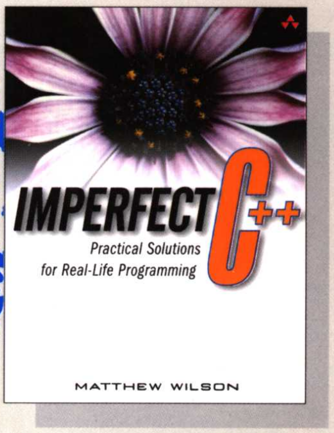
PEARSON
Addison Wesley

典藏原版书苑

Imperfect C++
Practical Solutions for Real-Life Programming

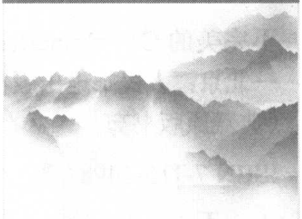
[美] Matthew Wilson 著

Imperfect C++
(英文版)



CD-ROM

人民邮电出版社
POSTS & TELECOM PRESS



典藏原版书苑

Imperfect C++

（英文版）

Imperfect C++: Practical Solutions for Real-Life Programming

[美]　Matthew Wilson　著

人民邮电出版社

图书在版编目（CIP）数据

不完美的 C++=Imperfect C++/（美）维尔逊（Wilson, M.）著.
—北京：人民邮电出版社，2006.8
（典藏原版书苑）

ISBN 7-115-14981-X

Ⅰ. 不…　Ⅱ. 维…　Ⅲ. C 语言—程序设计—英文　Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2006）第 076828 号

典藏原版书苑

Imperfect C++（英文版）

◆ 著　　[美] Matthew Wilson
　责任编辑　刘映欣

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
　邮编　100061　　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京顺义振华印刷厂印刷
　新华书店总店北京发行所经销

◆ 开本：800×1000　1/16
　印张：39
　字数：813 千字　　　2006 年 8 月第 1 版
　印数：1 – 2 500 册　　2006 年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2006-3201 号

ISBN 7-115-14981-X/TP・5541

定价：65.00 元（附光盘）

读者服务热线：(010)67132705　印装质量热线：(010)67129223

内容提要

即便是 C++阵营里最忠实的信徒，也不得不承认：C++语言并不完美。实际上，世界上也没有完美的编程语言。

如何克服 C++类型系统的不足？在 C++中，如何利用约束、契约和断言来实施软件设计？如何处理被 C++标准所忽略的动态库、静态对象以及线程等有关的问题？隐式转换有何替代方案？本书将为你一一解答这些问题。针对 C++的每一个不完美之处，本书都具体地分析原因，并探讨实用的解决方案。书中也不乏许多作者创新的、您从未听说或使用过的技术，但这些确实能帮助您成为 C++方面的专家。

本书适合有一定经验的 C++程序员和项目经理阅读，也适合对 C++编程的一些专门话题或高级话题感兴趣的读者参考。

To Chris, for all the lessons
To Dad, for the inspiration and the resolve
To Grandma, for the pride
To James, for the brotherhood
To John, for the conviction
To Mum, for the wisdom and the facilities
To Sarah, for the support and the love (and the boys)
To Suzanne, for setting the targets and lighting the way

And to my boys: may you take only the blessings

前　言

或许我不像喜欢我的孩子们那样喜爱 C++，甚至我对 C++的喜爱都比不上对骑自行车在坡度为 32° 、光滑度为 10%的柏油路上爬坡的热衷，[1]尽管有时这些喜爱之情的确十分接近。我庆幸我有这样的人生，让我得以将生命中的部分时间用来实践或阐释 Frederick P.Brooks 的名言：“尽量发挥想象力进行创造”。我更要感激的是我能够与这种如此强大、危险却又诱人的语言相伴。

这些话听起来似乎很华丽动听，但你仅仅可能是因为看到本书的书名才买它的，以为本书是一本抨击 C++的图书。你可能是 Java 或 C 或其他主流语言的热衷者，因而买这本书可能是想从中找到支持你远离 C++的理由。倘若果真如此，你会失望的，因为本书并不是在举办 C++的批判大会。不过先别急着离开，因为你或许能够从中找到令你愿意接触 C++的理由。

你将从中学到什么

我写这本书的目的在于给予开发者同行们一些能量。它以虽批评但却具建设性的眼光来看待 C++及其不完美之处，并给出实际的措施以避免或改善这种不完美。我希望你在读完本书后能够对下面这些问题有更深刻的认识：

- 如何克服 C++类型系统中的某些不足。
- 模板编程在提高代码灵活性和健壮性上的强大能力。
- 如何在（当前标准尚不予置理的）未定义行为的险恶丛林中生存下来，包括动态库、静态对象以及线程等。
- 隐式转换的代价，它所带来的问题，以及其替代方案，即基于显式转换的、高效且易于控制的泛用性编程。
- 如何编写能够与（或更容易令它们与）其他编译器、库、线程模型等兼容的软件。
- 编译器“在幕后”都做了些什么？你对编译器可以施加什么样的影响。

[1] 自行车爱好者会明白我为什么这样说。

- 数组和指针之间微妙而棘手的互操作性，以及用于防止它们的行为变得彼此相似的技术。
- C++对 RAII（资源获取即初始化，Resource Acquisition Is Initialization）机制的支持，以及该机制可以被运用到的各种问题领域。
- 如何通过最大限度地榨取编译器的侦错能力来节省你的精力和时间。

有了上面这些“装备”，你编写的代码将更有效率、更具可维护性、更健壮、更具灵活性。

我的意图是，即便是经验丰富的 C++实践者也能从本书中发现新思想和新技术，从而引发他们的灵感并提高其现有的实战能力。经验较少的程序员则能领会其中包含的有关原则并将相关技术应用到自己的工作中，在增长见识的同时弥补自己对一些技术细节理解上的疏漏。

我并不指望你们所有人都完全同意我所说的一切，但我期望哪怕是最有争议的内容也能激发你去真正弄明白自己对这门强大语言的使用问题。

假设你们已经知道

除非有人想写一本很厚的书，否则他必须假设许多知识是读者已经知道的。当然，规定大家必须首先读完某些书籍的做法过于粗鲁，但我假设你们的知识和经验能使你们轻松地理解 Scott Meyer 的 *Effective C++*系列和 Herb Sutter 的 *Exceptional C++*系列中讲述的绝大部分概念。我还假设你们拥有一本 C++“圣经”，即 Bjarne Stroustrup 的 *The C++ Programming Language*。我并不指望你们仔细阅读该书的每一页（我都还没有做到这一点），但你们应当将这本书作为这门语言的终极参考，因为它的确是字字珠玑。

本书中包含有相当多的模板代码（哪一本现代 C++书籍不是如此呢），但我并没有假设你是大师级的人物，[1] 或掌握了高阶元编程知识。话虽如此，我还是建议你们最好熟悉如何使用模板，比如组成 C++标准库中最流行的部分的那些内容。我努力将对模板的使用控制在一个合理的水平，但我们必须承认，正是对模板的支持使得 C++具有“自我修复”的能力，而这也是本书得以诞生的主要原因。

由于灵活性和实用性对我而言非常重要，因此书中的代码并非只能在少数最新编译器上编译；实际上，书中几乎所有代码都可以在任意一款“还算可以”的现代编译器（见附录 A）上编译。当然，有一些很好的编译器是可以免费获得的，并且你也可以相信你的编译器能够编译这些代码。

只要有可能，我就会尽量避免涉及特定的操作环境、库和技术。然而我也略微谈到了一些，所以，了解以下内容将会有所帮助（尽管并非必不可少）：COM 和 CORBA、动态库（UNIX 和 Win32 的）、STL、线程（POSIX 和 Win32 的）、UNIX 以及 Win32。参考书目中有许多包含相关内容的好书。此外，熟悉不止一种机器架构也是有帮助的，当然这同样并非不可或缺。

由于 C 目前仍是语言间交互以及操作系统开发的通用语言，因此它继续作为一门极其重要的语言存在着。尽管本书是关于 C++的，但在某些领域，C 和 C++之间的共性会变得很重要，我认为在这些领域选择兼顾 C/C++是合理的。事实上，正如本书第二部分中提到的，我们有时需要求助于 C 来支持 C++的

[1] 如果你愿意付出努力，“参考书目”中列出的若干书籍可在你成为大师级人物的漫漫征途中助你一臂之力。

一些高级用法。

此外，我还作了一个重要的假定。我假定你们也认为工作的质量是非常重要的，并且有动力去寻找达到这一目标的新途径。本书不敢妄称是这些所谓新途径的惟一源泉。确切地说，它提供了一种实践性的、有时甚至是异端的视角来看待我们在 C++中遇到的问题。本书或许能成为你的精华书库中的成员之一。最终的责任落在你自己的肩上，剩下来要做的就是去寻找最好的工具来支持你的工作。

本书的组织方式

本书的主要内容分为 6 个部分，每一部分由一个绪论和 5～7 章组成，每一章又可以被进一步划分为若干节。

既然本书取名为“Imperfect C++”，那么我就会在书中尽量凸显出实际的不完美之处，这就是你们在本书的通篇都会发现一些所谓的“不完美（Imperfection）”的原因。在书的前几部分，这些不完美之处出现得比较密集，这说明它们自身和它们的解决之道尚且是比较简单的。每个小节都对应着语言的某个特定的特性，并且通常会介绍它的一个不完美之处。只要有可能，我就会提供一些具体的技巧或技术来解决这些问题，或至少为开发者提供控制问题的方法。随着本书内容的逐步展开，这些不完美之处将会变得不再像前面那样琐碎，而是显得更为重大，从而伴随有更大篇幅、更加详细的讨论。

本书并没有采用时髦的“自助餐”式的写法，也不存在一条必须从头读到尾的贯穿全书的主线。当然，大部分后续章节的内容是根据前面章节的内容进行描述的，有时甚至建立在之前的章节之上，所以除非你故意作对，否则最好还是按顺序阅读。然而，一旦你读完一遍后，回头再来参考其中的某些部分时，你就可以根据需要跳至任意一处，而不再需要通读所有内容了。在每一章中，各节一般是按内容顺序排列的，所以建议按顺序阅读每一节。

在难度方面，第 1 部分到第 4 部分显然经历了一个从易到难的过程，从相当简单明了直到有相当高的要求。[1] 尽管第 5 部分和第 6 部分需要依赖第 3 部分和第 4 部分的一些内容，但它们相对来说不再那么具有挑战性了。你可以优哉游哉地一直读到附录。

主要内容介绍完之后是 4 个简短的附录。附录 A 介绍在探讨本书各项议题时使用的编译器和库的详细信息。附录 B 向你展示一个年轻的 C++工程师在刚踏入这一充满陷阱的领域时所犯下的一些令人目瞪口呆的错误。附录 C 介绍 Arturius 项目，这是一个免费的源码开放的编译器多路分发器，你也可以在随书光盘中找到它。最后，附录 D 介绍随书光盘的内容。

我的编码风格十分一致，甚至可以说是严格，你也可以说它过于“学究气”，我以前的同事和使用我库的人们早就这样说过。但我之所以采取这种风格是因为这样做代码中的所有东西都有其明确的位置，大家不至于会问“某某东西到哪里去了”这种问题，同时这也意味着当我几年后重新面对这些代码时，我还可以轻松搞定它。当然这么做也有缺点：我需要一个 21 英寸的显示器和一个工业用激光打印机。

为了尽量减少我的编码风格对阅读本书的影响，我在书中出现的代码示例中使用了一些自由风格。

[1] 第 3 部分和第 3 部分的某些内容直到现在还让我感到头疼呢！

你将会在示例中看到很多省略号（...），它们一般表示前面的有关例子中已经包含了该段代码，或者表示这些省却的代码是我们司空见惯的样板式代码（例如，禁止客户代码访问某些方法，见 2.2 节）。只有编码风格中对可靠性有着显著影响的方面才会被加以讨论（见第 17 章）。[1]

参考资料

我在阅读其他 C++书籍时感到不满的一件事是：作者指出事实的同时从不提及标准的相关部分。所以我在写作本书的过程中，除了会不断引用一些相关的书籍和文章外，还会在描述 C++语言行为的同时提供其在 C++（C++98）或 C（C99）标准中的相关参考。

补充材料

光盘

随书光盘中包含一些库、编译器（包括书中描述的大量编码技术）、测试程序、工具以及其他有用的软件，另外还有许多摘自各种出版物的相关文章。关于光盘的详细内容，请参考附录 D。

网上资源

你也可以通过如下网址来获取辅助资源：http://imperfectcplusplus.com。[2]

致谢

在几乎所有的书籍当中，都充满着对家人和朋友热情洋溢的致谢之辞，我可以向你保证这些言辞都是发自内心的真情流露。一本书要想得以完成，其背后必定隐藏着若干人的支持。

首先感谢我的母亲和 Suzanne，从当年的唧唧喳喳的小布谷鸟成长到成熟的年龄（27 岁），离开温暖的巢，飞向一个不同的世界，她们的容忍、支持以及养育之恩一直陪伴并支持着我。感谢母亲和 Robert 在一段艰苦但最终有所收获的岁月里一直帮助小布谷鸟和他的家庭。尤其感谢 Robert 在这段时间里的许多重要时刻帮我保持精神放松的状态。

谢谢 Piker 填补了这个大家庭的空缺，并且不遗余力地照看孩子，鼓励我们，并提供免费的午餐。同样感谢 Dazzle，他总是告诉我他对我极有信心，并难能可贵地放弃那些诱人的 DBA 大师活动，坚定地帮我做那些沉闷的审稿工作；他在看其他 Perl 和 Python 脚本的时候可决不会是这个状态！也要感谢 Besso 一直以来对我的计划的浓厚兴趣、以及鼓舞人心的观点。谢谢 Al 和 Cynth（亲家）提供许多免费的饭菜

[1] 如果你非要见识一下我的编码风格，你可以到随书光盘中的库里找到足够多的样例。

[2] 这个站点上也有一个勘误表页面，当然，这不代表这里一定会有错误。

和美味的巧克力（哦，我差点忘了我那辆自行车……）。

衷心感谢 808 State、Aim、Barry While、Billy Bragg、De La Soul、Fatboy Slim、George Michael、Level 42、Rush、Seal、Stevie Wonder 以及 The Brand New Heavies，没有他们我不可能从这个一年半的幻想中挣扎过来。

最重要的感谢要给予我美丽的爱妻 Sarah，感谢她抑制住合乎情理的担心和疑虑，而表现出完全的支持和信任。她是我心目中真正的明星！

感谢一些出色的人，他们对我的教育和事业的影响是深远的。感谢 Bob Cryan 教授，感谢他能够赏识一名天赋不错的学生，并在后来 3 年的研究生在读期间宽容他的逃课（去骑自行车）。

还要感谢 Richard McCormack 让我保持在概念上的优雅之外还看到了代码的高效之美。这段时间我有时会被别人责备说过于看重效率了，我回答说你要怪就怪 Richard 去吧。另外，感谢 Graham Jones（绰号“Dukey”）教我设置 vi，在那疯狂的 6 个月的时间里和我保持深厚的友谊以及提供开心的玩笑。这些东西，无可替代！

同样还要感谢 Leigh 和 Scott Perry，感谢他们向我介绍“螺栓”概念以及其他优秀的技术。

特别感谢 Andy Thurling。在我怀揣博士文凭和名不副实的软件工程技能等级证书去找工作时，Andy 对我的潜能表现出充分的信心。[1] Andy 还教给我或许在这个奇妙而令人畏惧的职业中最最重要的一课：“我们只是在尽量充分利用手头拥有的信息而已（skegging it out）”。[2] Chuck Allison 则以更为亲切的形式来表述这个意思，那是一句来自古老的印第安人部落中的箴言：“向一个不断学习的人学知识就好比是在一条生生不息的河流中畅饮”。

任何书籍的成功都离不开出版社、审稿人以及其他给出指导和建议的人们的帮助。感谢我的编辑 Peter Gordon 给予鼓励并包容一个热情而任性的作者在写他第一本书的过程中的情绪起落。同样感谢 Peter 的得力助手 Bernard Gaffney，他很好地控制了整个计划进程，并耐心地答复我每天发去的几封电子邮件。还要感谢 Addison Wesley 出版社的产品和市场部门的其他职员，包括 Amy Fleischer、Chanda Leary-Coutu、Heather Mullane、Jacquelyn Doucette、Jennifer Andrews、Kim Boedigheimer 以及 Kristy Hart。衷心感谢（并致歉）我的项目经理 Jessica Balch 不厌其烦地帮助我纠正书中糟糕的句法、蹩脚的幽默以及英式英语的拼写（例如有很多“ize”被写成了“ise”）。还要特别感谢 Debbie Lafferty，2002 年的某一个晚上我在睡梦中冒出了“Imperfect C++”这个念头，是 Debbie Lafferty 鼓励我付诸实现的。

感谢忠诚的评审者们，他们是 Chuck Allison、Dave Brooks、Darren Lynch、Duane Yates、Eugene Gershnik、Gary Pennington、George Frasier、Greg Peet、John Torjo、Scott Patterson 以及 Walter Bright。没有他们我肯定会到处磕磕碰碰，踉踉跄跄。他们中的一些人让我保持心情愉快，有些人则使我质疑这个写作计划是否明智，但无论如何他们的反馈信息都极大地有助于我改善最终的成果。我们生活在一个奇妙的世界中，来自各个不同国家的人们可以建立起友谊，虽然其中大部分人我未曾谋面，却能够给予我如此之多的帮助，这真是奇妙！

[1] 当时我连实模式与保护模式的区别都弄不清！

[2] “skegging it out”是北约克郡的一种用语，意思是“充分利用你当时拥有的信息”。

同样要感谢 Addison Wesley 的审阅者们，包括 Dan Saks、JC van Winkel、Jay Roy、Ron McCarty、Justin Shaw、Nevin Liber 以及 Steve Clamage。他们的反馈信息对于本书的篇幅能够降到 1000 页以下起到了至关重要的作用，而且帮助我避免了一些无聊的闲话和粗心大意的错误。作为一名作者，我早已领略了写作和审稿过程的艰辛，我知道一次彻底的审阅需要付出多少精力，真的非常感谢他们投入的时间和精力。

感谢 Peter Dimov 允许我引用他的名言（在第 26 章），同时还对第 5 部分各章给出了极好的建议。感谢 Kevlin Henney 对第 19 章和智能强制的一些有趣的讨论所给予的关注。感谢 Joe Goodman 仔细审阅有关 C++ ABI 的那一部分（第 7 章和第 8 章），使我最终能够呈上一份像样的讨论。感谢 Thorsten Ottosen 在第 1 章有关契约式设计部分提供的类似的帮助。

特别感谢 Chuck Allison、Herb Sutter、Joe Casad、John Dorsey 以及 Jon Erickson，他们在过去的几年中在各方面都给予我极大的支持和鼓励。

感谢 Bjarne Stroustrup 的鼓励，他还不时为我补上一点历史常识。哦，首先还是要感谢他发明了这门神奇的语言！

感谢 Walter Bright 在本书的写作过程中持续不断地改进他的优秀的 Digital Mars C/C++编译器，在世界上最缺乏耐心的家伙持续的质问面前保持责任心，而且还友好地将我引领入 D 语言开发世界，后者也给我这本书的写作带来了不少灵感。同样感谢 Greg Comeau，尽管他用不着对他的 Comeau 编译器做那么多的改进，Comeau 是目前业界最遵从标准的 C++编译器！他们除了是优秀而且反应迅速的编译器供应者之外，还在众多问题上持续不断地给予鼓励、建议以及其他信息。

感谢 CMP 出版公司允许我在书中使用我原先在他们刊物上发表的文章中的一些内容，并允许我将原先的几篇文章放在随书光盘中（见附录 D）。

感谢 Borland、CodePlay、Digital Mars、Intel、Metrowerks 以及 Microsoft 公司为我的研究、写作以及编写开源库提供编译器。

特别要感谢 Digital Mars、Intel 和 Watcom 公司授权我将他们的编译器放在随书光盘中（见附录 D）。另外，Greg Peet 在光盘的设计及内容方面的巨大帮助值得大大嘉赏。

谢谢 C/C++ User's Journal、Dr. Dobb's Journal、BYTE 以及 Windows Developer Network 的读者，他们为我的文章和专栏友好地提供了反馈信息和支持。

同样感谢整个 C++社群，以及那些关注各种与 C++有关的新闻组（见附录 A）的好心人。他们是 Attila Feher、Carl Young、Daniel Spangenberg、Eelis van der Weegen、Gabriel Dos Reis、Igor Tandetnik、John Potter、Massimiliano Alberti、Michal Necasek、Richard Smith、Ron Crane、Steven Keuchel、Thomas Richter 以及“tom_usenet”，或许还有很多人没有列出。尤其感谢 Ilya Minkov，是他要求我为 STLSoft 库增加属性（Properties）实现的，而我自己以前没有产生过这个念头。如果不是因为这个建议，我可能永远也不会实现这项我钟爱的技术（见第 35 章）。

最后，还要感谢 STLSoft 库的所有用户，没有他们的信息反馈，这个库中的许多特性就不会存在，而且本书中的某些部分也会变得复杂得多。

Matthew Wilson

Prologue: Philosophy of the Imperfect Practitioner

This book is about good practice as much as it is about C++ language techniques. It is not just about what is effective or technically correct in a specific situation, but what is safer or more practical in the long run. The message of the book is fourfold:

Tenet #1—C++ is great, but not perfect.

Tenet #2—Wear a hairshirt.

Tenet #3—Make the compiler your batman.

Tenet #4—Never give up: there's always a solution.

Together, these make up what I like to call the *Philosophy of the Imperfect Practitioner.*

C++ Is Not Perfect

I was taught very early, by a mother embarrassed by the overweening confidence of her youngest offspring, that if you're going to trumpet the good things to people, you'd also better be prepared to acknowledge the bad. Thanks, mum!

C++ is superb. It supports high-level concepts, including interface-based design, generics, polymorphism, self-describing software components, and meta-programming. It also does more than most languages in supporting fine-grained control of computers, by providing low-level features, including bitwise operations, pointers, and unions. By virtue of this huge spread of capabilities, coupled with its retaining a fundamental support of high efficiency, it can be justly described as the preeminent general purpose language of our time.[1] Nevertheless it is not perfect—far from it—hence the title of this book.

For very good reasons—some historical, some valid today—C++ is both a compromise [Stro1994] and a heterogeneous collection of unrelated, and sometimes incompatible, concepts. Hence it has a number of flaws. Some of these are minor; some of them are not so minor. Many come about as a result of its lineage. Others stem from the fact that it focuses—thankfully—on efficiency as a high priority. A few are likely fundamental restrictions to which any language would be subject. The most interesting set of problems comes about as a function of how complex and diverse a language it is becoming, things that no one could have anticipated.

This book meets this complex picture head on, with the attitude that the complexity can be tamed, and control wrested back to where it belongs, in the hands of the informed and experienced computing professional. The goal is to reduce the consternation and indecision that is experienced daily by software developers when using C++.

[1]Note that I'm not asserting that C++ is the best language in all specific problem domains. I wouldn't advise you to choose it over Prolog for writing Expert Systems, or over Python or Ruby for system scripts, or over Java for enterprise e-Commerce systems.

Imperfect C++ addresses problems that software developers encounter not as a result of inexperience or ignorance, but rather problems encountered by all members of the profession, from beginners through to even the most talented and experienced. These problems result partly from imperfections inherent in the language itself, and partly from common misapplications of the concepts that the language supports. They cause trouble for us all.

Imperfect C++ is not just a treatise on what is wrong with the language, along with a list of "do-nots"; there are plenty of excellent books available that take that approach. This book is about providing solutions to (most of) the flaws identified, and in so doing making the language even less "imperfect" than it is. It focuses on empowering developers: giving them important information regarding potential problem areas in the tools of their trade, and providing them with advice coupled with practical techniques and software technologies to enable them to avoid or manage these problems.

Hairshirt Programming

Many of the textbooks we read, even very good ones, tell you about the ways in which C++ can help you out if you use its features to their full extent, but all too often go on to say "this doesn't really matter" or "that would be taking [rigor] a bit far." More than one former colleague has engaged me in lively debate in a similar vein; the usual arguments amount to "I'm an experienced programmer, and I don't make the mistakes that XYZ would prevent me from doing, so why bother?"

Phooey!

This is flawed in so many ways. I'm an experienced programmer, and I do at least one stupid thing every day; if I didn't have disciplined habits, it would be ten. The attitude assumes that the code's never going to be seen by an inexperienced programmer. Further, it implies that the code's author(s) will never learn, nor change opinions, idioms, and methodologies. Finally, what's an "experienced programmer" anyway?[2]

These people don't like references, constant members, access control, `explicit`, concrete classes, encapsulation, invariants, and they don't code with portability or maintenance in mind. But they just love overloading, overriding, implicit conversions, C-style casts, using `int` for everything, globals, mixing typedefs, `dynamic_cast`, RTTI, proprietary compiler extensions, `friend`s, being inconsistent in coding styles, making things seem harder than they are.

Bear with me while I digress into some historical allegory. After being made Archbishop of Canterbury in 1162 by Henry II, Thomas À Beckett underwent a transformation in character, reforming from his materialistic ways and developing both a genuine concern for the poor and a profound sense of remorse for his former excesses. As his body was being prepared for burial, Beckett was found to be wearing a coarse, flea-infested hairshirt. It was subsequently learned that he had been scourged daily by his monks. Ouch!

Now, personally I think that's taking repentance and soul purging a tad far. Nevertheless, having in my earlier days made cavalier use of the power of C++ to create all manner of fell perversions (see Appendix B), these days I try to take a more temperate approach, and thus wear a bit of a hairshirt when programming.[3]

[2]It's hard to tell these days, when every vitae you see contains self-assessment matrices that are marked 10/10 for each metric.

[3]If the hairshirt analogy is too extreme for your tastes, you might like to think about yoga instead: tough, but worth the effort.

Of course, I don't mean I kneel on a gravel floor, or that I've punched nails through the back of my Herman-Miller, or have stopped blasting cooking dance music while I code. No, I mean I make my software treat me as harshly as I can whenever I try to abuse it. I like `const`—a lot of it—and use it whenever I can. I make things `private`. I prefer references. I enforce invariants. I return resources from where I got them, even if I know the shortcut is safe; *"Well, it worked fine on the previous version of the operating system. It's not my fault you upgraded!"* I've enhanced C++'s type checking for conceptual typedefs. I use nine compilers, through a tool (see Appendix C) that makes it straightforward to do so. I use a more potent `NULL`.

This is not done for a nomination for the "programmer of the year" award. It's simply a consequence of my being lazy, as all good engineers are wont to be. Being lazy means you don't want to find bugs at run time. Being lazy means you don't want to ever make the same mistake twice. Being lazy means making your compiler(s) work as hard as possible, so that you don't have to.

Making the Compiler Your Batman

A batman (as opposed to Batman) is a term derived from the days of the British Empire, and means an orderly or personal servant. If you treat it well, you can make the compiler your right-hand man, helper, conscience, your batman. (Or your superhero, if you prefer.)

The coarser your programming hairshirt, the better able your compiler is to serve you. However, there are times when the compiler, acting out of duty to the language, will stymie your intent, stubbornly refusing to do something you know to be sensible (or desirable, at least).

Imperfect C++ is also about allowing you to have the final choice by providing you with techniques and technologies to wrest control back from the compiler: getting what you want, not what you're given. This is not done blithely, or as a petulant thumbing of the nose from man to machine, but in recognition of the fact that it is the software developers who are the main players in the development process; languages, compilers, and libraries are merely tools that allow them to do their job.

Never Say Die

Despite most of my education being in the sciences, I'm actually much more of an engineer. I loved those early sci-fi books where the heroic engineers would "jury-rig" their way out of sticky situations. That's the approach taken in this book. There's theory, and we go with that first. But as often as not, when working on the borders of the language, most of the current compilers have problems with theory, so we have to code to their reality. As Yogi Berra said, "In theory, there's no difference between theory and practice. In practice, there is."

Such an approach can bring powerful results. Engineering effort, rather than academic induction, coupled with a stubborn refusal to live with the imperfections of C++, has led me (eventually) to a number of discoveries:

- The principles of explicit generalization via Shims (Chapter 20) and the resultant phenomenon of Type Tunneling (Chapter 34)
- An expansion of C++'s type system (Chapter 18), providing discrimination between, and overloading with, conceptually separate types sharing common implementation base types

- A compiler-independent mechanism providing binary compatibility between dynamically loaded C++ objects (Chapter 8)
- An incorporation of the principles of C's powerful `NULL` within the rules of C++ (Chapter 15)
- A maximally "safe" and portable `operator bool()` (Chapter 24)
- A fine-grained breakdown of the concept of encapsulation, leading to an expanded tool kit for the efficient representation and manipulation of basic data structures (Chapters 2 and 3)
- A flexible tool for efficiently allocating dynamically sized memory blocks (Chapter 32)
- A mechanism for fast, nonintrusive string concatenation (Chapter 25)
- An appreciation for how to write code to work with different error-handling models (Chapter 19)
- A straightforward mechanism for controlling the ordering of singleton objects (Chapter 11)
- A time-and-space efficient implementation of properties for C++ (Chapter 35)

Imperfect C++ does not attempt to be the complete answer to using the C++ language. Rather it takes the developer down the path of pushing beyond the constraints that exist to finding solutions to imperfections, encouraging a new way of thinking outside of the square.

I'm not perfect; none of us are. I do bad things, and I have a heretical streak. I have a poor habit of writing `protected` when I should write `private`. I prefer to use `printf()` when perhaps I should be favoring the IOStreams. I like arrays and pointers, and I'm a big fan of C-compatible APIs. Nor do I adhere religiously to the hairshirt programming philosophy. But I believe that having such a philosophy, and sticking to it wherever possible, is the surest and quickest way to achieve your aims.

The Spirit of *Imperfect C++*

As well as the tenets of *Philosophy of the Imperfect Practitioner,* this book reflects in general my own guiding principles in writing C++. These generally, though not completely, reflect the twin credos of the "Spirit of C" [Como-SOC]:

- *Trust the programmer: Imperfect C++* does not shy away from ugly decisions.
- *Don't prevent the programmer from what needs to be done: Imperfect C++* will actually help you achieve what *you* want to do.
- *Keep solutions small and simple:* most of the code shown is just that, and is highly independent and portable.
- *Make it fast, even if it is not guaranteed to be portable:* efficiency is given a high importance, although we do, on occasion, sacrifice portability to achieve it.

and the "Spirit of C++" [Como-SOP]:

- *C++ is a dialect of C with modern software enhancements:* We rely on interoperability with C in several important cases.
- *Although a larger language than C, you don't pay for what you don't use (so size and space penalties are kept to a minimum, and those that do exist must be put in perspective since what needs to be compared is equivalent programs, not feature X vs feature Y).* 1

- *Catch as many errors at compile time as possible: Imperfect C++* uses static assertions and constraints wherever appropriate.
- *Avoid the preprocessor when possible (inline, const, templates, etc. are the way to go in most cases):* we look at a variety of techniques for using the language, rather than the preprocessor, to achieve our aims.

Beyond these tenets, *Imperfect C++* will demonstrate an approach that, wherever possible:

- Writes code that is independent of compilers (extensions and idiosyncrasies), operating-systems, error-handling models, threading-models, and character-encodings
- Uses Design-by-Contract (see section 1.3) wherever compile-time error-detection is not possible

Coding Style

In order to keep the book to a manageable length, I've had to skip much of my normal strict—some might say pedantic—coding style in the examples given. Chapter 17 describes the general principles I tend to use in laying out class definitions. Other coding practices, such as bracing and spacing styles, are of less significance; if you're interested, you will be easily able to pick them up from much of the material included on the CD.

Terminology

Since computers speak the exact language of machine code, and human beings speak the various inexact languages of mankind, I'd like to define a few terms here that will be used throughout the remainder of the book:

Client code. This is code that uses other code, usually, but not limited to, the situation of application code using library code.

Compilation unit. The combination of all source from a source file and its entire dependent include files.

Compile environment. The combination of compiler, libraries, and operating system against which a given code set is compiled. Thanks to Kernighan and Pike [Kern1999] for this one.

Functor. This is a widely used term for Function Object or Functional, but it's not in the standard. I actually prefer Function Object, but I've been persuaded[4] that *Functor* is better because it's one short word, is more distinctive, and, most significantly, is more searchable, especially when searching online.

Generality. I've never properly understood the word *genericity,* at least in so far as a programming context, albeit that I sometimes bandy about the term with alacrity. I guess it means being able to write template code that will work with a variety of types, where those types are related by how they are used rather than how they are defined, and I restrict its use to that context. Generality [Kern1999] seems both to mean it better, and to apply the concept in a much broader sense: I'm just as interested in my code working with other headers and other libraries than merely with other (template parameterizing) types.

[4]Blame several of my reviewers for this.

In addition to these conceptual terms, I'm also including some specific language-related terms. I don't know about you, but I find the nomenclatural clutter in C++ more than a little confusing, so I'm going to take a small time out to drop in some definitions. The following are derived from the standard, but presented in a simpler manner for all our understanding, not least my own. Several of these definitions overlap, since they address different concepts, but all form part of the vocabulary of the accomplished C++ practitioner, imperfect or not.

Fundamental Types and Compound Types

The *fundamental types* (C++-98: 3.9.1) are the integral types (`char`, `short`, `int`, `long` (`long long` / `__int64`), the (`signed` and) `unsigned` versions thereof,[5] and `bool`), the floating-point types (`float`, `double`, and `long double`) and the `void` type.

Compound types (C++-98: 3.9.2) are pretty much everything else: arrays, functions, pointers (of any kind, including pointers to nonstatic members), references, classes, unions, and enumerations.

I tend not to use the term *compound types,* since I think the term implies things that are made up of other things, which can't really be said for references and pointers.

Object Types

Object types (C++-98: 3.9; 9) include any type that is "not a function type, not a reference type, and not a void type." This is another term I avoid, since it does not mean "instances of class types" which one might think. I use instances to refer to such things throughout the book.

Scalar Types and Class Types

Scalar types include the (C++-98: 3.9; 10) "arithmetic types, enumeration types [and] pointer types." *Class types* (C++-98: 9) are those things declared with one of the three class-keys: `class`, `struct` or `union`.

A structure is a class type defined with the class-key `struct`; its members and base classes are public by default. A union is a class type defined with the class-key `union`; its members are public by default. A class is a class type defined with the class-key `class`; its members and base classes are private by default.

Aggregates

The standard (C++-98: 8.5.1; 1) describes an *aggregate* as "an array or a class with no user-defined constructors, no private or protected non-static data members, no base classes, and no virtual functions." As an array or a class type, it means that there is a bringing together of several things into one, hence aggregate.

[5]Note that char comes in three flavors: char, unsigned char, and signed char. We see how this can be problematic in Chapter 13.