深入C++系列

Effective C++、More Effective C++的作者
Scott Meyers亲自为本书作序！

# Exceptional C++
# 中文版

Herb Sutter 著
卓小涛 译

Addison Wesley

中国电力出版社
www.infopower.com.cn

深入C++系列

# Exceptional C++
# 中文版

Herb Sutter 著
卓小涛 译


中国电力出版社

**Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions (ISBN 0-201-61562-2)**

**Herb Sutter**



图书在版编目（CIP）数据

Exceptional C++中文版/（美）萨特著；卓小涛译. —北京：中国电力出版社，2003

(深入 C++系列)

ISBN 7-5083-1485-9

Ⅰ.E… Ⅱ.①萨…②卓… Ⅲ.C 语言 - 程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 017800 号

**责任编辑**：乔晶

**丛 书 名**：深入C++系列

**书　　名**：Exceptional C++中文版

**编　　著**：(美) 萨特

**翻　　译**：卓小涛

**出版发行**：中国电力出版社

地址：北京市三里河路6号　邮政编码：100044

电话：（010）88515918　传真：（010）88518169

**印　　刷**：北京地矿印刷厂

**开　　本**：787×1092 1/16　　**印　　张**：14.5　　**字　　数**：297 千字

**书　　号**：ISNB 7-5083-1485-9

**版　　次**：2003年3月北京第一版

**印　　次**：2003年9月第二次印刷

**定　　价**：35.00 元

# Foreword

# 序

这是一本非同寻常的书，但是直到我差不多读完全书之后，我才真正意识到它是多么非同寻常。这可能是第一本写给那些已经熟悉 C++（熟悉 C++的一切）的人看的书。从语言特性到标准库的组件，再到编程技巧，本书从一个主题跳到另一个主题，始终使我们略微偏离平衡，始终吸引着我们的注意力。这与真正的 C++程序异曲而同工。在这里，类的设计将遇到虚函数的行为，迭代器惯例会碰上名字查找规则，赋值操作符将撞上异常安全性，而编译依赖性又会邂逅导出模板。正如它们在真实程序中那样，结果也正如真实的程序。语言特性、标准库组件和编程技巧构成了混乱而精彩的、令人晕眩的“大漩涡”。

我将 GotW（Guru of the Week，每周大师）读得与“Gotcha”押韵（译注 1），这或许是适宜的。当我将自己对书中小智力题的解答与 Sutter 的答案相比较时，虽然不想承认，但我确实经常掉进他（和 C++）设置的陷阱中。对于我犯的每个错误，我几乎总能看见 Herb 在微笑并且轻轻地说“Gotcha!”。一些人可能由此认为我不太了解 C++。另外一些人则可能认为，这说明了对于任何一个人而言，C++都太复杂了，确实不容易掌握。而我却认为，这说明了当我们使用 C++进行工作时，必须仔细地考虑正在做的事情。C++是被设计来帮助解决复杂问题的功能强大的语言，尽可能细致地琢磨这门语言、标准库以及编程习惯用法方面的知识，是非常重要的。本书主题的广度和独一无二的测试型格式在此方面必将对读者有所裨益。

C++新闻组的老读者们都知道选出一位“Guru of the Week”是多么困难。老资格的参

---

译注 1：意即将 Gotw 读成“got wa:”。Gotcha 即 got you，美国俚语，原意为“抓到你了”、“把你搞定了”，作为名词指陷阱、误区。这是当今开发人员喜用的黑话。

与者对此肯定更有体会。当然，在网上每周只会有一位 Guru 产生，但是在本书内容的帮助下，我们有理由希望每次编程时都能写出 Guru 质量的代码。



Scott Meyers

# Preface

# 前　言

本书将通过举例讲述如何开始实施坚实的软件工程。除了补充了大量其他材料之外，本书主要涵盖了 Internet 上流行的 C++专栏 Guru of the Week（简称为 GotW）的前 30 个问题的扩充版本。这是一系列，阐明了特定设计和编码技巧自成一体的 C++工程问题及其解决方案。

本书不是装满代码难题的随机摸彩袋；它主要是使用 C++语言揭示现实世界企业级软件设计的一个向导。它使用了“问题/解答”的形式，因为这是我所知道的最有效的方法，可以将你（亲爱的读者）引到问题背后的思想和指导准则背后的原因。虽然这些条款覆盖了各种各样的主题，但是你将注意到重复出现的主题都集中在企业级开发问题上，尤其是异常安全性、健全的类和模块设计、适当的优化以及可移植且符合标准的代码的编写。

我希望在日常工作中你将发现这些材料是有用的。但是我同时希望，你至少能发现一些极好的想法和精美的技术，而且当你通读这本书时，你会突然发出这样的赞叹：“啊！真复杂呀！”。谁说软件工程索然无趣来着？

## 怎样阅读本书？

我假定你已经掌握了 C++基本知识。如果还没有，应该从一本好的 C++介绍性和概述性的书籍开始（经典巨著，如 Bjarne Stroustrup 的《The C++ Programming Language,Third Edition》或者 Stan Lippman 和 Josee Lajoie 的《C++ Primer,Third Edition》（译注 1）都是一时之选），然后应该阅读介绍编程风格的指导书籍，如 Scott Meyers 的经典著作《Effective

译注 1：中文版《C++ Primer（3RD）中文版》已由中国电力出版社出版。

C++》和《More Effective C++》（译注 2）（我发现基于浏览器的 CD 版本非常方便合用）。

本书的每个条款都以引导性的标题来提出一个问题，如下所示：

> 第##条：主题名称　　难度：X
>
> 主题名称和难度系数（通常从 3~9½，按满分 10 计算）提示我们将要遇到什么。注意，这个难度系数只是我根据大部分人可能的反应而做的主观猜测，因此可能会发现对你而言，一个难度为 7 的问题比另一个难度为 5 的问题更容易。尽管如此，当你见到难度系数为 9½时，最好还是做好最坏的打算。
>
> 没有必要按照顺序来阅读这些章节和问题，但是在几个地方有一些相关的“小系列”，它们被标明为“之一”、“之二”等等。有些地方甚至有“之十”。最好作为一个整体阅读这些小系列。

## 本书的来龙去脉：GotW 和 PeerDirect

C++ Guru of the Week 已经有很长的历史了。GotW 最初创建于 1996 年底，用来为我们自己在 PeerDirect 的开发小组提供一些有趣的挑战，同时进行继续教育。我编写它来提供一种寓教于乐的学习工具，包括异常安全性和继承的正确用法之类的详细讲述。随着时间的推移，我也把它当作一种工具来向我们小组说明 C++标准会议上对 C++所做的改动。自那以来，GotW 作为 Internet 新闻组 comp.lang.C++.moderated 的一个固定专栏对广大的 C++开发人员发布，在这个新闻组里，你可以发现每一期的问题和答案（以及许多有趣的讨论）。

在 PeerDirect 公司里用好 C++很重要，这与在你的公司用好 C++很重要的理由相同，但是目标可能不同。我们的业务是构建系统软件（用于分布式数据库和数据库复制），其中诸如可靠性、安全性、可移植性、高效率以及其他一些企业级特性关乎软件的成败。我们写的软件必须可以在不同的编译器和操作系统上进行移植。在出现数据库事务死锁、通信中断和程序异常的情况下，软件必须是安全和健壮的。用户可以使用它来管理位于智能卡和 POS 机或者 PalmOS 和 WinCE 设备上的微型数据库，部门的 Windows NT、Linux 和 Solaris 服务器，乃至用于 Web 服务器的大规模并行的后台 Oracle 数据库以及数据仓库——使用的都是相同的软件，相同的可靠性，相同的代码。现在，当我们逐渐编写出大量紧凑而无注



释行的代码时，所面临的就是可移植性和可靠性的挑战。

对于过去几年中一直在 Internet 上阅读 Guru of the Week 的读者，我要说两件事情：

- 感谢你们的兴趣、支持、电子邮件、称赞、更正、评论、批评和提问——尤其是你们对将 GotW 系列编辑成书的要求。它应运而生了，希望你们能喜欢这本书。

---

译注 2：中文版《More Effective C++中文版》与本书同时由中国电力出版社出版。

■ 这本书包含了比 Internet 上多得多的内容。

本书不只是网络空间中原有的陈旧 GotW 问题的拷贝和粘贴。所有的问题和解答都已经相当大程度上被修改和重新处理过——例如，关于异常安全性的条款 8~17 最初只是一个 GotW 难题，而现在已经变成了一个深入、有十个部分的小系列。每个问题和解答都经过了仔细检查，以符合最新修改的正式 C++标准。

因此，即使你以前是 GotW 的老读者，这本书仍然会为你提供大量的新内容。再一次感谢所有忠实的朋友，另外，我希望这份材料有助于你继续磨励和扩展自己的软件工程和 C++编程技术。

## 致谢

首先，当然要感谢所有 comp.lang.C++.moderated 上的 GotW 读者和热爱者，尤其是那些参与了为本书选择名字的人。我想特别感谢其中对最后的书名有帮助的两位读者：Marco Dalla Gasperine 建议的书名是“Enlightented C++”；Rob Stewart 建议的书名是“Practical C++ Problems and Solutions”。考虑到本书反复强调异常安全性，所以在这些名字的基础上再进一步，很自然地最终选择双关语 exceptional。

同样感谢丛书编辑 Bjarne Stroustrup，感谢 Marina Lang、Debbie Lafferty 以及其他 Addison Wesley Longman 出版公司的编辑成员，感谢他们对本项目持续的兴趣和热情，感谢他们在 1998 年 Santa Cruz C++标准会议上举行的一次非常好的招待会。

我也想感谢所有的审读人员（其中许多是标准委员会的成员），他们提出的深思熟虑、一针见血的建议大大改善了本书的内容。特别要感谢 Bjarne Stroustrup 和 Scott Meyers, Andrei Alexandrescu、Steve Clamage、Steve Dewhurst、Cay Horstmann、Jim Hyslop、Brendan Kehoe 以及 Dennis Mancl，感谢他们出色的洞察力和宝贵的意见。

最后，衷心感谢我的家人和朋友以各种方式陪伴着我。

Herb Sutter



### 请注意

本书索引所列页码，皆为英文原书页码。

# Content

# 目　录

CHAPTER 1

# 第 1 章
# 泛型程序设计与 C++标准库

我们先来思考泛型领域的一些经过精挑细选的主题。这些问题的焦点是模板、迭代器和算法的有效使用，以及如何使用和扩展标准库设施。这些想法很自然地导出了接下来的章节，这些章节将在编写异常安全模板的大背景中分析异常安全性。

## 第 1 条：迭代器　　难度：7

每个使用标准库的程序员都必须知道以下常见的和不那么常见的迭代器（iterator）错误。你能发现多少个？

如下程序至少有四个与迭代器相关的问题。你能发现几个？

```
int main()
{
  vector<Date> e;
  copy( istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ) );
  vector<Date>::iterator first =
        find( e.begin(), e.end(), "01/01/95" );
  vector<Date>::iterator last =
        find( e.begin(), e.end(), "12/31/95" );
  *last = "12/30/95";
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
  e.insert( --e.end(), TodaysDate() );
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
}
```

## 解 答

```
int main()
{
  vector<Date> e;
  copy( istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ) );
```

目前一切正常。Date 类的编写者提供了一个原型（signature）为 operator>>(istream&, Date&)的抽取函数（extractor function），istream_iterator<Date>正是使用这个函数从 cin 流读取 Date 数据。这个 copy()算法只是将 Date 填充到向量中。

```
vector<Date>::iterator first =
      find( e.begin(), e.end(), "01/01/95" );
vector<Date>::iterator last =
      find( e.begin(), e.end(), "12/31/95" );
*last = "12/30/95";
```

错误：带阴影的这句可能是非法的，因为 last 可能等于 e.end()，因此可能不是一个可解除引用的（dereferenceable）迭代器。

如果值没有找到，那么 find()算法将返回它的第二个参数（范围的终止迭代器）。在这种情况下，如果“12/31/95”不存在于 e 中，那么 last 将等于 e.end()，它指向这个容器最后一个对象的下一个位置（one-past-the-end），这不是一个有效的迭代器。

```
copy( first,
      last,
      ostream_iterator<Date>( cout, "\n" ) );
```

错误：这可能是非法的，因为[first,last)可能不是一个有效的范围；实际上，first 确实可能在 last 之后。

例如，如果在 e 中没有找到“01/01/95”而找到了“12/31/95”，那么迭代器 last 所指向的集合内的对象（等于“12/31/95”的 Date 对象）将比迭代器 first 指向的（最后一个对象的下一个位置）更早。然而，copy()算法要求在相同的集合中 first 必须比 last 指向更早的位置——也就是说，[first,last)必须是一个有效的范围。

如果这种情况发生了，那么可能的故障现象是在 copy()函数过程之中或之后的某个时候，出现一个难以诊断的核心转储（core dump）（译注 1），除非你正在使用的标准库是一个检验过的版本，可以检测出某些此类的问题。

```
e.insert( --e.end() TodaysDate() );
```

第一个错误：表达式“--e.end()”可能是非法的。

---

译注 1：指出现程序错误时，内存中原始数据的输出。

原因很简单，虽然有一些隐晦：在标准库的流行实现产品中，vector<Date>:: iterator 经常只是一个 Date*，而且 C++语言不允许修改内置类型的临时变量。例如，如下代码也是非法的：

```
Date* f();        // 函数返回一个 Date*
p = --f();        // 错误，但可以写成 "f() - 1"
```

幸运的是，vector<Date>::iterator 是一个随机存取迭代器，因此编写如下正确代码将不会有任何效率损失。

```
e.insert( e.end() - 1, TodaysDate() );
```

第二个错误：如果 e 是空的，那么任何获得“e.end()之前某位置迭代器”的尝试[不论是写成“--e.end()”还是“e.end()-1”]，所得都将是一个无效的迭代器。

```
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
}
```

错误：first 和 last 可能不再是有效的迭代器。

vector 是以“块状（chunk）”的方式增长的，因此它并不是每次插入对象都必须重新分配缓冲区。然而，当 vector 已满时，再插入对象，将触发重新分配。

这里，e.insert()操作的结果是 vector 可能增长也可能不增长，这意味着它的内存可能移动也可能不移动。由于这种不确定性，我们必须认为容器的任何已存在的迭代器都是无效的。在这种情况下，如果内存确实移动了，那么这个问题多多的 copy()函数通常会再次生成难以诊断的核心转储。

**原 则**

永远不要解除无效迭代器的引用。

小结：使用迭代器时，注意四个主要问题。

1．有效值：迭代器可以解除引用吗？例如，写成“*e.end()”代码永远是编程错误。

2．有效生存期：使用迭代器时，仍然是有效的吗？或者这样思考：自从我们获得它以来，是否有一些操作已经使它无效了？

3．有效范围：一对迭代器是一个有效范围吗？first 确实在 last 之前（或相等）吗？两个迭代器确实都指向同一个容器的元素吗？

4．非法的内置类型操作：例如，是否有代码企图去修改内置类型的临时对象，正如上述的“--e.end()”。（幸运的是，编译器通常会捕获这种错误，对于类类型的迭代器，为了语法方便，库的作者通常会选择允许这种操作。）

## 第2条：不区分大小写的字符串——之一 难度：7

你需要一个不区分大小写的字符串类吗？如果选择同意，那么你的任务就是编写一个。

本条款由三个相关要点组成。

1.“不区分大小写”是什么意思？

2. 编写一个与标准的 std::string 类完全相似的 ci_string 类，但是它与通常提供的扩展函数 stricmp()（注 1）一样是不区分大小写的。ci_string 应该能够如下使用：

```
ci_string s( "AbCdE" );
// 不区分大小写
//
assert( s == "abcde" );
assert( s == "ABCDE" );
// 当然大小写依然不变
//
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

3. 区分大小写作为对象的属性，这是个好主意吗？

### 解 答

这三个问题的答案如下：

1.“不区分大小写”是什么意思？

“不区分大小写”实际上完全依赖于你的应用程序和所使用的语言。例如，一些语言根本就没有大小写的区分。对于那些区分大小写的语言，你仍然必须决定重音字符是否等同于非重音字符以及诸如此类的事。关于怎样为标准的字符串类实现不区分大小写，使它在各个方面都适用于你的情况，本条款提供了一些原则。

2. 编写一个与标准的 std::string 类完全类似的 ci_string 类，但是它与通常提供的扩展函数 stricmp()一样是不区分大小写的。

“怎样才能生成一个不区分大小写的字符串呢？”这个问题非常普遍，以至于应该专为它设一个 FAQ（常见问答）了——因此就有了这个条款。



以下是我们需要完成的：

注 1：stricmp()是不区分大小写的字符串比较函数，不是 C 或 C++标准的一部分，但是它是许多 C 和 C++编译器中都有的扩展函数。

```
ci_string s( "AbCdE" );
// 不区分大小写
//
assert( s == "abcde" );
assert( s == "ABCDE" );
// 当然大小写依然不变
//
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

此中关键是理解标准 C++里 string 的真实意义。看一下可以信赖的 string 头文件，将看到如下代码：

```
typedef basic_string<char> string;
```

因此 string 实际上不是一个类，它是一个模板的 typedef（类型定义）。而 basic_string<> 模板的声明又如下（可能带有其他与实现相关的模板参数）：

```
template<class charT,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_string;
```

因此“string”实际上指的是“basic_string <char, char_traits <char> , allocator<char> >”，可能带有与正在使用的实现相关的其他默认模板参数。我们不用管 allocator 部分，关键是 char_traits，因为 char_traits 定义了字符间相互作用和相互比较的方式。

那么让我们来比较 string 吧。basic_string 提供了有用的函数来比较一个 string 是等于另一个，还是小于另一个，诸如此类。这些 string 比较函数是以由 char_traits 模板提供的字符比较函数为基础的。尤其是，char_traits 模板提供了名为 eq()和 lt()的字符比较函数来进行相等和小于比较，提供了 compare()和 find()函数来比较和查找字符序列。

如果我们想要这些函数有不同的行为表现，所要做的只是提供一个不同的 char_traits 模板。这是最容易的方法：

```
struct ci_char_traits : public char_traits<char>
              // 只需继承无需替换的其他函数

{
  static bool eq( char c1, char c2 )
    { return toupper(c1) == toupper(c2); }
  static bool lt( char c1, char c2 )
    { return toupper(c1) <  toupper(c2); }
  static int compare( const char* s1,
                      const char* s2,
                      size_t n )
    { return memicmp( s1, s2, n ); }
          // 如果平台上有的话
          // 否则得自己写了
```

```
    static const char*
    find( const char* s, int n, char a )
    {
      while( n-- > 0 && toupper(*s) != toupper(a) )
      {
          ++s;
      }
      return n > 0 ? s : 0;
    }
};
```

最后，连起来：

```
typedef basic_string<char, ci_char_traits> ci_string;
```

我们所做的只是创建了一个名为 ci_string 的 typedef。除了使用 ci_char_traits 代替 char_traits<char>来取得自己的字符比较规则之外，它的操作完全类似于标准 string（毕竟，在大部分方面它仍然“是”标准 string）。由于已经方便地将 ci_char_traits 规则定为不区分大小写的了，我们也就已经使 ci_string 自身成为不区分大小写的了，无需任何进一步的外科手术。也就是说，我们根本不用触及 basic_string，就有了一个不区分大小写的字符串类。这就是所谓的可扩展性。

3．区分大小写作为对象的属性，这是个好主意吗？

让不区分大小写成为比较函数的而不是对象的属性，通常更为有用。例如，考虑如下代码：

```
string    a = "aaa";
ci_string b = "aAa";
if( a == b ) /* ... */
```

对于给定的一个适当的 operator==()，表达式“a == b”的值应该为 true 还是 false 呢？我们很容易接受这种观点：如果任意一边都不区分大小写，那么这个比较应当也是不区分大小写的。但是如果我们对这个例子只做一点点修改，引入 basic_string 的另一个实例化进行第三种方式的比较，又会怎么样呢？

```
typedef basic_string<char, yz_char_traits> yz_string;

ci_string b = "aAa";
yz_string c = "AAa";
if( b == c ) /* ... */
```

现在，再次考虑这个问题：表达式“a == b”的值应该为 true 还是 false 呢？在这种情况下，我们应当任意选择一个对象的优先级高于另一个对象，这并不那么显而易见，我想你会同意这一点吧。



相反，考虑一下，如果这个例子如下编写，将会变得多么清楚。

```
string a = "aaa";
string b = "aAa";
if( stricmp( a.c_str(), b.c_str() ) == 0 ) /* ... */
string c = "AAa";
if( EqualUsingYZComparison( b, c ) ) /* ... */
```

在许多情况下，使区分大小写成为比较操作的特性更加有用。但是我在实践中遇到了一些情况，在这些情况中，区分大小写成为对象的特性（尤其是当大部分或所有的比较都是使用 C 风格 char*字符串时）可能更有用，因为这样我们就可以自然而简单地比较这些值，比如“if(a == "text")等，而不用每次都要记住使用不区分大小写的比较。

本条款主要就 basic_string 模板的工作原理及其在实践中的灵活运用提供了一些帮助。如果你需要的功能与那些 memicmp()和 toupper()函数所能提供的不同，那么用自己的代码替换这里列出的五个函数，让它执行适合应用程序的字符比较功能好了。