PEARSON

C和C++实务精选
品味岁月积淀，读享技术菁华

C++编程规范

101条规则、准则与最佳实践

[加] | Herb Sutter
[罗] | Andrei Alexandrescu | 著
刘基诚 | 译

C++ Coding Standards:
101 Rules, Guidelines, and Best Practices

- C++领域20年集大成之作
- 两位世界顶级专家联袂巨献
- 适合所有层次C++程序员

中国电力出版社
POWER PRESS

PEARSON

[加] | Herb Sutter
[罗] | Andrei Alexandrescu | 著
刘基诚 | 译

C++编程规范

101条规则、准则与最佳实践

C++ Coding Standards:
101 Rules, Guidelines, and Best Practices

人民邮电出版社
北京

图书在版编目（CIP）数据

C++编程规范 : 101条规则、准则与最佳实践 / (加)萨特 (Sutter,H.), (罗) 安德烈亚历克斯安德莱斯库 (Alexandrescu,A.) 著 ; 刘基诚译. -- 北京 : 人民邮电出版社, 2016.3
ISBN 978-7-115-35135-7

Ⅰ. ①C… Ⅱ. ①萨… ②安… ③刘… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第007172号

◆ 著 ［加］Herb Sutter ［罗］Andrei Alexandrescu
译 刘基诚
责任编辑 傅道坤
责任印制 张佳莹 焦志炜
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷
◆ 开本：800×1000 1/16
印张：14.25
字数：325 千字 2016 年 3 月第 1 版
印数：1－3 000 册 2016 年 3 月北京第 1 次印刷
著作权合同登记号 图字：01-2005-3574 号

定价：39.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315

内 容 提 要

在本书中，两位知名的C++专家将全球C++界20年的集体智慧和经验凝结成一套编程规范。这些规范可以作为每一个开发团队制定实际开发规范的基础，更是每一位C++程序员应该遵循的行事准则。书中对每一条规范都给出了精确的描述，并辅以实例说明；从类型定义到错误处理，都给出了最佳的C++实践，即使使用C++多年的程序员也会从本书中受益匪浅。

本书适合于各层次C++程序员使用，也可作为高等院校C++课程的教学参考书。

前　言

尽早进入正轨：以同样的方式实施同样的过程。不断积累惯用法。**将其标准化**。如此，你与莎士比亚之间的唯一区别将只是掌握惯用法的多少，而非词汇的多少。

——Alan Perlis[①]

标准最大的优点在于，它提供了如此多样的选择。

——出处尚无定论

我们之所以编写本书，作为各开发团队编程规范的基础，有下面两个主要原因。

- 编程规范应该反映业界最久经考验的经验。它应该包含凝聚了经验和对语言的深刻理解的公认的惯用法。具体而言，编程规范应该牢固地建立在大量丰富的软件开发文献的基础之上，把散布在各种来源的规则、准则和最佳实践汇集在一起。
- 不可能存在真空状态。通常，如果你不能有意识地制定合理的规则，那么就会有其他人推行他们自己喜欢的规则集。这样产生的编程规范往往具有各种最不应该出现的属性。例如，许多这样的编程规范都试图强制尽量少地按 C 语言的方式使用 C++。

许多糟糕的编程规范都是由一些没有很好地理解语言、没有很好地理解软件开发或者试图标准化过多东西的人制定的。糟糕的编程规范会很快丧失可信度，如果程序员不喜欢或者不同意其中一些糟糕的准则，那么即使规范中有一些合理的准则，也可能被不抱幻想的程序员所忽略，这还是最好的情况，最坏的情况下，糟糕的标准可能真会被强制执行。

如何使用本书

三思而行。应该遵循好的准则，但是不要盲从。在本书的各准则中，请注意“例外情况”部分阐明了该准则可能不适用的不太常见的情况。任何准则，无论如何正确（当然，我们自认为本

① 
Alan Perlis（1922—1990）“因为在高级程序设计技术和编译器构造领域的影响”而获得 1966 年首届图灵奖。他是影响深远的计算机科学先驱之一，曾担任 ALGOL 语言设计委员会主席，倡议创办了 *Communications of ACM* 杂志并担任首任主编，对计算机成为独立学科起到了关键性的作用。1982 年他在 *SIGPLAN* 杂志上发表的“Epigrams in Programming”（编程格言）一文，由 130 条格言组成，凝练隽永，多年来一直被广泛引用，本书亦然。——译者注

书中的准则是正确的），都不能代替自己的思考。

每个开发团队都应该制定自己的标准，制定标准的时候都应该尽职尽责。这项工作是整个团队的事情。如果你是团队负责人，应该让团队成员都参与制定标准。人们当然更愿意遵守“自己的”标准，而非别人强加的一堆规矩。

编写本书的目的是为各开发团队提供编程规范的基础和参考。它并不是要成为终极编程规范，因为不同的团队会有适合特定群体或者特定任务的更多准则，应该大胆地将这些准则加入本书的条款中。但是我们希望本书能够通过记载和引用广泛接受的、权威的、几乎可以通用的（“例外情况”指出的除外）实践经验，减少读者制定或重新制定自己的编程规范的工作量，从而帮助提高读者所用编程规范的质量和一致性。

让团队人员阅读这些准则及其原理阐释（也就是本书全文，根据需要还包括所选条款引用的其他书籍和论文），共同决定是否有团队根本无法接受的内容（比如，由于某些项目特殊的情况），然后实践其余规范。一旦采纳，如果未与整个团队协商，任何人不得违反团队编程规范。

最后，团队还需定期复查这些准则，加入实际应用中得出的经验和反馈。

编程规范与人的关系

好的编程规范能够带来下列许多相互关联的优点。

- *改善代码质量*。鼓励开发人员一贯地正确行事，从而能够直接提高软件的质量和可维护性。
- *提高开发速度*。开发人员不需要总是从一些基本原则出发进行决策。
- *增进团队精神*。有助于减少在一些小事上不必要的争论，使团队成员更容易阅读和维护其他成员的代码。
- *在正确的方向上取得一致*。使开发人员放开手脚，在有意义的方向上发挥创造性。

在压力和时间的要求下，人们将按所受到的训练行事。他们会求助于习惯。这正是医院的急诊室之所以要雇佣有经验的、训练有素的人员的原因所在，知识再渊博的新手到时候也会手足无措。

作为软件开发人员，我们总是面临着工期压缩的巨大的压力。在进度压力下，我们按所受过的训练和习惯工作。平时不知道软件工程良好实践的（或者不习惯应用这些实践的）马虎程序员，在压力下将编写出更加马虎、错误更多的代码。相反，养成良好习惯并经常按此工作的程序员将保持自己的组织性，快速提交高质量的代码。

本书所介绍的编程规范是集编写高质量C++代码准则之大成。它们是C++社区丰富集体经验的精华总结。这一知识体系中的大量内容，此前要么只能零零碎碎地从不同的书中找到，要么需要依靠口口相传。编写本书的目的就是要将这些知识收集起来，汇成一组简练合理、易于理解、容易实施的规则。

当然，即使有最佳编程规范，还是有人会编写出糟糕的代码。任何语言、过程或者方法皆然。

但是，好的编程规范集能够培养超越规则本身的良好习惯和纪律。这种基础一旦打好，就将打开通往更高层次的大门。这里没有捷径可走：在学会写诗之前必须首先扩大词汇量，熟悉语法。我们只希望本书能够对读者所经历的这一过程有所裨益。

我们希望本书适用于各种层次的C++程序员。

如果你是初级程序员，我们希望你能够发现，这些规则及其原理阐释有助于理解C++语言对哪些风格和惯用法的支持最为自然。我们为每一规则和准则都提供了简洁的原理阐释和讨论，这是为了鼓励你重在理解，而不是死记硬背。

对于中级或者高级程序员，我们下了很大功夫为每一规则都提供了详细的准确引用列表。这样，你就能够对规则在C++类型系统、语法和对象模型中的来历作进一步的研究。

总而言之，你很可能工作在一个复杂项目的团队中。这正是编程规范的用武之地——你可以使用这些规范将团队统一提高到同一层次，并为代码审查打下基础。

关于本书

我们为本书制定了以下设计目标。

- 一寸短，一寸强。篇幅极长的编程规范很难被人接受，而短小精悍的才会有人阅读和使用。同样，长的条款容易被人忽视，而短的条款才会被阅读和使用。
- 每个条款都必须是无可争议的。本书的目的是为了记载已达成广泛共识的标准，而不是凭空发明一些规范。如果有什么准则并非在所有情况下都适用，我们会明确指出（比如用“考虑……”而不是“应该……”来陈述），并提供普遍接受的例外情况。
- 每个条款都必须具有权威性。本书中的准则均引用已出版著作作为支持。编写本书的目的也包括提供C++文献的索引。
- 每个条款都必须有阐述的价值。我们不为肯定会做的事情（比如编译器已经强制要求或者检查的事情）或者其他条款已经涵盖的事情定义准则。
  例如，“不要返回自动变量的指针/引用”是一个不错的准则，但是我们没有将它放在本书中，因为我们测试过的所有编译器都会对此发出警告，所以这一问题已经涵盖在更广的第1条“在高警告级别干净利落地进行编译”中了。
  例如，“使用编辑器（或者编译器，或者调试器）”是一个不错的准则，但是你当然会使用这些工具，这是不言自明的。但前四个条款中有两个是关于其他工具的：“使用自动构建系统”和“使用版本控制系统”。
  例如，“不要滥用goto语句”是一个很好的条款，但是根据我们的经验，程序员普遍都知道这一点，对此毋庸多言。

每个条款都遵照下面的格式。

- 条款标题：最简单而又意味深长的“原音重现”，有助于更好地记忆规则。
- 摘要：最核心的要点，简要陈述。
- 讨论：对准则的展开说明。通常包括对原理的简要阐释，但是请记住，原理阐释的主要

部分都有意识地留在参考文献中了。

- *示例（可选）*：说明规则或者有助于记忆规则的实例。
- *例外情况（可选）*：规则不适用的任何（通常是比较罕见的）情况。但是要小心，不要掉入过于匆忙而不加思考的陷阱："噢，我很特殊，所以这条规则对我的情况并不适用。"这种推理很常见，但是通常都是错的。
- *参考文献*：可以参考其中提到的C++文献的章节，进而获得完整的细节和分析。

每一部分中，我们都会选择推荐一个"最有价值条款"。通常，它是该部分中的第一个条款，因为我们尽量将重要的条款放在每一部分的前面。但是有时候，出于连贯和可读性的考虑，我们不能将重要的条款前置，因此需要采取这样的办法突出它们，以引起特别注意。

致谢

非常感谢丛书[①]主编 Bjarne Stroustrup、本书的编辑 Peter Gordon 和 Debbie Lafferty，还有 Tyrrell Albaugh、Kim Boedigheimer、John Fuller、Bernard Gaffney、Curt Johnson、Chanda Leary-Coutu、Charles Leddy、Heather Mullane、Chuti Prasertsith、Lara Wysong，以及 Addison-Wesley 团队的其他成员，感谢他们在本书写作过程中给予的协助和坚持。能和他们共事很荣幸。

各条款标题中"原音重现"的灵感有许多来源，包括[Cline99]的幽默风格和[Peters99]中经典的**"import this"**，还有具传奇色彩的 Alan Perlis 和他被广为引用的格言。

我们特别想感谢在本书技术审阅方面做出贡献的人，他们的工作使本书许多部分增色不少。从选题开始直到最终成稿，丛书主编 Bjarne Stroustrup 所提出的尖锐而又透彻的意见对本书影响至深。我们要特别感谢 Dave Abrahams、Marshall Cline、Kevlin Henney、Howard Hinnant、Jim Hyslop、Nicolai Josuttis、Jon Kalb、Max Khesin、Stan Lippman、Scott Meyers 和 Daveed Vandevoorde 积极参与审阅，并对原稿多个版本的草稿都提出了详细的意见。其他有价值的意见和反馈要归功于 Chuck Allison、Samir Bajaj、Marc Barbour、Damian Dechev、Steve Dewhurst、Peter Dimov、Alan Griffiths、Michi Henning、James Kanze、Matt Marcus、Petru Marginean、Robert C."Uncle Bob" Martin、Jeff Peil、Peter Pirkelbauer、Vladimir Prus、Dan Saks、Luke Wagner、Matthew Wilson 和 Leor Zolman。

与往常一样，书中仍然会存在错误、疏漏和含混之处，对此作者负全责。

Herb Sutter
Andrei Alexandrescu
2004 年 9 月
于美国华盛顿州西雅图市

① 指 Addison-Wesley 公司出版的著名的 *C++ In-Depth* 丛书，包括 *Essential C++*、*Accelerated C++*、*Exceptional C++*、*Modern C++ Design* 等，这些书多已在国内引进出版。——译者注

目　录

组织和策略问题

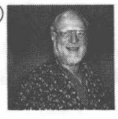
如果人们按照程序员编程的方式修建房屋，
那么一只啄木鸟就能毁灭整个文明。

——Gerald Weinberg[①]

为了遵从 C 和 C++的伟大传统，我们从 0 开始编号。首要的指导原则，也就是第 0 条，阐明了我们认为对编程规范而言最为基本的建议。

接下来，这个导论性部分的其他条款将主要讲述几个精心选择的基本问题，这些问题大多数与代码本身并没有直接关系，它们讨论的是编写坚实代码所必需的工具和技术。

本部分中我们选出的最有价值条款是第 0 条：“不要拘泥于小节”（又名：“了解哪些东西不应该标准化”）。

① 
Gerald Weinberg 是著名的软件思想家（他自称 thinker），因《程序开发心理学》等书名世，以对软件开发的深刻洞察而著称。——译者注

第0条
不要拘泥于小节[①]（又名：了解哪些东西不应该标准化）

摘要

只规定需要规定的事情：不要强制施加个人喜好或者过时的做法。

讨论

有些问题只是个人喜好，并不影响程序的正确性或者可读性，所以这些问题不应该出现在编程规范中。任何专业程序员都可以很容易地阅读和编写与其习惯的格式略有不同的代码。

应该在每个源文件乃至每个项目中都使用一致的格式，因为同一段代码中要在几种编程风格（style）之间换来换去是很不舒服的。但是无需在多个项目或者整个公司范围内强制实施一致的格式。

下面列举了几种常见的情况，在这里重要的不是设定规则，而是与所维护的文件中已经使用的体例保持一致。

- *不要规定缩进多少，应该规定要用缩进来体现代码的结构。*缩进空格的数量可以遵照个人习惯，但是至少在每个文件中应该保持一致。
- *不要强制行的具体长度，应该保证代码行的长度有利于阅读。*可以遵照个人习惯来决定行长，但是不要过长。研究表明，文字长度不超过 10 个单词最利于阅读。
- *不要在命名方面规定过多，应该规定的是使用一致的命名规范。*只有两点是必需的：（1）永远不要使用“晦涩的名称”，即以下划线开始或者包含双下划线的名称；（2）总是使用形如 ONLY_UPPERCASE_NAMES 的全大写字母表示宏，不要考虑使用常见的词或者缩略语作为宏的名称（包括常见的模板参数，比如 T 和 U；像#define T *anything* 这样的代码是极容易混淆的）。此外，应该使用一致的、有意义的名称，遵循文件的或者模块的规范。（如果你无法决定自己的命名规范，可以尝试如下命名规则：类、函数和枚举的名称形如 LikeThis，即单词首字母大写；变量名形如 likeThis，即第一个单词首字母小写，第二个单词首字母大写；私有成员变量名形如 likeThis_；宏名称形如 LIKE_THIS。）
- *不要规定注释风格（除非需要使用工具从特定的体例中提取出文档），应该编写有用的注释。*尽可能编写代码而不是写注释（比如，见第 16 条）。不要在注释中重复写代码语义，这样很容易产生不一致。应该编写的是解释方法和原理的说明性注释。

最后，不要尝试强制实施过时的规则（见例 3 和例 4），即使它们曾经在一些比较陈旧的编程规范中出现过。

示例

例 1　*大括号的位置。*以下代码在可读性方面并不存在区别：

---

① 出自 Richard Carlson 的畅销书 *Don't Sweat the Small Stuff... and It's all Small Stuff*，中文版译名为《别为小事抓狂》。——译者注

```
void using_k_and_r_style() {
  //……
}①

void putting_each_brace_on_its_own_line()
{
  //……
}②

void or_putting_each_brace_on_its_own_line_indented()
  {
  //……
  }③
```

任何一个专业程序员都能够毫无困难地阅读和编写这些体例中的任何一种。但是应该保持一致：不要随意地或者以容易混淆作用域嵌套关系的方式放置大括号，要尽量遵循每个文件中已经使用的体例。在本书中，对大括号位置的选择主要是为了能够在编辑允许范围内得到最佳可读性。

例 2 *空格与制表符*。有些团队禁用制表符（比如[BoostLRG]），因为不同的编辑器中制表符的设定是不同的，如果使用不当，会将缩进变为“缩出”和“无缩进”。其他同样受人尊敬的团队则允许使用制表符，并采取了一些能够避免其潜在缺陷的规定。这都是合理的，其实只要保持一致即可：如果允许使用制表符，那么要确保团队成员维护彼此的代码时，不会影响代码的清晰和可读性（见第 6 条）。如果不允许使用制表符，应该允许编辑器在读入源文件时将空格转换为制表符，使用户能够在编辑器中使用制表符，但是在将文件写回时，一定要将制表符转换回空格。

例 3 *匈牙利记法*。将类型信息并入变量名的记法，是混用了类型不安全语言（特别是 C）中的设施，这在面向对象语言中是可以存在的，但是有害无益，在泛型编程中则根本不可行。所以，任何 C++编程规范都不应该要求使用匈牙利记法，而在规范中选择禁用该法则是合理的。

例 4 *单入口，单出口*（SESE，Single Entry, Single Exit）。历史上，有些编程规范曾经要求每个函数都只能有一个出口，也就意味着只能有一个 return 语句。这种要求对于支持异常和析构函数的语言而言已经过时了，在这样的语言中，函数通常都有多个隐含的出口。取而代之，应该遵循类似于第 5 条那样的标准，即直接提倡更简单的、更短小的函数，这样的函数本身更易于理解，更容易防错。

参考文献

[BoostLRG] • [Brooks95]§12④ • [Constantine95]§29 • [Keffer95]p.1 • [Kernighan99] §1.1, §1.3, §1.6-7 • [Lakos96]§1.4.1, §2.7 • [McConnell93] §9⑤, §19 • [Stroustrup94] §4.2-3 • [Stroustrup00] §4.9.3, §6.4, §7.8, §C.1 • [Sutter00] §6, §20 • [SuttHysl01]

① 这是纯 C（K&R）风格的括号位置，如函数名所示。——译者注

② 每个括号一行，如函数名所示。——译者注

③ 每个括号一行并缩进，如函数名所示。——译者注

④《人月神话》第 12 章“干将莫邪”（Sharp Tools）主要讨论工具问题。其中谈到：“项目经理必须考虑、计划、组织的工具到底有哪些呢？……需要语言，语言的使用准则必须明确。”——译者注

⑤ *Code Complete* 一书（此书新版已经于 2004 年底出版）的第 9 章“Data Names”。——译者注

第 1 条
在高警告级别干净利落地进行编译

摘要

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

讨论

编译器是你的朋友。如果它对某个构造发出警告，一般表明代码中存有潜在的问题。

成功的构建应该是无声无息的（没有警告的）。如果不是这样，你很快就会养成不仔细查看输出的习惯，从而漏过真正的问题（见第 2 条）。

排除警告的正确做法是：（1）把它弄清楚；然后，（2）改写代码以排除警告，并使代码阅读者和编译器都能更加清楚，代码是按编写者的意图执行的。

即使程序一开始似乎能够正确运行，也还是要这样做。即使你能够肯定警告是良性的，仍然要这样做。因为良性警告的后面可能隐藏着未来指向真正危险的警告。

示例

例 1　第三方头文件。无法修改的库头文件可能包含引起警告（可能是良性的）的构造。如果这样，可以用自己的包含原头文件的版本将此文件包装起来，并有选择地为该作用域关闭烦人的警告，然后在整个项目的其他地方包含此包装文件。例如（请注意，各种编译器的警告控制语法并不一样）：

```
// 文件：myproj/my_lambda.h —— 包装了Boost的lambda.hpp
// 应该总是包含此文件，不要直接使用lambda.hpp。
// 注意：我们的构建现在会自动检查grep lambda.hpp <srcfile>。
// Boost.Lambda 会产生一些已知无害的编译器警告。
// 在改正以后，我们将删除以下的编译指示，但此头文件仍然存在。
//
#pragma warning(push)          // 仅禁用此头文件
 #pragma warning(disable:4512)
 #pragma warning(disable:4180)
 #include <boost/lambda/lambda.hpp>
#pragma warning(pop)           // 恢复最初的警告级别
```

例 2 “未使用的函数参数”（Unused function parameter）。检查一下，确认确实不需要使用该函数参数（比如，这可能是一个为了未来扩展而设的占位符，或者是代码没有使用的标准化函数签名中的一个必需部分）。如果确实不需要，那直接删除函数参数名就行了。

```
// ……在一个用户定义的allocator中未使用hint ……
// 警告：unused parameter 'localityHint
pointer allocate( size_type numObjects, const void *localityHint = 0 ) {
   return static_cast<pointer>( mallocShared( numObjects * sizeof(T) ) );
}
// 消除了警告的新版本
pointer allocate( size_type numObjects, const void * /* localityHint */ = 0 ) {
   return static_cast<pointer>( mallocShared( numObjects * sizeof(T) )      );
}
```

例 3 “定义了从未使用过的变量”（Variable defined but never used）。检查一下，确认并不是真正要引用该变量。（RAII 基于栈的对象经常会引起此警告的误报，见第 13 条。）如果确实不需要，经常可以通过插入一个变量本身的求值表达式，使编译器不再报警。（这种求值不会影响运行时的速度。）

```
// 警告：variable 'lock' is defined but never used
void Fun() {
   Lock lock;
   // ……
}
// 可能消除了警告的新版本
void Fun() {
   Lock lock;
   lock;
   // ……
}
```

例 4 “变量使用前可能未经初始化”（Variable may be used without being initialized）。初始化变量（见第 19 条）。

例 5 “遗漏了 return 语句”（Missing return）。有时候编译器会要求每个分支都有 return 语句，即使控制流可能永远也不会到达函数的结尾（比如：无限循环，throw 语句，其他的返回形式等）。这可能是一件好事，因为有时候你仅仅是认为控制不会运行到结尾。例如，没有 default 情况的 switch 语句不太适应变化，应该加上执行 assert(false) 的 default 情况（见第 68 条和第