用比较通用的正斜线“/”而不要使用仅在 Windows 下可用的反斜线“\”。

## 建议 112：让条件编译为开发出力

在我们的 C++ 程序中，编译预处理是一个不可缺少的步骤。在原程序中包含着的一些编译命令可以告诉编译器对源程序如何进行编译。虽然编译预处理命令不能算是 C++ 语言体系的一部分，但它确实扩展了 C++ 程序的设计能力，合理地使用编译预处理功能，可以使得编写的程序更便于阅读、修改、移植和调试。条件编译就是其中一种。

条件编译中的预处理命令主要包括 #if、#ifndef、#ifdef、#endif 和 #undef 等，它们的主要功能是在程序编译时进行有选择性的挑选，注释掉一些指定的代码，以达到版本控制、防

止对文件重复包含等目的。

条件编译命令主要包括如下几种使用格式：

□ 如果 identifier 为一个定义了的符号，your code 就会被编译，否则不予处理，如下所示：

```
#ifdef identifier
your code
#endif
```

□ 如果 identifier 为一个未定义的符号，your code 就会被编译，否则不予处理，如下所示：

```
#ifndef identifier
your code
#endif
```

□ 如果 expression 非零，your code 就会被编译，否则不予处理，如下所示：

```
#if expression
your code
#endif
```

□ 如果 identifier 为一个定义了的符号，your code1 就会被编译，否则 your code2 就会被编译，如下所示：

```
#ifdef identifier
your code1
#else
your code2
#endif
```

□ 如果 expression1 非零，就编译 your code1；如果 expression2 非零，就编译 your code2，否则就编译 your code3，如下所示：

```
#if expression1
your code1
#elif expression2
your code2
#else
your code3
#endif
```

条件编译在众多库中都得到了广泛的应用，比如说我们最为熟悉的 STL 库中关于 NULL 的定义：

```
#ifndef NULL
#define NULL ((void *)0)
#endif
```

这三行段代码能够保证符号 NULL 有且只有一次定义。

**请记住：**

正确且合理地使用条件编译命令，在程序编译时进行有选择的挑选，可以达到版本控制、防止对文件重复包含等目的。

### 建议 113：使用 .inl 文件让代码整洁可读

一般来说，我们将实现代码放到了 .cpp 文件中，而在 .h 文件中仅包含对应的声明。不过对于内联函数，这个通用准则似乎有些不太适用。如果内联函数的定义比较短小、逻辑比较简单，那么就会推荐将其实现代码放在 .h 文件中。例如，类属性存取函数的实现：

```
class CComplex
{
public:
    CComplex(float real = 0, float imagin = 0);
    ~CComplex();
    void SetValue(float real, float imagin)
    {
        m_real = real; m_imagin = imagin;
    }
    float GetReal() const {return m_real;}
    float GetImagin() const {return m_imagin;}
private:
    float m_real;
    float m_imagin;
};
```

然而为了便于实现和调用，我们亦可以把一些较复杂的内联函数也放到 .h 文件中，但是这样带来的直接后果就是让我们的代码变“丑陋”了。如何才能达到“鱼与熊掌”兼得的效果呢？这就要用到 .inl 文件了。

.inl 文件是内联函数的源文件。我们可以将这部分略显丑陋与笨重的具体定义分离到单独的 .inl 文件中，然后在该头文件的末尾将其用 #include 引入。

.inl 文件还可用于模板的定义。对于较大的工程，模板函数、模板类的声明一般存放在一个或少数几个文件中，而它们的定义部分则存放在对应的 .inl 文件中，然后会在相应的头文件中将其包含进来，从而使模板定义的可读性增强。比如：

```
// CTemplateClass.h
template<class T1, class T2>
class CTemplateClass
{
public:
    CTemplateClass()
    {
```

```

    ...
}

virtual ~CTemplateClass()
{
    ...
}

void DoSomething()
{
    ...
}

... // other methods
private:
    ...
};

}

```

未使用 .inl 文件的版本，模板的声明和定义是在同一个文件中的，这样就严重影响了代码的整洁性。而 .inl 文件正是解决模板定义文件整洁性的一大利器（至于为什么不能将模板的声明与实现分离开来，请参见建议 61）。下面按照上述内容对上面的代码片段进行改造：

```

// 文件 CTemplateClass.h
template<class T1, class T2>
class CTemplateClass
{
public:
    CTemplateClass();
    virtual ~CTemplateClass();
    void DoSomething();
    ...
private:
    ...
};

#include "CTemplateClass.inl"

// 文件 CTemplateClass.inl
template<class T1, class T2>
CTemplateClass<T1, T2>::CTemplateClass()
{
    ...
}

template<class T1, class T2>
CTemplateClass<T1, T2>::~CTemplateClass()
{
    ...
}

template<class T1, class T2>
void CTemplateClass<T1, T2>::DoSomething()
{
}

```

```

    ...
}
```

这让我们看到了模板声明与实现分离的假象，代码变得清晰整洁，不再笨重了。

#### 请记住：

---

.inl 文件可以将头文件与内联函数的复杂定义隔离开来，使代码整洁可读。如果将其用于模板定义，这一优点更加明显。

---

### 建议 114：使用断言来发现软件问题

随着软件行业的迅猛发展，软件测试也逐渐受到越来越多软件公司的重视。软件的可测试性是指在一定的时间和成本前提下，进行测试设计、测试执行，以此来发现软件存在的问题，并对故障进行定位、隔离的能力。简单来说，软件的可测试性就是指一个计算机程序能够被测试的难易程度。

提高代码的可测试性，除了依靠良好的设计之外，断言绝对是一个有力的武器。正如 C++ 前辈告诉我们的：

Use assert() for debugging, but use exceptions for runtime errors.

断言可以快速发现并定位软件问题，同时可对系统错误进行自动报警。断言可以对在系统中隐藏很深、用其他手段极难发现的问题进行定位，从而缩短软件问题的定位时间，提高系统的可测试性。在众多的工具当中，断言无疑是具有最高“性价比”<sup>⊖</sup>的那个。

我们可以使用断言来检查程序正常运行时不应发生但在测试时有可能发生的非法情况，比如用断言检查函数参数是否有效：

```

int DataProcessing(int* pData)
{
    assert( pData != NULL ); // 在使用指针前对其进行空检查
    ... // processing codes
}
```

处理的数据 pData 不为空是一个基本的假设。然而“在一个充满变化的应用程序中任何事情都有发生的可能，断言可以帮助我们确定‘显然为真’的事情确实为真”<sup>⊖</sup>。

为了更好地确定问题所在的位置，我们可以在断言中添加一些特殊的提示信息。比如在

---

<sup>⊖</sup> 这里的性与价分别是指工具的有效性和为此付出的代价。

<sup>⊖</sup> 引自图书《C++ 编程规范 101 条规则、准则与最佳实践》，译者：刘基诚。

下面的例子中，两个函数的参数都是 int\* pData，并且在处理数据之前都对参数进行了判空检查。为了更好地区分是哪一个函数中的参数出了问题，我们在断言中增加了区分信息：

```
int DataProcess(int* pData)
{
    assert( pData != NULL && "In Function DataProcess" );
    //... // other codes
}
// 两个函数中间相隔十万八千里 ...
int UseData(int* pData)
{
    assert( pData != NULL && "In Function UseData");
    //... // other codes
}
```

这主要是因为大多数编译器会在出现错误时输出这个信息，&&“Information”的使用方式正是利用了编译器的这一特点，即通过输出的字符串信息来取代注释，这让我们可以更加简便地获知出现问题的函数信息。

除了发现问题以外，断言还可以在处理错误之后报告错误的发生：

```
if( retrun_msg == MSG_HANDLE_SUCCESS ) // 正确处理
{
    ... // do something
}
else // 出现错误
{
    assert( 0 && "MSG_HANDLE_ERROR");
    ... // handle error
}
```

断言还有一个不得不说的优点，那就是它只会在调试模式下生成代码，而在 Release 版本中则直接无视之，这使得软件的运行速度更快！

**请记住：**

断言是强大的，广泛地使用断言来检查代码中的假设吧，它可以帮助你发现存在的问题。

## 建议 115：优先选择编译和链接错误

在我们的 C/C++ 编程经历中，会遇到各式各样的错误。一般说来，主要包括以下几种：编译错误、链接错误和运行时错误。相对于运行时错误，我们宁可要编译错误和链接错误。这是为什么呢？

首先得从编程语言的静态检查和动态检查说起。

所谓的静态检查，是这么阐述的：“编译器必须检查源程序是否符合源语言规定的语法规和

语义要求。”由此看来，静态检查的主要工作就是语义分析，它是独立于数据和控制流的，可信度相对较高，而且不会增加程序的运行时开销。C++ 语言体系中最强有力的静态检查工具是静态类型检查（C/C++ 是典型的静态类型语言<sup>⊖</sup>）。

而动态检查则是在运行时刻对程序的正确性、安全性等做检查，比如内存不足、溢出、数组越界、除 0 等。这类检查对于数据和控制流比较依赖，需要有足够可靠并具有代表性的测试用例来支持，而且你要不断地测试程序，一遍又一遍。不过，即使这样也无法保证所有的可能性都被覆盖了。这对于一个简单的应用程序来说是令人生畏的。

对于静态检查和动态检查孰优孰劣的争论一直没有间断过，静态检查支持者认为静态检查可以消除大多数的运行时错误，让应用程序的鲁棒性更强；而动态检查的拥趸们则认为静态检查只能检测出一小部分的错误，但付出的代价却有些大：失去了一个相对宽松的编程环境。

这样的争论暂且搁置不议，因为我们是在用 C/C++ 语言写程序。C/C++ 语言属于一种静态语言，结构规范，便于调试，类型安全，提供了比较强大的静态检查系统，但对于运行时的自动动态检查则鲜有支持，需要程序员人工完成。所以，一个设计较好的 C++ 程序应该是较少地依赖动态检查，更多地依赖静态检查。这就是我们宁可要编译错误和链接错误，也不愿与运行时错误打交道的最主要原因。

也许有人会说，错误不是我们可以选择的。确实如此，如果可以选择，我们不要任何错误。其实，我们可以通过改变设计方法用静态检查来代替动态检查，比如考虑用编译时多态代替运行时多态，如果经常用 `dynamic_cast`（或者 `static_cast`）向下强制转型则考虑重新设计接口，考虑使用枚举类型来表示符号变量等。

看看大师 Scott Meyers 是如何教我们用静态检查来代替动态检查的。他要用类 `Date` 来表达日期，最初的版本是：

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

对于这个版本的 `Date` 类，在构造时我们必须对 `day` 和 `month` 的值进行有效性检查，以避免写出类似 “`Data today (32, 15, 2011);`” 这样的语法正确但是毫无意义的代码。改进版本克服了这一问题，它用编译时检查代替了运行时检查，而且保证了足够的安全性：

---

<sup>⊖</sup> 静态类型语言是指数据类型在编译期间进行检查，C/C++ 就是静态语言的典型代表。与之对应的是动态类型语言，其数据类型检查是在运行期间完成的，比如脚本语言 Python、Lua 等。

```

class Month
{
public:
    static const Month Jan() { return 1; }
    static const Month Feb() { return 2; }
    ...
    static const Month Dec() { return 12; }
private:
    Month(int number) : monthNumber (number) {}
    int monthNumber;
};

class Date
{
public:
    Date(int day, const Month& month, int year);
    ...
};

```

这个版本阻止了用户产生新的月份（Month 构造函数为私有），只能通过 Month 的 static 函数获得，同时 month 也是不可修改的（const 修饰），它实现了 month 数值是否有效的编译时检查。

当然，如果想全部代替动态的运行时检查那也是不可能、不切实际的。所以，我们还是不得不面对一些运行时错误。

---

#### 请记住：

尽可能多地利用 C++ 提供的静态检查，当然也不能完全抛弃动态检查。合理地使用这两种错误检查方式，会让我们的开发更轻松得力！

---

## 建议 116：不放过任何一条编译器警告

编译器是程序员的朋友，可以帮助我们发现很多潜在的问题，并给出相应警告，让我们防患于未然！成功的 Build，应该是清新干净没有警告的。如果编译器给出了警告，就要对出现警告的代码多加注意，即使应用程序能够正确执行。如果我们平时能够注意这些警告信息，并采取一定的措施清除之，将会为我们减少很多麻烦。

因此，强烈建议：

- 把编译器的警告级别调至最高。
- 不要放过编译器的任何一条警告信息。

我们平时会遇到各种各样的警告，但是以下这四种绝对是出现最为频繁的：

warning C4101: “XXX”：未引用的局部变量

warning C4700：使用了未初始化的局部变量“XXX”

```
warning C4018: "<" : 有符号 / 无符号不匹配
warning C4715: "Function" : 不是所有的控件路径都返回值
```

其中，XXX表示变量名，Function表示函数名。这样的警告处理起来相对比较简单，只需要找到对应的位置修改即可。

如果应用程序应用到了第三方的头文件，而这些警告恰恰出自这些无法修改的库文件，此时我们该如何清除这些烦人的警告呢？方法也很简单，用自己新写的头文件将原文件封装起来，并有选择地关闭这些没法改变的警告。就像下面的代码片段：

```
/* -----
** 文件: Rewrite_3rdParty.h
** 包含第三方库文件 3rdParty.h
** 屏蔽此文件中的无关 warning
-----*/
#pragma warning(push)
#pragma warning(disable:4018) // 禁止警告 4018
#pragma warning(disable:4512) // 禁止警告 4512
#include <3rdParty.h>
#pragma warning(pop) // 恢复至最初的警告级别
```

#### 请记住：

编译器警告是程序员的好朋友，它能帮助我们发现代码问题，防患于未然。在实践中，我们需要把编译器的警告级别调至最高，不要放过编译器的任何一条警告信息。

### 建议 117：尽量减少文件之间的编译依赖

在 C++ 开发中，是否有人遇到过这样的问题：只是对一个头文件中的定义进行了简单修改，却导致很多文件重新编译，浪费了大量的时间？这是因为那些文件依赖了你修改的头文件，所以要尽量减少文件之间的编译依赖，不要在头文件中直接包含要使用的类的头文件（除了标准库），直接包含头文件这种方式相对简单方便，但是会耗费大量的编译时间，特别是对于大工程来说。

推荐使用类的前向声明（Forward Declaration）来减少文件之间的编译依赖。

编译器在编译期间必须知道对象的大小，如果自定义类的成员变量是一个其他类型的自定义类，建议使用指针来隐藏类细节，因为编译器知道指针的大小。我们只需告诉编译器这是一个自定义类型就可以了，而不需要让它知道具体的实现。比如：

```
#include "Address.h"

class CStudent
```

```
{
    ...
private:
    string m_strName;
    char m_nAge;
    CAddress m_cAddress;
};
```

使用前向声明进行修改后，上述代码就变成了：

```
class CAddress;
class CStudent
{
    ...
private:
    string m_strName;
    char m_nAge;
    CAddress* m_pAddress;
};
```

用对类声明的依赖替代对类定义的依赖，这是减少编译依赖的原则。

所谓“条条大路通罗马”，减少头文件依赖的方法也不是只有前向声明一个，比如柴郡猫技术也是一个不错的选择。也许有人对“柴郡猫”这三个字比较陌生，但是说到 PImpl 手法肯定大家会有所耳闻。

PImpl 是 Private Implementation 的缩写，它使用指向实现的指针来隐藏实现细节，即在主类定义中只定义了接口，私有成员封装在一个实现类中，主类中使用一个形如“`struct XxxxImpl* pimpl_`”的不透明指针来存储私有成员，解开类接口和实现的耦合，真正实现了接口和实现的分离。例如：

```
// Test.h
class CTestImp; // PImpl 前向声明

class CTest
{
public:
    CTest ();
    ~CTest();

    void Public_Method();

private:
    CTestImp *pimpl_; // PImpl 指针
};

// Test.cpp
class CTestImp
{
```

```

public:
    void  Private_Method()  {}

    int    private_var_;
};

CTest::CTest()
: pimpl_( new CTestImp() ) {}

CTest::~CTest()
{
    delete pimpl_;
}

void CTest::Public_Method()
{
    pimpl_->Private_Method();

    pimpl_->private_var_ = 0;
}

```

如果想将此技术应用得更加专业，还需要注意以下两点：

- (1) 将 XxxxImpl 类定义成一个 local 类。
- (2) 主类中使用智能指针代替普通的 PImpl 指针。

除了上述两种技巧外，使用接口类也可以达到降低头文件依赖的目标，可只依赖接口头文件，对于具体的实现类，则无须注意，此处不多赘述。

#### 请记住：

为了加快编译进程，减少时间的浪费，我们应该尽量减少头文件依赖，其中的可选方案包括前向声明、柴郡猫技术等。

## 建议 118：不要在头文件中使用 using

名空间是 C++ 提供的一种机制，可以有效地避免函数名污染。然而在应用时要十分注意：任何情况下都不应在头文件中使用“using namespace XXX”这样的语句。

这是为什么呢？来看下面的这个例子，我们在名空间 ABC 和 XYZ 中分别定义了一个 sort 函数，如下所示：

```

// 文件 abc.h
namespace ABC
{
    void sort() { ... }
}

```

```

    ...
}

// 文件 xyz.h
namespace XYZ
{
    void sort() { ... }
    ...
}

```

如果我们在一个头文件 data.h 中用到了名空间 ABC 中的 sort 函数：

```

// 文件 data.h
#include "abc.h"
using namespace ABC;

void DoSomething(){...} // use sort() to do something

```

麻烦马上就要出现了！如果我们在另一个文件中同时用到了 data.h 中的 DoSomething() 和 xyz.h 中的 sort()，编译就会报错：“error C2668: ‘XYZ::sort’：对重载函数的调用不明确”。

```

#include "data.h"
#include "xyz.h"

using namespace XYZ;
int DataProcessing()
{
    sort();
    doSomething();
    ...
}

```

这就是传说中的名空间污染。

之所以会出现这个问题，是因为在包含头文件 data.h 的源文件中相当于也用了“using namespace ABC”，而这样的名空间引入是在不知不觉中完成的。

因此，最好不要在头文件里面使用“using namespace XXX”，而应该在定义时直接用全称：

```

// 修改前版本：
#include <vector>
using namespace std; // 不推荐
class Data
{
    ...
private:
    vector<int> m_vecData;
};

// 修改后版本：

```

```
#include <vector>
class Data
{
    ...
private:
    std::vector<int> m_vecData; // 推荐
};
```

**请记住：**

任何情况下都不应在头文件中使用“using namespace XXX”，以避免可能的名空间污染，推荐使用全称：XXX::ABC。

## 建议 119：划分全局名空间避免名污染

在工程实践中，我们会定义各式各样的名称，或变量，或函数，或数据类型。然而，随着工程的扩大，特别是需要团队合作的大项目中，名称冲突问题总会不期而至。

比如，在某项工程中，工程师甲在模块 A.h 中定义了一个变量，如下所示：

```
const int MAX_COUNT = 100;
```

而工程师乙在模块 B.h 中也定义了这么一个同名变量。工程师丙负责两个模块的连调，那他就悲剧了，原本在工程师甲和乙那里运行良好的两个模块，在工程师丙这里却直接罢工。编译器给出了如下的错误提示：

```
error C2370: "MAX_COUNT": 重定义: 不同的存储类
```

这就是前面所讲的名称冲突或名污染。

这就像一个学校里出现了两个 Li Lei，你喊 Li Lei，就会不可避免地引起混淆，因为他不知道你要找的是哪一个。为了解决因为重名而引起的冲突，老师给出了解决方法，把他俩分到不同的班级，这样你就可以通过“1 班的 Li Lei”和“5 班的 Li Lei”这样的称呼来解决冲突问题了。

在 C++ 语言中，C++ 的发明者借鉴了实际生活中的经验，允许我们把变量、函数等名称分配到不同的“班级”，这个“班级”就是名空间（namespace）。所以上面所讲的工程师丙遇到的问题就可以按照如图 11-1 所示的方式来解决。

冲突名称 MAX\_COUNT 被放到了两个不同的名空间内，在使用时我们只需指定是哪一个空间的 MAX\_COUNT 就可以了。

在客户代码中使用特定名空间的符号有三种比较通用的方式，还是以上面的 MAX\_COUNT 为例。

```
// 方式 1
using namespace A;
int count = MAX_COUNT;

// 方式 2
using A::MAX_COUNT;
int count = MAX_COUNT;

// 方式 3
int count = A::MAX_COUNT;
```

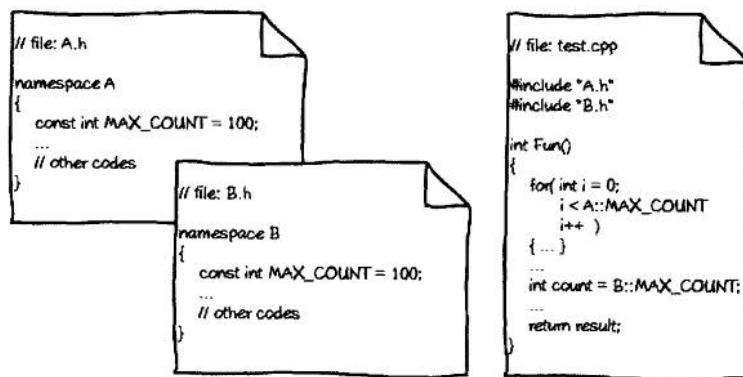


图 11-1 分隔全局名空间解决名称冲突

虽然名空间进入 C++ 标准的时间相对较短，但这并不意味着它不重要。如果它不重要，标准委员会也不会辛辛苦苦地对一些原有库进行重新组织，把它们放入名空间 std 中去了。

所以，如果你身处一个团队合作的大项目，为了避免麻烦，最好使用自己的名空间，而不要将所有的符号和名称统统扔进全局名空间里。

---

#### 请记住：

使用自己的名空间将全局名空间合理划分，会有效地减少名污染问题，因此，请不要简单地将所有的符号和名称统统扔进全局名空间里。

---





## 第三部分

# 程序架构和思想篇

### 本部分内容

第 12 章 面向对象的类设计

第 13 章 返璞归真的设计艺术

# 第 12 章 面向对象的类设计

当今世界，存在着各式各样的语言，有人类之间沟通所用的中文、英语，也有与机器交流所需的 Java、C++，诸多语言“虽变化万端，而理为一贯”，它们都为我们提供了一种表达思想的方法。当我们用 C++ 编写代码时，我们就需要用 C++ 的方式去思考设计问题，去描述和表达现实世界。

作为一种高级编程语言，面向对象绝对是 C++ 的重要特征之一。当 C++ 遇上了面向对象，类就会应运而生。所以，接下来的建议还是围绕 C++ 语言中的重中之重“类”来阐述。

相较于前面的 11 章，接下来的建议层次要更深更高一些，如果将之前的 119 条建议称为“术”，接下来的建议则就有点“道”的味道了。老子有言曰，“道之为物，惟恍惟惚”，虽然这里的“道”可能还有些浅显，但希望起到抛砖引玉的作用，引导你去品味那些代码之后的思想珠玑，洞察语言底层的设计哲学。

## 建议 120：坚持“以行为为中心”的类设计

类是一种复杂的数据类型，它是将不同类型的数据和与这些数据相关的操作封装在一起的集合体，关于这一点我们已经非常熟悉。它定义了该集合中每个对象所共有的属性和方法。其中，属性强调的是数据，而方法强调的是行为。所以，每个实例化后的对象都是数据与行为的结合体。

网友 hlqyq（来自 CSDN 博客）对这一点的认识更为深刻，他认为：

“各成员变量的当前值构成了对象的状态，当各成员变量的值发生变化时，就有可能由量变发生质变，此时我们认为对象的状态发生了改变。对象的方法分两类，一种是改变对象的值（有可能导致对象状态发生变化），我们称之为命令；另一种是不改变对象的值，我们称之为查询。在进行类设计时，需要重点考虑状态、命令、查询这三者之间的关系。”

那么，我们在类的设计中是以数据为中心呢，还是以行为为中心呢？

主张“以数据为中心”的人会关注类的内部数据结构，他们习惯将 private 类型的数据写在前面，而将 public 类型的函数写在后面，如下所示：

```
class CPoint2D
{
private:
    int m_x;
    int m_y;
```

```

public:
    CPoint2D(int x, int y);
    ~CPoint2D();
    void PublicFunction();
};

}

```

而主张“以行为为中心”的人则将关注的重心放在了类的服务和接口上，他们习惯将 public 类型的函数写在前面，而将 private 类型的数据写在后面，如下所示：

```

class CPoint2D
{
public:
    CPoint2D(int x, int y);
    ~CPoint2D();
    void PublicFunction();

private:
    int m_x;
    int m_y;
};

}

```

正如那句著名的哲言所说：存在即合理。

这两种设计方式也各有各的道理。虽然很多的教科书上主张在设计类时“以数据为中心”。但是在当今著名的 IT 公司的编程规范中，都是坚持“以行为为中心”的，比如微软 Microsoft。Microsoft 公司的 COM 规范的核心就是接口设计。之所以更加强调类的行为，是因为类的这些行为是这类对象向其他对象所提供的服务，所以类的使用者会将更多的注意放在接口的功能上，而对其具体的实现并不会感兴趣。

所以，应坚持“以行为为中心”的类设计。

#### 请记住：

坚持“以行为为中心”的类设计，突出接口功能。同时，这也是对大公司设计规范的一种学习与遵循。

## 建议 121：用心做好类设计

C++ 给了我们一个自由的编程世界，能够有效地帮助程序员表达和实现自己的思想。在这个世界里，类是构建 C++ 应用程序大厦的砖与瓦，类的本质是一片逻辑上互相紧密关联的数据和行为的综合逻辑体。在 C++ 语言中，我们可以通过定义一个新的类来抽象一类对象集合。

所以，C++ 开发者的很大一部分精力要花在类的设计上，免不了要在类的设计上倾注心血。

正如 Scott Meyers 告诉我们的那样，一个类型应该做到“易于正确使用、难以错误使用”。设计一个良好的类是一份具有挑战性的工作。那么什么样的类才可配得上“良好”二字呢？良好的类应该拥有“简单自然的语法，符合直觉的语义，以及一个或更多高效的实现。”事实上，如果缺乏一个详细的规划设计，这样的目标是很难达到的。

所以，在设计每一个类时，我们需要深思熟虑。

首先，类的设计就是数据类型的设计，在数据类型的设计中，我们应该逐一解答以下几个问题：

- 类应该如何创建和销毁呢？这会影响到类的构造函数和析构函数的设计。首先应该确定类是否需要分配资源，如果需要，还要确定这些资源又该如何释放。
- 类是否需要一个无参构造函数？如果需要，而恰恰此时这个类已经有了构造函数，编译器就不会帮我们了，那么我们就得显式地写一个。
- 类需要复制构造函数吗？其参数上加上了 `const` 修饰吗？它是用来定义这个类传值（`pass-by-value`）的具体实现的。
- 所有的数据成员是不是都已经在构造函数中完成了初始化呢？
- 类需要赋值操作符吗？赋值操作符能正确地将对象赋给对象本身吗？它与初始化有什么不同？其参数上加上了 `const` 修饰吗？
- 类的析构函数需要设置为 `virtual` 吗？
- 类中哪些值的组合是合法的？合法值的限定条件是什么？在成员函数内部是否对变量值的合法性做了检查？

其次，类的设计是对现实对象进行抽象的一个过程。

我们要对类的属性和功能进行细致分析，既要充分地表现该对象集合的公有属性、行为操作，又要不出现属性和行为的冗余。在设计类的过程中，切忌面面俱到，不要妄图设计一个包含所有功能的巨类，这会使类的使用和维护变得困难。要保证外部接口简单，因为类库是按照一定的接口规范，向上层提供一定的功能服务的，所以，接口设计得越简单，上层用户使用起来就越方便。我们应该设身处地地为用户考虑一下，从技术、情感和心理学等方面认真考虑一下。

再次，数据抽象的过程其实是综合考虑各方面因素并进行权衡的一个过程。以 CFish 类<sup>Θ</sup> 为例进行说明。

可以把“鱼是否可以吃”作为鱼的一个属性来描述：

```
class CFish
{
public:
    void SetEatableFlag(bool eatable);
```

---

<sup>Θ</sup> 根据网站博客园中关于《颠覆传统：面向对象的设计思想》相关内容的讨论改编。

```

    bool GetEatableFlag();
private:
    bool m_bIsEatable;
};

```

当这条鱼能吃的时候，应该调用 SetEatableFlag (true)；设置属性变量 m\_bIsEatable 为 true；如果是条有毒的鱼，m\_bIsEatable 就该设为 false，使用简单方便。

然而如果细细考虑：鱼自己是不能决定自己能不能吃的，决定鱼能不能吃、好不好吃的应该是吃鱼的对象。也许从普通人的角度来看河豚是不能吃的，但是从高明的大厨角度来看河豚就是无比美味的食物了。如此看来将“能不能吃”作为属性变量又不甚合理，似乎更应该通过一个独立于 CFish 类的判定函数来完成：

```

bool IsEatable(const CFish& fish)
{
    ...
}

```

两种方式各有特点，前一种方式使用方便，后一种方法易于扩展。如何抉择，这就需要根据具体的应用情形在这两个因素之间做合理的权衡了。

#### 请记住：

类设计是一件非常具有挑战性的工作，这既是一个数据类型的设计过程，也是一个对显示对象抽象、综合各方面因素来权衡利弊的过程。

## 建议 122：以指针代替嵌入对象或引用

设计类的数据成员时，我们可以有三种选择：

- (1) 嵌入对象。
- (2) 使用对象引用。
- (3) 使用对象指针。

这三种方式有什么区别呢？我们从下面的例子说起：每辆汽车 (CCar) 都会有一个引擎 (CEngine)。我们分别用三种方式来实现汽车类 CCar 的设计，如下所示：

```

class CEngine
{
    /* details are not important for this example */
};

// 嵌入对象
class CCar
{

```

```

public:
    void Start();
    void Move();
    void Stop();
private:
    CEngine m_engine;
};

```

如果使用嵌入对象，我们必须通过 CEngine 构造函数创建 m\_engine，这一步是我们必须接受的，没有任何选择余地。它的生存周期受到了 CCar 对象的影响，m\_engine 将在 CCar 对象的生存周期中一直存在。如果 CEngine 接口发生了变化，CCar 则必须重新编译。如果用户创建了一个 CCar 对象，但是不使用 m\_engine，那么创建 m\_engine 就变成了我们不得不接受的无用功，并且要为此付出代价。再看下面一段代码：

```

class CEngine{ ... };

// 对象引用
class CCar
{
public:
    ...
private:
    CEngine& m_engine;
};

```

在这种方式中，我们使用的是 CEngine 的引用。一旦 CCar 对象创建，CCar 构造函数则必须将 m\_engine 绑定到一个合法的 CEngine 对象上，一旦完成了这个操作，m\_engine 和对应的 CEngine 对象就建立了绑定关系。相较于使用嵌入对象，使用引用的优点就是类 CCar 不再依赖于 CEngine 的大小了。即使 CEngine 的实现发生了变化，CCar 也不需要重新编译。但是由于 m\_engine 必须绑定到一个现存的 CEngine 对象上，在实际应用中，构造 CCar 之前，我们必须保证合法 CEngine 对象的存在，不管后来是否应用到它。接着看下面这一段代码：

```

class CEngine{ ... };

// 对象指针
class CCar
{
public:
    ...
private:
    CEngine* m_pEngine;
};

```

使用了指针，我们就没有了上面的烦恼：类 CCar 并不依赖于 CEngine 的大小。当创建 CCar 对象的时候，我们并不需要将 m\_pEngine 绑定到一个合法的对象上，在构造时，可以

将其置为 NULL。在需要使用 m\_pEngine 的时候，判断 m\_pEngine 是否为 NULL。如果是，则创建一个新的 CEngine 对象；如果为否，则直接使用它就可以了。也就是说现在是“按需创建”（惰性计算）了，程序速度也会因此而加快。此外，在 CCar 对象的生存周期内，同一个 CCar 对象可以使用不同的 CEngine 对象。

使用对象指针作为数据成员具备以下优点：

- 按需创建，惰性计算，构造函数工作较少，对象创建更快。
- 在 CCar 对象的生存周期内，可以使用不同的 CEngine 对象，灵活性更强。

正如那句英文谚语所说：“Every coin has two sides”，指针在带来便利的同时也会存在一些缺点：

- 使用 m\_pEngine 时，需要加倍小心，需要检查 m\_pEngine 是否为 NULL。
- 如果某个成员函数使用 new 创建了该 CEngine 对象，那么应该记得删除 m\_pEngine 所指向的内存空间；否则，就会发生内存泄露。

除此此外，使用指针还有一大好处，那就是可以应用数据成员的多态行为，如下所示：

```
class CEngine{ ... };
class CFerrariEngine : public CEngine{ ... };
class CWeiChaiEngine : public CEngine{ ... };
```

如果我们创建的 CCar 对象是一辆法拉利赛车，那么它使用的是法拉利引擎，我们只需将 m\_pEngine 指向 CFerrariEngine 对象即可；如果这个对象摇身一变成为了一辆普通的农用车，那么它使用的可能就是潍柴发动机，此时也只需将 m\_pEngine 指向 CWeiChaiEngine 对象，而对于类 CCar 的设计，我们无须做出任何改变。当然，这样的优点并非是指针所独有的，使用引用也可以获得多态行为。

所以，如果数据成员的多态行为非常重要，请选择引用或指针。

如果在类数据成员中使用到了自定义成员对象，请使用对象指针。

#### 请记住：

如果在类数据成员中使用到了自定义数据类型，使用指针是一个较为明智的选择。它有以下几方面的优点：

- (1) 成员对象类型的变化不会引起包含类的重编译。
- (2) 支持惰性计算，不创建不使用的对象，效率更高。
- (3) 支持数据成员的多态行为。

### 建议 123：努力将接口最小化且功能完善

在设计类的过程中，我们一方面希望它易于理解、使用和维护，另一方面我们又希望它

功能强大、全面。这种纠结致使类的设计过程也会很纠结：要易于理解，接口要精简，接口越少越好；要功能强大，增加接口数量是在所难免的。

那我们该如何取舍呢？

听从大师们的建议：类接口的目标是完整且最小。

精简接口函数个数，使每一个函数都具有代表性，并且使其功能恰好覆盖 class 的职能。同时又可以获得接口精简所带来的好处：

- 利于理解、使用，维护成本也相对较低。
- 可以缩小头文件长度，并缩短编译时间。

但是也不必因此而因噎废食，掌握一个合适的度即可。而这个度，只可意会不可言传，需要在长期的实践中慢慢揣摩体会。

**请记住：**

类接口的设计目标是完整且最小。我们应该尽量向这一目标靠近。

## 建议 124：让类的数据隐藏起来

OO 思想已经出现了 20 多年，它已成为了软件开发的必然趋势。面向对象编程（OOP）的一个关键原则就是封装，把暴露的数据封装起来，尽可能地让对象管理它们自己的状态，因为过多的依存性会造成系统的紧耦合，这会使任何一点小小的改动都可能造成许多无法预料的结果。而封装机制则是一个控制对象数据和状态强有力的方法，它对外部世界隐藏了其内部细节，即使内部实现发生了变化，需要了解这一个变化的对象也会比较少，变化也就相对简单容易了。所以信息隐藏被认为是优秀软件工程的关键。

在类的设计中，如何实现所谓的数据封装和信息隐藏呢？我们首先想到的就是将所有的数据成员设为私有：

```
class CClassExample
{
public:
    return_type PublicFunctions();

    ...

private:
    type m_dataMembers;
};
```

私有数据是类保持其不变式的最佳方式。

至于公用数据，其本身就是接口，这些数据是无法控制的，这种方式很容易造成状态的

异步，从而引起对象状态的非法。比如下面的这段示例代码：

```
class CString
{
    ...
public:
    char* m_pChar;
    int m_nSize;
};
```

因为成员变量是公有的，所以我们可以通过 `Object.m_pChar = "Hello C++ Tips"` 这样的方式实现对 `CString` 对象成员变量 `m_pChar` 的赋值，但是字符串数量 `m_nSize` 却没有进行同步的更新。所以，公有数据很难保持类的不变式，应当果断拒绝。

保护数据也存在这样的问题，只不过读取修改保护数据的变成了自身的派生子孙。

私有数据只是数据隐藏的最初级阶段。在这么多年的工程开发中，各专家总结出了一些新的技巧来帮助我们实现更高级的数据隐藏，比如柴郡猫技术（Cheshire Cat Idiom）。在《Lewis Carroll's Alice in Wonderland》中柴郡猫逐渐消失了，什么也没有留下，除了它的微笑。而在这里，留下的变成了实现指针（如图 12-1 所示）。

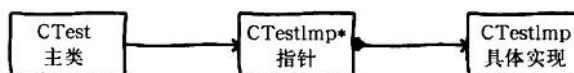


图 12-1 柴郡猫技术实现示意图

柴郡猫技术是将目标类型分为接口部分和实现部分；这种设计可以实现数据的真正隐藏，如下所示：

```
class CTestImp
{
public:
    void Method() {}
    int m_var;
};

class CTest
{
public:
    CTest();
    ~CTest();

    void Public_Method();

private:
    CTestImp *pimpl; // Pimpl 指针
};
```

**请记住：**

坚持数据封装，坚持信息隐藏，杜绝公有、保护属性的存在。推荐 Pimp 的二层设计。

## 建议 125：不要让成员函数破坏类的封装性

在建议 124 中，我们已经阐述了数据封装、信息隐藏的重要性，以及为达到这一目标而必须采取的措施。可是，如果在设计成员函数时不够小心，我们的所有努力都会前功尽弃。

看下面的这个例子，CStudent 类中有一个成员变量 m\_strName 表示该对象的姓名，为了将这一信息隐藏，我们把它设为私有。为了方便用户获得对象的该属性，我们为其设置了接口 GetName()。于是我们得到了如下代码：

```
class CStudent
{
public:
    CStudent(const std::string& name):m_strName(name){}
    std::string& GetName(){ return m_strName; }
    ...
private:
    std::string m_strName;
    ...
};
```

细心的你也许会发现，接口 GetName 返回的是一个引用。之所以这么设计，是基于效率上的考虑。可是，正是这个返回引用的函数给我们带来了巨大的麻烦，如下所示：

```
CStudent boy("Li Lei");
std::string& name = boy.GetName();
name = "Han Meimei";
```

英俊可爱的男同学 Li Lei 被我们叫成了 Han Meimei，m\_strName 的 private 属性似乎失去了它本该有的效力，变成了 public。这都要归结于接口 GetName 返回的引用，是它将我们辛苦封装好的类破坏掉了，留给了用户一个可以直达类内部实现的通道。

不要以为只有引用才会出现上述问题，指针也一样，如下所示：

```
class CStudent
{
public:
    std::string* GetName(){ return &m_strName; }
    ...
private:
    std::string m_strName;
    ...
};
```

```
CStudent boy("Li Lei");
std::string* pName = boy.GetName();
std::string name("Han Meimei");
(* pName) = name;
```

Li Lei 再次被改成了 Han Meimei。

对于这类问题，解决方法很简单：将函数的返回值加上 `const` 修饰，而且这不会改变属性变量的访问限制，如下所示：

```
class CStudent
{
public:
    const std::string* GetName(){ return &m_strName; }
    const std::string& GetName(){ return m_strName; }
    ...
};
```

---

#### 请记住：

小心类的成员函数返回属性变量的“直接”句柄，它会破坏辛辛苦苦搭建维护的封装性。

---

## 建议 126：理解“virtual + 访问限定符”的深层含义

访问限定符（`public`、`private`、`protected`）的作用是控制类成员对外部的可见性，而 `virtual` 关键字则是 C++ 中用于实现多态的重要机制，其核心理念就是通过基类访问派生类定义的函数。`virtual` 和访问限定符本该是各司其职、互不影响的，但是当 `virtual` 碰上这些访问限定符时，设计的灵光就由此迸发出来。接下来就让我们去挖掘这些隐藏在关键字背后的深刻意义吧。

还是从示例开始，分析如下一段代码：

```
class Base
{
public:
    void TestFunction() { Print(); }
private:
    virtual void Print() { std::cout<<"Base\n"; }
};

class Derived : public Base
{
private:
    virtual void Print() { std::cout<<"Base\n"; }
};
```

这段代码中将 Print 函数声明为了 private virtual。声明为 private 表示基类不想让子类看到这个函数，但是又声明为 virtual，表示基类想让这个函数实现多态，所以需要在派生类中对实现细节进行修改。一方面不希望被看到，另一方面又想被修改，似乎是一件矛盾的事儿。实际上这是基类在告诉派生类“覆不覆盖我由你自己决定，但是你绝不可以调用我的实现”。

如果将上面代码片段中的访问控制关键字 private 换作 protect，即：

```
class Base
{
public:
    void TestFunction() { Print(); }
protected:
    virtual void Print() { std::cout<<"Base\n"; }
};

class Derived : public Base
{
protected:
    virtual void Print() { std::cout<<"Derived\n"; }
};
```

以上代码将 Print 函数声明为 protected 表示基类“需要”子类看到这个函数。这带有一定的“强制性”，表示其实现细节必须在派生类中被修改。

而对于 public virtual，我们并不推荐这种方式，虽然在基类中设置 public virtual 函数是那么容易自然的事（Java 中甚至专门为这种做法建立了 interface 机制）。“对待虚函数要像对待数据成员一样，先把它们设为 private，直到设计上要求使用更宽松的访问控制时再去做调整。要知道对于存取级别，由 private 到 public 易，由 public 到 private 难啊。”

综上，我们可以得出以下几点结论：

- 基类中的虚拟私有成员函数，表示实现细节是可以被派生类修改的。
- 基类中的虚拟保护成员函数，表示实现细节是必须被派生类修改的。
- 基类中的虚拟公有成员函数，则表示这是一个接口，不推荐，建议用 protected virtual 来替换。

这些深刻的思想被大师们提炼升华，总结成了经典的设计模式——模板方法模式<sup>⊖</sup>。该模式定义了一个操作中的算法骨架，并将一些步骤的实现延迟到了子类中；模板方法使得派生类在不改变一个算法结构的情况下即可重定义算法的某些特定步骤。如下面的代码片段所示：

```
class CBaseTemplate
{
```

---

<sup>⊖</sup> 可参考 GOF 的经典大作《设计模式》。

```

private:
    void Step_1(){ ... }           // 不可被派生类修改
    virtual void Step_2(){...}      // 可以被派生类修改

protected:
    virtual void Step_3() = 0;     // 必须被派生类修改

public:
    void Function()               // 算法骨架函数
    {
        Step_1();
        Step_2();
        Step_3();
    }
};

```

所以，在采用模板方法模式时，可以这样设计：

- (1) 对于算法骨架中不可变更的一部分，可以将其设计为基类的私有函数，并且在基类公共骨架函数中调用该函数。
- (2) 对于算法骨架某环节的一个默认实现，可以考虑将其设计为基类的私有虚函数，派生类可以改写它，也可以不改写。
- (3) 对于算法骨架中要求在子类中拥有不同实现的部分，可以将其设计为基类的保护(纯)虚函数，这表示派生类必须改写它。

#### 请记住：

注意体会虚函数和访问限定符背后所隐藏的深层意义：

- (1) 基类中的一个虚拟私有成员函数，表示实现细节是可以被派生类修改的。
- (2) 基类中的一个虚拟保护成员函数，表示实现细节是必须被派生类修改的。
- (3) 基类中的一个虚拟公有成员函数，则表示这是一个接口，不推荐，建议用 `protected virtual` 来替换。

学习使用模板方法模式。

## 建议 127：谨慎恰当地使用友元机制

通常说来，类中的私有成员一般是不允许外面访问的。但是友元可以超脱这条禁令，它可以访问该类的私有成员。所带来的最大好处就是避免了类成员函数的频繁调用，节约了处理器的开销，提高了程序的效率。但是，硬币的两面性在这里再次得到体现。它在带来一定好处的同时，也对类的封装性产生了一定的影响。

通常，大家认为“友元破坏了类的封装性”。

这是因为友元使得原本 private 的私有数据不再私有了，而变成了朋友间的共享。友元就像封装这堵大墙上被凿开的一个窗户，类内部的实现不再具有绝对的隐私性了，所以类的封装性被削弱了。

这样的观点貌似正确合理，但是在《C++ FAQs》中，作者却告诉我们：如果它被适当的使用，实际上是可以增强封装的。

那么，友元机制到底会对类的封装性产生何种影响呢？

首先从友元机制的工程需求说起。采用友元机制，一般是基于这样的需求：一个类的部分成员需要对个别其他类公开。比如，妻子对于丈夫的一些细节信息是了如指掌的，所以我们将 CWife 设为 CHusband 的友元类：

```
class CWife{ ... };

class CHusband
{
public:
    ...
private:
    type m_details;

    friend class CWife;
};
```

这样，CWife 对象就可以自由访问 CHusband 的 m\_details 了。

如果不使用友元机制，要实现这两个类的信息互通，可以用以下两种方式。

(1) 改变访问限定符（将成员变量 m\_details 由 private 改为 public），如下所示：

```
class CHusband
{
public:
    ...
public:
    type m_details;
};
```

(2) 提供相应的 Get/Set 函数（const type& GetDetails() 函数），如下所示：

```
class CHusband
{
public:
    ...
    const type& GetDetails() { return m_details; }
private:
    type m_details;
};
```

可是这样一来，原本只有夫妻间知道的小秘密变成了所有人公开的话题，原本只是对个

别类开放的属性，变成了对所有类开放了。类的封装性瞬间分崩离析，消失殆尽。

从这一角度来看，友元机制可让原本是向所有类开放的属性只对部分类开放，这促进了更好的封装。

但是，我们还需要清楚得出以上结论是基于这样一个前提的：“被适当的使用”。如果友元机制使用不当，它对封装性的影响则另当别论了。比如，过度地使用友元机制，会使类的封装千疮百孔，使类的设计复杂混乱。所以，应该谨慎、恰当地评估两个类之间的关系，恰当地使用友元机制。

---

#### 请记住：

**谨慎、恰当地评估两个类之间的关系，恰当地使用友元机制，促进类的封装。**

---

## 建议 128：控制对象的创建方式

栈与堆是对象的主要分布区，它们对应着两种基本的对象创建方式：以 new 方式手动管理的堆创建和只需声明就可使用的栈创建。针对不同的需求，我们会选择对应的对象创建方式。比如，为了避免内存泄漏的发生，我们不会在堆中分配对象。如果不在于类的设计上给予一定的保障，而单靠程序员的理解和自觉来处理，这绝对是一件不怎么靠谱的事儿。

有没有什么机制可以控制对象的创建方式呢？

答案是，YES！

#### □ 要求在堆中建立对象

为了执行这种限制，必须找到一种方法保证调用 new 是建立对象的唯一手段。这一点很容易做到。非堆对象是在定义它时自动构造的，而且是在生存期结束时自动释放的。按照这样的思路，我们只要禁止使用隐式的构造函数和析构函数，就可以实现上述目标。

禁止使用构造函数和析构函数最简单的方法就是将其声明为 private。一般说来，如果将二者都设为 private 会带来较大的副作用，最好的方法是将析构函数声明为 private，而构造函数保持为 public，如下所示：

```
class CStudent
{
public:
    CStudent(const std::string& name):m_strName(name){}
    ...
private:
    ~CStudent() {};
    ...
private:
    std::string m_strName;
    ...
};
```

如果使用如下方式声明一个对象，编译器则表示不能通过，如下所示：

```
CStudent Girl("Han Meimei");
```

而采用 new 的方式，编译器就欣然同意了，如下所示：

```
CStudent* p = new CStudent("Lilei");
```

然而，在手动释放所申请的内存时，却又遇到了上面的麻烦：“无法访问 private 成员”。这时我们需要一点技巧来帮助我们设计——引入析构函数 Destroy，如下所示：

```
class CStudent
{
public:
    void Destroy() { delete this; }
    ...
};
```

在释放对象时，我们通过调用 Destroy 代替 delete：

```
p->Destroy();
```

为什么不是将构造函数都声明为 private 呢？这是因为一个类一般有许多构造函数，如果类的作者必须把它们全部声明为 private，工作量会相对较大，可析构却只有一个。因此声明析构函数为 private 是推荐的方式。

需要注意的是，这种设计方式会影响类的继承。比如：

```
class CHighSchoolStudent : public CStudent
{
    ...
};
```

编译时，编译器会报错：error C2512：“CHighSchoolStudent”：没有合适的默认构造函数可用。相同的情况还会发生在对象包含（Has A Student）时，如下所示：

```
class CSchool
{
    ...
private:
    CStudent m_aStudent[30];
};
```

如果遇到这样的情况该怎么解决呢？还是读者自己去思考一下吧。

#### 禁止在堆中建立对象

按照上面的思路，要禁止调用 new 来建立对象，可以通过将 operator new 函数声明为 private 来实现，如下所示。

```

class CStudent
{
    ...
private:
    static void *operator new(size_t size);
    static void operator delete(void *ptr);
    ...
};

```

现在用户仅仅可以做允许它们做的事情，如下所示：

```

CStudent Boy("Li Lei");           // OK
static CStudent Girl("Han Meimei"); // OK
CStudent *p = new CStudent("Lucy"); // ERROR

```

如果也想禁止堆对象数组，可以把 `operator new[]` 和 `operator delete[]` 也声明为 `private`。

这在类继承时同样会出现问题：如果 `operator new` 和 `operator delete` 没有在派生类中进行改写，且声明为 `public`，派生类就会继承基类中 `private` 的版本；但是如果重写了，基类中原有的 `private` 版本就不再有效了。

#### 请记住：

掌握控制类在堆上创建对象的方法，满足特殊情况下的特殊设计。

### 建议 129：控制实例化对象的个数

在游戏设计中，我们采用类 `CGameWorld` 来抽象化我们所处的游戏世界。然而与其他类有所不同的是，在游戏运行的过程中，`CGameWorld` 的实例化对象有且只有一个。如果按照往常一样来设计 `CGameWorld` 类，当实例化的对象多于一个时，编译器则不会提供任何的错误信息，甚至是警告了。有没有什么设计方法可以帮助我们达到控制 `CGameWorld` 对象个数的目的呢？一旦有人违背了这一设计初衷，编译器就会干脆直接地告诉他：NO，这样行不通！

对于对象的实例化，有一点是十分确定的：要调用构造函数来构建对象。所以，如果想控制 `CGameWorld` 的实例化对象有且只有一个，最简单的方法就是将构造函数声明为 `private`，同时提供一个 `static` 对象，如下所示：

```

class CGameWorld
{
public:
    bool Init();
    void Run();
    ...
};

```

```

private:
    CGameWorld();
    CGameWorld(const CGameWorld& rhs);
    ...

    friend CGameWorld& GetSingleGameWorld();
};

CGameWorld& GetSingleGameWorld()
{
    static CGameWorld s_game_world;
    return s_game_world;
}

```

这个设计有三个要点：

- 类的构造函数是 private，阻止对象的建立。
- GetSingleGameWorld 函数被声明为类的友元，避免了私有构造函数引起的限制。
- s\_game\_world 为一个静态对象，对象唯一。

当用到 CGameWorld 的唯一实例化对象时，可以通过 GetSingleGameWorld 来获取：

```

GetSingleGameWorld().Init();      // 初始化
GetSingleGameWorld().Run();       // 运行

```

如果有人对 GetSingleGameWorld 是一个全局函数有些不爽，或者不想使用那个“破坏”封装的 friend<sup>Θ</sup>，将其声明为类 CGameWorld 的静态函数也可以达到上述目的，如下所示：

```

class CGameWorld
{
public:
    bool Init();
    void Run();
    static CGameWorld& GetSingleGameWorld();
    ...

private:
    CGameWorld();
    CGameWorld(const CGameWorld& rhs);
    ...
};

```

这就是设计模式中著名的单件模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点（如图 12-2 所示）。

如果我们想让对象产生的个数不是一个，而

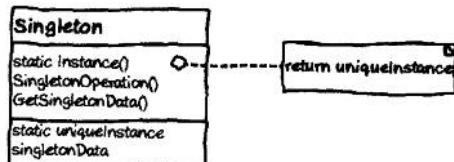


图 12-2 单件模式示意图

<sup>Θ</sup> 作者并不认为它破坏了类的封装。friend 关键字与封装之间的关系详见建议 127。

是 N 个呢（N 为大于 0 的任意整数）？这就需要我们重新寻找设计思路了。

我们可以在类内部设置一个静态计数量，在调用构造函数时，该变量加 1，当调用析构函数时，则该变量减 1。通过检查这个计数变量的值即可判断对象个数是否超出了限制，如下所示（设计并不完善，仅仅提供一个思路）：

```
class CObject
{
public:
    CObject();
    ~CObject();
    ...

private:
    static size_t m_nObjectsCount;
    ...
};

CObject::CObject()
{
    if(m_nObjectsCount > N)
        throw;

    Init();
    m_nObjectsCount++;
}

CObject::~CObject()
{
    Release();
    m_nObjectsCount--;
}
```

#### 请记住：

掌握控制类实例化对象个数的方法。当实例化对象唯一时，采用设计模式中的单件模式；当实例化对象为 N ( $N > 0$ ) 个时，设置计数变量是一个思路。

## 建议 130：区分继承与组合

对于 C++ 程序而言，设计孤立的类是比较容易的，难的是正确地设计相互关联的类，例如基类和派生类。这其中会涉及两个重要概念：继承（Inheritance）和组合（Composition）。因为二者具有一定的相似性，以致很多程序员在二者之间混淆不清。

### □ 继承

C++ 的“继承”特性可以提高程序的可复用性。如果 Base 是基类，Derived 是 Base 的

派生类，那么 Derived 将继承 Base 的数据和函数。这一点我们已经相当熟悉。例如：

```
class Base
{
public:
    void TestFunction_1();
private:
    int m_nData_1;
};

class Derived : public Base
{
public:
    void TestFunction_2();
private:
    int m_nData_2;
};
```

继承，在逻辑上表达的是“是一种（Is-A）”的关系。老师是人，学生也是人，所以类 CTeacher 和类 CStudent 都可以从类 CHuman 派生，如图 12-3 所示。

在 UML 的术语中，这种关系被称为泛化（Generalization）。

逻辑上看起来比较简单。但是在实际应用中却可能遭遇意外，比如在 OO 界中著名的“鸵鸟是不是鸟”和“圆是不是椭圆”的问题。这样的问题说明了程序设计和现实世界之间的逻辑差异。

例如从生物学的角度来讲，鸵鸟（Ostrich）是鸟（Bird）的一种。这一点，我们任何人都没有异议。既然是 Is-A 的关系，类 COstrich 应该可以从类 CBird 派生。虽然鸵鸟不能飞，但是它却从 CBird 那里继承了接口函数 Fly，如下所示：

```
class CBird
{
public:
    virtual void Fly();
    ...
};

class COstrich : public CBird
{
    ...
};
```

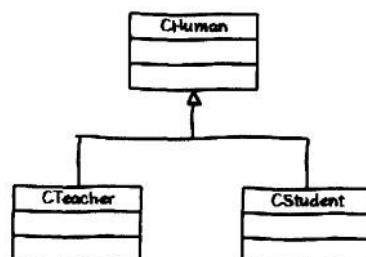


图 12-3 泛化关系 UML 图

再说说“圆是不是椭圆”这个问题。从数学角度讲，圆（Circle）是一种特殊的椭圆（Ellipse），所以呢，类 CCircle 应该可以从类 CEllipse 派生。但是这样导致圆具有了继承自

椭圆但对自己毫无意义的长轴和短轴，画蛇添足！代码如下所示：

```
class CEllipse
{
public:
    ...
private:
    int m_nLongAxle;
    int m_nShortAxle;
};

class CCircle : public CEllipse
{
    ...
};
```

所以更加严格的继承规则应当是：若在逻辑上 B 是一种 A，并且 A 的所有功能和属性对 B 而言都有意义，则允许 B 继承 A 的功能和属性。

继承易于修改或扩展那些被复用的实现。但它的这种“白盒复用”却容易破坏封装性，因为这会将父类的实现细节暴露给子类。当父类的实现更改时，子类也不得不随之更改。所以，从父类继承来的实现将不能在运行期间进行改变。

#### □ 组合

组合，在逻辑上表示的是“有一个（Has-A）”的关系，即 A 是 B 的一部分。

例如眼（Eye）、鼻（Nose）、口（Mouth）、耳（Ear）是头（Head）的一部分，所以类 CHead 应该由类 CEye、CNose、CMouth、CEar 组合而成，如下所示：

```
class CEye
{
public:
    void Look();
};

class CNose
{
public:
    void Smell();
};

class CMouse
{
public:
    void Eat();
};

class CEar
{
```

```

public:
    void Listen();
};

class CHead
{
public:
    void Look();
void Smell();
void Eat();
void Listen();
private:
    CEye m_eye;
    CNose m_nose;
    CMouse m_mouse;
    CEar m_ear;
};

```

其 UML 图如图 12-4 所示。

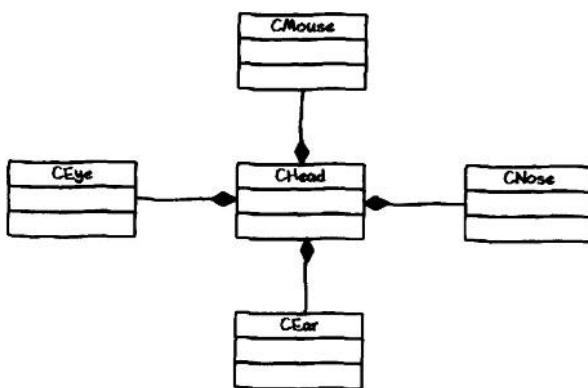


图 12-4 CHead 组合

实心的菱形代表了一种坚固的关系。被包含类的生命期受包含类的控制，被包含类会随着包含类的创建而创建，随包含类的消亡而消亡。

如果采用继承，CHead 也可以实现看（Look）、闻（Smell）、吃（Eat）、听（Listen）的功能，代码简单，并且运行正确，但是并不推荐。

```

class CHead : public CEye, public CNose,
              public CMouse, public CEar
{
    ...
}

```

组合属于“黑盒”复用，被包含对象的内部细节对外是不可见的。所以，它的封装性相

对较好，实现上的相互依赖性比较小。并且可以通过获取指向其他的具有相同类型的对象引用，在运行期间动态地定义组合。而其缺点就是致使系统中的对象过多。

综上所述，Is-A 关系用继承表达，Has-A 关系用组合表达，优先使用（对象）组合。

#### 请记住：

Is-A（是一个）和 Has-A（有一个）表示着不同的逻辑，所以有着不同的表达方式：前者继承，后者组合。

### 建议 131：不要将对象的继承关系扩展至对象容器

在前一节中，我们讲述了 Is-A 的两个特殊例子，并介绍了在程序世界中 Is-A 的逻辑和现实世界中的“是一种”关系略微不同。这里不再重复上一节的老套故事了，下面来看一个全新的问题。

车（Car）是一种交通工具（Vehicle），并且交通工具所具备的一些功能车也具备，所以我们就可以确定它们之间的继承关系：

```
class CVehicle
{
    ...
};

class CCar : public CVehicle
{
    ...
};
```

那么，我们能不能由此推断出停车场（CarParkingLot）是一种交通工具的停泊场（VehicleParkingLot）呢？代码如下所示：

```
class CVehicleParkingLot
{
public:
    void AddNewVehicle(const CVehicle& vehicle);

};

class CarParkingLot : public CVehicleParkingLot
{
    ...
};
```

直觉上是可以的。因为不仅在现实逻辑上成立，基类函数的成员函数 AddNewVehicle 在

派生类中也有意义。

可是，这虽然很好地执行了上节的建议，对程序世界中 Is-A 关系的认识也比较深刻，但是却忽略了一点：成员函数 AddNewVehicle 的参数。在基类中，它添加的是交通工具，也就是说既可以是飞机大炮，也可以是小车战船。如果派生类 CarParkingLot 继承了基类的这一能力，那么停车场停的将不再是车，停车场也就不再是停车场了。

所以呢，车是一种交通工具，但是停车场不是交通工具的停泊场。换句话说，A 是 B 的基类，B 是一种 A，但是 B 的容器却不能是这种 A 的容器。正如 C++ FAQ 中讲的那样：“一袋苹果不是一袋水果；如果一袋苹果能够被传递给一袋水果的话，就可以把香蕉放入袋中了，即使它被认为里面只能放苹果！”

所以，不要将对象的继承关系扩展至对象容器。

#### 请记住：

Is-A（是一个）表示类之间的继承关系，但是这种类间的关系不能扩展至对应的对象容器，“一袋苹果不是一袋水果”。

## 建议 132：杜绝不良继承

封装、继承、多态是面向对象技术的三大机制，其中封装是基础，继承是关键，多态是延伸。继承作为关键的一部分，如果我们对其机制理解不够深刻，很容易造成不良继承的出现，从而影响了软件的设计。

以建议 130 提到的“圆是不是椭圆”这一著名问题为例。在数学上，圆可以被看作是一种特殊的椭圆，如下所示：

```
class CEllipse
{
public:
    void SetSize(float x, float y);
    ...
};

class CCircle : public CEllipse
{
```

但是，椭圆具有一个设置长短轴的成员函数 SetSize(x, y)，通过这个函数可以改变圆率。而圆的圆率是不能被改变的。椭圆能做某些圆不能做的事，所以圆 CCircle 继承自椭圆 CEllipse 的设计不是很合理。

那面对“圆是 / 不是一种椭圆”这个两难问题，我们该怎么办呢？

很多人会选择用代码技巧来弥补设计缺陷，也许会重新定义 CCircle::SetSize(x, y)，或者使其抛出异常，或者调用 abort()，或者让它做些你想让它做的工作。但是这些技巧会让用户吃惊不已，这违背了接口设计的“最小惊讶”原则。所以，建议禁用！

你的选择只有三个：

(1) 改变观念，认为圆可以是不对称的。

这样的观念对于思维严谨的程序员来说有些困难，因为从认识圆的那一刻起，对称就作为其最重要的一个特征印在了我们的脑海里。

(2) 将基类中的成员函数 SetSize(x, y) 删除。

这样做的一个前提是：保持“圆是一种椭圆”的继承关系对你来说非常重要。

(3) 去掉它们之间的继承关系。

去掉继承关系，并不代表圆类和椭圆类完全无关。两个类可以继承自同一个类 COvalShape，不过，该基类不能执行不对称的 SetSize 运算，如下所示：

```
class COvalShape
{
public:
    void SetSize(float size);
    ...
};

class CEllipse : public COvalShape
{
public:
    void SetSize(float x, float y);
    ...
};

class CCircle : public COvalShape
{
    ...
};
```

其中，椭圆类 CEllipse 中增加了特有的 SetSize(x, y) 运算。

当然也可以使两个类完全不相干，椭圆可以从不对称图形类 CAbsymmetricShape 派生，而圆则从对称图形类 CSymmetricShape 派生。代码如下所示：

```
class CAbsymmetricShape
{
public:
    void SetSize(float x, float y);
    ...
};
```

```

class CEllipse : public CSymmetricShape{ ... };

class CSymmetricShape
{
public:
    void SetSize(float size);
    ...
};

class CCircle : public CSymmetricShape{ ... };

```

不良继承出现的根本原因在于对继承的理解不够深刻，错把直觉中的“是一种（Is-A）”的概念当成了学术中“子类型（subtyping）”概念。在继承体系中，派生类对象必须是可以取代基类对象的。而在圆和椭圆的例子中，成员函数 SetSize(x,y) 违背了这个可置换性，即 Liskov 替换原则<sup>⊖</sup>。

所有的不良继承都可以归结成“圆是不是椭圆”这一著名的具有代表性的问题上，这也许有些难以置信，但的确是事实。在不良的继承中，基类总会有一个额外的能力，而派生类却无法满足它，这一额外能力通常是一个或多个成员函数所允许的操作。要解决这一问题，要么使基类弱一些，派生类强一些，要么消除继承关系，这需要根据具体情形来选择。

所以，要仔细体会“圆是不是椭圆”这一问题，它会帮助你发现设计中的不良继承。

#### 请记住：

“圆是不是椭圆”这一问题是不良继承设计的典型代表。深刻理解这一问题，体会 Is-A 的深层含义，杜绝设计中的不良继承。

### 建议 133：将 RAII 作为一种习惯

在 C++ 中，资源管理必须要做到善始善终，申请了资源就要记得归还。然而在实际中，这却是很难保证的。RAII 为此提供了实用的解决方案。

RAII（Resource Acquisition Is Initialization），资源获取即初始化。该短语本身包括两个要点：资源是其一，初始化是其二。RAII 是 C++ 语言的一种管理资源、避免泄漏的惯用方法。C++ 标准可保证在任何情况下，已构造的对象最终会被销毁，即它的析构函数最终会被调用。简单地说，RAII 的做法是使用一个对象，在其构造时获取资源，在对象生命周期中控制对资源的访问，使之始终保持有效，最后在对象析构时释放资源。实现这种功能的类即采

<sup>⊖</sup> 详见建议 136。

用了 RAII 方式，这样的类被称为封装类。比如：

```
class CResource
{
public:
    CResource(const char *name) :
        _resource(alloc_resource(name)) { }
    ~CResource() { release_resource(m_resource); }

private:
    type m_resource;
};
```

RAII 封装类的使用方法也非常简单。程序员要做的，就是用一个 RAII 语义的类来表达一类资源，然后为该资源设置一个生命周期范围，如下所示：

```
void Transaction( const char *res_name)
{
    CResource resource(res_name);
    ... // 使用资源 resource
}
```

这样的方式在表达逻辑上做到了资源和事务逻辑的“同生共死”。不管程序体内资源的使用逻辑如何复杂，退出路径如何繁多，C++ 语言保证了资源对象在超出生命周期时必定得到析构，资源必定得到释放。

如果资源的生命周期无法静态决定，智能指针类的引用计数便可披甲上阵，它基本上可以解决百分之九十以上的问题。

由此可见，RAII 仅需少量的管理代码，即可普遍适用于各种资源对象的应用情形，为资源管理提供一个统一的模式。

RAII 在 C++ 编程技术中已经是一种不可或缺的核心技术了。RAII 几乎无处不在，这不仅仅是因为 C++ 之父的大力提倡，更是因为这一技术本身的简单高效，特别是在配合智能指针时，双剑合璧，将无所不能！通过 RAII，我们可以把资源管理问题弱化，让程序员的关注点放在如何表达事务逻辑上，而这恰恰是程序员的主要任务。

所以，使用 RAII 应该成为 C++ 程序员的基本习惯。

#### 请记住：

使用 RAII 是 C++ 程序员的良好习惯，所以我们应将使用 RAII 作为一种基本习惯。

### 建议 134：学习使用设计模式

设计模式是用来“封装变化、降低耦合”的工具，它是面向对象设计时代的产物，其本

质就是充分运用面向对象的三个特性（即：封装、继承和多态），并进行灵活的组合。

设计模式这个术语是由 Erich Gamma 等人在 20 世纪 90 年代从建筑设计领域引入的。它是对软件设计中普遍存在的各种问题所提出的解决方案。设计模式并未直接用来完成代码的编写，而是描述了在各种不同情况下，要怎么解决问题。面向对象的设计模式通常是以类或对象来描述其中的关系和相互作用的，但不涉及用来完成应用程序的特定类或对象。设计模式能使不稳定依赖于相对稳定、具体依赖于相对抽象，避免会引起麻烦的紧耦合，增强软件设计面对变化及适应变化的能力。

设计模式是前人经验实践的结晶，可以帮助我们优化设计。关于设计模式的论著最为著名的就是 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides（常被称作“四人帮”GoF，Gang of Four）合作出版的《Design Patterns : Elements of Reusable Object-Oriented Software》。书中共收录了 23 个设计模式，这些模式被划分成为三大类：创建型、结构性、行为型，如图 12-5 所示。

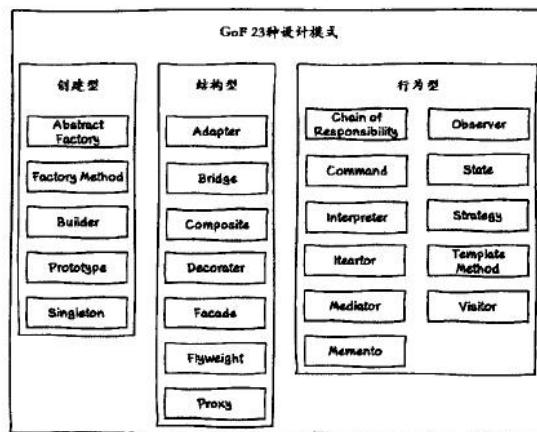


图 12-5 GoF 23 种设计模式

下面选取创建型的 5 种设计模式做简要分析：

□ Abstract Factory 抽象工厂模式（如图 12-6 所示）

意图：

提供一个创建一系列相关或相互依赖对象的接口，无须指定它们具体的类。

适用情形：

- (1) 当一个系统要独立于它的产品的创建、组合和表示时。
- (2) 当一个系统要由多个产品系列中的一个来配置时。

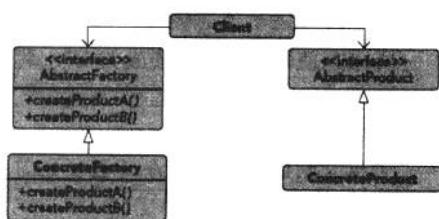


图 12-6 抽象工厂模式示意图

- (3) 当要强调一系列相关的产品对象的设计以便进行联合使用时。
- (4) 当提供了一个产品类库，但只想显示它们的接口而不是实现时。

**Factory Method 工厂方法模式** (如图 12-7 所示)

意图：

定义一个用于创建对象的接口，让子类决定实例化哪一个类。该模式使得类的实例化延迟到了其子类。

适用情形：

- (1) 当一个类不知道它所必须创建的对象的类时。
- (2) 当一个类希望由它的子类来指定它所创建的对象时。
- (3) 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将“哪一个帮助子类是代理者”这一信息局部化时。

**Builder 生成器模式** (如图 12-8 所示)

意图：

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

适用情形：

- (1) 当创建复杂对象的算法应该独立于该对象的组成部分及它们的装配方式时。
- (2) 当构造过程必须允许被构造的对象有不同的表示时。

**Prototype 代理模式** (如图 12-9 所示)

意图：

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

适用情形：

- (1) 当要实例化的类是在运行时刻指定时。
- (2) 当一个类的实例只能有几个不同状态组合中的一种时。

**Singleton 单件模式** (如图 12-10 所示)

意图：

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

适用情形：

当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

GoF 的 Design Patterns 虽然经典，但是却有一个缺点：晦涩难懂。从风格上来讲，与其

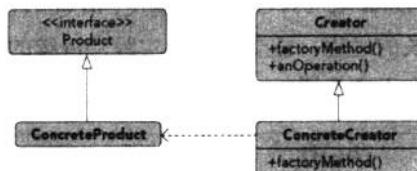


图 12-7 工厂方法模式示意图

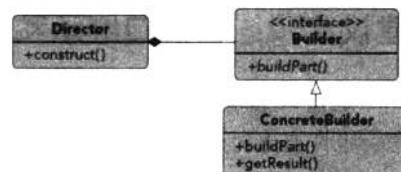


图 12-8 生成器模式示意图

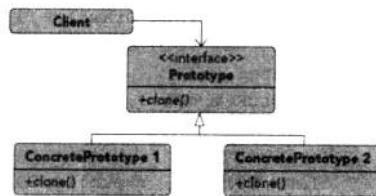


图 12-9 代理模式示意图

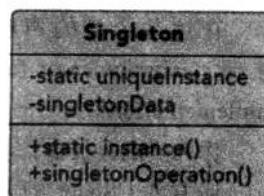


图 12-10 单件模式示意图

说是为学习者而写作的教程范本，还不如说是给学术界人士看的学术报告，它严谨有余，生动不足，所以需要慢慢体会。

除了 GoF 的 23 个著名模式之外，还有很多别的著名模式，比如说 Wrapper 模式、DAO（Data Access Object）模式、MVC（Model-View-Control）模式等。模式的学习是一个艰苦漫长的过程，需要大量的实践，持续的学习。如果你想在短时间内精通它、熟悉它，这绝对是个天方夜谭！

#### 请记住：

设计模式是前人经验实践的结晶，可以帮助我们优化设计，封装变化，降低耦合。但是设计模式的理解、应用又具有一定的难度，慢慢学习体会，将其融入到你的设计中。

### 建议 135：在接口继承和实现继承中做谨慎选择

是接口继承还是实现继承，这是一个问题。这其中蕴含着很深刻面向对象的设计理念。也许有人对于这两种选择的不同不是很了解，所以下面先从示例程序讲起。

CShape 是一个几何图形的基类，对于任何一个几何图形来说，绘制和设置颜色都是合理操作。因此可按照如下方式设计类：

```
class CShape
{
public:
    virtual void Draw() = 0;
    virtual void SetColor(const COLOR& color);
private:
    COLOR m_color;
};

class CCircle : public CShape{};

class CEllipse : public CShape{};
```

在这几个类的声明中就很好地展示了继承的两个相互独立的部分：函数接口的继承（inheritance of function interfaces）和函数实现的继承（inheritance of function implementations）。

作为类的设计者，我们必须深思熟虑。

所有的 CShape 对象都是可以绘制出来的，同时基类 CShape 本身不能为 Draw 这个函数设置一个合理的默认实现，所以我们将 Draw 设置成了纯虚函数。这就意味着它必须被任何继承它的具体类重新声明，派生类仅仅继承了一个接口而已。圆和椭圆有着自己的绘制方式，圆类 CCircle 和椭圆类 CEllipse 继承基类 CShape 的绘制接口 Draw，而对于具体的实

现，两个类则各自在内部完成。

对于 CShape::SetColor，我们将其设置为普通的虚函数。派生类设计者由此得到的信息是：“你应该支持一个 SetColor 函数，如果你不想自己写，可以求助于基类 CShape 中的默认版本”。这样的设计避免了代码重复，使得通用特性更加清楚明白，提升了未来的可扩展性，简化了长期的维护工作。

因此，在接口继承和实现继承之间进行选择时，我们需要考虑的一个因素就是：基类的默认版本。对于那些无法提供默认版本的函数接口我们选择函数接口继承；而对于那些能够提供默认版本的，函数实现继承就是最佳选择。

---

#### 请记住：

接口继承和实现继承相互独立，代表着一定的设计意义：对于那些无法提供默认版本的函数接口我们选择函数接口继承；而对于那些能够提供默认版本的，函数实现继承就是最佳选择。

---

## 建议 136：遵循类设计的五项基本原则

面向对象的设计（OOD）是多少年来一直被讨论的一个热点话题。在面向对象的设计中，我们希望能够通过很小的设计改变来应对设计需求的变化。经过多年的实践和思考，不少 OO 先驱和前辈提出了很多有关面向对象的设计原则，用于指导 OO 的设计和开发。其中，就包括指导类设计的五项基本原则。

### □ 单一职责原则（Single Responsibility Principle, SRP）

专注，是一个人优良的品质；同样，单一也是一个类的优良设计。单一职责的核心思想是：一个类，最好只做一件事。

单一职责原则可以看作是低耦合、高内聚在面向对象原则上的引申，它将职责定义为了引起变化的原因，以提高内聚性来减少引起的变化。职责过多，可能引起它变化的原因就会越多，这将导致职责依赖，它们相互之间就产生影响，从而会大大损伤其内聚性和耦合度。通常意义上的单一职责，指的是只有一种单一功能，不要为类实现过多的功能点，保证实体只有一个能引起它变化的原因。

交杂不清的职责将使代码看起来特别别扭，而且牵一发而动全身，同时又有失美感，必然会导致丑陋的系统错误风险。

### □ 开闭原则（Open Closed Principle, OCP）

正如牛顿三大定律在经典力学中的地位一样，开闭原则是面向对象设计的基石。一个优秀的软件实体应当对扩展开放，对更改关闭。换句话说，在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展。软件设计追求的目标是封装变化、降低耦合，而开

放封闭原则就是这一目标的最直接体现。

实现开放封闭原则的关键在于抽象化。对一个事物抽象化，实质上是在概括、归纳、总结它的本质。将总结得到的特征作为抽象层，会相对稳定，不需更改，从而满足了“对修改关闭”的原则；而从抽象类导出的具体类可以改变系统的行为，从而满足了“对扩展开放”。

因此，在设计时要尽量考虑接口封装机制、抽象机制和多态技术。

“需求总是变化的”，如果遵循封闭开放原则，合理设计就能封闭变化满足需求，同时还能保证内部封装体系稳定，不被需求的变化所影响。

#### □ 替换原则（Liskov Substitution Principle, LSP）

子类应当可以替换父类并出现在父类能够出现的任何地方。这个原则是由 Liskov 于 1987 年提出的。它同样可以从 Bertrand Meyer 的 DBC（Design by Contract）的概念推出。

对于 Liskov 替换原则，其核心思想是：子类必须能够替换其基类。这一思想体现为对继承机制的约束规范：只有子类能够替换基类，才能保证系统在运行期内能识别子类，这是保证继承复用的基础。在父类和子类的具体行为中，必须严格把握继承层次中的关系和特征，要使得将基类替换为子类，程序的行为不会发生任何变化。但是，这一约束反过来则是不成立的，也就是说，子类可以替换基类，但是基类不一定能替换子类。

Liskov 替换原则，主要着眼于将抽象和多态建立在继承的基础上，因此只有遵循了 Liskov 替换原则，才能保证继承复用是可靠的。其实现的方法是面向接口编程：通过提取纯虚类（Extract Abstract Class），将公共部分抽象为基类接口或抽象类在子类中以覆盖父类的方法实现新的方式来支持同样的职责。

Liskov 替换原则能够保证系统具有良好的拓展性，同时可实现基于多态的抽象机制，能够减少代码冗余，避免运行期的类型判别。

#### □ 依赖倒置原则（Dependency Inversion Principle, DIP）

其核心思想是：依赖于抽象。具体而言就是高层模块不依赖于底层模块，而是二者都依赖于抽象，即抽象不依赖于具体，具体依赖于抽象。依赖倒转原则是对传统的过程性设计方法的“倒转”，是高层次模块复用及其可维护性的有效规范。

依赖一定会存在于类与类、模块与模块之间。当两个模块之间存在紧密的耦合关系时，最好的方法就是分离接口和实现：在依赖之间定义一个抽象的接口使得高层模块调用接口，底层模块实现接口的定义，从而有效控制耦合关系，达到依赖于抽象的设计目的。

依赖于抽象，就是不对实现编程，而对接口编程。依赖于抽象是一个通用的原则，而某些时候依赖于细节则是在所难免的，所以我们必须在抽象和具体之间进行取舍。

#### □ 接口分离原则（Interface Segregation Principle, ISP）

该原则的核心思想是：使用多个小的专门的接口，而不要使用一个大的总接口。具体而言，接口应该是内聚的，应该避免“胖”接口。一个类对另外一个类的依赖应该建立在最小的接口上，不要强迫依赖不用的方法，这是一种接口污染。

接口有效地将细节和抽象隔离开来，体现了对抽象编程的一切好处，接口隔离强调接口的单一性。而胖接口存在明显的弊端，会导致实现的类型必须完全实现接口的所有方法、属性等。事实上，在某些时候，实现类型并非需要所有的接口定义，这在设计上是一种“浪费”，而且在实施上会带来潜在的问题，对胖接口的修改将导致一连串的客户端程序需要修改，有时候这是一种灾难。在这种情况下，将胖接口分解为多个定制化方法，使客户端仅仅依赖于它们实际调用的方法，不会依赖于它们不用的方法。

分离的手段主要有以下两种方式：

- (1) 利用委托分离接口。
- (2) 利用多重继承分离接口。

以上就是5个基本的设计原则，它们就像面向对象程序设计中的金科玉律，遵守它们可以使我们的代码易于复用，易于拓展，灵活优雅。设计原则代表着程序设计的永恒灵魂，需要在实践中时时刻刻遵守。就如ARTHUR J.RIEL在《OOD启示录》中所说的：“你不必严格遵守这些原则，违背它们也不会被处以宗教刑罚。但你应当把这些原则看作警铃，若违背了其中的一条，那么警铃就会响起。”

---

#### 请记住：

##### 遵循类设计的五项基本原则：

- (1) SRP，单一职责原则，一个类应该有且只有一个改变的理由。
  - (2) OCP，开放封闭原则，你应该能够不用修改原有类就能扩展一个类的行为。
  - (3) LSP，Liskov替换原则，派生类要与其基类自相容。
  - (4) DIP，依赖倒置原则，依赖于抽象而不是实现。
  - (5) ISP，接口隔离原则，客户只要关注它们所需的接口。
-

# 第 13 章 返璞归真的程序设计

虽然本章的名字有些玄虚，但是这并不代表接下来的建议空洞无物。这些建议有些仅局限于 C 语言，有一些则超越了特定语言的限制。你可以将它们看作是一些编程的习惯，更可以将它们当作习惯背后的设计哲学。这些建议很多是大师们实践的积累与经验的升华<sup>⊖</sup>，将会有对我们的程序生涯大有裨益。

## 建议 137：用表驱动取代冗长的逻辑选择

表驱动法（Table driven method），是一种不必用很多的逻辑语句（if 或 Case）就可以把表中信息找出来的方法。它是一种设计模式，可用来代替复杂的 if/else 或 switch-case 逻辑判断。从某种意义上来说，任何信息都可以通过“表”来挑选。在表相对简单的情况下，逻辑语句往往更简单而且更直接。但随着逻辑链的复杂，表就变得越来越富有吸引力了。

### □ 初识表驱动

下面先从一个例子<sup>⊖</sup>说起，假设要设计一个函数，该函数的功能比较明确，是获得每个月的天数。首先进入我们脑海的，是最为简单而直接的方法，即循环使用 if 语句：

```
int GetMonthDays(int iMonth)
{
    int iDays;
    if(1 == iMonth)      {iDays = 31;}
    else if(2 == iMonth) {iDays = 28;}
    else if(3 == iMonth) {iDays = 31;}
    else if(4 == iMonth) {iDays = 30;}
    else if(5 == iMonth) {iDays = 31;}
    else if(6 == iMonth) {iDays = 30;}
    else if(7 == iMonth) {iDays = 31;}
    else if(8 == iMonth) {iDays = 31;}
    else if(9 == iMonth) {iDays = 30;}
    else if(10 == iMonth) {iDays = 31;}
    else if(11 == iMonth) {iDays = 30;}
    else if(12 == iMonth) {iDays = 31;}
    return iDays;
}
```

<sup>⊖</sup> 我所做的主要工作就是将这些大师的零碎经验重新编排，安排到了这本书层次最高的一章中。感谢各位前辈。

<sup>⊖</sup> 改写自 baidu 百科词条“表驱动”示例。

也许思维缜密逻辑清晰的读者会说，这个函数功能存在问题，它没有区分闰年的特殊情况。这的确是个问题，可却不是我们关注的重点。我们关注的是复杂的 if/else 逻辑链。上述方法绝对是一个比较笨的方法，本来应该是一件很简单的事情，代码却这么冗余，同时这也导致了代码可读性的降低。有没有好的解决方法呢？

答案是有。

我们可以先定义一个静态数组，这个数组用来保存一年十二个月的天数，然后用表来更简洁地解决这个问题：

```
static int s_nMonthDays[12] =
    {31,28,31,30,31,30,31,31,30,31,30,31};
int GetMonthDays(int iMonth)
{
    return s_nMonthDays[(iMonth - 1)];
}
```

这就是表驱动方法。

□ 为什么表驱动优于“函数封装或宏”？

简短的 switch-case 或 if/else 用起来还是比较顺手的，所以还是继续用吧。但是对于分支太多的长 switch-case 或 if/else 最好能想办法将其化解开。化解长 switch-case 的方法有很多种，是选择函数封装？！宏？！还是表驱动？！看下面的代码：

```
// 版本1 - case 分支版:
int ProcessControl(UINT function_no, void* para_in, void* para_out)
{
    int result;
    switch(function_no)
    {
        case PROCESS_A:
            result = ProcessA(para_in,para_out);
            break;
        case PROCESS_B:
            result = ProcessB(para_in,para_out);
            break;
        case PROCESS_C:
            result = ProcessC(para_in,para_out);
            break;
            //.....
        default:
            result = UN_DEFINED;
            break;
    }
    return result;
}

int ProcessA(void* para_in, void* para_out)
```

```

{
    //code....
}

int ProcessB(void* para_in, void* para_out)
{
    //code...
}

int ProcessC(void* para_in, void* para_out)
{
    //code....
}

```

分支越多，可读性越差，维护起来也越麻烦！看起来也不太美观、不太优雅，这离我们想要的优雅的 C++ 代码相比相差甚远！

那么考虑一下宏定义，代码如下所示：

```

// 版本 2 - 宏定义版:
#define DISPATCH_BEGIN(func)      switch(func) \
{
#define DISPATCH_FUNCTION(func_c, function) case func_c: \
    result = function(para_in,para_out); \
    break;
#define DISPATCH_END(code)         default: \
    result = code; \
}

int ProcessControl(UINT function_no, void* para_in,
                   void* para_out)
{
    int result;

    DISPATCH_BEGIN(function_no)
        DISPATCH_FUNCTION(PROCESSA,ProcessA)
        DISPATCH_FUNCTION(PROCESSB,ProcessB)
        DISPATCH_FUNCTION(PROCESSC,ProcessC)
        // .....
    DISPATCH_END(UN_DEFINED)

    return result;
}

// ProcessA、ProcessB、ProcessC 定义同上（略）

```

这个版本的代码稍稍有了点清爽的感觉。但是用函数封装或宏取代 case 块是治标不治本的方法。如果想彻底治疗这种顽症，最有效的方法还是使用表驱动。

首先，定义一个数据结构来表示函数序号 fun\_no，并处理函数之间的对应关系；接着，用表将这种对应关系存储起来。当需要调用函数时，只需查表寻找。代码如下所示：

```
// 版本 3 - 表驱动版：
typedef struct tagDispatchItem
{
    UNIT func_no;
    ProcessFuncPtr func_ptr;
}DISPATCH_ITEM;

DISPATCH_ITEM dispatch_table[MAX_DISPATCH_ITEM];

int ProcessControl(UINT function_no, void* para_in, void* para_out)
{
    int i;

    for(i = 0; i < MAX_DISPATCH_ITEM; i++)
    {
        if(function_no == dispatch_table[i].func_no)
        {
            return
                dispatch_table[i].func_ptr(para_in,para_out);
        }
    }
    return UN_DEFINED;
}
```

上述方法中采用的是数组形式，还可以换为高级数据结构，如下所示：

```
// 版本 4 - 表驱动版（高级数据结构）：
typedef std::hash_map<UINT, ProcessFuncPtr> CmdHandlerMap;
CmdHandlerMap HandlerMap;

void InitHandlerMap()
{
    HandlerMap[PROCESSA] = ProcessFuncPtr(&ProcessA);
    HandlerMap[PROCESSB] = ProcessFuncPtr(&ProcessB);
    HandlerMap[PROCESSC] = ProcessFuncPtr(&ProcessC);
    // .....
}

int ProcessControl(UINT function_no, void* para_in,
                  void* para_out)
{
    CmdHandlerMap::iterator it =
        HandlerMap.find(function_no);

    if(it!=HandlerMap.end())
    {
        ProcessFuncPtr pHandler = it->second;
```

```

        return (*pHandler)(para_in,para_out);
    }
    return UN_DEFINED;
}

```

使用表驱动的好处就是 ProcessControl 的代码就这么几行，如果要添加新的功能，只需要维护驱动表 dispatch\_table 或 HandlerMap 就行了，这就摆脱了冗长乏味的 switch-case。

所以，表驱动优于函数封装和宏代替。

#### □ 研究一下 MFC 中的表驱动

也许有人会用 MFC 做相当漂亮的的应用程序，也许有人认为自己对 MFC 很熟悉，但是有人发现了程序中隐藏的表驱动没？有人知道它是怎么实现的么？

在 MFC 的程序声明文件中，你会经常使用这么一句话：DECLARE\_MESSAGE\_MAP()，而在对应的定义文件中，也会加上这么几句：

```

BEGIN_MESSAGE_MAP(CTestMFCApp, CWinAppEx)
    ON_COMMAND(ID_APP_ABOUT, &CTestMFCApp::OnAppAbout)
    ON_COMMAND(ID_FILE_PRINT, &CWinAppEx::OnFilePrint)
END_MESSAGE_MAP()

```

其实，表驱动就隐藏在这些语句的后面，查看 DECLARE\_MESSAGE\_MAP() 的宏定义你会发现：

```

#define DECLARE_MESSAGE_MAP()
protected: \
static const AFX_MSGMAP* PASCAL GetThisMessageMap();
virtual const AFX_MSGMAP* GetMessageMap() const; \

```

它定义了两个函数，返回值是一个 AFX\_MSGMAP\*。这又是什么呢？继续往下搜寻：

```

struct AFX_MSGMAP
{
    const AFX_MSGMAP* (PASCAL* pfnGetBaseMap)();
    const AFX_MSGMAP_ENTRY* lpEntries;
};

```

这是 Window message map，而这就是我们寻找已久的“表”啊。

再返回去，看看 BEGIN\_MESSAGE\_MAP(CTestMFCApp, CWinAppEx) 和 END\_MESSAGE\_MAP() 到底是什么：

```

#define BEGIN_MESSAGE_MAP(theClass, baseClass) \
PTM_WARNING_DISABLE \
const AFX_MSGMAP* theClass::GetMessageMap() const \
{ return GetThisMessageMap(); } \
const AFX_MSGMAP* PASCAL theClass::GetThisMessageMap() \
{ \
    typedef theClass ThisClass; \
}

```

```

typedef baseClass TheBaseClass; \
static const AFX_MSGMAP_ENTRY _messageEntries[] = \
{

#define END_MESSAGE_MAP() \
{ 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 } \
}; \
static const AFX_MSGMAP messageMap = \
{ &TheBaseClass::GetThisMessageMap,\ 
    &_messageEntries[0] };\
return &messageMap; \
} \
PTM_WARNING_RESTORE

```

把宏替换掉，得到代码：

```

const AFX_MSGMAP* theClass::GetMessageMap() const
{
    return GetThisMessageMap();
}

const AFX_MSGMAP* PASCAL theClass::GetThisMessageMap()
{
    typedef theClass ThisClass;
    typedef baseClass TheBaseClass;
    static const AFX_MSGMAP_ENTRY _messageEntries[] =
    {
        { WM_COMMAND, CN_COMMAND, (WORD)id, (WORD)id, AfxSigCmd_v, static_
            cast<AFX_PMSG> (memberFxn) },
        { WM_COMMAND, CN_COMMAND, (WORD)id, (WORD)id, AfxSigCmd_v, static_
            cast<AFX_PMSG> (memberFxn) },
        { WM_COMMAND, CN_COMMAND, (WORD)id, (WORD)id, AfxSigCmd_v, static_
            cast<AFX_PMSG> (memberFxn) },
        //.....
        { 0, 0, 0, 0, AfxSig_end, (AFX_PMSG)0 }
    };

    static const AFX_MSGMAP messageMap =
    {
        &TheBaseClass::GetThisMessageMap,
        &_messageEntries[0]
    };

    return &messageMap;
}

```

突然之间，豁然开朗！

## □ 总结

Table-drive 表驱动法是一种替代逻辑语句（if / else）的编程模式，其优点是简单清晰，

尤其是在判断分支较多的情况下，另一个显在的优势是可以放在文件中读取，这样一来，就可以在不改变源代码的前提下更改一些表内容了，正合元编程思想。

---

**请记住：**

用表驱动替代冗长的逻辑语句，遵循元编程思想。

---

### 建议 138：为应用设定特性集

一个更好的 C (A Better C) 是 C++ 语言的设计目标之一，所以 C++ 被认为是 C 的优化，是 C 的扩充。与 C 语言相比较，C++ 语言具备很多梦幻的高水平特征。这些特征在带来便利的同时，也会要我们为此付出一定的代价，例如过分复杂、编译及运行时性能的损耗、维护成本的增加等。

那么我们是否需要这些梦幻特征呢？

C++ 之父 Stroustrup 曾有言曰：

*Just because you can do it, doesn't mean that you have to.*

翻译过来就是：你可以使用并不意味着你必须使用。

所以在对待 C++ 高级特性的态度上一定要谨慎，是否有必要使用多重继承、异常、RTTI、模版及模版元编程，一定要做一个审慎的评估，以便为应用选择合适的特征集，避免使用过分复杂的设计和功能，否则将会使得代码难于理解和维护。

---

**请记住：**

C++ 为我们提供了丰富的基础设施，但是对于一些高级特性我们可以选择放弃，可以根据应用的具体情况来设定特定的特征集。

---

### 建议 139：编码之前需三思

在文章《关于编程，鲜为人知的真相》中，作者给出了这样一段话：

“编程是个很难的工作。是一种剧烈的脑力劳动。好的程序员 7×24 小时地思考他们的工作。他们最重要的程序都是在淋浴时、睡梦中写成的。因为这最重要的工作都是在远离键盘的情况下完成的，所以软件工程不可能通过增加在办公室的工作时间或增加人手来加快进度。”

这的确是事实，是一个为众多程序员所认同的事实。

C++ 之父 Bjarne Stroustrup 也有过类似的言论。他说：“我不是使用支持工具进行巧妙设

计的信徒，但是我强烈支持系统地使用数据抽象、面向对象编程和泛型编程。不拥有支持库和模板，不进行事先的总体设计，而是埋头写下一页页的代码，这是在浪费时间。这是给维护增建困难。”

所以需要在编码前进行思考，我们需要三思而后行。

我们需要认真的思考，细致的设计。就像《Want to write some code? Get away from your computer!》<sup>⊖</sup>文中所说的那样，“最好的写程序的地方不是在你的计算机前，不是使用你的编译器、IDE或其他一些工具。最好的地方是一个远离这些工具的场所——是某个能让你认真的思考的地方。对于一个你很熟悉的编程语言，你很容易把你脑子里已经构思好的程序转换成编译器 / 解释器可以编译 / 解释的程序——难就难在如何在脑子里先把程序编好。”（如图 13-1 所示）。

也许有人已经养成了这样一种编程流程：写点乱糟糟的代码→编译运行→测试→调试发现 Bug→修复 Bug→重复上述过程（如图 13-2 所示）。

这会导致程序的可读性极差，同时还很有可能会潜藏着很多问题。

解决这一问题的最有效建议就是：远离计算机，迫使你在大脑里周全地思考所有的问题，在纸上好好地规划一下程序，把思路画在纸上，思考它，做出高质量的、高效的、没有问题的程序，最后将它从纸上拷贝到计算机编译器里。

所以要先想好，后编程（Think first, Program later），就像 Wisconsin 大学的 Roy Carlson 所说的那样：

你开始编码的时间越早，项目持续的时间就越长。<sup>⊖</sup>

最后，请牢记：编码之前先三思。

### 请记住：

做任何事之前都要三思而后行，编程也不例外。在让电脑运行你的程序之前，先让你的大脑编译运行。



图 13-1 Want to write some code? Get away from your computer!

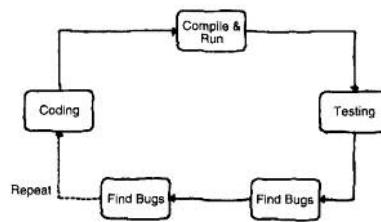


图 13-2 编程流程图

<sup>⊖</sup> 它是 blog.rtwilson.com 上的一篇博客。

<sup>⊖</sup> The sooner you start to code, the longer the program will take.

## 建议 140：重构代码

每一行代码都浸透着作者深厚的思想。看似简单的符号，却“孕育在苦思冥想之中，生长于严谨的设计之下，绽放在千锤百炼的雕琢之后”。正是经过了如此这般的先天孕育和后天雕琢，才使得代码具有了艺术品般的美丽。细细品味，每一段代码无不闪烁着智慧的光辉。

当你闲暇下来回顾整理几天前所写的代码时，会发现很多地方写得都不够好，代码中竟然存在着如此之多的“坏味道”。所以，我们需要对代码进行重构。代码之于程序员就好比艺术品之于艺术家，程序员对代码品质的追求绝不亚于艺术家对美丽的渴望，只有经过精雕细琢、千锤百炼，才可以让你的代码更加美丽！

看下面的这个例子，一起来探究什么是精雕细琢、千锤百炼的好代码：

```
void PrintHello()
{
    CTextHandler strHandler;
    strHandler.SendText("Hello 2012", true);
    // ...
}
```

函数 `SendText` 中的第二个参数代表什么含义？作为读者的我们很难猜明白。所以，我们只好转到函数声明处：

```
class TextHandler
{
public:
    void SendText( const std::string& msg,
                   bool send_new_line );
    //...
};
```

原来，它表示“是否要自动加上一个回车换行”。

源代码最主要的用途是交流，是对意图的交流。正如著名的计算机科学家和作家 Harold Abelson 与 Gerald Jay Sussman 在《The Structure and Interpretation of Computer Programs》所说的那样，“代码的主要功能是供别人阅读，其次才是计算机执行（Programs must be written for people to read, and only incidentally for machines to execute）”。上述的代码片段在交流上似乎存在着一定的问题，`bool` 型参数不能传递任何有用的信息给读者，它没有“使用明确、详细、具体的语言<sup>⊖</sup>”来实现艺术的交流。那么，我们来尝试着改变一下这个设计。

最容易想到的方法就是去掉第二个参数，让用户自己加‘\n’，代码如下所示：

---

<sup>⊖</sup> William Strunk Jr. and E.B. White. *The Elements of Style* (MacMillan Publishing Co. Ltd, 1979)

```

class TextHandler
{
public:
    void SendText( const std::string& msg );
    //...
};

CTextHandler strHandler;
strHandler.SendText("Hello 2011\n"); // 换行
strHandler.SendText("Hello 2012");   // 不换行

```

这样的设计看似完美，但是一旦 string 换作其他类型或 bool 类型所表示的意义有变化，这样的方法就会宣告失败：

```
void SendText( const std::string& msg,    bool apply_bold );
```

其中，第二个参数表示是否加粗显示。这样一来上述方案就进入了死胡同。

如果为不同的功能提供不同的函数呢，这样的方式是否行得通？代码如下所示：

```

void SendText( const std::string & msg );
void SendBoldText ( const std::string & msg );

```

这样做的话，一旦参数增加，我们需要提供的函数就会一下子增加很多，如下所示：

```

class TextHandler
{
public:
    void SendText( const std::string& msg,
                  bool apply_bold,
                  bool apply_italics );
    //...
};

CTextHandler strHandler;
strHandler.SendText("Hello 2012", true, false);

```

函数参数依旧不明确、不详细、不具体，而且，上述函数中的第二个和第三个参数都是 bool，容易混淆，要是出现参数顺序错误，编译器也绝对不会发现。采用这个方案虽然可以工作，但是需要提供太多的函数，繁缛难当，如下所示：

```

void SendBoldItalicsText( const std::string& msg );
void SendBoldText( const std::string& msg );
void SendItalicsText( const std::string& msg );
void SendText( const std::string& msg );

```

同时也违背了 Don't Repeat Yourself 原则。

在设计过程中，我们不仅要实现需求中的功能，还要注意代码的可读性和可扩展性。显然上述方案都难以让我们满意，这两种方案的代码设计很难称得上“美”。

那该怎么办呢？

使用枚举！代码如下所示：

```
class CTextHandler
{
public:
    enum TEXT_FORMAT
    {
        WITH_NEW_LINE,
        NO_NEW_LINE,
    };
    void SendText( const std::string &, TEXT_FORMAT );
};

CTextHandler t;
t.SendText( "Hello, ", CTextHandler::WITH_NEW_LINE );
t.SendText( "world", CTextHandler::NO_NEW_LINE );
```

这种写法很好，代码可以做到 self-documenting，不需注释，其意图和结果就能表达得很清楚、可读性较好。同时，其扩展性也有所加强，如果需要显示为红色字体，只需要在 enumeration 类型中加上 WITH\_RED\_COLOR 即可，现有的代码无须任何改变，如下所示：

```
CTextHandler t;
t.SendText( "Hello, ", CTextHandler::WITH_NEW_LINE );
t.SendText( "world", CTextHandler::NO_NEW_LINE );
t.SendText( "Hello China", CTextHandler::WITH_RED_COLOR );
```

经过如此的反复斟酌，我们终于得到了目前最好的 SendText 函数设计。而这个过程就是代码重构。这就是代码精雕细琢、千锤百炼的结果，代码在重构中已完成蜕变。

#### 请记住：

重构无止境，重构你的代码，精雕细琢，千锤百炼。

### 建议 141：透过表面的语法挖掘背后的语义

通常，语法是表象，语义是表象后面隐藏的东西，而这些隐藏的语义往往更具有价值。举个例子，是关于 public 继承与 private 继承的，如下所示：

```
// public 继承（汽车是一种交通工具）
class CVehicle
{
public:
    CVehicle();
    void Move();
```

```

};

class CCar : public CVehicle
{
public:
    CCar() : CVehicle(){}
};

// private 继承（车需要有一个引擎）
class CEngine
{
public:
    CEngine(int nCylinders);
    void Start();
};

class CCar : private CEngine
{
public:
    CCar() : CEngine(4){}
    using CEngine::Start;
};

```

**public** 继承与 **private** 继承在语法方面似乎没有更多的东西值得探讨，它们的区别仅仅在于改变了继承得到的成员的可见性，但是从语义方面来分析，它们就相差太远了，**private** 继承在语义上来讲是“通过基类来实现自己”，即“实现继承”，在这种继承关系中，基类和子类的关系是很薄弱的，私有继承可以看作是组合（Composition）在语法表达上的一个变种。在这种“类组合”关系中，每个 CCar 只能有一个引擎 CEngine，它不支持多引擎设计。而 **public** 继承在语义上则是我们所熟知的“Is-A”关系，它体现了基类和子类之间的亲密性，也正是这种“Is-A”关系为多态性提供了基础。

再比如，建议 126 中关于访问限定符与 **virtual** 组合，其背后也有着深刻的语义：

- (1) 基类中的一个虚拟私有成员函数，表示实现细节是可以被派生类修改的。
- (2) 基类中的一个虚拟保护成员函数，表示实现细节是必须被派生类修改的。
- (3) 基类中的一个虚拟公有成员函数，则表示这是一个接口，不推荐，建议用 **protected virtual** 来替换。

所以，通过表面的语法来挖掘其背后的语义很有意义，挖掘出这些语义，能够加深我们对 C++ 语言设计的理解，帮助我们在以后的设计中做出恰当的抉择。

#### 请记住：

通过表面的语法来挖掘其背后的语义，加深我们对 C++ 语言设计的理解，帮助我们在以后的设计中做出恰当的抉择。

## 建议 142：在未来时态下开发 C++ 程序

我不敢掠美，这是大师 Scott Meyers<sup>Θ</sup>留给我们的一条重要建议。如果将这条建议翻译成为中文成语的话，那就是“未雨绸缪”，即为我们现在写下的代码的未来做出一些必要的考虑，这样，我们的代码才会保持与时俱进，才能适应新的变化，满足新的需求。

也就是说眼光要放长远一点，把 C++ 代码将来的需求变化纳入到我们设计类时所考虑的范畴之内。

一般说来，以下几种情况可能会导致我们开发的 C++ 程序库发生变化（如图 13-3 所示）。

所以，在未来时态下开发 C++ 程序，我们需要考虑代码的可重用性、可维护性、健壮性，以及可移植性。

要实现以上这几点，我们需要在以下几方面做出努力：

对我们不想让类具备的一些行为进行明令禁止。莫菲法则——

“只要是能被人做的，就会有人这么做”告诉我们，使用简单的文档说明是绝对不够的，我们需要用 C++ 的方式来进行约束，使得我们设计的类“易于被正确使用而难以误用”。比如：

(1) 控制对象是否在堆上的创建，可采用建议 128。

(2) 控制对象的生成个数，可采用建议 129。

(3) 如果拷贝构造和赋值声明无意义，则将其声明为私有。

.....

努力让类的操作和函数具有自然的语法和直观的语义，严格遵守最小惊讶法则，这就要求我们在类的命名上下点工夫了，运算符重载遵循着其通常的含义。

尽量避免使用一些影响代码可移植性的语言特征，尽可能地使用封装，为将来可能的平台移植做好准备。

提供完备的类，为将来可能出现的新需求做好准备。像建议 140 那样重构代码，优化设计，提高代码的扩展性。

如果没有限制你不能通用化你的代码，那么通用化它。

.....

当然这些还要与正在进行时时态的约束条件相互比较，并进行取舍；既要满足现阶段的需求，也要尽量保证软件的长期生存能力，在二者之间找出一个平衡点。

### 请记住：

在设计中对将来时态的需求做一定的考虑，让我们的代码与时俱进，适应新的变化，满足新的需求。

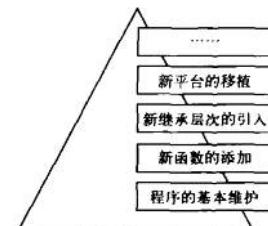


图 13-3 C++ 程序未来时态的可能变化

<sup>Θ</sup> Effective C++ 系列的作者，不多说，你们都懂的。

## 建议 143：根据你的目的决定造不造轮子

软件领域有一个著名的描述软件重用的谚语：Don't reinvent the wheel（不要重复造轮子）。这也是很多软件工程专家给我们的建议。在面向对象的语言中已为我们提供了大量的库和类，如果我们善于利用它们，不仅能节省我们的成本，还可以使重用和更新变得更方便容易。

这也是一个各行各业通用的道理。不重复造轮子，是因为对大多数行业内的问题而言，彼此间相像的地方要多于彼此间有差异的地方。这意味着，用较少的技巧就可以解决范围较大的问题。将这些经验技巧总结升华，就形成了解决这类问题的通用工具箱，也就是轮子。在编程语言中这些轮子表现为大量的通用类和库。

所以，在工程实践中，不要造轮子，只要用轮子就好了。

但是，如果你是一个计算机专业的学生或某项技术的爱好者研究者，目的是学习，那造轮子则是必不可少的。因为“用轮子”之于“造轮子”，就好比“知其然”之于“知其所以然”，我们要在造轮子的过程中学习其设计思想，提高创新。

所以，如果是在学习研究中，不要只用轮子，更要造轮子。

**请记住：**

在工程实践中，不要重复造轮子；而在学习研究中，鼓励重复造轮子。

## 建议 144：谨慎在 OO 与 GP 之间选择

Bjarne Stroustrup 是这样描述自己发明的 C++ 语言的：

它是一种多用途编程语言，它支持过程式编程、数据抽象、面向对象编程和泛型编程。

面向对象（OO）和泛型编程（GP）是 C++ 提供给程序员的两种矛盾的思考模式。OO 是我们难以割舍的设计原则，世界是对象的，我们面向对象分析、设计、编程；而泛型编程则关注于产生通用的软件组件，让这些组件在不同的应用场合都能很容易的重用。

也许你会注意到上面的“矛盾”二字，之所以用这两个字来形容二者之间的关系，是因为二者拥有相左的设计思路：在面向对象的编程中，正确地定义类型举足轻重、不可或缺；而在泛型编程中，类型系统则倾向于逃离程序设计者的控制。这些类型往往彼此缺乏联系，即使它们之间存在着事实上的紧密关系。两种设计规则对类型的态度似乎是水火不两立的。

那么，OO 和 GP 哪个更好呢？

这样的争端充斥在诸多的 C++ 技术社区里，但是都没有一个合理明确的答案。因为我们不能够完全抛弃其中的任何一个，要成功设计实现一个大型的软件系统，就必须同时借助 OO 原则和泛型编程二者的力量，缺少其中任何一个，都不会是有前途的优秀方案。

正如下面这段话所说的那样：

优秀的工程设计，总是包含了很多妥协。优秀的、可交付并投入使用的产品，从来都不会仅仅依靠某项定律或技术标准；高超的工程艺术，永远不会是寻找并应用单独某项方法，而是清晰知道自己的可选择项。

因此，OO 和 GP 我们都得依靠，我们需要在 OO 纯度和 GP 通用及高效之间审慎权衡。

#### 请记住：

OO 和 GP 是 C++ 语言支持的两种矛盾的设计思想；虽然矛盾，但是各有千秋，不要争论孰优孰劣，请在合适的时候使用合适的特性。

## 建议 145：让内存管理理念与时俱进

C/C++ 语言的内存管理历来被认作是程序开发中的难点。为了更好地利用内存这一最宝贵的系统资源，C++ 前辈们为此付出了很多的努力，进行了几次比较重大的变革，但是稍有遗憾的是相关技术尚未成熟。

### 第一次变革：从 malloc/free 到 new/delete

这场变革是 OOP 技术兴起的产物。C++ 是强类型语言，new/delete 的主要成果也就是加强了类型观念，减少了强制转型的需求。但是从内存管理的角度来看，这个变革并没有多少突破性。

### 第二次变革：从 new/delete 到内存配置器（allocator）

内存池是一个非常基础也非常关键的底层库，在建议 35 中我们已经详细阐述过。设计一个优秀的内存池对提高系统的效率和稳定性非常有帮助，尤其是针对小内存对象（一般低于 128 字节）而言。因为对象的分配和释放非常频繁，只用简单的 malloc/free 或 new/delete 来处理非常影响效率，这不是一个优秀的设计。所以就有了新的解决方案 allocator 的诞生。自从 STL 被纳入 C++ 标准库后，C++ 世界就发生了巨大的变化。而从内存管理的角度来看，allocator 的引入也是 C++ 内存管理的一个突破。整个 STL 所有组件的内存均从 allocator 分配。也就是说，STL 并不推荐使用 new/delete 进行内存管理，而是推荐使用 allocator。

SGI STL 中的 allocator 被认为是目前设计得最优秀的 C++ 内存分配器之一。它是一个提供内存管理相关服务的类，其功能包括内存分配、释放、回收等。它的基本思路是这样的：

(1) 设计一个 free\_list[16] 数组，负责管理从 8 bytes 到 128 bytes 的不同大小的内存块 (chunk)，每一个内存块都由连续的固定大小的很多内存块组成，并用指针链表串接起来，

如图 13-4 所示。

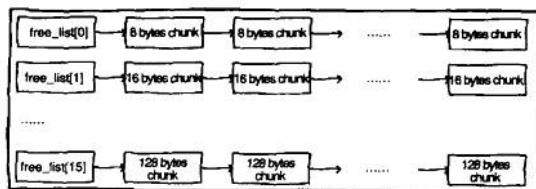


图 13-4 allocator 原理示意图 (1)

(2) 当用户想要获取特定大小的内存时，就在对应大小的 free\_list 链表中找一个最近的内存块回传给用户，同时将此块从链表中删除，即把这个块的前后内存指针链接起来，如图 13-5 所示。

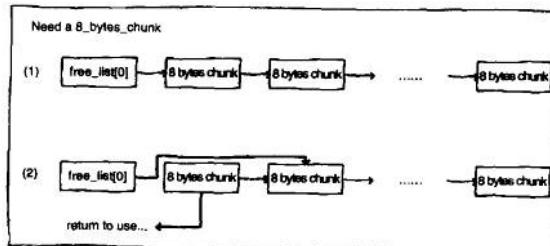


图 13-5 allocator 原理示意图 (2)

(3) 当用户使用完毕，释放该内存块后，该内存块就会重新插入到 free\_list 中（如图 13-6 所示）。

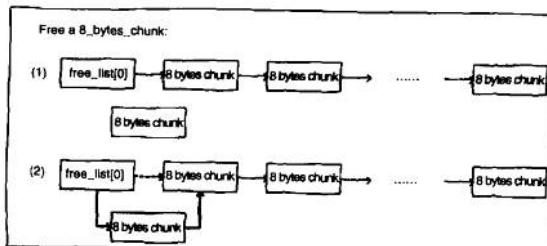


图 13-6 allocator 原理示意图 (3)

(4) 当 free\_list 中的内存不够时，allocator 会自动再分配一块新的较大的内存区块来加入。能够管理多种大小不同的内存块、内存池，使其可以根据需求的变化自动增长，这就是 SGI STL 内存分配器设计的特点。

STL 内存管理器 allocator 的引入使得内存管理从容器的实现中独立了出来，它虽然没能

为广大的 C++ 程序员带来观念上的转换，没能让我们从使用 new/delete 进行繁琐的内存分配 / 释放的过程中解放出来，却有着比较深刻的意义。

每个类都有着自身的特点，所以也有着与之对应的、最合适它的内存管理机制。allocator 就是根据具体情况具体分析，为 STL 容器类定制的特殊的内存申请方式。这与 C++ 传统的做法——使用一个全局的 new/delete 很是不同（虽然我们可以为类定义自己的 new/delete，但是其理念和层次完全不同）。

所以，在坚持 new/delete 老思想的基础上，接受 STL allocator 的新思维，让它们更好地为我们的设计服务。

---

#### 请记住：

学习 STL allocator，更新内存管理理念。

---

## 建议 146：从大师的代码中学习编程思想与技艺

读万卷书，行万里路。这句话之于程序员依旧有效。著名的 Lisp 程序员 Peter Norvig <sup>Θ</sup> 曾有言曰：阅读伟大的代码之于程序员，就如同阅读伟大作品之于作家。阅读别人的代码是很有价值的，大师们的思想和技艺都凝聚在他们的代码里；阅读他们的代码，从中学习他们的思想与技艺，就等于站在巨人的肩膀上，优化我们的设计，提高我们的能力。

阅读代码还是一种能力。

要想当一个称职的开发人员，熟练地阅读代码非常重要。现在的开发项目一般都是团队协作的成果，你需要阅读团队其他成员的代码，理解它们的设计意图，或修改，或扩展，或维护。

阅读代码需要耐心。

第一次接触一个重要的代码库时，通常需要花很长的时间去读懂，你要保持你的耐心，数天、数周，甚至是数月，坚持不懈，直至成功。

阅读代码需要方法。

阅读代码不是盯着看，还需要消化理解，将其融入到自己的血液中。就像我们读书一样，阅读代码也要先“从厚读薄”，然后再“由薄读厚”。换句话说，刚开始不要纠结于代码的细节，将关注的重点放在代码的高层结构上，理解代码的构建过程；之后，再有重点的深入研究，理解对象的构造，明晰算法的步骤，尝试着深入理解其中的繁杂细节。当然，在这个过程中，最好穿插着实践，写几个小段程序，调试一下，可加深理解。

阅读代码需要正确的态度。

---

<sup>Θ</sup> 专家一个，现任 Google 研发总监。

端正态度对于任何一件事来说都非常重要，代码阅读也不例外。如果你不重视代码阅读，对其嗤之以鼻，那么它确实是分文不值的；如果你将它当成是一次与大师交流的机会，那么你就会从中得到很多。态度虽然不能决定一切，但是会决定很多。

优秀的代码是我们成长过程中的武林秘籍；代码阅读也是我们成为 C++ 专家过程中不可或缺的阶段与步骤。正确地对待代码阅读，从大师的代码中学习程序之中浸染的思想和技艺。

---

请记住：

读万卷书，行万里路；端正对代码阅读的态度，学习大师的思想与技艺。

---

### 建议 147：遵循自然而然的 C++ 风格

C++ 语言之父 Bjarne Stroustrup 曾经苦口婆心地告诉我们：

C++ 语言是一个面向解决实际问题的语言，而不是效忠于某个编程泛型的宗教语言，不要过分地执迷与所谓的奇技淫巧，而应当加强基础设计建设，具体问题具体分析。

换句话说，当你面对一个特定问题的时候，应采用最自然有效的方式来解决。

C++ 为我们提供了足够丰富的特性，我们的选择可能是面向对象，可能是基于模板，或者是源自 C 语言的过程化设计。不必执着于所谓的高级技术，选择实现最简单的那一个。当然，当问题变得复杂时，我们就要充分利用标准库的基础设施了，如果灵活自如地组合函数、模板和面向对象的机制，恰到好处地运用指针，就可以优雅地解决这些问题。不必为了特定的思想而扭曲设计，死守陈规，使用那些过度聪明的设计。

软件的设计之“道”，不在于设计有多么的华丽、精巧，而在于其朴实、自然，这是“无招胜有招”的至高境界。

道法自然。

---

请记住：

用最自然有效的方式解决特定的 C++ 设计问题，遵循自然而然的 C++ 风格，道法自然。

---

### 建议 148：了解 C++ 语言的设计目标与原则

在日常的学习中，我们一直强调：“知其然，更要知其所以然”。对于 C++ 的学习也不例外。接下来，我们要讨论的是 C++ 语言的设计目标与原则，这是 C++ 之父在创造 C++ 语言时曾深思过的，了解这些，可以让我们对 C++ 语言本身有一个更加深刻的认知。

### □ C++ 的设计目标 (C++ Design Aims)

C++ 的设计目标，就是要让 C++ 既具有适合系统程序设计的 C 语言所拥有的可适应性和高效性，又能在其程序组织结构方面具有像 Simula 那样的语言设施（面向对象的程序设计风格）。并且为了使 Simula 的高层次的程序设计技术能够应用于系统程序的设计之中，在设计时还做了很大的努力。也就是说，C++ 所提供的抽象机制能够应用于那些对效率和可适应性具有极高要求的程序设计任务之中。

这些设计目标可以总结为如下两点：

(1) 对于要解决实际问题的程序员而言，C++ 使程序设计变得更有乐趣。

(2) C++ 是一门通用目的的程序设计语言，它：

- 是一个更好的 C；
- 支持数据抽象；
- 支持面向对象程序设计；
- 支持范型程序设计。

关于对范型程序设计的支持，在 C++ 设计的后期才被作为一个明确、独立的目标来实现。而在 C++ 演化过程的大部分时间里，一直是把范型程序设计及支持它的语言特性划归在“数据抽象”的大标题之下的。

### □ C++ 的设计原则 (Design Principles)

在 Stroustrup 关于 C++ 涉及规则的阐述中，将其分为了基本规则、基于设计的规则、语言的技术性规则，以及基于低层次程序设计的规则四个方面：

#### 基本规则 (General rules)

- (1) C++ 的每一步演化和发展必须是由于实际问题所引起的。
- (2) C++ 是一门语言，而不是一个完整的系统。
- (3) 不能无休止地一味追求完美。
- (4) C++ 在其存在的“当时”必须是有用处的。
- (5) 每一种语言特性必须有一个有根据的、明确的实现方案。
- (6) 总能提供一种变通的方法。
- (7) 能为意欲支持的每一种程序设计风格提供易于理解的支持方法。
- (8) 不强制于人。

基本规则的最后三条其实是在强调 2 个问题：一是对适用于真实世界中各种应用的便捷工具的强调；二是对程序员的技术和取向（偏好）的充分考虑。从 C++ 诞生伊始，它面向的就是那些要做实际项目的程序员。所谓的“完美”被认为是不可能达到的，这是因为 C++ 用户在需求、背景和待解决的问题上存在着太大的不同。况且，在一门通用目的的程序设计语言的整个生存期之内，对“完美”一词的诠释都可能会有极大的改变。由此可知，在语言的演化过程中，来自用户的反馈和语言实现者们积累的经验才是最为重要的。

### 基于设计的规则 (Design-support rules)

- (1) 支持良好的设计方案。
- (2) 提供用于程序组织的语言设施。
- (3) 心口如一 (Say what you mean)。
- (4) 所有的语言特性必须具有切实有效的承受能力。
- (5) 开启一个有用的特性比避免所有的误用更为重要。
- (6) 能将独立开发的部件组合成完整的软件。

C++ 的一个目标就是提供更易用并具有一定承受能力的设计思想和程序设计技术，进一步提高程序的质量。这些技术中的绝大部分都源自 Simula，并通常会被作为面向对象程序设计和面向对象设计思想来讨论。然而，C++ 的设计目标总还是在于要支持一定范围内的各种程序设计风格和设计思想，这与一般在语言设计方面的观点形成了一定的对比。一般在语言设计上总是试图将所有系统内建于被单独重点支持的、带有强制性的程序设计风格之中。

### 语言的技术性规则 (Language-technical rules)

- (1) 与静态型别系统 (Static type system) 没有内在的冲突。
- (2) 像对内建 (built-in) 类型一样对用户自定义类型提供很好的支持。
- (3) 个性化 (locality) 行为是可取的。
- (4) 避免产生顺序上的依赖关系。
- (5) 在对语言产生疑惑时，可以选取其特性中最易掌握的部分。
- (6) 可以因为不正当的语法使用而产生问题 (Syntax matters (often in perverse ways))。
- (7) 削弱对预处理器的使用。

这些规则要具体结合更多关于基本目标的上下文环境来考虑。应该注意的是，“与 C 有较高的兼容性”、“不损失效率”，以及“具有便捷的可用性来解决实际问题”这三个方面的要求与“完整的型别安全性”、“完全的通用性”及“完善的抽象之美”这三个方面的要求形成了对立。

C++ 从 Simula 中借鉴了用户自定义类型和类层次机制。然而，在 Simula 及许多类似的语言中，其对用户自定义类型的 support 与其对内建类型的支持存在着根本上的不同。例如，Simula 中不允许在栈中为用户自定义类型的对象分配空间，并且只允许通过指针（在 Simula 中称为引用——reference）来对这些对象进行访问。至于其内建类型的对象则只能在栈中分配空间，不能在动态存储区中分配，并且不能使用指针指向它。这种在对待内建类型与对待用户自定义类型上的差异，暗示着对效率问题的严格考虑。比如，当一个在动态存储区中被分配的对象被作为引用使用时，如果该对象属于自定义类型，那么就会为运行期及空间带来负荷，而且这些负荷在有些应用中被认为是不可接受的。这些正是 C++ 意欲解决的问题。同时，二者在用法上的不同也决定了：不可能在范型程序设计中统一对待那些语义上近似的类型。

在维护一个较庞大的程序时，一个程序员不可避免地会基于某些不完整的知识来对程序

做一些修改，而在做这些修改时，最好只关注全部程序代码中的一小部分。基于此，C++ 提供了 class、namespace 和访问控制，这使得设计决策的各差异化成为可能。

在基于一趟编译（one-pass compilation）的语言中，某些顺序上的依赖性是不可避免的。例如在 C++ 中，一个变量或函数在被声明之前是无法使用的。然而，C++ 中类成员的名字规则和重载解析（overload resolution）的规则还是在独立于声明顺序的原则下被制定出来的，以便将发生混乱和错误的可能性降至最低。

#### 基于低层次程序设计的规则（Low-level programming support rules）

- (1) 使用传统的（笨拙的）连接器（linker）。
- (2) 与 C 语言不存在无故的不兼容性。
- (3) 不给 C++ 层级之下的更低层语言留出余地（汇编语言除外）。
- (4) 你不会为你所不使用的部分付出代价（零负荷规则）。
- (5) 在产生疑惑时，能提供完全自主控制的途径。

在 C++ 的设计中只要是不严重影响其对强类型检查（strong type checking）支持的地方，都尽量做到与 C 的“source-link”方式相兼容。除了某些微小的细节差别之外，C++ 将 C 作为一个子集包含了进来。C++ 与 C 的兼容性使得 C++ 程序员立刻就能有一个完整的语言和工具集可用。还有两点也很重要，一是有大量关于 C 的高质量的教学素材已经存在；二是 C++ 程序员可以利用 C++ 与 C 的兼容性直接有效地使用大量现成的程序库。在决定将 C 作为 C++ 的基础时，C 还没有像后来那样出类拔萃，所以在考虑这个问题的时候，与 C 语言所提供的可适应性和高效性相比，C 语言的流行程度只是个次要的考虑因素。

然而，与 C 的兼容性也使得 C++ 在某些语法和语义上保留了 C 的一些瑕疵之处。比如，C 语言的声明语法就实在远达不到优美程序的要求；而其内建类型的隐式转换规则也是混乱无章法的。还有另一个大问题，就是许多从 C 转向 C++ 的程序员并没有认识到，代码质量上的显著提高只能通过在程序设计风格上的显著改变来达到。

---

#### 请记住：

目的与原则对于一个语言的特性有着决定性的作用。了解 C++ 语言的设计目的及设计原则，可加深对 C++ 语言本身的认知。

---

### 建议 149：明确选择 C++ 的理由

在编程语言争奇斗艳的今天，为什么要选用 C++ 呢？更何况，Linux 之父 Linus Torvalds 曾经很是不满地炮轰 C++，称其为“糟糕程序员的垃圾语言”。我们是不是上错了船呢？

我们需要给自己一个坚定的理由。

也许，你首先想到的就是：效率。

1979 年 Bjarne Stroustrup 发明了一门新编程语言 C with classes，它被定位为 a better C。它就是我们现在说的 C++，它继承了 C 的特色，既为高级语言，又含低级语言功能，可同时作为系统和应用编程语言。所以，在一般的场合下，C 和 C++ 语言的主要用途就是系统级软件的开发。其中的最大原因就是：效率。然而，C 和 C++ 无论是在时间效率还是空间效率上都不相上下，甚至是 C 更具优势，因为通常情况下用 C 语言写成的程序运行得更快。

那么是不是在所有注重效率的领域，选择 C 就一定比选择 C++ 更好呢？这也未必，我们还是要具体情况具体分析。真正影响选择语言的因素是业务逻辑。我们需要在 C++ 的 OOP/GP 和 C 的数据和过程之间做一些权衡：是使用 OOP/GP 来表达逻辑更胜一筹，还是用数据和过程来表达逻辑更为简洁清晰？

针对一些高层应用，抽象至少是和效率一样重要的。

C++ 作为一门通用编程语言，支持多种编程范式，包括过程式、面向对象（Object-oriented Programming, OP）、泛型（Generic Programming, GP），它是图灵完备的。所以，应用 C++ 语言更容易达到程序设计的 KISS 原则（此乃作者一家之言，尊重读者的其他观点）。

KISS 的精髓在于用清晰的代码清晰地表达设计思想，这就意味着要用最适合的工具来做事情，以尽量直接简洁的方式来表达思想，同时又不降低代码的可读性，保持代码容易理解。

借用 Matthew Wilson 的著作 *STL 扩展技术手册（卷 1）：集合和迭代器中的例子* 来说明这一点：

```
// C 语言版
DIR* dir = opendir(".");
if(NULL != dir)
{
    struct dirent* de;
    for(; NULL != (de = readdir(dir)); )
    {
        struct stat st;
        if( 0 == stat(de->d_name, &st) &&
            S_IFREG == (st.st_mode & S_IFMT))
        {
            remove(de->d_name);
        }
    }
    closedir(dir);
}

// C++ 语言版
readdir_sequence entries(".", readdir_sequence::files);
std::for_each(entries.begin(), entries.end(), ::remove);

// C++09 语言版
```

```
std::for_each(readdir_sequence(".",  
    readdir_sequence::files), ::remove);
```

对于不太关心底层细节的高层应用而言，由 C 到 C++，程序变得简单清晰，更加符合“让简单的事情保持简单”的 KISS 理念。然而恰恰也是因为抽象层次的提高，使得开发者要弄清楚“下面实际发生的事情”变得困难起来。比如：

```
std::string str_src("Hello C++");  
std::string str_copy = str_src;
```

虽然知道上面这段代码是创建两个内容相同的字符串副本，调用了 std::string 的拷贝赋值函数，但是却没有一个人能够在不了解更多信息的情况下清楚地描述背后所发生的事情，因为不同的 STL 对于 string 的实现方式不同。

语言没有好不好，只有合适不合适。如果你既想拥有低级语言的高效，又想具备高级语言的抽象，那么请选择 C++。

**哪些程序适宜使用 C++ 呢？**

按应用领域来说，C++ 适用于开发服务器软件、桌面应用、游戏、实时系统、高性能计算、嵌入式系统等。也就是说其应用范围比较广，我们没有上错船。

C++ 之父官方主页的这些应用案例也许能给你更强的信心：

- **Adobe Systems**: 所有主要应用程序都使用 C++ 程序开发而成，比如 Photoshop & ImageReady、Illustrator 和 Acrobat 等。
- **Maya**: 知道“蜘蛛人”、“指环王”的电脑特技是使用什么软件做出来的吗？没错，就是 Maya。
- **Amazon.com**: 使用 C++ 程序开发大型电子商务软件。
- **Apple**: 部分重要“零件”采用 C++ 程序编写而成。
- **AT&T**: 美国最大的电讯技术提供商，主要产品采用 C++ 程序开发。
- **Google**: Web 搜索引擎采用 C++ 程序编写。
- **KDE**: K Desktop Environment (Linux)。
- **Symbian OS**: 最流行的蜂窝电话 OS 之一。

所以，不要被“C++ 太过复杂”这样的言论所吓倒，因为“人们需要用相对复杂的语言去解决绝对复杂的问题”。

请相信，在一些应用中，C++ 确实是你不错的选择。

**请记住：**

为什么选择 C++？你要明确理由。记住，语言没有优劣，只有合适不合适。



专业成就人生  
立体服务大众

www.hzbook.com

填写读者调查表 加入华章书友会

获赠精彩技术书 参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名：

书号：7-111-( )

姓名：	性别： <input type="checkbox"/> 男 <input type="checkbox"/> 女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

朋友推荐 书店 图书目录 杂志、报纸、网络等 其他

2. 您从哪里购买本书：

新华书店 计算机专业书店 网上书店 其他

3. 您对本书的评价是：

技术内容	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
文字质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
版式封面	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
印装质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
图书定价	<input type="checkbox"/> 太高	<input type="checkbox"/> 合适	<input type="checkbox"/> 较低	<input type="checkbox"/> 理由_____

4. 您希望我们的图书在哪些方面进行改进？

\_\_\_\_\_

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

\_\_\_\_\_

6. 您有没有写作或翻译技术图书的想法？

是，我的计划是\_\_\_\_\_ 否

7. 您希望获取图书信息的形式：

邮件 信函 短信 其他\_\_\_\_\_

请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收

邮编：100037 电话：(010) 88379512 传真：(010) 68311602 E-mail:hzjsj@hzbook.com