HZ BOOKS

PEARSON Addison Wesley

Modern C++ Design C++ 设计新思维

C++标准库扩展权威指南

The C++ Standard Library Extensions

A Tutorial and Reference

（美） Pete Becker 著

史晓明 译

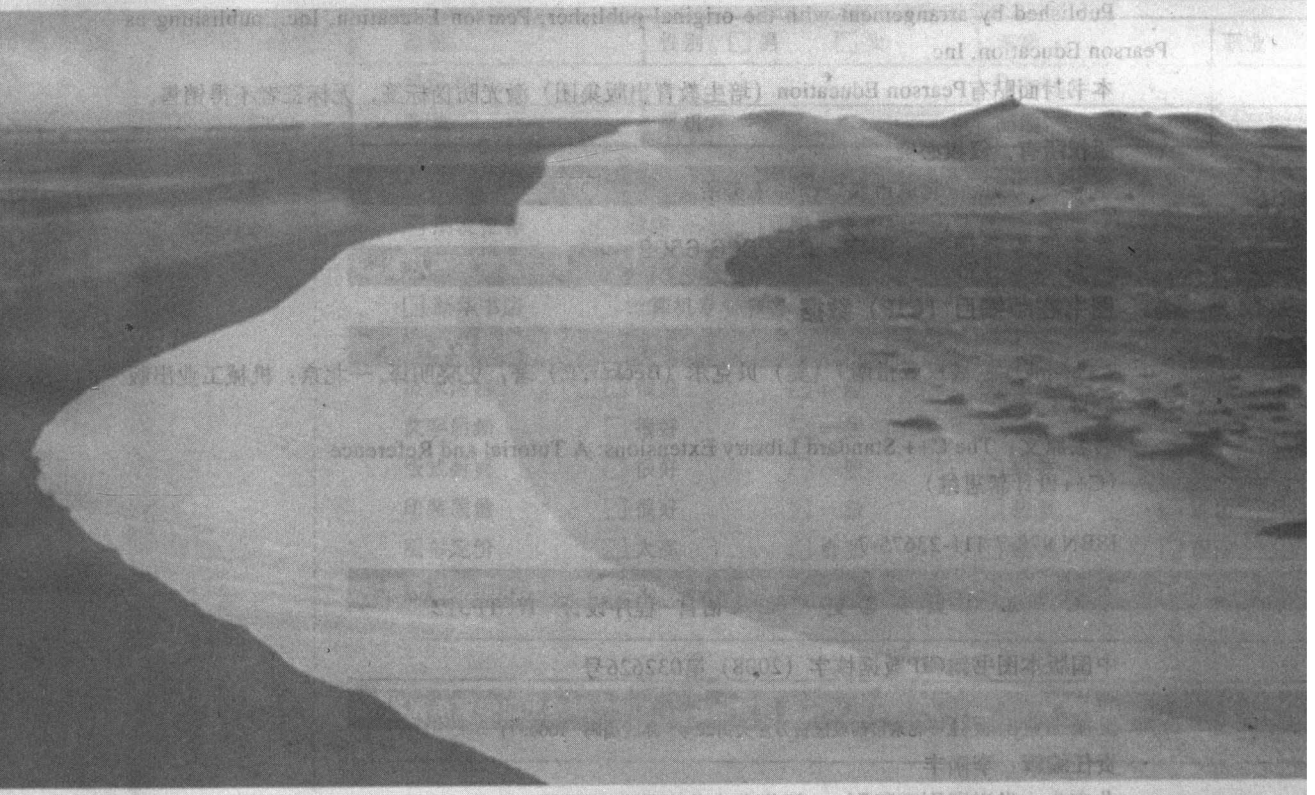
机械工业出版社
China Machine Press

Modern C++ Design
C++ 设计新思维

C++标准库扩展权威指南

The C++ Standard Library Extensions

A Tutorial and Reference



(美) Pete Becker 著
史晓明 译

机械工业出版社
China Machine Press

本书是对TR1进行了完整的介绍。全书共22章，包括元组、智能指针、类模板array、无序关系容器、调用包装器基础、mem_fn函数模板、reference_wrapper类模板、类模板function、函数模板bind、类型特性、数值函数、随机数生成器、头文件<regex>、正则表达式对象、查找、格式化和文件替换、正则表达式的定制以及C语言兼容等内容。全书不仅对枯燥的标准文本给出了浅显易懂的解释，还提供了很多示例和练习来帮助我们理解。每个C++程序员都可以从本书中获益匪浅。

本书版权登记号：图字：01-2006-6509

图书在版编目（CIP）数据

C++标准库扩展权威指南 /（美）贝克尔（Becker, P.）著；史晓明译. —北京：机械工业出版社，2008.4

书名原文：The C++ Standard Library Extensions: A Tutorial and Reference

（C++设计新思维）

ISBN 978-7-111-23675-7

Ⅰ. C…　Ⅱ. ① 贝…　② 史…　Ⅲ. C语言—程序设计　Ⅳ. TP312

中国版本图书馆CIP数据核字（2008）第032626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：李南丰

北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行

2008年4月第1版第1次印刷

186mm × 240mm · 26.5印张

标准书号：ISBN 978-7-111-23675-7

定价：56.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

译 者 序

在和C++朝夕共处的10多年中，我对它总是又爱又恨。作为一种语言来说，C++不仅具有极强的表达能力，还能支持多种编程范式，用它来描写自己心中的想法总是那么自然。然而，一旦牵涉到库，那就是另一番局面了。如果所要的功能没有在C++标准库或是（所有）目标编译器附带的类库中提供，那就是一场灾难。就拿几乎每个大型项目都需要的智能指针来说，虽然有那么多的选择，但是其行为、实现和使用方式各不相同，更不用说能否为我们的目标编译器所支持了。因此很多时候，自己编写库是唯一自然的选择。

TR1库是C++通向完善之路上迈出的又一步，它为C++标准库加入了很多新功能。虽然其各个部件早已在各种类库（尤其是boost）中初见雏形，但是被纳入标准的重要意义是不言而喻的。本书是对TR1库的完整介绍，不仅对枯燥的标准文本给出了浅显易懂的解释，还提供了很多示例和练习来帮助我们理解。相信每个严肃的C++程序员都可以像我一样从本书中获益良多。

在本书的翻译过程中，给我帮助最大的莫过于我的好朋友李敏。在近1年的时间里，她逐字逐句地阅读了每一个章节并且提出了很多宝贵的意见，每当在技术和表达上遇到困难时，她也总是我默认的讨论对象。没有她，本书的翻译也就不可能及时完成。在此，我想对她说声谢谢。

我还想感谢本书的原作者Pete Becker和机械工业出版社的陈冀康编辑，是他们让我能有机会接触并翻译这样一本好书。

在翻译本书的过程中，我还得到了很多人的无私帮助，他们是陈恩惠、陈冀康、董平、康飞、陆海根、陆如江、陆如娟、陆如琴、陆如珍、史友良、史宗霖、邵玉玲、王雅梦、谢之易、徐金方、徐云峰、Christopherc Zimmerman和Nick Londey。感谢他们在本书翻译过程中给予我的各种帮助。

最后，我想感谢我父母多年的养育，也感谢他们为我所做的一切！

史晓明

2008年2月22日

前　言

不少程序员都曾为C++标准库的某些限制感到灰心丧气，本书就是为这些人而写的。国际标准化组织[⊖]（International Organization for Standardization，ISO）于2006年通过了第一份C++库技术报告（Technical Report，TR）。对于由这份报告所描述的库来说，本书同时起到了简介和参考手册的作用。本书所讨论的库（我称之为TR1库）并不是C++标准的一部分。TR1只不过是一份“信息性文档”，但是我们可以预期TR1库将会随着某些编译器一起发布，或是由各个库生产商以附加库的形式提供。我们也可以预期这个库中的很多部分都会被加入到下一个C++标准[⊜]的标准库中。

TR1库对C++标准库进行的扩展主要表现为在下列领域中所加入的新功能：

- **基本工具**（utility）：使用引用计数的智能指针；类模板tuple，它是std::pair类模板的一般化实现，可以对不同个数的参数进行处理。
- **容器**：使用散列表实现的set和map；固定大小的数组实现array。
- **调用包装器**（call wrapper）：一系列更为强大和灵活的模板。可以使用这些模板对函数、成员函数以及其他函数对象进行包装，从而使我们能够更方便地把它们作为参数传递给算法。
- **类型特性**（type trait）：一系列可以提取类型属性或是在编译时修改类型的模板，它们有助于对模板元编程进行简化。
- **数值**（numeric）：各种各样的随机数生成器；高等数学特殊函数；数值功能（与C语言在1999年所加入的相类似）。
- **正则表达式**（regular expression）：一系列类和函数，可以用它们来描述和查找文本中的模式。
- **C语言兼容**：一系列类型、函数和宏（与C语言在1995年和1999年所加入的相类似）。

TR1库的正式工作始于2001年。当时，C++标准委员会（C++ Standards Committee）通过其库工作组（Library Working Group）征求提案。最后纳入库中的大多数提案都是由Boost组织[⊝]的成员提出的。一部分标准委员会的成员一直在试图寻找一种不受标准化过程约束的方式来为C++开发新库，他们在1998年建立了Boost组织。在正规过程外部进行工作会更为灵活，因而Boost库的各个部分能够以更为迅速而独立的进度进行开发，而这在标准委员会内部是不可能的。Boost是一个稳步发展的组织，其所提供的库中还有很多有用的功能尚未包含在TR1库中。

标准委员会每年举行两次会议。在2001年～2005年间，库工作组不仅把他们在这些会议上的绝大多数时间都花在和TR1库相关的工作上，还在不召开会议时使用E-mail进行了大量的联系。在这段时间里，库工作组对收到的提案在以下方面进行了改进：尽可能地加以简化；对提案进行重写使得它们不仅更为明晰，还能符合ISO标准所需要的正规形式；统一库中各个部分的表达

⊖ 请参见www.iso.ch

⊜ 在2006年4月，TR1库中除了特殊数学函数以外的所有部分都已加入到下一个C++标准的草案中。

⊝ 请参见www.boost.org

方式。

到了2002年末，我所工作的公司Dinkumware, Ltd.开始实现TR1库。Dinkumware销售C、C++和Java的标准库，因此实现TR1库对于我们来说是很自然的举动。我们的工作也对技术报告的改进有所帮助，因为我们发现一些部分的描述不够清晰或是过于详细，有时甚至被遗漏了[一]。Dinkumware已经完整地实现了TR1库，在本书的所有例子里，我都会使用这一实现。

关于本书

本书分为七个部分，每个部分都会对某个特定领域中的新功能进行详细的讨论。它们都会先对所包含的功能进行简要的介绍。这通常会包含一些关于其历史的简短讨论，有时还会为TR1库为什么没有包含某些明显的功能给出理由。

每个部分都会有一个头文件摘要，正是这些头文件定义了这个部分中所讨论的功能。头文件摘要通常并不是可编译的代码，它只是为这个头文件所定义的组件提供了一个概览，列出了这个头文件中定义的所有模板、类、非成员函数、对象、常量等。随后我们会对它们进行更为详细的定义。然而，即使是后者通常也不是可编译的代码。很多细节都是实现相关的，在这些细节上花费太多精力并不会为我们带来任何收获。

所有摘要都使用灰色表示。我们在大部分摘要后面附上了C++标准的正式要求。这些正式要求将使用缩进方式表示。我们不会在那些非正式的讨论上使用缩进。譬如：

```
class C
{
};
```

这段文本给出了类C的正式要求。通常这不仅难以理解而且非常技术化。虽然其措辞非常明确，但是往往很难读懂。非正式的文本没有缩进。它提供了一个更容易理解的描述，虽然有时不是很精确。

我们在大部分章节的最后提供了一系列的练习，可以用来回顾章节中所介绍的概念。有些练习很简单，而有些则有意地设计得很难。因此，即使读者无法做出所有的练习也无需担心。

为了避免混淆，我通常使用规范化语言来指定正在讨论的对象。由于讨论的参与者并不知道他们正在讨论的究竟是模板、类还是对象而导致讨论陷入困境是很常见的。譬如，TR1库中有一个名为function的模板，它可以包含函数对象。为了不让读者猜测function这个单词究竟指的是这个模板还是使用这个模板创建的对象，我始终采用“类模板function”、“类模板function的特化”以及“function<T>类型的对象”之类的短语。

所有的代码示例都是完整的：只需要合适的编译器和库就可以对它们进行编译。不仅如此，具有main函数的例子还能够连接和运行。我们在Microsoft C/C++编译器[二]7.1版和GNU项目的gcc编译器[三]3.4.3版对这些示例进行了测试。在测试中我们使用了Dinkumware, Ltd.[四]提供的

[一] 这并不是在批评Boost成员所作的工作；他们先为所提交的库编写文档，然后再把这些文档改编为提案的规范草案。第二步非常困难，因而很少能够顺利地进行。

[二] www.microsoft.com

[三] www.gnu.org

[四] www.dinkumware.com

Dinkum TR1库1.0版。读者可以从我的网站[㊀]下载这些示例的源码。

ISO、符合标准和TR1库

ISO C++标准是C++程序设计语言的规范，它不仅对一个C++程序是否正确进行了定义，还定义了在特定条件下这一程序的行为。TR1库并不是C++标准的一部分，虽然我们几乎可以肯定这个库中的绝大部分都会成为C++标准下一个修订版本（大约在2010年）的一部分。在此之前，符合C++标准的编译器并不需要包含TR1库[㊁]。

C++标准在对一个程序是否正确以及一个正确的程序意味着什么进行讨论时定义和使用了几个技术术语。库技术报告也使用了同样的术语。程序员经常会混淆这些术语的含义[㊂]，虽然它们并不很复杂。如果读者对此感兴趣，就请继续读下去吧。

诊断信息是指由于违反了标准所规定的规则而产生的任何编译器[㊃]输出信息。编译器也可以给出其他信息，但是标准要求编译器在其文档中指出哪些信息是诊断信息。

当C++标准说某些代码会导致未定义行为时，意味着当编译器在编译一个包含这类代码的程序时，标准不对其作出任何要求。不幸的是，具有未定义行为的代码所生成的程序其行为通常都和我们预期的完全一样。这会让调试变得非常困难，因此最好避免写出具有未定义行为的代码。

如果标准中的任意规则并没有说“不要求诊断信息”，并且也不会导致任何未定义行为，它就是一条诊断规则。如果我们的代码违反了一个不需要任何诊断信息的规则，C++标准允许编译器作任何决定；因为我们编写了一个C++标准无法辨别的程序，我们只能自己去承担这一后果了。如果我们的代码违反了一条诊断规则，编译器必须给出至少一条诊断信息[㊄]。这并不意味着编译器必须报告每一个必须给出诊断信息的错误，也不意味着如果编译器所编译的程序不包含任何必须给出诊断信息的错误，它就不能给出诊断信息。从技术上说，一个在每次编译代码时都给出“发生错误”的编译器符合C++标准。当然，只要能够避免，没有人会使用这样的编译器。

唯一定义规则要求应该相同的定义实际上也必须相同。譬如，如果我们在一个源文件中定义了一个名为data的结构，它包含两个int类型的成员，并且在另一个源文件中也定义了一个名为data的结构，它包含3个double类型的成员，那么我们就违反了唯一定义规则。这类违规行为并不需要给出一个诊断信息。它们会导致非常奇怪的问题。

㊀ www.petebecker.com/tr1book

㊁ 从技术角度来说，它们不应该包含TR1库，因为TR1库所加入的名字可能会和一些已经在正确的程序中使用的名字产生冲突。实际上这并不会成为一个显著的问题。TR1库把它的名字都放在tr1命名空间中，而tr1则被嵌套包含在std命名空间中。因此，using namespace std;这一语句会把tr1这个名字提升到全局命名空间中，如果现存的代码也使用了这个名字，就可能会产生冲突。不仅如此，如果程序代码中所定义的宏和TR1库中的任意内容相同也会导致问题。当然，由于我们都使用大写字母来为宏命名，因此这应该不会引起任何问题。

㊂ 事实上，很多最后进入TR1库的原始提案并没有正确地使用这些术语，虽然库工作组试图修正这些用法错误，但还是有可能在技术报告中遗留了一部分错误。

㊃ C++标准使用“实现”这一术语来统称编译器、连接器以及在把源码转换为可执行程序时所需要的任何其他工具。我将使用“编译器”这一传统简称来表示同样的意思。

㊄ 读者可能经常听说编译器应该拒绝编译那些违反了诊断规则的代码，这并不正确。标准对此唯一的要求是编译器必须给出一个诊断信息。这是对语言进行扩展的钩子：一旦编译器给出了一个诊断信息，它就可以做任何编译器编写者觉得合适的事情了。

一个正确的程序就是指一个既没有违反任何诊断规则也没有违反唯一定义规则的程序。而如果一个程序不是正确的，它就是一个错误的程序。

编译器所运行的系统具有有限的存储空间，因此它们会受到资源的限制，这使得编译器不能编译过于复杂的程序。C++标准为这些限制的最小值提出了一些建议，例如，复合语句的最多嵌套层次（256），switch语句中最多的case标签个数（16 384），模板实例化的最多嵌套层次（17）。标准要求编译器在文档中列出任何已知的限制。然而，现在的编译器已经不再使用固定大小的表了，而是在需要时动态地为其内部数据结构分配内存；这意味着这些限制随着这个程序其他部分功能的改变而改变。这导致我们不能认为一个正确的程序如果能够通过一个编译器的编译，它也能通过另一个编译器的编译，这也包括“同一编译器”的两个不同版本。当我们改变编译器时，我们的程序可能会面对更为有限的资源。

如果一个编译器能够正确地编译不超过其资源限制的正确程序，它就是一个符合C++标准的实现。当然，编译得到的可执行文件在行为上必须与C++标准的要求一致。然而，这并不像看上去那么明确；很多情况下，标准允许从同样的源代码编译出的可执行文件具有不同的行为。

有时C++标准会提到某段代码具有未指定行为。通常，这意味着在这种情况下有不少合理的做法并且C++标准并不要求我们必须遵循它们中的某一个。譬如，计算函数参数的顺序就是未指定的。这意味着对于f(g(),h())这样的代码来说，编译器可以决定是在调用函数h之前调用函数g还是在调用g之前调用h。如果我们的代码假定g会在h之前调用，那么当我们更换编译器或是改变正在使用的编译器的优化设置时，就可能会大吃一惊。

C++标准还指出某些代码具有由实现定义的行为。正如具有未指定行为的代码一样，这时通常也有几个合理的选择，并且标准并没有要求我们必须遵循它们中的某一个。然而，标准要求实现在文档中说明其行为。譬如，基本类型char既可以是signed的也可以是unsigned的，编译器必须在文档中告诉我们究竟是哪一个。

最后，我们应该从标准中的规范文本中获得对语法和语义的要求。脚注中的文本、示例、在“[Note:...-- end note]”之间的文本以及章节的标题都不是规范文本；它们有助于我们阅读C++标准，但是本身并不带有任何要求。例如，如果一个脚注和一段普通文本之间有冲突，那么我们应该以普通文本为准。

致谢

我想感谢库工作组所有成员对TR1库所作的贡献，并向Matt Austern致以特殊的谢意，作为库工作组主席，他不仅让我们能把全部精力都投入到工作中，还为我们之间的（有时会变得非常激烈的）讨论进行仲裁。我也要向那些为TR1库撰写了开创性论文（seminal paper）的人们致以特殊的谢意，他们是Matt Austern（无序容器，unordered containers）、Walter Brown（数学特殊函数）、Greg Colvin（shared_ptr）、Beman Dawes（shared_ptr）、Peter Dimov（reference_wrapper、shared_ptr、mem_fn和bind）、Doug Gregor（reference_wrapper、function和bind）、Jaakko Järvi（tuple和bind）、John Maddock（类型属性和正则表达式）、Jens Maurer（随机数发生器）、Alisdair Meredith（array）、P.J. Plauger（C语言兼容）和Gary Powell（bind）。

我还想感谢Bjarne Stroustrup，是他发明了这一奇妙的语言并把我引荐给Addison-Wesley出版社。我也要感谢Addison-Wesley出版社中每一位为本书的出版作出贡献的人，他们是Peter

Gordon、Kim Boedigheimer、Elizabeth Ryan、Evelyn Pyle、John Fuller、Chanda Leary-Coutu和Marie McKinley。

感谢Doug Gregor、John Maddock、Matt Austern和Marc Paterno审阅了本书的草稿。他们提出的每条建议都非常深刻，对我很有帮助。

感谢P.J. Plauger和Tana Plauger（Dinkumware, Ltd.的经营者），是他们鼓励我并给了我足够的时间编写本书。

最后我想感谢我的妻子Angela Pao，她为我所做的事难以计数，例如不时地提醒我休息一下。

进一步阅读

- *ISO/IEC Standard 14882:2003, Programming Languages—C++* [Int03a]。这是C++语言和C++标准库最权威的技术参考资料。
- *ISO/IEC Standard 9899:1990, Programming Languages—C* [Int90]。这是在我们编写当前C++标准时C语言和C标准库最权威的技术参考资料。C++语言和C++标准库中有很大一部分是对这份文档的引用。
- *Amendment 1 to ISO/IEC Standard 9899:1990, Programming Languages—C* [Int95]。它为C语言对国际化的支持提供了大量的改进。
- *ISO/IEC Standard 9899:1999, Programming Languages C*—[Int99]。这是C语言和C标准库最权威的技术参考资料，它同时在语言和标准库这两方面为浮点数学的支持提供了更为严格的定义。TR1库包含了C99对C语言标准库的所有修改。
- *The C++ Standard Library* [Jos99]。这是一本C++标准库教程和参考手册，不仅内容全面而且通俗易懂。

目　录

译者序
前言

第一部分　基本工具

第1章　元组 ……1
1.1　头文件<tuple>的摘要 ……2
1.2　tuple类模板 ……2
1.3　像tuple一样对std::pair进行存取 ……10
1.4　练习 ……12
第2章　智能指针 ……16
2.1　定义 ……16
2.2　关于示例 ……17
2.3　头文件<memory>的摘要 ……18
2.4　shared_ptr类模板 ……19
2.5　类模板weak_ptr ……32
2.6　类模板enable_shared_from_this ……41
2.7　bad_weak_ptr类 ……43
2.8　转换 ……43
2.9　受控资源的析构 ……48
2.10　异常 ……50
2.11　多线程 ……52
2.12　练习 ……52

第二部分　容　器

第3章　基础知识 ……59
3.1　STL部件 ……59
3.2　容器 ……63
3.3　进一步阅读 ……66
3.4　练习 ……66
第4章　类模板array ……68
4.1　类模板array概述 ……70
4.2　信息获取 ……73
4.3　访问 ……73
4.4　修改 ……75
4.5　迭代 ……76
4.6　内嵌类型名称 ……77
4.7　比较 ……79
4.8　与tuple类似的接口 ……80
4.9　练习 ……80
第5章　无序关系容器 ……82
5.1　对散列表进行标准化 ……82
5.2　散列表 ……83
5.3　关系容器和无序容器 ……85
5.4　对无序容器的要求 ……85
5.5　头文件<unordered_map>和<unordered_set> ……90
5.6　类模板hash ……91
5.7　无序容器的实例化 ……93
5.8　构造函数 ……94
5.9　容器操作 ……94
5.10　负载因子和重散列 ……97
5.11　优化 ……98
5.12　进一步阅读 ……99
5.13　练习 ……99

第三部分　调用包装器

第6章　调用包装器基础 ……101
6.1　术语 ……101
6.2　对可调用类型的要求 ……103
6.3　头文件<functional>的摘要 ……105
6.4　类模板result_of ……106
6.5　与现存的函数对象进行交互 ……108
6.6　练习 ……111
第7章　mem_fn函数模板 ……114
第8章　reference_wrapper类模板 ……120
8.1　创建 ……121

8.2 内嵌类型 ……123
8.3 调用 ……124
8.4 练习 ……125
第9章 类模板function ……127
9.1 构造一个function对象 ……128
9.2 访问 ……130
9.3 修改 ……130
9.4 比较 ……132
9.5 内嵌类型 ……133
9.6 调用 ……133
9.7 目标对象 ……134
9.8 练习 ……135
第10章 函数模板bind ……137
10.1 占位符 ……138
10.2 unspecified bind(...) ……139
10.3 对bind进行扩展 ……146
10.4 练习 ……146

第四部分 类型特性

第11章 类型特性概述 ……149
11.1 头文件<type_traits> ……151
11.2 辅助类型 ……153
11.3 基本类型 ……154
11.4 复合类型 ……156
11.5 类型属性 ……156
11.6 类型关系 ……162
11.7 类型变换 ……163
11.8 对齐 ……165
11.9 进一步阅读 ……166
11.10 练习 ……166

第五部分 数 值

第12章 数值函数 ……169
12.1 关于示例 ……169
12.2 浮点值的表示 ……170
12.3 管理浮点环境 ……173
12.4 无穷大、非规范数、NaN和比较 ……179
12.5 定义域和值域错误 ……180
12.6 新的重载规则 ……180
12.7 基本数学函数 ……181
12.8 数学特殊函数 ……194
12.9 复函数 ……203
12.10 进一步阅读 ……208
12.11 练习 ……208
第13章 随机数生成器 ……213
13.1 随机数引擎 ……215
13.2 TR1中的引擎模板 ……218
13.3 TR1库中的random_device类 ……224
13.4 TR1库中的预定义引擎 ……225
13.5 随机数分布 ……226
13.6 离散分布 ……228
13.7 连续分布 ……231
13.8 类模板variate_generator ……234
13.9 进一步阅读 ……238
13.10 练习 ……239

第六部分 正则表达式

第14章 头文件<regex> ……241
第15章 正则表达式语法 ……245
15.1 正则表达式结构 ……246
15.2 语法特性 ……250
15.3 正则表达式细节 ……250
15.4 关于练习 ……256
15.5 练习 ……257
第16章 正则表达式对象 ……259
16.1 定义 ……259
16.2 头文件<regex>的部分摘要 ……261
16.3 语法选项 ……261
16.4 basic_regex类模板 ……265
16.5 预定义的basic_regex类型 ……272
16.6 错误处理 ……272
16.7 练习 ……274
第17章 查找 ……276
17.1 头文件<regex>的部分摘要 ……278
17.2 完整匹配 ……279
17.3 查找 ……280

17.4 查找选项 ……281
17.5 练习 ……286
第18章 查找结果 ……289
18.1 头文件<regex>的部分摘要 ……290
18.2 sub_match类模板 ……291
18.3 预定义的sub_match类型 ……299
18.4 类模板match_results ……300
18.5 练习 ……307
第19章 重复查找 ……309
19.1 暴力查找 ……311
19.2 regex_iterator类模板 ……317
19.3 regex_token_iterator类模板 ……326
19.4 练习 ……335
第20章 格式化和文本替换 ……337
20.1 格式化选项 ……340
20.2 格式化文本 ……341
20.3 文本替换 ……342
20.4 练习 ……345
第21章 正则表达式的定制 ……347
21.1 字符特性 ……348
21.2 locale ……348
21.3 字符匹配 ……348
21.4 归并 ……349
21.5 字符类别 ……350
21.6 regex_traits类模板 ……351

第七部分 C语言兼容

第22章 C语言兼容 ……353
22.1 整数类型 ……353
22.2 64位整数类型 ……354
22.3 固定大小的整数类型 ……357
22.4 文本转换 ……362
22.5 格式化说明符 ……363
22.6 格式化I/O ……368
22.7 字符分类 ……370
22.8 布尔类型 ……371
22.9 练习 ……371

附 录

附录A 头文件 ……373
附录B 辅助头文件 ……402
附录C 多线程 ……407
参考文献 ……410

第一部分 基本工具

第1章 元 组

类模板tuple（元组）是C++标准库中类模板pair的一般化（generalization）实现。模板pair接受两个参数；类型pair<T_1, T_2>的对象都具有两个成员：一个是T_1类型的，另一个则是T_2类型的。模板tuple对pair的扩展表现在于它可以接受任何数量的类型参数。类型tuple<T_1, T_2, …, T_N>的对象具有N个成员，其类型分别为T_1，T_2，…、T_N[⊖]。

示例1.1 tuple的声明（tuples/typedefs.cpp）

```
#include <tuple>
using std::tr1::tuple;

tuple<> none;                              // holds no elements
tuple<int> one;                            // holds one element, of type int

int i; double d;
tuple<int&, const double&>
    two(i, d);                             // holds two elements, one of
                                           // type reference to int and one
                                           // of type reference to const double
tuple<int, int, int, int, int,
  int, int, int, int, int> ten;            // holds ten elements, all of type int
```

我们可以通过类型tuple的任何一个构造函数来创建tuple对象。不仅如此，make_tuple和tie这两个函数模板也会返回tuple对象，这个对象的类型取决于传递给这些函数的参数。我们会在第1.2.1小节中对这些创建tuple对象的技术进行讨论。

函数模板get返回对tuple对象中某个成员的引用。我们既可以使用其返回值来访问这个成员的值，也可以用它来修改一个非常量（non-const）tuple对象的成员。我们也可以通过把一个tuple对象赋值给另一个tuple对象来改变后者成员的值。我们将在第1.2.2小节中对这些访问机制（access mechanism）进行介绍。

可以使用类模板tuple_size和tuple_element来获得某个tuple特化（specialization）的大小以及每个成员的类型。我们会在第1.2.3小节中介绍这些模板。

如果两个tuple对象具有相同数量的成员，我们就可以通过对它们进行比较来判断它们是否相等或者得到它们的相对次序（relative order）。请参见第1.2.4小节。

⊖ 每个库实现都可以为N的值规定一个上限。TR1库建议这个上限至少为10。

1.1 头文件<tuple>的摘要

```
namespace std {
 namespace tr1 {
    // TUPLE CLASS TEMPLATES
template<class T1, class T2, ..., class TN>
  class tuple;
template<class Tuple> struct tuple_size;
template<int Idx, class Tuple> struct tuple_element;

    // TUPLE FUNCTION TEMPLATES
template<int Idx, class T1, class T2, ..., class TN>
  RI get(tuple<T1, T2, ..., TN>&);
template<int Idx, class T1, class T2, ..., class TN>
  RI get(const tuple<T1, T2, ..., TN>&);
template<class T1, class T2, ..., class TN>
  tuple<V1, V2, ..., VN>
  make_tuple(const T1&, const T2&, ..., const TN&);
template<class T1, class T2, ..., class TN>
  tuple<T1&, T2&, ..., TN&>
  tie(T1&, T2&, ..., TN&);

    // CONSTANTS
const unspecified ignore;

    // COMPARISON OPERATORS
template<class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
  bool operator==(const tuple<T1, T2, ..., TN>&,
    const tuple<U1, U2, ..., UN>&);
template<class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
  bool operator!=(const tuple<T1, T2, ..., TN>&,
    const tuple<U1, U2, ..., UN>&);
template<class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
  bool operator<(const tuple<T1, T2, ..., TN>&,
    const tuple<U1, U2, ..., UN>&);
template<class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
  bool operator<=(const tuple<T1, T2, ..., TN>&,
    const tuple<U1, U2, ..., UN>&);
template<class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
  bool operator>(const tuple<T1, T2, ..., TN>&,
    const tuple<U1, U2, ..., UN>&);
template<class T1, class T2, ..., class TN,
    class U1, class U2, ..., class UN>
  bool operator>=(const tuple<T1, T2, ..., TN>&,
    const tuple<U1, U2, ..., UN>&);
} }
```

1.2 tuple类模板

```
template<class T1, class T2, ..., class TN>
```

```
class tuple {
public:
  // CONSTRUCTORS
  tuple();
  explicit tuple(P1, P2, ..., PN);              // when N > 0
  tuple(const tuple&);
  template <class U1, class U2, ..., class UN>
    tuple(const tuple<U1, U2, ..., UN>&);
  template <class U1, class U2>
    tuple(const pair<U1, U2>&);                 // when N == 2

  // ASSIGNMENT OPERATORS
  tuple& operator=(const tuple&);
  template <class U1, class U2, ..., class UN>
    tuple& operator=(const tuple<U1, U2, ..., UN>&);
  template <class U1, class U2>
    tuple& operator=(const pair<U1, U2>&);      // when N == 2
  };
```

1.2.1 创建tuple对象

我们可以使用多种方法来创建tuple对象。如果所有成员类型都具有默认的构造函数，那么包含这些成员类型的tuple也具有默认构造函数，它所创建的tuple对象中所有的成员都是使用它们各自的默认构造函数来构造的。不仅如此，每个具有一个或多个成员的tuple类型还具有一个构造函数，其参数个数和这个tuple类型的成员个数相同，这个构造函数用相应的参数来构造tuple所包含的每一个成员。很明显，每个参数的类型都必须能够被转换为（convertible）与之对应的成员类型。最后，我们还可以通过拷贝另一个具有相同成员个数的tuple对象来构造一个tuple对象；当所构造的tuple对象正好包含两个成员时，我们还可以通过拷贝一个pair对象来构造它。

示例1.2 tuple构造函数（tuples/ctors.cpp）

```
#include <utility>
#include <tuple>
using std::tr1::tuple; using std::pair;

class C {
public:
  C() : val(0) {}
  C(int i) : val(i) {}
private:
  int val;
};

tuple<> t0;                     // default constructor
tuple<int> t1;                  // default constructor; element
                                // not initialized
tuple<int> t2(3);               // element initialized to 3
tuple<C> t3;                    // element initialized to C()
tuple<C> t4(C(1));              // element initialized to C(1)
tuple<C, C> t5(1, 2);           // first element initialized to C(1)
                                // second element initialized to C(2)
tuple<double> t6(t2);           // element initialized to 3.0
```

```
pair<int, int> p0(3, 4);        // first element initialized to 3
                                // second element initialized to 4
tuple<C, C> t7(p0);             // first element initialized to C(3)
                                // second element initialized to C(4)
```

有时，列出tuple的所有类型参数并不方便。当我们知道希望tuple对象所包含的值时，可以使用函数make_tuple来创建一个tuple对象，这个对象包含了make_tuple函数所有参数的拷贝。

如果读者运行了这个示例并且成功地读懂了编译器为这些tuple类型生成的冗长名字，很可能会注意到虽然我们传递给make_tuple的第二个参数的类型为对int的引用，但是对make_tuple的最后一次调用所返回的对象类型是tuple<int, int>。这是因为函数模板make_tuple并不会区分对象和对象的引用，两者都会得到和对象相同类型的成员。

示例1.3 函数模板maketuple（tuples/maketuple.cpp）

```
#include <tuple>
#include <typeinfo>
#include <iostream>
using std::tr1::tuple; using std::tr1::make_tuple;

template <class T> void show_type(T)
    {
    std::cout << typeid(T).name() << "\n\n";
    }

int main()
    {
    int i = 3;
    int& j = i;
    show_type(make_tuple());          // returns tuple<>
    show_type(make_tuple(1, 3.14));   // returns tuple<int,double>
    show_type(make_tuple(i, j));      // returns tuple<int,int>
    return 0;
    }
```

我们可以使用TR1库中的函数模板ref和cref来创建包含引用的tuple对象，它们在头文件<functional>中定义。我们将会在后面对这些函数的其他用途进行讨论（请参见第8.1节）。目前，读者只需要知道ref是一个包装器（wrapper），它可以告诉make_tuple相应的成员类型是ref函数参数类型的引用。同样地，cref告诉make_tuple所要创建的成员是一个对常量类型的引用。

示例1.4 使用ref和cref（tuples/refcref.cpp）

```
#include <tuple>
#include <functional>              // for ref, cref
using std::tr1::make_tuple;
using std::tr1::ref; using std::tr1::cref;

void test()
    {
    int i = 17;
    int j = 3;
```

```
    make_tuple(ref(i),cref(j)); // returns tuple<int&, const int&>
                               // first element is reference to i
                               // second element is reference to j
    }
```

有时我们想要创建一个tuple对象，使其仅仅包含对其他对象的引用。正如刚刚所看到的，我们可以使用ref来实现这一点，但是这会导致大量的输入。使用函数模板tie是一种比较方便的方法，它所创建的tuple对象包含了对其参数的引用。把值ignore作为参数传递给tie就可以让tie返回的tuple对象在赋值时忽略与这个参数相对应的元素。

示例1.5 函数模板tie（tuples/tie.cpp）

```
#include <tuple>
#include <iostream>
using std::tr1::make_tuple; using std::tr1::tie;
using std::tr1::ignore;

int i = 1;
int j = 2;
int k = 3;

void show()
  {
  std::cout << i << ' ' << j << ' ' << k << '\n';
  }

int main()
  {
  show();                          // 1 2 3
  tie(i, ignore, k) =
    make_tuple(5, 6, 7);
  show();                          // 5 2 7
  return 0;
  }
```

1.2.2 存取

把一个tuple对象赋值给另一个tuple对象可改变后者所包含的值。这两个对象不仅必须包含相同个数的元素，而且源对象中每个元素的类型都必须能够被转换为目标对象中对应元素的类型。

我们还可以通过把一个pair对象赋值给一个包含了两个元素的tuple对象来改变后者所包含的值。

示例1.6 为tuple对象赋值（tuples/assign.cpp）

```
#include <utility>
#include <iostream>
#include <tuple>
using std::tr1::tuple; using std::tr1::get;
using std::cout; using std::make_pair;

void show(int i, int j, const tuple<int,int&,int>& t)
```