C++ Gotchas

Avoiding Common Problems in Coding and Design

PEARSON Addison Wesley

C++语言

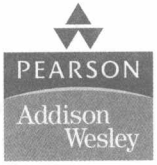
99个常见编程错误

Stephen C. Dewhurst 著

高博 译

PEARSON Education

清华大学出版社



C++ Gotchas

Avoiding Common Problems in Coding and Design

C++语言

99个常见编程错误

Stephen C. Dewhurst 著

高博 译

清华大学出版社
北京

EISBN: 0-321-12518-5

图书在版编目(CIP)数据

C++语言99个常见编程错误 / (美)杜赫斯特(Dewhurst, S. C.)著;高博译. —北京: 清华大学出版社, 2009.8
书名原文: C++ Gotchas: Avoiding Common Problems in Coding and Design
ISBN 978-7-302-19939-7

Ⅰ. C… Ⅱ. ①杜… ②高… Ⅲ. C语言—程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字(2009)第 058588 号

责任编辑: 龙啟铭
责任校对: 徐俊伟
责任印制: 李红英

出版发行: 清华大学出版社 **地 址**: 北京清华大学学研大厦 A 座
http://www.tup.com.cn **邮 编**: 100084
社 总 机: 010-62770175 **邮 购**: 010-62786544
投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn
质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者: 北京市昌平环球印刷厂
装 订 者: 北京市密云县京文制本装订厂
经 销: 全国新华书店
开 本: 185×230 **印 张**: 18.25 **字 数**: 404 千字
版 次: 2009 年 8 月第 1 版 **印 次**: 2009 年 8 月第 1 次印刷
印 数: 1～3000
定 价: 39.00 元

译者序

技术翻译——一种笔记体式的创作尝试

经过近一年的工作，这本几百页的小册子终于和大家见面了。

这本书从一个读者的角度来看，当然主要地可以视为是对于当之无愧的C++大师 Stephen C. Dewhurst 在近 15 年前原创的一本技术书籍的译作。但如果从译者的本意出发，它未尝不可以说是我本人 10 年来学习 C++、领悟 C++和运用 C++的一个小结。2005 年起，我开始陆续在论坛中发表一些零碎的技术文章和翻译作品，并在企业和大学里作了一些演讲。和真正的一线工程师，以及即将踏上工程师岗位的同道们作了一些比较深入的交流之后，我才真真切切地感受到他们对于将书本知识转化为真正实力的那种热切的渴求。现在每年出版的有关C++的书籍车载斗量，但是如何能把这些“知识”尽可能多地转化成工程师手中对付真正的项目需求的“武器”？我感到自己负有责任来做一些工作，来对这个问题做出自己尝试性的解答。那么，最好的方式是创作一本新书吗？经过再三的权衡，我认为并非如此。作为一个未在 *C/C++ Users Journal* 或是 *Dr. Dobb* 上发表过任何文字的人，原创很难企及自己欲达成的号召力。并且，原创的话就意味着要自己照顾一切技术细节，我绝没有自大到认为自己已经有了那种实力的程度。可是，是否仅仅再去翻译一本新的 C++著作呢？那也不是。C++近几年来已不比往昔，新著作的翻译效率简直高得惊人，但单纯的翻译工作其实并不能消除读书人的费解。那么，我就想到：为什么不能挑选一本书，一方面将它翻译过来，另一方面以它作为“蓝本”，将自己的见解以笔记的形式融入其文字，并引导读者参读其他的技术书籍呢？对于某一个特定的技术细节，我希望达到的效果是：读者能够从我的翻译这“小小的一隅”扩展开去，从深度而言他们能够参阅其他专门就此发力的技术资料，获得某种技术或习惯用法的历史背景、推导逻辑、常见形式等翔实、全面、准确的信息；从广度而言，他们可以了解到编码与设计、细节与全局的关系，从而做到取舍中见思路、简化中见智慧，真正地把 C++这种优秀的、有着长久生命力的程序设计语言背后的有关软件工程的科学和艺术的成分“提炼”出来，化为自己实实在在的内功提升。这样的工作，我才认为有它的价值所在，也是我这些年来下苦功夫研读了一二十种 C++的高质量书籍，以及使用 C++交付了一些成功的工程之后有实力完成的——这就是我创作本书的初衷和原动力——以技术翻译为主体，并进行“笔记体”的再创作给读者以诠释和阅读参考的附加值，这就是我的答案。

不过，选取这样的一本作为“蓝本”的书籍殊非易事。首先，它本身需要有相当的深度

和广度，否则难以面面俱到，从而也就难以体现 C++语言在各个层次上的大能。其次，它必须有相当的发散性，否则它就难以和已有的大量资料相结合，难以引导读者去重读他们之前已经看过，但未能充分理解的资料。再次，它还要有明确的主题组织，否则很可能会陷入空谈，使读者感觉难以理解和掌握，从而不能发挥应有的“知识”向“实力”的转化之效。最后，*C++ Gotchas* 落入我的视线，研读数次之后，我觉得它不仅完全符合“蓝本”的一切要求，并且 Stephen C. Dewhurst 大师还在数个方面给予了我太多的启迪：这本书所有的章节都从一个众所周知的、在日常编码或设计实践经常遭遇的问题入手，先是就事论事地指出其不足，再是对其背后思想中存在何种合理与不合理之处深入剖析，最后取其精华弃其糟粕，给出一个简洁、通用、美轮美奂的方案。有的条款中，大师会给出数种不同的解决之道，并一一评点其优劣之处，指出其适用场合；有的条款中，大师步步推进，先是给出一个去除错误的解，再进一步地优化它，直至与某种习惯用法和设计模式接壤作为点题之笔。从翻译的过程中，我自己真的是受益良多，希望我的读者能够收获更大。

在本书的翻译中，清华大学出版社的编辑给予了我很大的帮助和鼓励，并促成这本书最终完稿。微软亚洲研究院的徐宁研究员和 EMC 中国的柴可夫工程师通读了全书，并给予了全面的审阅意见，包括不少技术和文字的问题，在此向他们深深致谢。另外，Hewlett-Packard 总部的 Craig Hilderbrandt 经理、上海交通大学计算机系的张尧弼教授、Phoenix 中国的唐文蔚高级工程师、谷歌中国的龚理工程师、微软亚洲工程院的魏波工程师、微软全球技术中心的陈曦工程师和 SAP 中国的劳佳工程师也都在本书写作的过程中给了我不小的帮助，在此一并致谢。当然，书中的错误和纰漏在所难免，这些理应由我本人负全部责任。另外要感谢的还有我的家人和同事们，没有你们的支持，我不可能坚持到底。希望本书的出版能够给你们带来快乐。

高博
于微软亚洲工程院上海分院

前　言

本书之渊薮乃是近 20 年的小小挫折、大错特错、不眠之夜和在键盘的敲击中不觉而过的无数周末。里面收集了普遍的、严重的或有意思的 C++常见错误，共计九十有九。其中的大多数，（实在惭愧地说）都是我个人曾经犯过的。

术语“gotcha”[①]有其云谲波诡的形成历史和汗牛充栋的不同定义。但在本书中，我们定义它在 C++范畴里的含义为既普遍存在、又能加以防范的 C++编码和设计问题。这些常见错误涵盖了从无关大局的语法困扰，到基础层面上的设计瑕疵，到源自内心的离经叛道等诸方面。

大约 10 年前，我开始在自己教授的 C++课程的相关材料中添加个别常见错误的心得笔记。我的感觉是，指出这些普遍存在的误解和误用，配合以正确的用法指导就像给学生打了预防针，让他们自觉地与这些错误作斗争，更可以帮助新入门的 C++软件工程师避免重蹈他们前辈的覆辙。大体而言，这种方法行之有效。我也深受鼓舞，于是又收集了一些互相关联的常见错误，在会议上作演讲用。未想这些演讲大受欢迎（或是同病相怜之故也未可知？），于是就有人鼓励我写一本“常见错误之书”。

任何有关规避或修复 C++常见错误的讨论都涉及了其他的议题，最多见的是设计模式、习惯用法，以及 C++语言特征的技术细节。

这并非一本讲设计模式的书，但我们经常在规避或修复 C++常见错误时发现设计模式是如此管用的一种方法。习惯上，设计模式的名字我们把每个单词的首字母大写，比如模板方法设计模式（Template Method）或桥接设计模式（Bridge）。当我们提及一种设计模式的时候，若它不是很复杂，则简介其工作机制，而详细的讨论则放在它们和实际代码相结合的时候才进行。除非特别说明，本书不提供设计模式的完全描述或极为详尽的讨论，这些材料可以参考 Erich Gamma et al.，*Design Patterns*。无环访问者（Acyclic Visitor）、单态（Monostate）和空件（Null Object）等设计模式的描述请参见 Robert Martin，*Agile Software Development*。

从常见错误的视角来看，设计模式有两个可贵的特质。首先，它们描述了已经被验证为成功的设计技术，这些技术在特定的软件环境中可以采用自定义的手法搞出很多新的设计花样。其次，或许更重要的是，提及设计模式的应用，对于文档的贡献不仅在于使运用的技术一目了然，同时也使应用设计模式的原因和效果一清二楚。

举例来说，当我们看到在一个设计里应用了桥接设计模式时，我们就知道在一个机制层

① 译注：本书通译为“常见错误”，固然较之原文失之神韵，倒也算得通俗易懂。

里，一个抽象数据类型的实现被分解成了一个接口类和一个实现类。犹有甚者，我们知道这样做是为了强有力地把接口部分同底层实现剥离，因此底层实现的改变将不会影响到接口的用户。我们也知道这种剥离会带来运行时的开销、还知道此抽象数据类型的源代码应该怎么安排，以及很多其他细节。

一个设计模式的名字是关于某种技术极为丰富的信息和经验之高效、无疑义的代号。在设计和撰写文档时仔细而精确地运用设计模式及其术语会使代码洗练，也会阻止常见错误的发生。

C++是一门复杂的软件开发语言，而一种语言愈是复杂，习惯用法在软件开发中之运用就愈是重要。对一种软件开发语言来说，习惯用法就是常用的、由低阶语言特征构成的高阶语言结构的特定用法组合。总的来说，这和设计模式与高阶设计的关系差不多。因此，在 C++语言里我们可以直接讨论复制操作、函数对象、智能指针以及抛出异常等概念而不需要一一指出它们在语言层面上的最低阶实现细节。

有一点要特别强调一下，那就是习惯用法并不仅仅是一堆语言特征的常见组合，它更是一组对此种特征组合之行为的期望。比如，复制操作是什么意思呢？再比如，当异常被抛出的时候，我们能指望发生什么呢？本书中的大多数建议都是在提请注意以及建议应用 C++编码和设计中的习惯用法。很多这里列举的常见错误往往可以直接视作对某种 C++习惯用法的背离，而这些常见错误对应的解决方案则往往可以直接视作对某种 C++习惯用法的皈依（参见常见错误 10）。

本书在 C++语言的犄角旮旯里普遍被误解的部分着了重墨，因为这些语言材料也是常见错误的始作俑者。这些材料中的某些部分可能让人有武林秘笈的感觉，但如果不熟悉它们，就是自找麻烦，在通往 C++语言专家的阳关大道上也会平添障碍。这些语言死角本身研究起来也是其乐无穷，而且产出颇丰。它们被引入 C++语言总有其来头，专业的 C++软件工程师经常有机会在进行高阶的软件开发和设计时用到它们。

另一个把常见错误和设计模式联系起来的东西是，描述相对平凡的实例对于两者来说是差不多同等重要的。平凡的设计模式是重要的。在某些方面，它们也许比在技术方面更艰深的设计模式更为重要，因为平凡的设计模式更有可能被普遍应用。所以从对平凡设计模式的描述中获得的收益就会以杠杆方式造福更大范围的代码和设计。

差不多以完全相同的方式，本书中描述的常见错误涵盖了很宽范围内的技术难题，从如何成为一个负责的专业软件工程师的循循善诱（常见错误 12）到避免误解虚拟继承下的支配原则的苦口良言（常见错误 79）。不过，就与设计模式类似的情况看，专业负责当然比懂得什么支配原则要对日复一日的软件开发工作来得更受用。

本书有两个指导思想。第一个是反复强调有关习惯用法的极端重要性。这对于像 C++这样的复杂语言来说尤为重要。对业已形成的习惯用法的严格遵守使我们能够既高效又准确地和同行交流。第二个是对“其他人迟早会来维护我们写的代码”这件事保持清醒头脑。这种维护可能是直截了当的，所以这就要求我们把代码写得很洗练，以使那些称职的维护工程师一望即知；这种维护也可能是拐了好几道弯的，在那种情况下我们就得保证即使远在天边的某个变化影响了代码的行为，它仍然能够给出正确的结果。

本书中的常见错误以多组小的论说文章的形式呈现，其中每一组都讨论了一个常见错误

或一些相互关联的常见错误，以及有关如何规避或纠正它们的建议。由于常见错误这个主题内涵的无政府倾向，我不敢说哪本书可以特别集中有序地讨论它。然而，在本书中，所有的常见错误都按照其错误本质或应用（误用）所涉及的领域归类到相应的章节。

还有，对一个常见错误的讨论无可避免地会牵涉其他的常见错误。当这种关联有它的意义时——通常确实是有的——我就显式地作出链接标记。其实，这种每个常见错误的为了增强其关联性的描述本身也是有其讨厌之处的。比方说经常遇到一种情况就是还没来得及描述一个常见错误，自己倒先把为什么会犯这个错误的前因后果交代了一大篇。要说清这些个前因后果呢，好家伙，又非得扯上某种技术啦、习惯用法啦、设计模式啦或是语言细节什么的，结果在言归正传之前先要兜上一个更大的圈子。我已经尽力把这种发散式的跑题减到最少了，但要是说完全消除了这种现象，那我就没说实话。要把 C++程序设计做到很高效的境界，那就得在非常多水火不容的方面作出如履薄冰的协调，想在研究大量相似的主题前就对语言作出像样的病理学分析，那只能说是不现实的。

把这本书从第 1 个常见错误到第 99 个常见错误这么挨个地读下去，不仅是毫无必要的，而且也谈不上明智。一气儿服下这么一帖虎狼之剂恐怕会让你一辈子再也学不成 C++了。比较好的阅读方法应该是拣一条你不巧犯过的，或是你看上去有点儿意思的常见错误开始看，再沿着里面的链接看一些相关的。另一种办法就是你干脆由着性子，想看哪儿就看哪儿，这倒也行。

本书的文本里也使用了一些固定格式来阐明内容。首先，错误的和不提倡的代码以灰色背景来提示，而正确和适当的代码却没有任何背景。其次，这里作示意用的代码为了简洁和突出重点，都经过了编辑。这么做的一个结果是，这里示例用的代码若是没有额外的支撑代码往往不能单独通过编译。那些并非平凡无用的示例源代码则可以在作者的网站里找到：www.semantics.org。所有这样的代码都由一个相对路径引出，像这样：

```
//gotcha00/somecode.cpp
```

最后，提个忠告：你不要把常见错误的重要性提升到和习惯用法、设计模式一样[①]。一个你已经学会正确地使用习惯用法和设计模式的标志是，当某个习惯用法或是设计模式正好是你手头的设计或编码对症良方时，它就会“神不知鬼不觉地”在你最需要时从你的脑海里浮现出来。

对常见错误的清醒意识就好比是对危险的条件反射：一回错，二回过。就像对待火柴和枪械一样，你不必非得烧伤或是走火打中了脑袋才学乖。总之，只要加强戒备就行了。把我这本手册当作是你面对 C++常见错误时自我保护的武器吧。

Stephen C. Dewhurst
Carver, Massachusetts

① 译注：作者用心良苦，怕读者“近墨者黑”，好的没记住反而学会了坏的。所以特意提醒所有读者，常见错误有些奇技淫巧，但毕竟不登大雅之堂。

致　谢

编辑们经常在书的“致谢”里落得个坐冷板凳的下场，有时用一句“……其实我也挺感谢我那编辑的，我估计在我拼了命爬格子的时候此人大概肯定也是出过一点什么力的吧”就打发了。Debbie Lafferty，也就是负责本书问世的编辑。有一次，我拿着一本不足为道的介绍性的程序设计教材去找她搞个不足为道的合作提案，结果她反而建议我把其中一个有关常见错误的章节扩展成一本书。我不肯。她坚持。她赢了。值得庆幸的是，Debbie 在胜利面前表现得特别有风度，只是淡淡地说了一句站在编辑立场上的“你瞧，我叫你写的吧。”当然不止于此，在我拼了命爬格子的时候，她是颇出了一些力的。

我也感谢那些无私奉献了他们的时间和专业技能来使本书变得更好的审阅者们。审阅一本未经推敲的稿本是相当费时的，常常也是枯燥乏味的，有时甚至会气不打一处来，而且几乎肯定是讨不着什么好的（参见常见错误 12），这里要特别赞美一下我的审阅者们入木三分而又深中肯綮的修改意见。Steve Clamage、Thomas Gschwind、Brian Kernighan、Patrick McKillen、Jeffrey Oldham、Dan Saks、Matthew Wilson 和 Leor Zolman 对书中的技术问题、行业规矩、清出校样、代码片断和偶然出现的冷嘲热讽都提出了自己的宝贵意见。

Leor 在稿本出来之前很久就开始了对本书的“审阅”，书中的一些常见错误的原始版本只是我在互联网论坛里发的一些帖子，他针对这些帖子回复了不少逆耳忠言。Sarah Hewins，是我最好的朋友同时也是最不留情的批评家，不过这两个头衔都是在审阅我一改再改的稿本时获得的。David R. Dewhurst 在整个写作项目进行的时候经常把我拉回正轨。Greg Comeau 慷慨地让我有幸使用他堪称一流的标准 C++编译器来校验书里的代码（译注：这应该就是著名的 Comeau C/C++ Front/End 编译器）。

就像关于 C++的任何有意义的工作那样，本书也是集体智慧的结晶。这些年来，很多我的学生、客户和同事们为我在 C++常见错误面前表现的呆头呆脑和失足跌跤可没少数落过我，并且他们中的好多人都帮我找到了问题的解决之道。当然，这些特别可贵的贡献者中的大部分都没法在这里一一谢过，不过有些提供了直接贡献的人还是可以列举如下的：

常见错误 11 中的 Select 模板，和常见错误 70 中的 OpNewCreator 策略都取自 Andrei Alexandrescu，*Modern C++ Design*。

我在常见错误 44 中描述了有关返回一个常量形式参数的引用带来的问题[①]，此问题我初见于 Cline et al.，*C++ FAQs*（我客户的代码中在此之后马上就用上了这个解决方案）。此书还描述了我在常见错误 73 中提到的用于规避重载虚函数的技术。

常见错误 83 中的那个 Cptr 模板，其实是 Nicolai Josuttis，*The C++ Standard Library* 中 CountedPtr 模板的一个变形。

Scott Meyers 在他的 *More Effective C++*中，对运算符&&、||、和，的重载之不恰当性提出了比我在常见错误 14 中的描述更加深入的见解。他也在他的 *Effective C++*中，对我在常见错误 58 中讨论的二元运算符以值形式返回的必要性作了更细节的描述，还在 *Effective STL* 中描述了我在常见错误 68 里说的对 auto_ptr 的误用。在后置自增、自减运算符中返回常量值的技术，也在他的 *More Effective C++*中提到了。

Dan Saks 对我在常见错误 8 中描述的前置声明文件技术提出了最有说服力的论据，他也是区别出常见错误 17 中提及的“中士运算符”的第一人，他也说服了我在 enum 类型的自增和自减中不去做区间校验，这一点被我写进在常见错误 87 中。

Herb Sutter 的 *More Exceptional* C++中条款 36 促使我重读了 C++标准 § 8.5，然后修正了我对形式参数初始化的理解（见常见错误 57）。

常见错误 10、27、32、33、38～41、70、72～74、89、90、98 和 99 中的一些材料出自我先是在 *C++ Report*、后来在 *The C/C++ Users Journal* 撰写的 *Common Knowledge* 专栏。

① 译注：是个有关临时对象生存期的问题。

目　录

第1章

基础问题

说一个问题是基础的，并不就是说它不是严重的或不是普遍存在的。事实上，本章所讨论的基础问题的共同特点比起在以后章节讨论的技术复杂度而言，可能更侧重于使人警醒。这里讨论的问题由于它们的基础性，在某种程度上可以说它们存在于几乎所有的C++代码中。

常见错误1：过分积极的注释

很多注释都是画蛇添足。它们只会让源代码更难读、更难维护，并经常把维护工程师引入歧途。考虑下面的简单语句：

```
a = b;                    // 将b赋值给a
```

这个注释难道比代码本身更能说明语句的意义吗？其实它是完全无用的。事实上，它比完全无用还要坏。它是害人精。首先，这条注释转移了代码阅读者的注意力，增加了阅读量，从而使代码更费解。其次，要维护的东西更多了，因为注释也是要随着它描述的代码的更改而更改的。第三，对注释的更改常常被忽视了[①]。

```
c = b;                    // 将b赋值给a
```

仔细的维护工程师不会武断地说注释是错的[②]，所以他就被迫要去检视整个程序以确定注释到底是错了的呢，还是好意的呢（c可能是a的引用），还是微妙的呢（赋值给c可能引发一些传播效应以使a的值也发生相应变化）等等，总之这一行就根本不应该带注释。

```
a = b;
```

该代码本来的样子就最清楚地表明了其意义，也不用维护额外的注释。这在精神上也符合老生常谈，亦即“最有效率的代码就是根本不存在的代码”。这条经验对于注释也适用：最好的注释就是根本用不着写的注释，因为要注释的代码已经“自注释”了。

① 译注：因为如果注释没有随着代码更改，这个语句看起来就成了下面这个样子。
② 译注：说不定是代码错了呢？

另一些常见的非必要的注释的例子经常可以在类型的定义里见到，它们要么是病态的编码标准的怪胎，要么就是出自 C++新手：

```
class C {
    // 公开接口
    public:
        C();                        // 默认构造函数
        ~C();                       // 析构函数
        // ...
};
```

你会觉得别人在挑战你的智商。要是某个维护工程师连“public:”是什么意思都需要教，你还敢让他碰你的代码吗？对于任何有经验的 C++软件工程师而言，这些注释除了给代码添乱、增加需要维护的文本数量以外没有任何用处。

```
class C {
    // 公开接口
    protected:
        C( int );                   // 默认构造函数
    public:
        virtual ~C();               // 析构函数
        // ...
};
```

软件工程师还有一种强烈的心理趋势就是尽量不要“平白无故”地在源文件文本中多写哪怕一行。这里公布一个有趣的行业秘密：如果某种结构（函数、类型的公开接口等）能被塞在一“页”里，也就在三四十行左右[①]的话，它就很容易理解。假如有些内容跑到第二页去了，它理解起来就难了一倍。如果三页才塞得下，据估计理解难度就成原来的四倍了[②]。

一种特别声名狼藉的编码实践就是把更改日志作为注释插入到源文件的头部或尾部：

```
/* 6/17/02 SCD 把一个该死的 bug 干掉了 */
```

这到底是有用的信息，抑或仅仅是维护工程师的自吹自擂？在这行注释被写下以后的一两个星期，它怎么看也不再像是有用的了，但它却也许要在代码里粘上很多年，欺骗着一批又一批的维护工程师。顶好是用你的版本控制软件来做这种无用注释真正想做的事，C++的源代码文件里可没有闲地方来放这些东西。

想不用注释却又要使代码意义明确、容易维护的最好办法就是**遵循简单易行的、定义良好的命名习惯**来为你使用的实体（函数、类型、变量等）取个清晰的、能反映其**抽象含义**的

① 译注：亦即，在普通的屏幕上能在一页内显示得下。
② 译注：源文件文本长度与理解难度成指数关系，所以能少写一行不必要的注释就尽量少写一行。

名字。**（函数）声明中形式参数的名字尤其重要**。考虑一个带有三个同一类型参数的函数：

```
/*
从源到目的执行一个动作
第一个参数是动作编码(action code),第二个参数是源(source),第三个参数是目的
(destination)
*/
void perform( int, int, int );
```

这也不算太坏吧，不过如果参数是七八个而不是三个，你又该写多少东西呢？我们明明可以做得更好：

```
void perform( int actionCode, int source, int destination);
// 译注: 这很明显是 Herb Sutter 倡导的命名规则
```

这就好多了。按理，我们还需要写一行注释来说明这个函数的用途（而不是如何实现的）。形式参数的一个最引人入胜之处就是，不像注释，它们是随着余下的代码一起更改的，即使改了也不影响代码的意义。话虽然这么说，但我不能想象任何一个软件工程师在参数意义改变了的时候，会不给它取个新名字。但我能举出一串软件工程师来，他们改了代码但老是忘记维护注释。

Kathy Stark 在 *Programming in C++*中说得好：“如果在程序里用意义明确、脱口而出的名字，那么注释只是偶尔才需要。如果不用意义明确的名字，即使加上了注释也不能让代码更好懂一些。”

另一种最大程度地减少注释书写的办法是**采用标准库中的、或人尽皆知的组件**：

```
printf( "Hello, World!" );          // 在屏幕上打印"Hello, World"
```

上面这个注释不但是无用的，而且只在部分情况下正确。标准库组件不仅是“自注释”的，并且有关它们的文档汗牛充栋，有口皆碑。

```
swap( a, a+1 );
sort( a, a+max );
copy( a, a+max, ostream_iterator<T>(cout,"\n") );
```

因为 swap、sort 和 copy 都是标准库组件，对它们加上任何注释都是多余的，而且给定义得好好的标准操作规格描述带来了（非必要的）不确定性。

注释之害并非与生俱来。注释常常必不可少。但**注释必须（和代码一起）维护**。维护注释常常比维护他们所注解的代码要难。**注释不应该描述显而易见的事，或把在别的地方已经说清楚的东西再聒噪一遍**。我们的目标不是要消灭注释，而是在代码容易理解和维护的前提下，尽可能少写注释。