

Boost

程序库完全开发指南

——深入C++ “准”标准库

罗剑锋 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

Boost 是一个功能强大、构造精巧、跨平台、开源并且完全免费的 C++ 程序库，有着“C++ ‘准’标准库”的美誉。

它由 C++ 标准委员会部分成员所设立的 Boost 社区开发并维护，使用了许多现代 C++ 编程技术，内容涵盖字符串处理、正则表达式、容器与数据结构、并发编程、函数式编程、泛型编程、设计模式实现等许多领域，极大地丰富了 C++ 的功能和表现力，能够使 C++ 软件开发更加简洁、优雅、灵活和高效。

本书基于 Boost1.42 版，介绍了其中的所有 99 个库，并且详细深入地讲解了其中数十个库，同时实现了若干颇具实用价值的工具类和函数，可帮助读者迅速理解掌握 Boost 的用法以及应用于实际的开发工作中。

本书内容丰富、结构严谨、详略得当、讲解透彻，带领读者领略了 C++ 的最新前沿技术，相信会是每位 C++ 程序员的必备工具书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目 (CIP) 数据

Boost 程序库完全开发指南：深入 C++ “准”标准库 / 罗剑锋著. —北京：电子工业出版社，2010.9
ISBN 978-7-121-11577-6

I. ①B... II. ①罗... III. ①C 语言—程序设计—指南 IV. ①TP312-62

中国版本图书馆 CIP 数据核字 (2010) 第 157435 号

责任编辑：郭 立

特约编辑：顾慧芳

印 刷：北京市天竺颖华印刷厂

装 订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：37.5 字数：801 千字

印 次：2010 年 9 月第 1 次印刷

印 数：4000 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

推荐序

最近一年我电话面试了数十位 C++ 应聘者，惯用的暖场问题是“工作中使用过 STL 的哪些组件？使用过 Boost 的哪些组件？”。得到的答案大多集中在 `vector`、`map` 和 `shared_ptr`。如果对方是在校学生，我一般会问问 `vector` 或 `map` 的内部实现、各种操作的复杂度以及迭代器失效的可能场景。如果是有经验的程序员，我还会追问 `shared_ptr` 的线程安全性、循环引用的后果及如何避免、`weak_ptr` 的作用等。如果这些都回答得不错，进一步还可以问问如何实现线程安全的引用计数，如何定制删除动作等等。这些问题让我能迅速辨别对方的 C++ 水平。

我之所以在面试时问到 Boost，是因为其中的许多组件确实可以用于编写可维护的产品代码。Boost 包含近百个程序库，其中不乏具有工程实用价值的佳品。每个人口味与技术背景不一样，对 Boost 的取舍也不一样。就我的个人经验而言，首先可以使用绝对无害的库，例如 `noncopyable`、`scoped_ptr`、`static_assert` 等，这些库的学习和使用都比较简单，容易入手。其次，有些功能自己实现起来并不困难，正好 Boost 里提供了现成的代码，那就不妨一用，比如 `date_time` 和 `circular_buffer` 等。然后，在新项目中，对于消息传递和资源管理可以考虑采用更加现代的方式，例如用 `function/bind` 在某些情况下代替虚函数作为库的回调接口、借助 `shared_ptr` 实现线程安全的对象回调等等。这二者会影响整个程序的设计思路与风格，需要通盘考虑，如果正确使用智能指针，在现代 C++ 程序里一般不需要出现 `delete` 语句。最后，对某些性能不佳的库保持警惕，比如 `lexical_cast`。总之，在项目组成员人人都能理解并运用的基础上，适当引入现成的 Boost 组件，以减少重复劳动，提高生产力。

Boost 是一个宝库，其中既有可以直接拿来用的代码，也有值得借鉴的设计思路。试举一例：正则表达式库 `regex` 对线程安全的处理。

早期的 `Regex` 类不是线程安全的，它把“正则表达式”和“匹配动作”放到了一个类里边。

由于有可变数据，Regex 的对象不能跨线程使用。如今的 regex 明确地区分了不可变 (immutable) 与可变 (mutable) 的数据，前者可以安全地跨线程共享，后者则不行。比如正则表达式本身 (basic_regex) 与一次匹配的结果 (match_results) 是不可变的；而匹配动作本身 (match_regex) 涉及状态更新，是可变的，于是用可重入的函数将其封装起来，不让这些数据泄露给别的线程。正是由于做了这样合理的区分，regex 在正常使用时就不必加锁。

Donald Knuth 在 “*Coders at Work*” 一书里表达了这样一个观点：如果程序员的工作就是摆弄参数去调用现成的库，而不知道这些库是如何实现的，那么这份职业就没啥乐趣可言。换句话说，固然我们强调工作中不要重新发明轮子，但是作为一个合格的程序员，应该具备自制轮子的能力。非不能也，是不为也。

C/C++ 语言的一大特点是其标准库可以用语言自身实现。C 标准库的 strlen、strcpy、strcmp 系列函数是教学与练习的好题材，C++ 标准库的 complex、string、vector 则是类、资源管理、模板编程的绝佳示范。在深入了解 STL 的实现之后，运用 STL 自然手到擒来，并能自动避免一些错误和低效的用法。

对于 Boost 也是如此，为了消除使用时的疑虑，为了用得更顺手，有时我们需要适当了解其内部实现，甚至编写简化版用作对比验证。但是由于 Boost 代码用到了日常应用程序开发中不常见的高级语法和技巧，并且为了跨多个平台和编译器而大量使用了预处理宏，阅读 Boost 源码并不轻松惬意，需要下一番功夫。另一方面，如果沉迷于这些有趣的底层细节而忘了原本要解决什么问题，恐怕就舍本逐末了。

Boost 中的很多库是按泛型编程的范式来设计的，对于熟悉面向对象编程的人而言，或许面临一个思路的转变。比如，你得熟悉泛型编程的那套术语，如 concept、model、refinement，才容易读懂 Boost.Threads 的文档中关于各种锁的描述。我想，对于熟悉 STL 设计理念的人而言，这不是什么大问题。

在某些领域，Boost 不是唯一的选择，也不一定是最好的选择。比如，要生成公式化的源代码，我会首选用脚本语言写一小段代码生成程序，而不用 Boost.Preprocessor；要在 C++ 程序中嵌入领域特定语言，我会首选用 Lua 或其他语言解释器，而不用 Boost.Proto；要用 C++ 程序解析上下文无关文法，我会首选用 ANTLR 来定义词法与语法规则并生成解析器 (parser)，而不用 Boost.Spirit。总之，使用 Boost 时心态要平和，别较劲去改造 C++ 语言。把它有助于提高生产力的那部分功能充分发挥出来，让项目从中受益才是关键。

要学习 Boost，除了阅读其官方网站的文档、示例与源码之外，最好能有一本比较全面的中文书在手边随时翻阅。对于不谙英文的开发者，这更是可幸之至。您手上这本《Boost 程序库完全开发指南》是很好的使用指南与参考手册。作者由浅入深地介绍了 Boost 的大部分常用内容，能让读者迅速了解 Boost，并从中找到自己需要的部分。拿到这本书稿之后，我有粗有细地阅读

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真题解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总: **ASP.NET** 篇

.Net 技术精品资料下载汇总: **C#**语言篇

.Net 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子书下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

了一遍，总体来看，作者水平很高，也相当务实，对 C++ 和 Boost 的理解与运用很到位，我从这本书学到了不少新知识。为此，我乐于向希望学习 Boost 程序库的开发者推荐这本靠谱的书。

须知“功不唐捐”，作为一名现代 C++ 程序员，在 Boost 上投入的精力定能获得回报。

陈硕

《代码大全》译者之一

2010 年 8 月于中国·香港

前言

屈指算来，接触 C++ 语言至今已经有十余个年头了。回首往事，不禁感慨良多。

缘起

1996 年我上大学最开始学的是 Pascal，不得不说，Pascal 严谨的程序风格确实很适合作为一门教学语言，然而用于实际开发就不那么合适了（直到出现 Delphi）。由于当时学校并未开设 C 语言课程，因此在 Pascal 课程结束后我就买书自学 C/C++ 语言，并在次年报名国家计算机水平考试，靠着一点点编程和考试的“天分”获得了高级程序员资质（当年很热衷考级考证，后来就“淡定”多了）。虽然有了资格证，但我仍然算是个 C++ 的初学者，对于 C++ 的认识还处于 C 的面向过程和简单的基于对象层次上。

新千年伊始我考入了北京理工大学就读研究生，因为跟导师做项目开始接触 STL 与 C++ 标准库，大概是 2005 年从 1.33 版结识了 Boost，这才真正领略了 C++ 的精髓。那段时期 Java 和 C# 正在国内大行其道，C++ 则势单力薄，有关 STL 和 C++ 标准的技术书籍寥寥无几，而讲解 Boost 的书更是为零，故对 Boost 的学习基本只能靠自己的摸索与实践。好在 Boost 自带的文档相当丰富（虽然看全英文的资料十分辛苦），而且源码也写得比较清晰规范，在熟悉了 STL 的基础上学习 Boost 倒也并不算太难。

但 Boost 的一个最大的特点就是“庞大”，功能组件众多，要想把它全部装进脑子里融会贯通基本上是不可能的，使用时需要经常查阅英文文档，相当的麻烦。因此，在学习的过程中，我逐渐产生了编写学习笔记的想法。一开始只是一个简单的纯文本文件，记录了一些使用经验的片断，随着积累的不断增多，纯文本形式已经不能够满足知识整理的需求了，于是我又把这些文字迁移到了 Word 文档里，把使用经验分类编目，加以较系统地归纳梳理。慢慢地，这份学习笔记

居然有了上万字的规模，成为了一份很好的 Boost 备忘参考，在日常的开发工作中给了我很大的帮助——就像《设计模式》一书中所说的那样，捕获了很多使用 Boost 解决问题的实践经验，避免了重复发现。不过，这份资料一直是仅限于我个人使用，属于“自娱自乐”的作品，从未示人。

时间一晃到了 2010 年 1 月份的某天夜里，不知道是什么原因我忽然失眠了，躺在床上翻来覆去怎么也睡不着。突然，一个念头闯入了脑海：把 Boost 开发经验整理出版吧，让更多人能够分享这些知识，正所谓“独乐乐，与众乐乐，孰乐？”。这个大胆的想法的出现让我那天的失眠又延长了几个小时——关于书的各种构想在头脑中“肆虐横行”。

随后的几天里我就把这个想法付诸行动了，虽然以前也写过并发表很多文章，也在网上印刷了几本个人文集，但出版正式的书还是第一次。在把学习笔记进一步整理完善，编写出较完整的结构和一个样章后，我就开始联系出版社了。当初并没有多大的信心，毕竟我这个作者名不见经传，也没有什么资历、背景和名气（而且还是个“网盲”，从未跟随潮流开个人博客）。很幸运，发出的第一个 E-mail 就是电子工业出版社，而且编辑也在第一时间回复了我，这才给了我以持续写作完成全书的动力。

写作过程中我也进一步加深了对 Boost 的认识，澄清了许多原来未曾注意到的细节。原本只打算写 20 万字左右、三百多页，但写到中途才发现 Boost 库的博大精深远非当初的理解，也意识到了自己当初学习的肤浅。历经了近半年近乎不眠不休的努力，最终呈现给读者的是这本厚达 500 多页的图书，文字量是最初学习笔记的数十倍，内容也翔实丰满了很多——达成这个结果，我个人可以说是问心无愧了。

C++与 Boost

C++ 较 Java 和 C# 等语言的一个最大不同在于它并非是由某个公司或个人把持的，它的真正发展动力来自于广大程序员。Boost 就是这样的一个典范，它成功地填补了从 C++98 到 C++0x 这“失落的十年”间的空白，在竞争对手 Java 和 C# 不断更新版本新增特性的时候以库的形式极大地增强了 C++ 的能力，使 C++ 不至于因为标准规范的滞后而落后于时代，而且 Boost 还深层次挖掘了 C++ 的潜力，开创了泛型编程、模板元编程、函数式编程等崭新的境界。

就个人来说，我比较喜欢的 Boost 版本有两个，分别是 1.35 和 1.39：1.35 版增加了 asio、bimap、circular_buffer 等许多重要组件，而 1.39 版则增加了 signals2 库，这两个版本都在我的工作用机上停留了相当长的时间。落笔之时，Boost 已经更新到了 1.43 版，成长为了一个相当完善、全面、强大的 C++ 程序库。可以毫不夸张地说，现在的 C++ 程序员，如果不熟悉 Boost，那么至少丧失了一半使用 C++ 的好处，同时会多耗费数倍的开发精力和时间。

随着 C++0x 标准的即将来临，Boost 程序库的发展也出现了加速的趋势，由原来间隔数月

不定期更新版本，改为定期（每 3 个月左右）发布新版本，而且每个新版本都会包含大量极有价值的更新内容。因此，希望读者在阅读本书时及时访问 Boost 的官网（<http://www.boost.org>），以便获取最新的版本。

感谢读者选择本书，再说一句真心的“套话”（笑）：限于作者水平有限，书中错漏在所难免，敬请读者原谅、指正。

致谢

首先我要感谢整个 C++ 群体，特别是：C++ 语言的发明者 Bjarne Stroustrup 博士——他给我们带来了美妙的 C++；然后是 Alexander Stepanov 和 C++ 标准委员会——他们把 STL 引入了 C++，开创了 C++ 的现代编程风格；以及 Beman G. Dawes、Boost 程序库的所有作者和 Boost 社区——他们为我们奉献了如此高水准的程序库。

其次我要感谢电子工业出版社博文视点公司，他们给了我这个把自己的开发经验出版成书的机会，在把潦草的个人学习笔记变成正式图书的过程中他们付出了艰辛的努力。还要感谢陈硕先生，他审阅了本书的部分手稿，提出了很多有价值的参考意见，并慨然为本书撰写序言。

接下来我要感谢我的家人：感谢我的父母和弟弟，他们永远是我生命中最重要的人；感谢我的妻子，她自始至终都支持我的写作，并担负了大部分照顾孩子的家务（虽然偶有怨言）；还要对已满一岁半的女儿说声抱歉，为了写作本书，我已经牺牲了很多陪她玩耍的时间。

我还要感谢黄美华、冯薇、戚天龙、罗玉震、颜静、陈刚、张秋香、缪泽波等同事，长期的共事令我们建立了深厚的友谊。对后两位同事致以特别的感谢，他们对完成本书提供了大力的支持和帮助。

最后，感谢多年以来的好友岳大海、时吉斌、王峰，感谢我的中学老师邓英、杜爱芹、练鑫云、陈静，感谢我的研究生导师贾云得，以及所有在我成长过程中曾经给予我关心和帮助的朋友们！

罗剑锋

2010 年 6 月 7 日 于 北京 王府井

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真题解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总: **ASP.NET** 篇

.Net 技术精品资料下载汇总: **C#**语言篇

.Net 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子书下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目录

第 0 章 导读	1	2.2 timer	17
0.1 关于本书	1	2.2.1 用法	18
0.2 读者对象	1	2.2.2 类摘要	18
0.3 本书的术语与风格	2	2.2.3 使用建议	19
0.4 本书的结构	3	2.3 progress_timer	20
0.5 如何阅读本书	5	2.3.1 用法	20
第 1 章 Boost 程序库总论	7	2.3.2 类摘要	21
1.1 关于 Boost	7	2.3.3 扩展计时精度	21
1.1.1 什么是 Boost	7	2.4 progress_display	23
1.1.2 安装 Boost	8	2.4.1 类摘要	24
1.1.3 使用 Boost	8	2.4.2 用法	25
1.2 关于 STLport	9	2.4.3 注意事项	26
1.2.1 什么是 STLport	9	2.5 date_time 库概述	27
1.2.2 安装 STLport	10	2.5.1 编译 date_time 库	27
1.2.3 使用 STLport	10	2.5.2 date_time 库的基本概念	28
1.3 开发环境	10	2.6 处理日期	29
1.3.1 STLport 的编译方法	11	2.6.1 日期	29
1.3.2 Boost 的编译方法	11	2.6.2 创建日期对象	30
1.3.3 VisualStudio 2005 环境设置	15	2.6.3 访问日期	31
第 2 章 时间与日期	17	2.6.4 日期的输出	32
2.1 timer 库概述	17	2.6.5 与 tm 结构的转换	33
		2.6.6 日期长度	34
		2.6.7 日期运算	35

2.6.8	日期区间	36	3.3.3	使用建议	69
2.6.9	日期区间运算	38	3.4	shared_ptr	69
2.6.10	日期迭代器	39	3.4.1	类摘要	70
2.6.11	其他功能	40	3.4.2	操作函数	70
2.6.12	综合运用	40	3.4.3	用法	72
2.7	处理时间	43	3.4.4	工厂函数	74
2.7.1	时间长度	43	3.4.5	应用于标准容器	75
2.7.2	操作时间长度	45	3.4.6	应用于桥接模式	76
2.7.3	时间长度的精确度	46	3.4.7	应用于工厂模式	77
2.7.4	时间点	48	3.4.8	定制删除器	78
2.7.5	创建时间点对象	48	3.4.9	高级议题	80
2.7.6	操作时间点对象	49	3.5	shared_array	81
2.7.7	与 tm、time_t 等结构的转换	50	3.5.1	类摘要	81
2.7.8	时间区间	51	3.5.2	用法	82
2.7.9	时间迭代器	51	3.6	weak_ptr	82
2.7.10	综合运用	52	3.6.1	类摘要	82
2.8	date_time 库的高级议题	54	3.6.2	用法	83
2.8.1	编译配置宏	55	3.6.3	获得 this 的 shared_ptr	84
2.8.2	格式化时间	55	3.7	intrusive_ptr	85
2.8.3	本地时间	56	3.8	pool 库概述	85
2.8.4	序列化	58	3.9	pool	85
2.9	总结	58	3.9.1	类摘要	86
第 3 章	内存管理	61	3.9.2	操作函数	86
3.1	smart_ptr 库概述	61	3.9.3	用法	87
3.1.1	RAII 机制	61	3.10	object_pool	88
3.1.2	智能指针	62	3.10.1	类摘要	88
3.2	scoped_ptr	63	3.10.2	操作函数	88
3.2.1	类摘要	63	3.10.3	用法	89
3.2.2	操作函数	64	3.10.4	使用更多的构造参数	90
3.2.3	用法	65	3.11	singleton_pool	91
3.2.4	与 auto_ptr 的区别	66	3.11.1	类摘要	91
3.3	scoped_array	67	3.11.2	用法	92
3.3.1	类摘要	67	3.12	pool_alloc	93
3.3.2	用法	68	3.13	总结	93

第 4 章 实用工具	95	4.7 tribool	120
4.1 noncopyable	95	4.7.1 类摘要	121
4.1.1 原理	96	4.7.2 用法	122
4.1.2 用法	96	4.7.3 为第三态更名	122
4.2 typeof	97	4.7.4 输入输出	123
4.2.1 动机	97	4.7.5 与 optional<bool>的区别	124
4.2.2 用法	99	4.8 operators	125
4.2.3 向 typeof 库注册自定义类	100	4.8.1 基本运算概念	126
4.2.4 高级议题	101	4.8.2 算术操作符的用法	127
4.3 optional	101	4.8.3 基类链	129
4.3.1 “无意义”的值	101	4.8.4 复合运算概念	130
4.3.2 类摘要	102	4.8.5 相等与等价	131
4.3.3 操作函数	102	4.8.6 解引用操作符	133
4.3.4 用法	103	4.8.7 下标操作符	134
4.3.5 工厂函数	104	4.8.8 高级议题	135
4.3.6 高级议题	105	4.9 exception	136
4.4 assign	106	4.9.1 标准库中的异常	137
4.4.1 使用操作符+=向容器 增加元素	106	4.9.2 类摘要	137
4.4.2 使用操作符()向容器 增加元素	107	4.9.3 向异常传递信息	139
4.4.3 初始化容器元素	108	4.9.4 更进一步的用法	140
4.4.4 减少重复输入	110	4.9.5 包装标准异常	142
4.4.5 与非标准容器工作	111	4.9.6 使用函数抛出异常	143
4.4.6 高级用法	112	4.9.7 获得更多的调试信息	144
4.5 swap	113	4.9.8 高级议题	145
4.5.1 原理	113	4.10 uuid	146
4.5.2 交换数组	114	4.10.1 类摘要	147
4.5.3 特化 std::swap	114	4.10.2 用法	148
4.5.4 特化 ADL 可找到的 swap	115	4.10.3 生成器	150
4.5.5 使用建议	116	4.10.4 增强的 uuid 类	152
4.6 singleton	116	4.10.5 与字符串的转换	153
4.6.1 boost.pool 的单件实现	117	4.10.6 SHA1 摘要算法	154
4.6.2 boost.serialization 的 单件实现	119	4.11 config	155
		4.11.1 BOOST_STRINGIZE	155
		4.11.2 BOOST_STATIC_ CONSTANT	155

4.11.3 禁止编译器警告	156	5.4.1 类摘要	190
4.11.4 其他工具	157	5.4.2 用法	190
4.12 utility	157	5.4.3 分词函数对象	191
4.12.1 BOOST_BINARY	157	5.4.4 char_separator	192
4.12.2 BOOST_CURRENT_		5.4.5 escaped_list_separator	193
FUNCTION	158	5.4.6 offset_separator	193
4.13 总结	160	5.4.7 tokenizer 库的缺陷	195
第 5 章 字符串与文本处理	163	5.5 xpressive	196
5.1 lexical_cast	163	5.5.1 两种使用方式	197
5.1.1 用法	164	5.5.2 正则表达式语法简介	197
5.1.2 异常 bad_lexical_cast	165	5.5.3 类摘要	199
5.1.3 对转换对象的要求	166	5.5.4 匹配	201
5.1.4 应用于自己的类	166	5.5.5 查找	203
5.2 format	167	5.5.6 替换	204
5.2.1 简单的例子	168	5.5.7 迭代	206
5.2.2 输入操作符%	169	5.5.8 分词	207
5.2.3 类摘要	171	5.5.9 与 regex 的区别	208
5.2.4 格式化语法	172	5.5.10 高级议题	209
5.2.5 format 的性能	173	5.6 总结	211
5.2.6 高级用法	173	第 6 章 正确性与测试	213
5.3 string_algo	175	6.1 assert	213
5.3.1 简单的例子	175	6.1.1 基本用法	213
5.3.2 string_algo 概述	176	6.1.2 禁用断言	214
5.3.3 大小写转换	177	6.1.3 扩展用法	215
5.3.4 判断式 (算法)	178	6.1.4 BOOST_VERIFY	216
5.3.5 判断式 (函数对象)	179	6.2 static_assert	217
5.3.6 分类	180	6.2.1 用法	217
5.3.7 修剪	181	6.2.2 使用建议	218
5.3.8 查找	182	6.3 test	219
5.3.9 替换与删除	184	6.3.1 编译 test 库	219
5.3.10 分割	186	6.3.2 最小化的测试套件	220
5.3.11 合并	187	6.3.3 单元测试框架简介	221
5.3.12 查找 (分割) 迭代器	188	6.3.4 测试断言	222
5.4 tokenizer	189	6.3.5 测试用例与套件	223

6.3.6 测试实例	224	7.4.1 类摘要	261
6.3.7 测试夹具	226	7.4.2 基本用法	262
6.3.8 测试日志	228	7.4.3 值的集合类型	263
6.3.9 运行参数	229	7.4.4 集合类型的用法	264
6.3.10 函数执行监视器	230	7.4.5 使用标签类型	266
6.3.11 程序执行监视器	233	7.4.6 使用 assign 库	267
6.3.12 高级议题	234	7.4.7 查找与替换	268
6.4 总结	236	7.4.8 投射	269
第 7 章 容器与数据结构	239	7.4.9 高级议题	270
7.1 array	239	7.5 circular_buffer	271
7.1.1 类摘要	240	7.5.1 类摘要	271
7.1.2 操作函数	240	7.5.2 用法	272
7.1.3 用法	241	7.5.3 环型缓冲区	273
7.1.4 能力限制	242	7.5.4 空间优化型缓冲区	275
7.1.5 array 的初始化	242	7.6 tuple	275
7.1.6 实现 ref_array	243	7.6.1 最简单的 tuple: pair	276
7.1.7 ref_array 的用法	244	7.6.2 类摘要	276
7.2 dynamic_bitset	245	7.6.3 创建与赋值	277
7.2.1 类摘要	245	7.6.4 访问元素	278
7.2.2 创建与赋值	247	7.6.5 比较操作	279
7.2.3 容器操作	248	7.6.6 输入输出	280
7.2.4 位运算与比较运算	249	7.6.7 连结变量	281
7.2.5 访问元素	249	7.6.8 应用于 assign 库	282
7.2.6 类型转换	251	7.6.9 应用于 exception 库	282
7.2.7 集合操作	251	7.6.10 内部结构	282
7.2.8 综合运用	252	7.6.11 使用访问者模式	284
7.3 unordered	253	7.6.12 高级议题	285
7.3.1 散列集合简介	254	7.7 any	287
7.3.2 散列集合的用法	255	7.7.1 类摘要	287
7.3.3 散列映射简介	256	7.7.2 访问元素	288
7.3.4 散列映射的用法	256	7.7.3 用法	289
7.3.5 性能比较	257	7.7.4 简化的操作函数	290
7.3.6 高级议题	259	7.7.5 保存指针	291
7.4 bimap	261	7.7.6 输出	292
		7.7.7 应用于容器	294

7.8	variant	294	8.2.2	使用 tuples::tie	334
7.8.1	类摘要	294	8.3	minmax_element	334
7.8.2	访问元素	295	8.3.1	用法	334
7.8.3	用法	296	8.3.2	其他函数的用法	335
7.8.4	访问器	297	8.4	总结	336
7.8.5	与 any 的区别	300	第 9 章	数学与数字	337
7.8.6	高级议题	300	9.1	integer	337
7.9	multi_array	302	9.1.1	integer_traits	337
7.9.1	类摘要	302	9.1.2	标准整数类型	339
7.9.2	用法	304	9.1.3	整数类型模板类	341
7.9.3	多维数组生成器	306	9.2	rational	344
7.9.4	改变形状和大小	307	9.2.1	类摘要	344
7.9.5	创建子视图	308	9.2.2	创建与赋值	345
7.9.6	适配普通数组	310	9.2.3	算术运算与比较运算	346
7.9.7	高级议题	311	9.2.4	类型转换	346
7.10	property_tree	314	9.2.5	输入输出	347
7.10.1	类摘要	315	9.2.6	分子与分母	347
7.10.2	读取配置信息	316	9.2.7	与数学函数配合工作	347
7.10.3	写入配置信息	318	9.2.8	异常	348
7.10.4	更多用法	319	9.2.9	rational 的精度	348
7.10.5	XML 数据格式	320	9.2.10	实现无限精度的整数类型	348
7.10.6	其他数据格式	321	9.2.11	最大公约数和最小公倍数	353
7.10.7	高级议题	323	9.3	crc	353
7.11	总结	324	9.3.1	类摘要	354
第 8 章	算法	327	9.3.2	预定义的实现类	354
8.1	foreach	327	9.3.3	计算 CRC	355
8.1.1	用法	328	9.3.4	CRC 函数	356
8.1.2	详细解说	329	9.3.5	自定义 CRC 函数	357
8.1.3	使用 typeof	329	9.4	random	357
8.1.4	更优雅的名字	330	9.4.1	伪随机数发生器	358
8.1.5	支持的序列类型	331	9.4.2	伪随机数发生器的构造	359
8.1.6	一个小问题	332	9.4.3	伪随机数发生器的拷贝	360
8.2	minmax	332	9.4.4	随机数分布器	360
8.2.1	用法	333	9.4.5	随机数分布器类摘要	361

9.4.6 随机数分布器用法	363	10.3.14 文件流操作	399
9.4.7 变量发生器	364	10.4 program_options	400
9.4.8 产生随机数据块	365	10.4.1 编译 program_options 库	400
9.4.9 真随机数发生器	367	10.4.2 概述	401
9.4.10 实现真随机数发生器	368	10.4.3 选项值	403
9.5 总结	369	10.4.4 选项描述器	404
第 10 章 操作系统相关	371	10.4.5 选项描述器的用法	405
10.1 io_state_savers	371	10.4.6 分析器	407
10.1.1 类摘要	372	10.4.7 存储器	409
10.1.2 用法	372	10.4.8 使用位置选项值	409
10.1.3 简化 new_progress_timer	374	10.4.9 分析环境变量	411
10.2 system	374	10.4.10 分组选项信息	412
10.2.1 编译 system 库	375	10.4.11 高级用法	414
10.2.2 错误值枚举	375	10.5 总结	417
10.2.3 错误类别	376	第 11 章 函数与回调	419
10.2.4 错误代码	377	11.1 result_of	419
10.2.5 错误异常	379	11.1.1 原理	420
10.3 filesystem	380	11.1.2 用法	422
10.3.1 编译 filesystem 库	380	11.2 ref	422
10.3.2 类摘要	381	11.2.1 类摘要	423
10.3.3 路径表示	383	11.2.2 基本用法	424
10.3.4 可移植的文件名	384	11.2.3 工厂函数	425
10.3.5 路径处理	385	11.2.4 操作包装	425
10.3.6 异常	387	11.2.5 综合应用	426
10.3.7 文件状态	388	11.2.6 为 ref 增加函数调用功能	427
10.3.8 文件属性	390	11.3 bind	429
10.3.9 文件操作	391	11.3.1 工作原理	429
10.3.10 迭代目录	392	11.3.2 绑定普通函数	430
10.3.11 实例 1: 实现查找文件 功能	394	11.3.3 绑定成员函数	432
10.3.12 实例 2: 实现模糊查找 文件功能	395	11.3.4 绑定成员变量	433
10.3.13 实例 3: 实现拷贝目录 功能	397	11.3.5 绑定函数对象	433
		11.3.6 使用 ref 库	434
		11.3.7 高级议题	435
		11.4 function	437

11.4.1 类摘要	437	12.1.13 高级议题	488
11.4.2 function 的声明	438	12.2 asio	493
11.4.3 操作函数	439	12.2.1 概述	493
11.4.4 比较操作	440	12.2.2 定时器	494
11.4.5 用法	440	12.2.3 定时器用法	495
11.4.6 使用 ref 库	441	12.2.4 网络通信简述	498
11.4.7 用于回调	442	12.2.5 IP 地址和端点	499
11.4.8 与 typeof 的区别	445	12.2.6 同步 socket 处理	500
11.5 signals2	445	12.2.7 异步 socket 处理	502
11.5.1 类摘要	446	12.2.8 查询网络地址	506
11.5.2 操作函数	447	12.2.9 高级议题	507
11.5.3 插槽的连接与调用	448	12.3 总结	511
11.5.4 信号的返回值	449	第 13 章 编程语言支持	513
11.5.5 合并器	450	13.1 python 库概述	513
11.5.6 管理信号的连接	452	13.1.1 Python 语言简介	514
11.5.7 更灵活的管理信号连接	453	13.1.2 安装 Python 环境	515
11.5.8 自动连接管理	455	13.1.3 编译 python 库	515
11.5.9 应用于观察者模式	457	13.1.4 使用 python 库	516
11.5.10 高级议题	460	13.2 嵌入 Python	517
11.6 总结	465	13.2.1 初始化解释器	517
第 12 章 并发编程	467	13.2.2 封装 Python 对象	518
12.1 thread	467	13.2.3 执行 Python 语句	520
12.1.1 编译 thread 库	468	13.2.4 异常处理	521
12.1.2 使用 thread 库	468	13.3 扩展 Python	522
12.1.3 时间功能	469	13.3.1 最简单的例子	523
12.1.4 互斥量	469	13.3.2 导出函数	525
12.1.5 线程对象	472	13.3.3 导出重载函数	526
12.1.6 创建线程	473	13.3.4 导出类	528
12.1.7 操作线程	475	13.3.5 导出类的更多细节	530
12.1.8 中断线程	476	13.3.6 高级议题	532
12.1.9 线程组	479	13.4 总结	534
12.1.10 条件变量	480	第 14 章 其他 Boost 组件	537
12.1.11 共享互斥量	484	14.1 字符串和文本处理	537
12.1.12 future	485		

14.2 容器与数据结构	538	15.3 行为模式	552
14.3 迭代器	539	15.4 其他模式	555
14.4 函数对象与高级编程	539	15.5 总结	556
14.5 泛型编程	540	第 16 章 结束语	559
14.6 模板元编程	541	16.1 未臻完美的 Boost	559
14.7 预处理元编程	542	16.2 让 Boost 工作得更好	560
14.8 并发编程	542	16.3 工夫在诗外	563
14.9 数学与数字	543	附录 A 推荐书目	565
14.10 TR1 实现	543	附录 B 网络资源	567
14.11 输入输出	544	附录 C C++标准简述	569
14.12 杂项	544	附录 D STL 简述	571
14.13 总结	546	附录 E ref_array 实现代码	573
第 15 章 Boost 与设计模式	547		
15.1 创建型模式	547		
15.2 结构型模式	549		

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真题解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总: **ASP.NET** 篇

.Net 技术精品资料下载汇总: **C#**语言篇

.Net 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子书下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

第 0 章

导读

0.1 关于本书

C++是一种伟大的编程语言，某种程度上它甚至超越了编程语言的境界而升华为一种哲学。

C++又是一种多范式、可扩展的编程语言，能够支持多种编程风格（基于过程、基于对象、面向对象、泛型、模板元、函数式），非常自由灵活，易学难精，在不同编程风格间切换时必须小心谨慎以避免失误。

C++98 标准中出现的 STL（标准模板库）极大地改变了 C++ 程序员的编程思维，使“泛型”成为了 21 世纪以来程序开发界最流行的词汇之一。而 C++ 标准委员会成员所设立的 Boost 社区和开发的 Boost 程序库，更将“泛型”等现代 C++ 编程方法发挥到了极致。

Boost 程序库代表了目前 C++ 语言最新最前沿的技术，内容博大精深，丝毫不逊于经典的 STL，但同时也令很多人难以摸清门路，不得登堂入室而一窥究竟。有鉴于此，作者根据多年在实际开发中使用 Boost 库的经验编写了本书，想为广大 C++ 程序员了解 C++ 的最新技术进展尽一份自己的绵薄之力。

本书的定位是“指南”（Guide、Introduction），而不是参考手册（Reference）或者使用说明（Manual），能够解答 90% 但不是所有 Boost 库相关的问题。但作者尽量做到让本书接近一本参考手册，使读者在阅读时能够脱离计算机，不至于频繁使用鼠标和键盘查询自带在线帮助或者源代码。

0.2 读者对象

本书适合有一定 C++ 基础的读者。

阅读本书需要有一定的 C++ 知识，基本要求是了解面向对象编程技术，了解封装、继承等面向对象程序语言的特性，此外还要理解名字空间 (namespace)、异常 (exception)、模板 (template)^①、泛型编程 (generic programing) 等 C++ 高级特性 (但不必非常深入的了解)，最好还能够了解设计模式和 C++ 标准库 (STL) 提供的容器、算法等各种组件。

如果读者是 C++ 初学者或还不具备以上所列的知识，建议先阅读附录推荐书目所列的技术书籍，然后再学习本书。如果手头刚好有推荐书目中的一本或两本，则可以一边翻阅推荐书目，一边学习本书。

无论读者是 C++ 哪一层次的用户，现在或将来本书都会给您带来帮助。

0.3 本书的术语与风格

本小节列出了本书中经常用到的专业术语和编程风格，以期与读者获得阅读的共识。

Boost 库并不是一个单一、平面化的程序库，而是有着复杂的内部结构，每个“库”可能是由其他许多更小的“库”组成的。因此，本书中把程序库中所有有机组成部分统称为“组件”，“库” (Library) 与“组件” (Compoment) 这两个术语有时会通用。

namespace 这个 C++ 术语有译称作“命名空间”、“名称空间”、“名字空间”，本书依据推荐书目 [3] [4] 称做“名字空间”。这只是作者个人习惯而已，如果在阅读过程中给读者造成了小小的困扰，还请谅解。

在使用 template 定义模板类或者模板函数时，本书统一使用 typename 而不是常见的 class，因为 typename 能够更清楚地向代码阅读者表明这是一个类型参数，而不一定是一个类 (class)。但例外的是书中列出 Boost 源代码，会尽量保持其原始形式。

在命名函数或者类时，本书遵循 C++ 标准库和 Boost 的惯例，均采用小写形式，单词间以下画线分隔，如 demo_class、rand_bytes()，但并不要求读者也必须遵循这种命名方式，通常自己编写的类使用大写字母开头的单词命名会更好。

在 for 循环递增变量、指针或者迭代器对象的时候，本书统一使用 ++ 操作符的前置用法 (例如 ++i)，而不是后置用法 (i++)，因为前者不需要返回一个临时对象，执行效率更高。

① 模板类和模板函数一般形式是 class_or_func<T>，例如 vector<int>。<> 在编程语言中通常被用于大小比较，因此其形式对于初学者很难接受，但这种形式具有良好的可读性，<> 可以被读作英文 of，比如 vector<int> 可以读作 vector of int，清晰地表明了类/函数和它的模板类型的关系，这样可以减少一部分对尖括号的不适应感，作者经常这么做，读者也可以试一下，很有效。

“未定义行为”一词经常用来指代某些操作可能导致的不正确结果，如使用已失效的迭代器、错误地使用指针等等。对于“未定义行为”的一个较好（但不太精确）的定义是：程序在开发人员面前运行正常，在测试人员面前运行正常，但在老板或者最终用户面前运行时崩溃了。读者应当小心并尽量避免“未定义行为”，它是代码中的“定时炸弹”，如果它在调试的过程中爆炸了，那通常是最好的结果，因为它明确地告诉了你代码存在问题。

本书中“C++标准”一词指1998年通过的C++ ISO国际标准，编号为ISO/IEC 14882-1998，有时也会称为“C++98”或者“C++98 标准”。标准中定义的标准库则称为“C++标准库”或者“STL”，但严格意义上 STL 与标准库并不等价，STL 只是标准库中的一个（很大的）子集，这么称呼有时候只是为了行文上的方便。

为使读者对 C++标准能够有简单的了解，作者编写了一份 C++标准和 STL 的简要介绍，作为本书附录供参考。

一般情况下使用 C++标准库都必须包含相应的头文件并且加上 using 语句（using namespace std;），但标准库已经成为了 C++软件开发的基础设施，应用得非常频繁，因此书中的代码示例片断一般会将其略去。但有的情况下为了特别强调，会加上 std 名字空间前缀，如 std::vector。读者可以认为书中所有代码都默认包含了如下的头文件：

```
///stdcpp.hpp
#include <string>
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <algorithm>
using namespace std;
```

对于微软出品的 VisualStudio 200X，由于本书只用到其中的 C++开发环境，有的时候作者会将之简称为 VS200X 或 VCX。具体到本书所用的 VisualStudio 2005，则简称为 VS2005 或 VC8。

0.4 本书的结构

Boost 库组件繁多，相互关联也较多，如何排列其顺序是作者面临的一个颇为棘手的问题。

Boost 官方提供了两种基本方式，一种是按照组件的字母顺序，另一种是按照功能用途分类

顺序,但这两种方式都不是组织本书结构的最佳手法。经反复斟酌,作者决定以难易度和实用程度对 Boost 库组件分类排序,采用由浅入深循序渐进的方式,先介绍较简单易用且实用程度高的库,然后逐步深入,介绍用法复杂的库,以期帮助读者尽快掌握 Boost 的使用方法。

对于每个 Boost 组件,本书通常首先简要介绍其功能,然后说明其头文件和编译方法(如果需要编译的话),列出类的声明概要,再使用例子讲解详细用法和注意事项,涉及其他 Boost 库组件时则以交叉引用的方式指明参考页码,最后是对该库的总结。

本书共分 16 章,每章的内容简介如下:

■ 第 0 章: 导读

也就是读者正在阅读的这一章,介绍本书的基本内容和一些注意事项。

■ 第 1 章: Boost 程序库总论

本章简要介绍 Boost 的历史、特点和获取方式,并论述了一个 STL 的免费高效实现——STLport,最后介绍本书的开发环境和如何编译 STLport 和 Boost。

■ 第 2 章至第 14 章

第 2 章至第 14 章分门别类、由浅入深地介绍 Boost 库的各个组件,占据了本书的大部分篇幅,也是读者需要仔细阅读的内容。其中既包括如 timer、swap 等简单的小工具,也包括 test、thread、python 等用法复杂且功能强大的组件,Boost1.42 版全部 99 个库在本书都可以找到阐述。

■ 第 15 章: Boost 与设计模式

本章结合之前介绍的所有 Boost 库组件简要论述了推荐书目[1]中的 23 个设计模式和几个其他常用模式,以及 Boost 库组件使用这些设计模式的方法,从设计模式的抽象层次来加深理解 Boost 库。

■ 第 16 章: 结束语

本章简单展望了 Boost 今后的发展,介绍了其他可与 Boost 互为补充的开源 C++库,并对如何做一个好的程序员提出了一些自己的见解。

■ 附录

书末的附录也很有价值,列出了作者认为值得阅读的 C++经典书籍,它们也是作者编写本书时的案头必备参考资料。还有就是网络资源、C++标准和 STL 简述,以及书中开发的 ref_array 完整实现代码清单。

这些内容被放在附录部分并不是因为它们不重要，而仅仅是因为它们与全书的主题关联不大，但很值得阅读，读者会从中发现一些很有用的东西。

0.5 如何阅读本书

本书是一本介绍程序库的书籍，其中的很多组件是彼此独立的，所以在阅读完第 0 章（即本章）和第 1 章后，可以随意自由阅读其他章节。

读者既可以按照书的物理顺序循序渐进地逐页阅读，也可以查阅目录，然后直接跳到感兴趣的章节。不过对于大多数读者来说，作者还是推荐第一种方式，因为这可能会是学习效率更高、学习曲线更平滑的方式。

涉及编程语言，尤其是 C++，书中不可避免地会出现大量的代码片断，有的还可能很长，希望读者阅读时能有足够的耐心。这些代码都是精心编制的范例，将会指导读者如何使用 Boost 编写正确、高效的代码。

阅读时还需要注意一点：本书并不包含 Boost 库的所有接口说明，如果在阅读过程中对某些地方有疑虑，请参考 Boost 说明文档或者直接阅读 Boost 实现代码。

最后，欢迎来到 Boost 的世界!!!

第 1 章

Boost 程序库总论

1.1 关于 Boost

本小节将带领读者快速浏览 Boost 程序库，了解它的历史、组成和基本使用方式。

1.1.1 什么是 Boost



Boost 库是一个功能强大、构造精巧、跨平台、开源并且完全免费的 C++ 程序库。

1998 年，Beman G. Dawes (C++ 标准委员会成员之一) 发起倡议并建立了 Boost 社区，目的是向 C++ 程序员提供免费的 (free)、同行审查的 (peer review)、可移植的 (portable) 高质量 C++ 源程序库。Boost 强调程序库要与 C++ 标准库很好地共同工作，建立在“既有的实践”之上和提供参考实现，使得 Boost 库可以适合最后的标准化。自创立以来，Boost 社区的工作已经取得了卓越的成果，C++0x TR1 中有三分之二来自 Boost 库，而且将来还会有更多的库进入新标准。

C++ 三十余年的发展历史中产生了数不清的程序库，有影响力的也不计其数（如微软的 MFC、Borland 的 OWL），然而没有一个能够与 Boost 相提并论，Boost 有着与其他程序库无法比拟的优点。

首先，许多 Boost 库的作者本身就是 C++ 标准委员会成员，因此 Boost 天然成为了标准库的后备，负责向新标准输送组件，也使得 Boost 获得了“准”标准库的美誉。其次，Boost 库独特的同行审查制度保证了每一个 Boost 库组件都经过了严格的审查和验证，使库具有很高的工业强度，甚至超过大多数商业产品的实现。Boost 库采用了类似 STL 的编程范式，但却并没有

STL 那样晦涩难懂，代码格式优美清晰、易于阅读，而且附带丰富的说明文档——它既是一个程序库，同时也是一个很有价值的学习现代 C++ 编程的范本。最后，Boost 的发布采用 Boost Software License，这是一个不同于 GPL、Apache 的非常宽松的许可证，允许库用户将 Boost 用于任何用途，既鼓励商业用途，也鼓励非商业用途。用户无须支付任何费用，即可轻松享有 Boost 的全部功能。

本书内容基于 Boost 官方于 2010 年 2 月发布的 Boost1.42 版，共包含 99 个库（组件），分为 20 大类，涵盖字符串与文本处理、容器、迭代器、算法、图像处理、模板元编程、并发编程等许多 C++98 未涉及的领域——使用 Boost，将大大增强 C++ 的功能和表现力。

虽然本书主要讨论 Boost1.42 版，但由于 Boost 库中的许多组件已经相当稳定，故书中的论述对 1.42 版之前和之后的版本也基本适用。^①

1.1.2 安装 Boost

从 Boost 网站 (www.boost.org) 下载 boost_1_42_0.7z，一个约 30MB 左右大小的压缩包文件，使用 7Zip、WinRAR 或者其他工具将该文件解压缩到硬盘任意位置即可完成安装，本书使用的路径是 D:\boost。

注意：Boost1.42 版解压缩后约有 200MB，请确保硬盘有足够的空间。

1.1.3 使用 Boost

Boost 库大部分组件（近 90%）不需要编译，直接包含头文件即可。例如，如果要使用 boost::tribool，只需要在 C++ 源文件中添加如下 include 语句即可（当然，接下来的代码可能还需要 using namespace boost;）：

```
#include <boost/logic/tribool.hpp>           //使用 tribool 库
```

细心的读者会发现，Boost 库的头文件与我们平常所用的头文件 (*.h) 或 C++ 标准库头文件（没有后缀名）不同，这正是 Boost 的独特之处。它把 C++ 类的声明和实现都放在了一个文件中，而不是分成两个文件，也就是 “.h+.cpp”，故文件的后缀是 .hpp。

之所以这么做当然是有理由的。首先就是与普通的 C 头文件 (*.h) 区分，另一个很重要的原因就是使 Boost 库不需要预先编译，直接引入程序员的工程即可编译链接，方便了库的使用。最后一个（无奈的）原因则是 C++ 编译器的限制，许多编译器尚不支持 C++ 标准提出的模板的分

^① 2010 年 5 月 Boost 发布了 1.43 版，但较 1.42 版没有太多的变化，仅增加了两个 functional 小组件，本书之后会以脚注的方式说明 1.43 版与 1.42 版之间的细微差异。

离编译模式 (export 关键字), 而 Boost 库大量使用了模板, 为了保持与各个编译器的兼容, 也不得不采用这种 .hpp 的头文件形式^①。

剩下的共十五个库 (包括 date_time、regex、program_options、test、thread、python 等) 必须编译成静态库或者动态库后才能使用。不过有个好消息, 其中有的库不需要编译也可以使用部分功能, 而更好的消息是有的库已经有了不需要编译的替代品 (xpressive 可替代 regex、signals2 可替代 signals)。

在 Windows 下的 VC 编译器支持自动链接技术, VC 程序员可以不必为链接静态库或动态库、调试库或发行库等问题而费心了。其他编译器就没有这样幸运, 必须在命令行上手工指定链接库。

1.2 关于 STLport

本小节将介绍 C++ 标准库的一个高效实现——STLport, 它是本书的默认标准库配置, 用于配合 Boost 程序库工作。

1.2.1 什么是 STLport

STLport 是一个完全符合 C++98 标准 (及 2003 年修订) 的一个免费的 C++ 标准库实现。它是由俄罗斯人 Boris Fomitchev 发起的开源项目, 目的是基于著名的 SGISTL 开发一个可移植到各种平台上使用的高效的 C++ 标准库。



STLport 具有很多其他 STL 实现所没有的优点。首先是高度的可移植性, 可以配合市面上几乎所有的操作系统和编译器使用, 使开发的程序能够在不同的编译平台上获得一致的标准库实现。其次是性能表现优秀, 其原始版本 SGISTL 就以高效而闻名, STLport 在移植时也特别注重性能与效率, 而且 100% 完全符合 C++ 标准规范。第三个优点是在标准之外增加了若干有用的扩展, 如 rope (增强的字符串类)、slist (单链表数据结构)、hash_map (散列映射容器), 以及支持线程安全。

STLport 以其优异的品质自发布以来获得了极大的成功, 许多编译器都采用 STLport 作为内置的标准库实现 (比如 Borland C++ Builder), 以至于 Boost 专门为 STLport 提供了编译选项和设置。

Windows 平台开发主流工具是 MSVC, 其自带的 STL 向来名声不佳, 虽然随着 VC 的版本升

① Java、C# 程序员应该对这种代码文件形式很熟悉, 这两种语言中都是在一个文件中编写所有代码 (*.java、*.cs)。

级而逐渐改善，但质量仍非一流水准。作者在工作用机上运行了简单的自测，结果是 VC8 自带的 STL (Dinkumware v405) 较 STLport5.21 慢大约一倍；而 VC9 自带的 STL (Dinkumware v503) 速度虽然有较大改善，基本与 STLport5.21 速度相当，但仍有大约 10% 以上的差距。综合各个方面来看，STLport 都较 VC 自带的 Dinkumware STL 实现好很多。

基于以上原因，本书使用 STLport 搭配 Boost，而没有使用 VC8 自带的 Dinkumware STL。

1.2.2 安装 STLport

STLport 目前的最新版本是 2008 年 12 月 10 日发布的 5.21 版。

从 STLport 网站 (www.stlport.org) 下载压缩包 STLport-5.2.1.tar.bz2，使用 WinRAR 或者其他工具解压缩到硬盘任意位置即可。本书使用的路径是 D:\STLport。

1.2.3 使用 STLport

STLport 早期 (5.0 版以前) 支持对本地 IO 流的 wrapper 模式，可以不需要编译，简单地定义一个宏即可使用。但自从 5.0 版后 STLport 取消了这个模式，用户只能选择使用流或者不使用流 (如有的平台不需要流输出的情形)，这也就意味着在 Windows、Linux 等主流操作系统下 STLport 必须要编译后才能使用。

在 Debug 模式下使用 STLport，需要定义宏 “`__STL_DEBUG`”。

在 Debug 模式下与 Boost 配合使用 STLport，需要定义宏 “`_STLP_DEBUG`”。

与 MFC 配合使用 STLport，需要定义宏 “`_STLP_USE_MFC`”。

更多有关 STLport 的使用信息请参阅 STLport 的说明文档。

1.3 开发环境

C++ 是一个大型语言，十分复杂。虽然 C++98 标准已经面世十余年，但仍然没有一个编译器敢宣称自己能够 100% 支持 C++ 的全部特性。

由于 Boost 大量使用了 C++ 高级特性 (如模板偏特化、ADL)，因此不是所有的编译器都能够很好地支持 Boost，并且每个组件对编译器的支持都不尽相同。虽然 Boost 已经针对平台和编译器的兼容性做了大量的工作，但仍有可能出现意外情况 (某些过“老”的编译器已经不再被支持)。

阅读本书和使用 Boost，读者需要一个能够较好地支持 C++ 标准的编译器和开发环境。

本书作者的主要开发环境是 WindowsXP + Visual Studio 2005, 使用的标准库为 STLport5.21 版。另一个开发环境是 Linux 2.6.9+ GCC 3.4.6, 使用 GCC 自带的标准库实现。全书所有例子代码均在 Windows 环境下编译通过, 大部分代码在 Linux 环境下编译通过。

对于还在使用 Visual C++ 6.0^①或其他开发环境的读者, 只能说抱歉了, 作者不能保证书中代码能够百分之百正确运行。请参考 Boost 说明文档查看对您正在使用的平台或编译器的支持情况。

下面就对 Windows 平台下的开发环境设置做详细介绍, 供读者参考。

1.3.1 STLport 的编译方法

本书使用默认选项编译生成 STLport 的多线程库, 具体步骤如下。

1. 从“开始”菜单运行 VS2005 工具的命令行提示符“Visual Studio 2005 Command Prompt”。
2. 执行命令“cd d:\STLport”, 进入 d:\STLport 目录。
3. 执行命令“configure msvc8”(如果使用 VC6、VC9 则执行 configure msvc6 或 msvc9) 配置编译环境。
4. 执行命令“cd d:\STLport\build\lib”。
5. 执行命令“nmake -f msvc.mak clean install”。

STLport 编译时间稍长, 大约需要数分钟。编译完成后会自动将编译出的*.dll 和*.lib 复制到 STLport\lib 和 STLport\bin 目录下, 之后可将 STLport\build\lib\obj 目录删除以节约硬盘空间。

1.3.2 Boost 的编译方法

Boost 的编译较 STLport 要复杂一些。它使用的不是已经成为公认标准的 make, 而是专门为 Boost 开发的工具 bjam (boost jam)。

要编译 Boost, 首先要获得 bjam 程序。

① STLport 可在 VC6 下正常编译运行, 但因不支持模板偏特化等许多新特性, 有的 Boost 库会缺少某些功能甚至不能使用。

编译 bjam

Boost 网站上提供各种平台上预编译好的 bjam 可执行程序, 可以直接下载, 不过我们最好还是从源码编译配合 Boost 的最佳 bjam 版本, 具体步骤如下。

1. 从“开始”菜单运行 VS2005 工具的命令行提示符“Visual Studio 2005 Command Prompt”。

2. 执行命令“cd D:\boost\tools\jam\src”。

3. 运行 build.bat, 编译 bjam。

编译完成后可执行文件位于 src\bin.ntx86 下。

无论是下载还是编译, 最后都要把 bjam 可执行文件 (bjam.exe) 拷贝到 Boost 根目录 d:\boost。

修改 Boost 配置

使用 bjam 之前需要修改 bjam 的配置文件: boost\tools\build\v2 下的 user-config.jam, 使用 Notepad、UltraEdit 或其他任意文本编辑器打开即可修改, 修改分两步进行。

1. 修改第 57 行, 去掉前面的#注释, 启用 msvc8.0

2. 修改第 75 行, 去掉前面的#注释, 启用 STLport, 修改 STLport 的头文件路径和 lib 路径, 注意路径应使用“/”分隔。例如 using stlport : : d:/stlport/srcd:/stlport/lib ;

如果读者不打算采用 STLport 作为 C++ 标准库的替代, 那么第二步可以省略。

完整编译 Boost

完成如上准备工作, 接下来就可以开始正式编译 Boost 库了, 在 Boost 根目录下执行如下命令:

```
bjam --toolset=msvc --build-type=complete stdlib=stlport stage
```

或者 (Linux):

```
bjam --toolset=gcc --build-type=complete stage
```

这样将开始对 Boost 的完整编译, 生成所有调试版、发行版的静态库和动态库^①。

① \boost\tools\jam\build_dist.bat 具有相同的效果, 将会完成 bjam 及 Boost 的所有编译工作, 但本书不推荐使用。

其中：

- `toolset` 选项指定编译器，如 `msvc`、`gcc` 等；
- `build-type` 选项指定编译类型，如不指定则默认使用 `release` 模式；
- `stdlib` 选项指定要搭配的标准库（如不使用 `STLport` 可不用该选项）；
- `stage` 选项指定 Boost 使用本地构建。如果使用 `install` 选项则编译后会吧 Boost 安装到默认路径下（Windows 是 `c:\boost`，Linux 下是 `/usr/local`）。

注意，对 Boost 库的完整编译所需要的时间和空间都是相当可观的，在目前主流级别 CPU 上需要数个小时，硬盘需要数十 GB 的空间，通常都会花费你半天左右的时间。这段时间里你大可关闭显示器，静下心来阅读本书的其余部分。

编译成功后可以在 `boost\bin.v2` 目录下找到生成的 `*.dll` 和 `*.lib`，拷贝到其他目录（如 `boost\vc8lib`），之后就可以把 `bin.v2` 目录删除。

部分编译 Boost

完整编译 Boost 费时费力，而且这些库并不可能在开发过程中全部用到，因此，`bjam` 也允许用户自行选择要编译的库。

执行“`bjam --show-libraries`”，可查看所有必须编译才能使用的库。在完全编译命令的基础上，使用 `--with` 或者 `--without` 选项可打开或者关闭某个库的编译，如：

```
bjam --toolset=msvc --with-date_time --build-type=complete stdlib=stlport stage
```

将仅编译 `date_time` 库。

`bjam` 还有其他很多选项，如指定安装路径、指定 `debug` 或 `release` 版等。对 `bjam` 的进一步介绍已经超出了本书的范围，读者可参考 `bjam` 的文档以获得更多信息，本书不再叙述。

Boost 库的命名规则

Boost 库在 VC 编译器下支持库自动链接技术（使用 `#pragma comment(lib, XXX)`），只要把所有生成的 `lib` 拷贝到 VC 的搜索路径下，不需要你费心，编译器会自动根据编译选项找到合适的库链接成可执行文件。

但如果读者使用的是 `GCC`、`XLC` 或者其他不支持自动链接技术的编译器，就有必要了解 Boost 库的命名规则，以便在链接时指定正确的库。

Boost 库的文件名遵循的规则十分清晰明了，基本形式如下：

libboost_filesystem-vc80-mt-sgdp-1_42.lib

前缀：统一为 lib，但在 Windows 下只有静态库有 lib 前缀；

库名称：以 “boost_” 开头的库名称，在这里是 boost_filesystem；

编译器标识：编译该库文件的编译器名称和版本，在这里是 -vc80；

多线程标识：支持多线程使用 -mt，没有表示不支持多线程；

ABI 标识：这个标识比较复杂，标识了 Boost 库的几个编译链接选项；

- s: 静态库标识
- gd: debug 版标识
- p: 使用 STLport 而不是编译器自带 STL 实现

版本号：Boost 库的版本号，小数点用下画线代替，在这里是 1_42；

扩展名：在 Windows 上是 .lib，在 Linux 等类 Unix 操作系统上是 .a 或者 .so。

在链接 Boost 程序库时可以有两种方式，一种是在编译命令行直接指明库文件全路径，另一种是用 -L 指定库文件所在路径，再用 -l 指定库文件名。

嵌入自己的工程编译

编译库的方法使用 Boost 费时费力，有时还要面临链接的烦恼。本书推荐使用作者自行实践的另一种方式：把 Boost 源代码嵌入到自己的工程中编译。

这种方法的原理类似 VC 的预编译技术（灵感来源于著名的 StdAfx.h），在工程中直接包含 Boost 库的 cpp 文件（大部分都很小很少），不但可以省略库的编译步骤，而且源代码还获得了独立于编译器、操作系统和 Boost 库版本的好处（任何一个因素发生变化时都不需要换编译器重新编译库），能够增强程序的可移植性。

不过这种方法也有小小的代价，就是每个工程都要重新编译 Boost 库，不如前两种方式那样可以（暂时的）“一劳永逸”，但作者认为其平台无关的优点还是大于缺点。在其后的章节里会向读者示范这种嵌入工程编译的方式如何具体实现，这里暂举一个小例子：

```
/// sysprebuild.cpp 一个嵌入编译源代码文件
#define BOOST_SYSTEM_NO_LIB                //禁用 Boost 的自动链接功能
#include <libs/system/src/error_code.cpp>    //包含嵌入编译源代码
```

以上代码实现了 boost.system 库的嵌入编译，读者只需要将该文件(sysprebuild.cpp)

加入到工程（或者 Makefile）中，无须使用 bjam 预先编译库就能享用 Boost 带来的好处。即使将来 Boost 程序库更新版本，也只需要简单地重新编译工程即可使用新版的函数，而无需耗费大量时间去重新编译链接 Boost 库。

在 VC 集成环境中使用嵌入工程编译的方式需要定义宏 `BOOST_ALL_NO_LIB` 或者 `BOOST_XXX_NO_LIB`（XXX 是某个库的名称），以指示 Boost 库不要使用自动链接功能。

1.3.3 VisualStudio 2005 环境设置

在编译完 STLport 和 Boost 后，还需要设置 VC 的环境选项，才能让 VC 识别 STLport 和 Boost 从而正常使用。

本书采用静态库链接、多线程、非 Unicode 的编译方式。

配置 VC8 的目录选项

打开菜单 Tools->Options，在“Projects and Solutions”的“VC++ Directories”页，选择 Include files，加入 `D:\STLport\stlport` 和 `D:\boost\`，并调到最前面；选择 Library files，加入 `D:\STLport\lib`，并调到最前面，如图 1.1 所示。

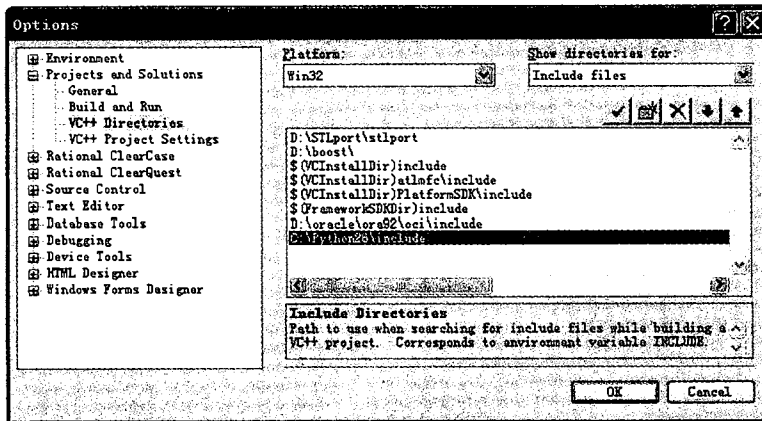


图 1.1 VC8 的目录选项界面

设置工程属性

打开菜单 Project->Properties，在“Configuration Properties”的“General”页，设置 Character Set（字符集）为 Not Set，如图 1.2 所示。

在“C/C++”的“Code Generation”页，选择 Runtime Library（运行库）为多线程（release 版是 /MT，Debug 版为 /MTd），如图 1.3 所示。

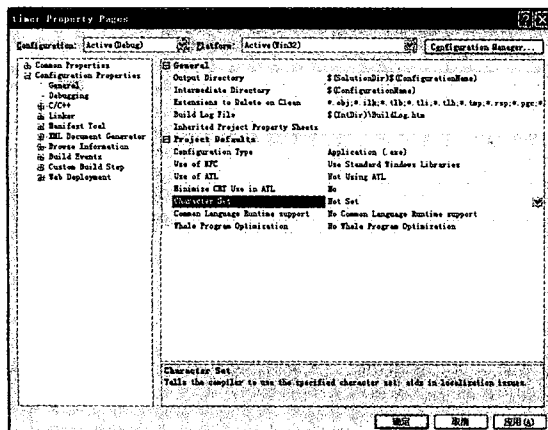


图 1.2 设置字符集为 Not Set

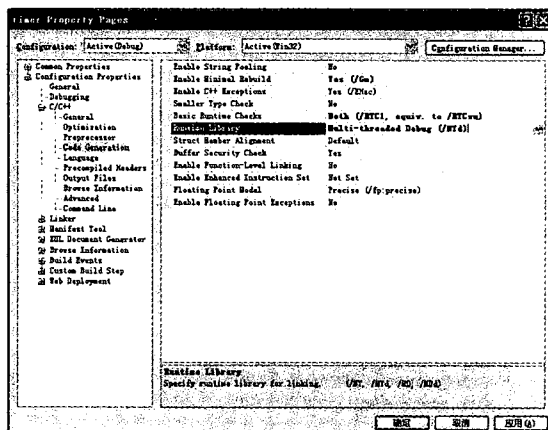


图 1.3 选择运行库为多线程

如果是 debug 版工程，不要忘记在 Preprocessor 页中定义宏 “_STLP_DEBUG” 和 “_STL_DEBUG” 以使用 STLport。

一切准备就绪，接下来，让我们正式接触 Boost。

第 2 章

时间与日期

C++一直以来缺乏对时间和日期的处理能力，而时间和日期又是现实生活中经常遇到的，C++程序员不得不求助于C，使用笨拙的结构和函数（`struct tm`、`time()`）。无法忍受这一情形的程序员则手工构造了自己的实现以满足开发所需，可以想象，有无数的程序员在这方面重复了大量的工作。

而现在，Boost使用`timer`和`date_time`库完美地解决了这个问题。

2.1 timer 库概述

`timer`是一个很小的库，提供简易的度量时间和进度显示功能，可以用于性能测试等需要计时的任务，对于大多数的情况它足够用。

`timer`库包含三个组件，分别是：计时器类 `timer`、进度计时器 `progress_timer` 和进度指示类 `progress_display`，以下将分别详述。

2.2 timer

`timer`类可以测量时间的流逝，是一个小型的计时器，提供毫秒级别的计时精度和操作函数，供程序员手工控制使用，就像是个方便的秒表。

`timer`位于名字空间`boost`，为了使用`timer`组件，需要包含头文件`<boost/timer.hpp>`，即：

```
#include <boost/timer.hpp>
using namespace boost;
```

2.2.1 用法

让我们通过一段示例代码来看一下如何使用 timer。

```
#include <boost/timer.hpp>           //timer 的头文件
using namespace boost;              //打开 boost 名字空间
int main()
{
    timer t;                        //声明一个计时器对象,开始计时
    cout << "max timespan:" << t.elapsed_max() /3600 << "h" <<endl;
                                     //可度量的最大时间,以小时为单位
    cout << "min timespan:" << t.elapsed_min() << "s" << endl;
                                     //可度量的最小时间,以秒为单位

    //输出已经流逝的时间
    cout << "now time elapsed:" << t.elapsed() <<"s" << endl;
}
```

上面的代码基本说明了 timer 的接口^①。timer 对象一旦被声明,它的构造函数就启动了计时工作,之后就可以随时用 elapsed() 函数简单地测量自对象创建后所流逝的时间。成员函数 elapsed_min() 返回 timer 测量时间的最小精度,elapsed_max() 返回 timer 能够测量的最大时间范围,两者的单位都是秒。

程序的输出如下所示:

```
max timespan:596.523h
min timespan:0.001s
now time elapsed:0.015s
```

2.2.2 类摘要

timer 类非常小,全部实现包括所有注释也不过 70 余行,真正的实现代码则只有不到 20 行。作为我们学习的第一个 Boost 组件,值得把源码全部列出来仔细研究:

```
class timer
{
public:
    timer() { _start_time = std::clock(); }
    void restart() { _start_time = std::clock(); }
    double elapsed() const
```

① 注意,在这里 main() 函数结束没有 return 0 语句,这是符合 C++ 标准的(但不符合 C 标准),本书之后的例子也都将使用这种形式。

```

    { return double(std::clock() - _start_time) / CLOCKS_PER_SEC; }

double elapsed_min() const
{ return double(1)/double(CLOCKS_PER_SEC); }
double elapsed_max() const
{
    return (double((std::numeric_limits<std::clock_t>::max)())
        - double(_start_time)) / double(CLOCKS_PER_SEC);
}

private:
    std::clock_t _start_time;
}; // timer

```

timer 的计时使用了标准库头文件<ctime>里的 `std::clock()` 函数，它返回自进程启动以来的 clock 数，每秒的 clock 数则由宏 `CLOCKS_PER_SEC` 定义。`CLOCKS_PER_SEC` 的值因操作系统而不同，在 Win32 下是 1000，而在 Linux 下则是 1000000，也就是说在 Win32 下的精度是毫秒，而 Linux 下的精度是微秒。

timer 的构造函数记录当前的 clock 数作为计时起点，保存在私有成员变量 `_start_time` 中。每当调用 `elapsed()` 时就获取此时的 clock 数，减去计时起点 `_start_time`，再除以 `CLOCKS_PER_SEC` 获得以秒为单位的已经流逝的时间。如果调用函数 `restart()`，则重置 `_start_time` 重新开始计时。

函数 `elapsed_min()` 返回 timer 能够测量的最小时间单位，是 `CLOCKS_PER_SEC` 的倒数。函数 `elapsed_max()` 使用了标准库的数值极限类 `numeric_limits`，获得 `clock_t` 类型的最大值，采用类似 `elapsed()` 的方式计算可能的最大时间范围。

timer 没有定义析构函数，这样做是正确且安全的。因为它仅有一个类型为 `clock_t` 的成员变量 `_start_time`，故没有必要实现析构函数来特意“释放资源”（也无资源可供释放）。

2.2.3 使用建议

timer 接口简单，轻巧好用，适用于大部分的程序计时任务。但使用时我们必须理解 `elapsed_min()` 和 `elapsed_max()` 这两个计时精度函数的含义，它们表明了 timer 的能力。timer 不适合高精度的时间测量任务，它的精度依赖于操作系统或编译器，难以做到跨平台。timer 也不适合大跨度时间段的测量，可提供的最大时间跨度只有几百个小时，如果需要以天、月甚至年作为时间的单位则不能使用 timer，应转向 2.5 小节，27 页的 `date_time` 库。



2.3 progress_timer

progress_timer 也是一个计时器，它继承自 timer，会在析构时自动输出时间，省去了 timer 手动调用 elapsed() 的工作，是一个用于自动计时相当方便的小工具。

progress_timer 位于名字空间 boost，为了使用 progress_timer 组件，需要包含头文件 <boost/progress.hpp>，即：

```
#include <boost/progress.hpp>
using namespace boost;
```

2.3.1 用法

progress_timer 继承了 timer 的全部能力，可以如 timer 那样使用，例如：

```
progress_timer t;           //声明一个 progress_timer 对象
...                         //任意计算、处理工作
cout << t.elapsed() << endl; //输出流逝的时间
```

但它有更简单的用法，不需要任何的调用，只要声明 progress_timer 对象就可以了：

```
#include <boost/progress.hpp>
int main()
{
    boost::progress_timer t; //声明对象开始计时
    //do something ...
}                             //退出作用域，调用 progress_timer 的析构函数
```

这样，在程序退出（准确地说是离开 main 函数局部域）导致 progress_timer 析构时，会自动输出流逝的时间。显示输出如下：

0.23 s

如果要在一个程序中测量多个时间，可以运用花括号 {} 以限定 progress_timer 的生命期：

```
#include <boost/progress.hpp>
int main()
{
    {
        boost::progress_timer t; //第一个计时

        //do something ...
    }                             //progress_timer 在这里析构，自动输出时间
}
```

```

        boost::progress_timer t;           //第二个计时
        //do something ...
    }                                     //progress_timer 在这里析构, 自动输出时间
    ...
}

```

就这些了, 只需要声明 progress_timer 的实例就完成了所需的所有工作, 非常容易使用。有了 progress_timer, 程序员今后在做类似性能测试等计算时间的工作时将会感到轻松很多。

2.3.2 类摘要

progress_timer 的类摘要如下:

```

class progress_timer : public timer, noncopyable
{
public:
    explicit progress_timer();
    progress_timer( std::ostream& os );
    ~progress_timer();
};

```

progress_timer 继承自 timer, 因此它的接口与 timer 相同, 也很简单。唯一需要注意的是构造函数 progress_timer(std::ostream& os), 它允许将析构时的输出定向到指定的 IO 流里, 默认是 std::cout。如果有特别的需求, 可以用其他标准库输出流 (ofstream、ostringstream) 替换, 或者用 cout.rdbuf() 重定向 cout 的输出 (但需记得及时恢复 cout 的状态以免发生未知错误, 可参考 10.1 小节, 371 页的 io_state_savers)。

例如, 下面的代码把 progress_timer 的输出转移到了 stringstream 中, 它可以被转换为字符串供其他应用使用:

```

stringstream ss;                                     //一个字符串流对象
{
    progress_timer t(ss);                             //要求 progress_timer 输出到 ss 中
}                                                     //progress_timer 在这里析构, 自动输出时间
cout << ss.str();

```

2.3.3 扩展计时精度

progress_timer 用于计时非常方便, 但可惜的是它的对外输出精度只有小数点后两位, 即只精确到百分之一秒。但实际上 timer 的内部计时 _start_time 是个 double 值, 从 timer.elapsed_min() 可以知道, 在 Win32 平台上精确度最小能够达到毫秒, Linux 更可以精确到微秒。

如果需要更高的时间精度，可以直接修改 `progress_timer` 代码，把输出精度改为所需的值，但这不是一个好主意，原则上程序库的代码是不应该被用户修改的。合理的做法是使用“开一闭”原则扩展 `progress_timer`。

“开一闭”原则的思想是对扩展开放而对修改关闭，但很遗憾，`progress_timer` 也并没有为扩展而预留发挥的空间。因此，我们只能使用模板技术仿造 `progress_timer` 编写一个新类：`new_progress_timer`，以实现任意精度的输出。

`new_progress_timer` 的实现基本复制了 `progress_timer` 的代码。它同样继承自 `timer`，只是变成了模板类。模板参数 `N` 指明了输出精度，默认值是 2，与 `progress_timer` 相同。

```
/// new_progress_timer.hpp
#include <boost/progress.hpp>
#include <boost/static_assert.hpp>

///使用模板参数实现 progress_timer
template<int N = 2 >
class new_progress_timer:public boost::timer
{
public:
```

`new_progress_timer` 的构造函数初始化输出流 `m_os`，并使用了另一个 Boost 组件 `static_assert`（静态断言）保证 `N` 的取值范围在 0 和 10 之间（虽然在 Win32 平台大于 3 已经没有意义）。关于 `static_assert` 具体用法，可参考 6.2 小节，217 页。

```
new_progress_timer(std::ostream & os = std::cout)
: m_os(os)
{
    BOOST_STATIC_ASSERT(N >= 0 && N <= 10);           //静态断言
}
```

`new_progress_timer` 的析构函数代码是其核心功能，处理流程也与 `progress_timer` 基本相同：保存 IO 流状态，然后设定输出精度，完成输出后恢复 IO 流的状态。IO 流的状态保存与恢复工作比较麻烦，但又不得不做，本书之后会使用 `boost.io_state_saver` 库（将在 10.1 小节，371 页评述）来简化这段代码。

```
~new_progress_timer(void)
{
    try
    {

        //保存流的状态
        std::istream::fmtflags old_flags
```

```

    = m_os.setf( std::istream::fixed, std::istream::floatfield );
    std::streamsize old_prec = m_os.precision( N );

    //输出时间
    m_os << elapsed() << " s\n"           //"s" 表示秒
        << std::endl;

    //恢复流状态
    m_os.flags( old_flags );
    m_os.precision( old_prec );
}
catch (...) {} //析构函数绝对不能抛出异常, 非常重要
}
private:
    std::ostream & m_os;
};

```

代码的最后我们提供一个 `new_progress_timer` 的模板特化实现, 对于精度为 2 的 `new_progress_timer`, 直接继承自 `progress_timer`。

```

///使用模板特化
template<>
class new_progress_timer<2>:public boost::progress_timer
{};

```

使用 `new_progress_timer`, 之前的例子可以改成如下:

```

#include "new_progress_timer.h"
int main()
{
    new_progress_timer<10> t;
    //do something ...
}

```

程序运行结果如下:

```
0.1500000000 s
```

如果用户还想使用原来的 `progress_timer`, 只需要把 `t` 的模板参数改为 2, 即 `new_progress_timer<2>`。或者不提供值, 使用默认的模板参数, 那样更简单, 也就是 `new_progress_timer<>`。

2.4 progress_display

`progress_display` 可以在控制台上显示程序的执行进度, 如果程序执行很耗费时间, 那么它能够提供一个友好的用户界面, 不至于让用户在等待中失去耐心、甚至怀疑程序的运行是否

出了问题。

`progress_display` 位于名字空间 `boost`，为了使用 `progress_display` 组件，需要包含头文件 `<boost/progress.hpp>`，即：

```
#include <boost/progress.hpp>
using namespace boost;
```

2.4.1 类摘要

`progress_display` 是一个独立的类，与 `timer` 库中的其他两个组件——`timer` 和 `progress_timer` 没有任何联系，它的类摘要如下：

```
class progress_display : noncopyable
{
public:
    progress_display( unsigned long expected_count );
    progress_display( unsigned long expected_count,
                     std::ostream& os,
                     const std::string & s1 = "\n",
                     const std::string & s2 = "",
                     const std::string & s3 = "" );
    void restart( unsigned long expected_count );
    unsigned long operator+=( unsigned long increment );
    unsigned long operator++() ;
    unsigned long count() const ;
    unsigned long expected_count() const;
};
```

`progress_display` 的构造函数接受一个 `long` 型的参数 `expected_count`，表示用于进度显示的基数，是最常用的创建 `progress_display` 的方法。

另一种形式的构造函数除了基数和流输出对象外，还接受三个字符串参数，定义显示的三行首字符串。但这个构造函数有点小问题，流输出对象通常都应该是 `cout`，把进度输出到文件或者其他用户看不到的地方似乎没有多大的意义。

在构造后 `progress_display` 对象会显示出一个类似 Windows 进度条的界面，用两行以字符的形式显示百分比进度，像这样：

```
0%   10   20   30   40   50   60   70   80   90  100%
|----|----|----|----|----|----|----|----|----|
```

重载的加法操作符 `operator+=` 和 `operator++` 用来增加计数，并在第三行用字符 `*` 显示进度百分比 `count()/expected_count`。成员函数 `count()` 可以返回当前的计数，当计数到达基数 `expected_count` 时表示任务已经完成，进度为 100%。

2.4.2 用法

假设我们要把一些存储在 `std::vector` 中的字符串以每个一行的形式写入到文本文件中，因为字符串数量很多，而且有的很大，因此可能操作会耗费很多时间。`progress_display` 可以让程序有一个友好的人机界面，让用户知道程序的进度。

使用 `progress_display` 首先要调用它的构造函数，传入进度基数，如：

```
progress_display pd(v.size());
```

随后就可以使用它重载的加法操作符 (`++pd`) 来根据程序的运行情况增加计数，直到完成任务。

示范 `progress_display` 用法的代码如下：

```
#include <boost/progress.hpp>
using namespace boost;
int main()
{
    vector<string> v(100);                //一个字符串向量
    ofstream fs("c:\\test.txt");          //文件输出流

    //声明一个 progress_display 对象,基数是 v 的大小
    progress_display pd(v.size());

    //开始迭代遍历向量,处理字符串,写入文件
    vector<string>::iterator pos;
    for (pos = v.begin(); pos != v.end(); ++pos)
    {
        fs << *pos << endl;
        ++pd;                             //更新进度显示
    }
}
```

当程序处理了 75 个字符串时，控制台的显示大概如下：

```
0%   10   20   30   40   50   60   70   80   90  100%
|----|----|----|----|----|----|----|----|----|
*****
```

如果改用 `progress_display` 另一种形式的构造函数，如：

```
progress_display pd(v.size(),cout, "%%%", "+++", "???");
```

那么显示就会变成这样：

```
%%%0%   10   20   30   40   50   60   70   80   90  100%
+++|----|----|----|----|----|----|----|----|----|
???*****
```

这种输出形式颇有点“画蛇添足”的味道，建议读者最好不要这么用。

在本书的 10.3.13 小节 (397 页) 有 `progress_display` 的另一个使用例子，读者可以参考。

2.4.3 注意事项

`progress_display` 可以用作基本的进度显示，但它有个固有的缺陷：无法把进度显示输出与程序的输出分离。

这是因为 `progress_display` 和所有 C++ 程序一样，都向标准输出 (cout) 输出字符，如果使用 `progress_display` 的程序也有输出操作，那么 `progress_display` 的进度显示就会一片混乱。

仍然使用刚才的写入文本文件的例子，我们希望显示进度的同时还能报告空的字符串的行号，像这样：

```
0%   10   20   30   40   50   60   70   80   90  100%
|---|---|---|---|---|---|---|---|---|---|
*****
null string # 10
null string # 23
...
```

代码实现如下：

```
#include <boost/progress.hpp>
int main()
{
    vector<string> v(100, "aaa");

    v[10] = ""; v[23] = "";
    ofstream fs("c:\\test.txt");
    progress_display pd(v.size());
    vector<string>::iterator pos;
    for (pos = v.begin(); pos != v.end(); ++pos)
    {
        fs << *pos << endl;
        ++pd;
        if (pos->empty())
        {
            cout << "null string # "
                  << (pos - v.begin()) << endl;
        }
    }
}
```

似乎一切都很好，但实际的运行结果却是：

```

0%   10   20   30   40   50   60   70   80   90  100%
|----|----|----|----|----|----|----|----|----|
*****null string # 10
*****null string # 23
*****

```

很显然，for 循环中的输出打乱了 progress_display 的正常输出，而 progress_display 对此毫不知情，仍然按部就班地显示着百分比进度。

这个显示混乱的问题很难解决，因为我们无法预知庞大的程序之中哪个地方会存在一个可能会干扰 progress_display 的输出。一个可能（但远非完美）的办法是在每次显示进度时都调用 restart() 重新显示进度刻度，然后用 operator+= 来指定当前进度，而不是简单地调用 operator++，例如：

```

pd.restart(v.size());
pd += (pos - v.begin() + 1);

```

2.5 date_time 库概述

日期和时间在程序中就像整数和字符串一样经常出现，被作为一种基础设施广泛地用在很多地方，例如作为随机数的种子值。但精确并准确地操纵时间非常困难，因为时间本身是一个难以度量的实体，它有许多变化。

我们使用的基本时间度量依据地球的自转，但地球的自转是不均匀的（有时候非常剧烈的地质运动也会影响地球的自转），需要用闰秒、闰月和闰年进行调整，不同的地区还有夏令时、时区等的人为规定。现实生活中存在着很多个时间度量体系，如儒勒历、格里高利历、农历、印加帝国的太阳历、UTC、ISO、国际原子时间等等，非常复杂。想要实现一个可以计算各种时间日期相关问题的库即使不是不可能的，难度也将是非常之大。

date_time 库勇敢地面对了这个挑战，并成功地解决了大部分问题。它是一个非常全面且灵活的日期时间库，基于我们日常使用的公历（即格里高利历），可以提供时间相关的各种所需功能，如精确定义的时间点、时间段和时间长度、加减若干天/月/年、日期迭代器等等。date_time 库还支持无限时间和无效时间这种实际生活中有用的概念，而且可以与 C 的传统时间结构 tm 相互转换，提供向下支持。

2.5.1 编译 date_time 库

date_time 库是 Boost 中少数需要编译的库之一。

直接编译 date_time 需运行的 bjam 命令如下：

```
bjam --toolset=msvc --with-date_time --build-type=complete stdlib=stdlibport stage
```

编译后会生成形如 `libboost_date_time-vc80-mt-sgdp-1_42.lib` 的静态库或者动态库。

如不想使用 `bjam` 预编译库，可以直接在工程内包含 `date_time` 库的实现 `cpp` 源文件，使用嵌入工程编译的方式。`cpp` 源代码如下：

```
///dateprebuild.cpp
#define BOOST_DATE_TIME_SOURCE

#include <libs/date_time/src/gregorian/greg_names.hpp>
#include <libs/date_time/src/gregorian/date_generators.hpp>
#include <libs/date_time/src/gregorian/greg_month.hpp>
#include <libs/date_time/src/gregorian/greg_weekday.hpp>
#include <libs/date_time/src/gregorian/gregorian_types.hpp>
```

在工程的其他源代码使用 `date_time` 库时，需要在包含头文件之前定义宏 `BOOST_DATE_TIME_SOURCE`、`BOOST_DATE_TIME_NO_LIB` 或者 `BOOST_ALL_NO_LIB`。例如：

```
#define BOOST_DATE_TIME_SOURCE
#include <boost/date_time/gregorian/gregorian.hpp>
```

2.5.2 date_time 库的基本概念

时间的处理很复杂，因此在使用 `date_time` 库之前，我们需要明确一些基本概念。

如果把时间想象成一个向前和向后都无限延伸的实数轴，那么时间点就是数轴上的一个点，时间段就是两个时间点之间确定的一个区间，时长（时间长度）则是一个有正负号的标量，它是两个时间点之差，不属于数轴。

时间点、时间段和时长三者之间可以进行运算，例如时间点+时长=时间点，时长+时长=时长，时间段 \cap 时间段=时间段、时间点 \in 时间段等等，但有的运算也是无意义的，如时间点+时间点、时长+时间段等等。这些运算都基于生活常识，很容易理解，但在编写时间处理程序时必须注意。

`date_time` 库支持无限时间和无效时间（NADT, Not Available Date Time）这样特殊的时间概念，类似于数学中极限的涵义。时间点和时长都有无限的值，它们的运算规则比较特别，例如 $+\infty$ 时间点+时长 $=+\infty$ 时间点，时间点 $+\infty$ 时长 $=+\infty$ 时间点。如果正无限值与负无限值一起运算将有可能是无效时间，如 $+\infty$ 时长 $-\infty$ 时长=NADT。

`date_time` 库中用枚举 `special_values` 定义了这些特殊的时间概念，它位于名字空间 `boost::date_time`，并被 `using` 语句引入其他子名字空间：

■ `pos_infin`: 表示正无限；

- `neg_infin` : 表示负无限;
- `not_a_date_time`: 无效时间;
- `min_date_time` : 可表示的最小日期或时间;
- `max_date_time` : 可表示的最大日期或时间。

2.6 处理日期

我们首先研究 `date_time` 库的日期处理部分, 因为日期只涉及年月日, 较时间少处理时分秒三个量, 相当于数轴上的整数, 要容易学习一些。

`date_time` 库的日期基于格里高利历, 支持从 1400-01-01 到 9999-12-31 之间的日期计算 (很遗憾, 它不能处理公元前的日期, 不能用它来研究古老的历史)。它位于名字空间 `boost::gregorian`, 为了使用 `date_time` 库的日期功能, 需要包含头文件 `<boost/date_time/gregorian/gregorian.hpp>`, 即:

```
#define BOOST_DATE_TIME_SOURCE
#include <boost/date_time/gregorian/gregorian.hpp>
using namespace boost::gregorian;
```

2.6.1 日期

`date` 是 `date_time` 库处理日期的核心类, 使用一个 32 位的整数作为内部存储, 以天为单位表示时间点概念。它的类摘要如下:

```
template<typename T, typename calendar, typename duration_type_>
class date {
public:

    //构造函数
    date(year_type, month_type, day_type);
    date(const ymd_type &);

    //基本操作函数
    year_type year() const;
    month_type month() const;
    day_type day() const;
    day_of_week_type day_of_week() const;
    ymd_type year_month_day() const;
    bool operator<(const date_type &) const;
    bool operator==(const date_type &) const;
```

```

bool is_special() const;
bool is_not_a_date() const;
bool is_infinity() const;
bool is_pos_infinity() const;
bool is_neg_infinity() const;

special_values as_special() const;
duration_type operator-(const date_type &) const;

//...其他日期运算
};

```

`date` 是一个轻量级的对象，很小，处理效率很高，可以被拷贝传值。`date` 也全面支持比较操作和流输入输出，因此我们完全可以把它当作是一个像 `int`、`string` 那样的基本类型来使用。

2.6.2 创建日期对象

有很多种方式可以创建一个日期对象。

空的构造函数会创建一个值为 `not_a_date_time` 的无效日期^①；顺序传入年月日值则创建一个对应日期的 `date` 对象，例如：^②

```

date d1;                                //一个无效的日期
date d2(2010,1,1);                      //使用数字构造日期
date d3(2000, Jan , 1);                 //也可以使用英文指定月份
date d4(d2);                            //date 支持拷贝构造

assert(d1 == date(not_a_date_time));    //比较一个临时对象
assert(d2 == d4);                      //date 支持比较操作
assert(d3 < d4);

```

`date` 也可以从一个字符串产生，这需要使用工厂函数 `from_string()` 和 `from_undelimited_string()`。前者使用分隔符（斜杠或者连字符）分隔年月日格式的字符串，后者则是无分隔符的纯字符串格式。例如：

```

date d1 = from_string("1999-12-31");
date d2 ( from_string("2005/1/1") );
date d3 = from_undelimited_string("20011118") ;

```

① 如果不希望 `date` 的缺省构造出无效日期，则可以在包含头文件前定义宏 `DATE_TIME_NO_DEFAULT_CONSTRUCTOR`，从而禁止缺省构造函数。

② 从本小节开始，本书中的示例代码将大量使用 `assert` 宏，用来断言表达式成立。如果读者对 `assert` 宏不够了解，可以参阅 6.1 小节，231 页。

`day_clock` 是一个天级别的时钟,它也是一个工厂类,调用它的静态成员函数 `local_day()` 或 `universal_day()` 会返回一个当天的日期对象,分别是本地日期和 UTC 日期。`day_clock` 内部使用了 C 标准库的 `localtime()` 和 `gmtime()` 函数,因此 `local_day()` 的行为依赖于操作系统的时区设置。例如:

```
cout << day_clock::local_day() << endl;
cout << day_clock::universal_day() << endl;
```

我们也可以使用特殊时间概念枚举 `special_values` 创建一些特殊的日期,在处理如无限期的某些情形下会很有用:

```
date d1(neg_infin);           //负无限日期
date d2(pos_infin);          //正无限日期
date d3(not_a_date_time);    //无效日期
date d4(max_date_time);      //最大可能日期 9999-12-31
date d5(min_date_time);      //最小可能日期 1400-01-01
```

使用 `cout` 将它们输出,显示将会是:

```
-infinity
+infinity
not-a-date-time
9999-Dec-31
1400-Jan-01
```

如果创建日期对象时使用了非法的值,例如日期超过了 1400-01-01 到 9999-12-31 的范围、使用不存在的月份或日期,那么 `date_time` 库会抛出异常(而不是转换为一个无效日期),可以使用 `what()` 获得具体的错误信息。

下面的 `date` 对象构造均会抛出异常:

```
date d1(1399,12,1);          //超过日期下限
date d2(10000,1,1);          //超过日期上限
date d3(2010,2,29);          //不存在的日期
```

2.6.3 访问日期

`date` 类的对外接口很像 C 语言中的 `tm` 结构,也可以获取它保存的年、月、日、星期等成分,但 `date` 提供了更多的操作。

成员函数 `year()`、`month()` 和 `day()` 分别返回日期的年、月、日:

```
date d(2010,4,1);
assert(d.year() == 2010);
assert(d.month() == 4);
assert(d.day() == 1);
```

成员函数 `year_month_day()` 返回一个 `date::ymd_type` 结构, 可以一次性地获取年月日数据:

```
date::ymd_type ymd = d.year_month_day();
assert(ymd.year == 2010);
assert(ymd.month == 4);
assert(ymd.day == 1);
```

成员函数 `day_of_week()` 返回 `date` 的星期数, 0 表示星期天。`day_of_year()` 返回 `date` 是当年的第几天 (最多是 366)。`end_of_month()` 返回当月的最后一天的 `date` 对象:

```
assert(d.day_of_week() == 4);
assert(d.day_of_year() == 91);
assert(d.end_of_month() == date(2010, 4, 30));
```

成员函数 `week_number()` 返回 `date` 所在的周是当年的第几周, 范围是 0 至 53。如果年初的几天位于去年的周那么则周数为 53, 即第 0 周:

```
assert(date(2010, 1, 10).week_number() == 1 );
assert(date(2010, 1, 1).week_number() == 53 );
assert(date(2008, 1, 1).week_number() == 1);
```

`date` 还有五个 `is_xxx()` 函数, 用于检验日期是否是一个特殊日期, 它们是:

- `is_infinity()` : 是否是一个无限日期;
- `is_neg_infinity()` : 是否是一个负无限日期;
- `is_pos_infinity()` : 是否是一个正无限日期;
- `is_not_a_date()` : 是否是一个无效日期;
- `is_special()` : 是否是任意一个特殊日期。

它们的用法如下:

```
assert(date(pos_infin).is_infinity() );
assert(date(pos_infin).is_pos_infinity() );
assert(date(neg_infin).is_neg_infinity() );
assert(date(not_a_date_time).is_not_a_date() );
assert(date(not_a_date_time).is_special() );
assert(!date(2010, 10, 1).is_special() );
```

2.6.4 日期的输出

`date` 对象可以很方便地转换成字符串, 它提供了三个自由函数:

- `to_simple_string(date d)`: 转换为 YYYY-mm-DD 格式的字符串, 其中的 mm 为 3 字符的英文月份名;
- `to_iso_string(date d)`: 转换为 YYYYMMDD 格式的数字字符串;
- `to_iso_extended_string(date d)`: 转换为 YYYY-MM-DD 格式的数字字符串。

`date` 也支持流输入输出, 默认使用 YYYY-mm-DD 格式。例如:

```
date d(2008,11,20);

cout << to_simple_string(d) << endl;
cout << to_iso_string(d) << endl;
cout << to_iso_extended_string(d) << endl;
cout << d << endl;

cin >> d;
cout << d;
```

程序的运行结果如下:

```
2008-Nov-20
20081120
2008-11-20
2008-Nov-20
2010-Jan-02 (用户的输入)
2010-Jan-02
```

2.6.5 与 tm 结构的转换

`date` 支持与 C 标准库中的 `tm` 结构相互转换, 转换的规则和函数如下:

- `to_tm(date)`: `date` 转换到 `tm`。 `tm` 的时分秒成员 (`tm_hour`, `tm_min`, `tm_sec`) 均置为 0, 夏令时标志 `tm_isdst` 置为 -1 (表示未知)。
- `date_from_tm(tm datetm)`: `tm` 转换到 `date`。只使用年、月、日三个成员 (`tm_year`, `tm_mon`, `tm_mday`), 其他成员均被忽略。

下面的代码示范了 `date` 与 `tm` 的相互转换:

```
date d(2010,2,1);
tm t = to_tm(d);
assert(t.tm_hour == 0 && t.tm_min == 0);
assert(t.tm_year == 110 && t.tm_mday == 1);

date d2 = date_from_tm(t);
assert(d == d2);
```

2.6.6 日期长度

日期长度是以天为单位的时长，是度量时间长度的一个标量。它与日期不同，值可以是任意的整数，可正可负。基本的日期长度类是 `date_duration`，它的类摘要如下：

```
class date_duration
{
public:

    //构造函数
    date_duration(long);
    date_duration(special_values);

    //成员访问函数
    long days() const;
    bool is_special() const;
    bool is_negative() const;

    bool operator==(const date_duration &) const;

    //...其他操作符定义

    static date_duration unit() ;
};
```

`date_duration` 可以使用构造函数创建一个日期长度，成员函数 `days()` 返回时长的天数，如果传入特殊时间枚举值则构造出一个特殊时长对象。`is_special()` 和 `is_negative()` 可以判断 `date_duration` 对象是否为特殊值、是否是负值。`unit()` 返回时长的最小单位，即 `date_duration(1)`。

`date_duration` 支持全序比较操作（`==`、`!=`、`<`、`<=`等），也支持完全的加减法和递增递减操作，用起来很像一个整数。此外 `date_duration` 还支持除法运算，可以除以一个整数，但不能除以另一个 `date_duration`，其他的数学运算如乘法、取模、取余则不支持。

`date_time` 库为 `date_duration` 定义了一个常用的 `typedef: days`，这个新名字更好地说明了 `date_duration` 的含义——它是一个天数的计量。

示范 `days(date_duration)` 用法的代码如下：

```
days dd1(10), dd2(-100), dd3(255);

assert(dd1 > dd2 && dd1 < dd3);
assert(dd1 + dd2 == days(-90));
assert((dd1 + dd3).days() == 265);
```

```
assert(dd3 / 5 == days(51));
```

为了方便计算时间长度, `date_time` 库还提供了 `months`、`years`、`weeks` 等另外三个时长类, 分别用来表示 n 个月、 n 个年和 n 个星期, 它们的含义与 `days` 类似, 但行为不太相同。

`months` 和 `years` 全面支持加减乘除运算, 使用成员函数 `number_of_months()` 和 `number_of_years()` 可获得表示的月数和年数。`weeks` 是 `date_duration` 的子类, 除了构造函数以 7 为单位外其他的行为与 `days` 完全相同, 可以说是一个 `days` 的近义词。

示范这三个时长类基本用法的代码如下:

```
weeks w(3); //3 个星期
assert(w.days() == 21);

months m(5); //5 个月
years y(2); //2 年

months m2 = y + m; //2 年零 5 个月
assert(m2.number_of_months() == 29);
assert((y * 2).number_of_years() == 4);
```

2.6.7 日期运算

`date` 支持加减运算, 两个 `date` 对象的加操作是无意义的 (`date_time` 库会以编译错误的方式通知我们), `date` 主要是与时长概念配合运算。

例如, 下面的代码计算了从 2000 年 1 月 1 日到 2008 年 8 月 8 日的天数, 并执行其他的日期运算:

```
#define BOOST_DATE_TIME_SOURCE
#include <boost/date_time/gregorian/gregorian.hpp>
using namespace boost::gregorian;
int main()
{
    date d1(2000,1,1), d2(2008,8,8);
    cout << d2 - d1 << endl; //3142 天
    assert(d1 + (d2 - d1) == d2);

    d1 += days(10); //2000-1-11
    assert(d1.day() == 11);
    d1 += months(2); //2000-3-11
    assert(d1.month() == 3 && d1.day() == 11);
    d1 -= weeks(1); //200-3-4
    assert(d1.day() == 4);

    d2 -= years(7); //2001-8-8
```

```

    assert(d2.year() == d1.year() + 1);
}

```

日期与特殊日期长度、特殊日期与日期长度进行运算的结果也会是特殊日期:

```

date d1(2010,1,1);

date d2 = d1 + days(pos_infin);
assert(d2.is_pos_infinity());

d2 = d1 + days(not_a_date_time);
assert(d2.is_not_a_date());
d2 = date(neg_infin);
days dd = d1 - d2;
assert(dd.is_special() && !dd.is_negative());

```

在与 months、years 这两个时长类进行计算时要注意: 如果日期是月末的最后一天, 那么加減月或年会得到同样的月末时间, 而不是简单的月份或者年份加 1, 这是合乎生活常识的。但当天数是月末的 28、29 时, 如果加減月份到 2 月份, 那么随后的运算就总是月末操作, 原来的天数信息就会丢失。例如:

```

date d(2010,3,30);
d -= months(1);           //2010-2-28, 变为月末, 原 30 的日期信息丢失
d -= months(1);           //2010-1-31
d += months(2);           //2010-3-31
assert(d.day() == 31);    //与原来日期不相等

```

使用 days 不会出现这样的问题, 如果担心 weeks、months、years 这些时长类被无意使用进而扰乱了代码, 可以 undef 宏 BOOST_DATE_TIME_OPTIONAL_GREGORIAN_TYPES, 这将使 date_time 库不包含它们的定义头文件 <boost/date_time/date_duration_types.hpp>。

2.6.8 日期区间

date_time 库使用 date_period 类来表示日期区间的概念, 它是时间轴上的一个左闭右开区间, 端点是两个 date 对象。区间的左值必须小于右值, 否则 date_period 将表示一个无效的日期区间。

date_period 的类摘要如下:

```

class date_period
{
public:
    period(date, date);
    period(date, days);

```

```

date begin() const;
date end() const;
date last() const;
days length() const;
bool is_null() const;

bool operator==(const period &) const;
bool operator<(const period &) const;

void shift(const days &) ;
void expand(const days &) ;

bool contains(const date &) const;
bool contains(const period &) const;
bool intersects(const period &) const;
bool is_adjacent(const period &) const;
bool is_before(const date &) const;
bool is_after(const date &) const;
period intersection(const period &) const;
period merge(const period &) const;
period span(const period &) const;
};

```

`date_period` 可以指定区间的两个端点构造，也可以指定左端点再加上时长构造，通常后一种方法比较常用，相当于生活中从某天开始的一个周期。例如：

```

date_period dp1(date(2010,1,1), days(20));
date_period dp2(date(2010,1,1), date(2009,1,1));           //无效
date_period dp3(date(2010,3,1), days(-20));                 //无效

```

成员函数 `begin()` 和 `last()` 返回日期区间的两个端点，而 `end()` 返回 `last()` 后的第一天，与 STL 中的 `end()` 含义相同，是一个“逾尾的位置”。`length()` 返回日期区间的长度，以天为单位。如果日期区间在构造时使用了左大右小的端点或者日期长度是 0，那么 `is_null()` 函数将返回 `true`。例如：

```

date_period dp(date(2010,1,1), days(20));

assert(!dp.is_null());
assert(dp.begin().day() == 1);
assert(dp.last().day() == 20);
assert(dp.end().day() == 21);
assert(dp.length().days() == 20);

```

`date_period` 可以进行全序比较运算，但比较不是依据日期区间的长度，而是依据区间的端点，即第一个区间的 `end()` 和第二个区间的 `begin()`，判断两个区间在时间轴上的位置大小。如果两个日期区间相交或者包含，那么比较操作无意义。

`date_period` 也支持输入输出操作符, 默认的输入输出格式是一个 `[YYYY-mm-DD/YYYY-mm-DD]` 形式的字符串。例如:

```
date_period dp1(date(2010,1,1), days(20));
date_period dp2(date(2010,2,19), days(10));

cout << dp1;                      //[2010-Jan-01/2010-Jan-20]
assert(dp1 < dp2);
```

2.6.9 日期区间运算

`date_period` 同 `date`、`days` 一样, 也支持很多运算。

成员函数 `shift()` 和 `expand()` 可以变动区间: `shift()` 将日期区间平移 `n` 天而长度不变, `expand()` 将日期区间向两端延伸 `n` 天, 相当于区间长度加 `2n` 天。例如:

```
date_period dp(date(2010,1,1), days(20));

dp.shift(days(3));
assert(dp.begin().day() == 4);
assert(dp.length().days() == 20);

dp.expand(days(3));
assert(dp.begin().day() == 1);
assert(dp.length().days() == 26);
```

`date_period` 还可以使用成员函数判断某个日期是否在区间内, 或者计算日期区间的交集:

- `is_before()`、`is_after()`: 日期区间是否在日期前或后;
- `contains()`: 日期区间是否包含另一个区间或者日期;
- `intersects()`: 两个日期区间是否存在交集;
- `intersection()`: 返回两个区间的交集, 如果无交集返回一个无效区间;
- `is_adjacent()`: 两个日期区间是否相邻;

示范这几个成员函数用法的代码如下:

```
date_period dp(date(2010,1,1), days(20));          //1-1 至 1-20

assert(dp.is_after(date(2009,12,1)));
assert(dp.is_before(date(2010,2,1)));
assert(dp.contains(date(2010,1,10)));

date_period dp2(date(2010,1,5), days(10));         //1-5 至 1-15
```

```
assert(dp.contains(dp2));

assert(dp.intersects(dp2));
assert(dp.intersection(dp2) == dp2);

date_period dp3(date(2010,1,21), days(5));          //1-21 至 1-26
assert(!dp3.intersects(dp2));
assert(dp3.intersection(dp2).is_null());

assert(dp.is_adjacent(dp3));
assert(!dp.intersects(dp3));
```

date_period 提供了两种并集操作：

- merge() : 返回两个区间的并集，如果区间无交集或者不相邻则返回无效区间；
- span() : 合并两日期区间及两者间的间隔，相当于广义的 merge()。

示范这两个并集函数的用法和区别的代码如下：

```
date_period dp1(date(2010,1,1), days(20));
date_period dp2(date(2010,1,5), days(10));
date_period dp3(date(2010,2,1), days(5));
date_period dp4(date(2010,1,15), days(10));

assert(dp1.contains(dp2) && dp1.merge(dp2) == dp1);
assert(!dp1.intersects(dp3) && dp1.merge(dp3).is_null());
assert(dp1.intersects(dp2) && dp1.merge(dp4).end() == dp4.end());
assert(dp1.span(dp3).end() == dp3.end());
```

2.6.10 日期迭代器

date_time 库为日期处理提供了迭代器的概念，可以用简单的递增或者递减操作符连续访问日期，这些迭代器包括 date_iterator、week_iterator、month_iterator 和 year_iterator，它们分别以天、周、月和年为单位增减。

日期迭代器的用法基本类似，都需要在构造时传入一个起始日期和增减步长（可以是一天、两周或者 N 个月等等，默认是 1 个单位），然后就可以用 operator++、operator-- 变化日期^①。迭代器相当于一个 date 对象的指针，可以用解引用操作符*获得迭代器当前的日期对象，也可以用->直接调用日期对象的成员函数。

① 出于运算效率的考虑，日期迭代器只提供前置式的++、--操作符，不提供后置式，读者应该习惯于这种高效的编码书写方式。

为了方便用户使用，日期迭代器还重载了比较操作符，不需要用解引用操作符就可以直接与其他日期对象比较大小。

示范日期迭代器基本用法的代码如下：

```
date d(2006,11,26);
day_iterator d_iter(d); //增减步长默认为1天

assert(d_iter == d);
++d_iter; //递增1天
assert(d_iter == date(2006,11,27));

year_iterator y_iter(*d_iter, 3); //增减步长为3年
assert(y_iter == d + days(1));
++y_iter; //递增3年
assert(y_iter->year() == 2009);
```

2.6.11 其他功能

类 `boost::gregorian::gregorian_calendar` 提供了格里高利历的一些操作函数，基本上它被 `date` 类在内部使用，用户通常很少用到。但它也提供了几个有用的静态函数：成员函数 `is_leap_year()` 可以判断年份是否是闰年，`end_of_month_day()` 给定年份和月份，返回该月的最后一天。例如：

```
typedef gregorian_calendar gre_cal; //typedef 以简化代码书写
cout << "Y2010 is "
    << (gre_cal::is_leap_year(2010)?"":"not")
    << " a leap year." << endl;
assert(gre_cal::end_of_month_day(2010, 2) == 28);
```

`date_time` 库还提供了很多有用的日期生成器，如某个月的最后一个星期天或者第一个星期一等，它们封装了一些常用但计算起来又比较麻烦的时间概念，限于篇幅本书不做过多的介绍。

2.6.12 综合运用

`date_time` 库比较复杂，有很多的概念和对应的类，因为日期的处理本身就存在很多复杂的因素。本小节将综合运用日期相关的所有类，给出一些具体的使用例子。

显示月历

首先我们实现一个打印月历的功能。用户指定一个日期，可以得到该月的起始和结束日期，然后我们构造一个日期迭代器，使用 `for` 循环打印出日期。

```
date d(2008,11,20); //实际运行时日期可以从cin获得
```



```

date d_start(d.year(), d.month(), 1);           //当月第一天
date d_end = d.end_of_month();                 //当月最后一天

for(day_iterator d_iter(d_start);              //构造日期迭代器
    d_iter != d_end; ++d_iter)                 //循环结束条件
{
    cout << *d_iter << " " <<               //输出日期和星期
        d_iter->day_of_week() << endl;
}

```

简单的日期计算

下面的程序计算一个人十八岁的生日是星期几，当月有几个星期天，当年有多少天：

```

date d(2008,11,20);                             //声明日期对象

date d18years = d + years(18);                   //加上 18 年
cout << d18years << " is "
    << d18years.day_of_week() << endl;

int count = 0;                                    //星期天的计数器
for (day_iterator d_iter(date(d18years.year(),11,1));
    d_iter != d18years.end_of_month(); ++d_iter)
{
    if (d_iter->day_of_week() == Sunday)          //是星期天则计数增加
    {
        ++count;
    }
}
cout << "total " << count << " Sundays." << endl;

count = 0;                                         //计数器归 0
for (month_iterator m_iter(date(d18years.year(),1,1));
    m_iter < date(d18years.year() + 1,1,1); ++m_iter)
{
    count += m_iter->end_of_month().day();         //累加月份的天数
}
cout << "total " << count << " days of year." << endl;

```

程序的运行结果如下：

```

2026-Nov-20 is Fri
total 5 Sundays.
total 365 days of year.

```

其实计算当年的天数没有必要累加每月的天数，可以简单地只判断该年是否是闰年就可以了，例如：

```
cout << (gre_cal::is_leap_year(d18years.year())?365:366);
```

计算信用卡的免息期

最后我们实现一个较复杂的程序，计算信用卡的最长免息期。

先来简单了解一下信用卡免息期的计算规则：使用信用卡的当天称为消费日，信用卡每月有一个记账日，在记账日之后有一个固定的免息还款期限，通常为 20 天，因此每笔信用卡交易的免息期就是消费日到下一个记账日的时间再加上还款期限，最长可以达到 50 天。

我们使用类 `credit_card` 来代表信用卡，它保存了信用卡的基本信息，包括发卡银行名和记账日：

```
class credit_card
{
public:
    string bank_name;           //银行名
    int bill_day_no;            //记账日
    credit_card(const char* bname, int no):    //构造函数
        bank_name(bname), bill_day_no(no){}
```

我们使用成员函数 `calc_free_days()` 来计算信用卡的免息期，它依据传入的消费日得到“下一个”记账日，并算出免息期：

```
int calc_free_days(date consume_day = day_clock::local_day()) const
{
    date bill_day(consume_day.year(), consume_day.month(), bill_day_no);
                                                    //得到记账日
    if (consume_day > bill_day)                    //消费日是否已经过了记账日
    {
        bill_day += months(1);                    //如果过了则是下个月的记账日
    }

    return (bill_day - consume_day).days() + 20; //计算免息期
}
```

为了支持比较操作，还需要为 `credit_card` 增加小于比较操作符重载：

```
friend bool operator<(const credit_card& l, const credit_card& r)
{
    //比较免息期
    return l.calc_free_days() < r.calc_free_days();
}
};
```

最后，我们在 `main()` 函数中创建两个信用卡对象，并使用标准库的算法 `std::max()` 来比

较这两个信用卡，从而决定使用免息期最长的那张信用卡：

```
int main()
{
    credit_card a("A bank", 25);           //A 银行记账日是每月的 25 号
    credit_card b("B bank", 12);           //B 银行记账日是每月的 12 号

    credit_card tmp = std::max(a, b);
    cout << "You should use " << tmp.bank_name
        << ", free days = " << tmp.calc_free_days() << endl;
}
```

通过这个程序，我们可以知道：如果今天是 5 日，那么应该使用 A 银行的卡，它的免息期是 40 天；如果今天是 15 日，那么应该使用 B 银行的卡，它的免息期是 48 天。

`credit_card` 的 `calc_free_days()` 函数还可以直接传入指定的日期，以计算任意时间点的免息期，例如：

```
cout << a.calc_free_days(date(2010,5,26));
```

读者可以进一步改进这个程序，把 `credit_card` 放入标准容器，使用 `std::sort()` 算法来管理更多的信用卡（提示：使用函数对象保存消费日期，作为比较谓词传递给算法）。

2.7 处理时间

`date_time` 库在格里高利历的基础上提供微秒级别的时间系统，但如果需要，它最高可以达到纳秒级别的精确度。

`date_time` 库的时间功能位于名字空间 `boost::posix_time`，为了使用时间组件，需要包含头文件 `<boost/date_time/posix_time/posix_time.hpp>`，即：

```
#include <boost/date_time/posix_time/posix_time.hpp>
using namespace boost::posix_time;
```

从概念上来说，（广义的）时间是日期的进一步细化，相当于在日期“天”的量级下增加了时分秒的分辨率，因此，我们首先介绍时间长度 `time_duration` 类，它表述了时分秒的度量，然后再介绍时间点 `ptime` 类。

2.7.1 时间长度

与日期长度 `date_duration` 类似，`date_time` 库使用 `time_duration` 度量时间长度。

`time_duration` 很像 C 中 `tm` 结构的时分秒部分，可以度量基本的小时、分钟和秒钟，在

秒以下精确到微秒。如果在头文件<boost/date_time/posix_time/posix_time.hpp>之前定义了宏 BOOST_DATE_TIME_POSIX_TIME_STD_CONFIG, 那么它可以精确到纳秒级。以下我们主要叙述微秒精度的 time_duration。

time_duration 的类摘要如下:

```
class time_duration
{
public:
    time_duration();
    time_duration(hour_type, min_type, sec_type = 0,
                  fractional_seconds_type = 0);
    time_duration(special_values);

    hour_type hours() const;
    min_type minutes() const;
    sec_type seconds() const;
    sec_type total_seconds() const;
    tick_type total_milliseconds() const;
    tick_type total_nanoseconds() const;
    tick_type total_microseconds() const;
    fractional_seconds_type fractional_seconds() const;

    duration_type invert_sign() const;
    bool is_negative() const;

    bool operator<(const time_duration &) const;
    bool operator==(const time_duration &) const;

    tick_type ticks() const;
    bool is_special() const;
    bool is_pos_infinity() const;
    bool is_neg_infinity() const;
    bool is_not_a_date_time() const;
    impl_type get_rep() const;

    static duration_type unit() ;
    static tick_type ticks_per_second() ;
    static time_resolutions resolution() ;
    static unsigned short num_fractional_digits() ;
};
```

为了方便使用, time_duration 也有几个子类, 可以度量不同的时间分辨率, 它们的名字很容易记, 分别是: hours、minutes、seconds、millisec/milliseconds、microsec/microseconds 和 nanosec/nanoseconds。

time_duration 支持全序比较操作和输入输出, 而且比 date_duration 要支持更多的算

术运算，可以进行加减乘除全四则运算。

2.7.2 操作时间长度

`time_duration` 可以在构造函数指定时分秒和微秒来构造，例如创建一个 1 小时 10 分钟 30 秒 1 毫秒（1000 微秒）的时间长度：

```
time_duration td(1,10,30,1000);
```

时、分、秒等值可以是任意的数量，不一定必须在它们的限度里，超出的时间会自动进位或借位，例如，下面的代码表示 2 小时 01 分 06.001 秒：

```
time_duration td(1,60,60,1000*1000* 6 + 1000);
```

使用 `time_duration` 的子类可以更直观地创建时间长度：

```
hours h(1);           //1 小时
minutes m(10);        //10 分钟
seconds s(30);        //30 秒钟
millisec ms(1);       //1 毫秒

time_duration td = h + m + s + ms;           //可以赋值给 time_duration
time_duration td2 = hours(2) + seconds(10); //也可以直接赋值
```

使用工厂函数 `duration_from_string()`，`time_duration` 也可以从一个字符串创建，字符串中的时、分、秒和微秒需要用冒号隔开：

```
time_duration td = duration_from_string("1:10:30:001");
```

`time_duration` 里的时分秒可以用 `hours()`、`minutes()` 和 `seconds()` 成员函数访问。`total_seconds()`、`total_milliseconds()` 和 `total_microseconds()` 分别返回时间长度的总秒数、总毫秒数和总微秒数。`fractional_seconds()` 以 `long` 返回微秒数：

```
time_duration td(1,10,30,1000);
assert(td.hours() == 1 && td.minutes() == 10 && td.seconds() == 30);
assert(td.total_seconds() == 1*3600+ 10*60 + 30);
assert(td.total_milliseconds() == td.total_seconds()*1000 + 1);
assert(td.fractional_seconds() == 1000);
```

`time_duration` 可以取负值，专门有一个成员函数 `is_negative()` 来判断它的正负号。成员函数 `invert_sign()` 可以将时间长度改变符号后生成一个新的时间长度：

```
hours h(-10);
assert(h.is_negative());

time_duration h2 = h.invert_sign();
```

```
assert(!h2.is_negative() && h2.hours() == 10);
```

`time_duration` 也可以赋值为特殊时间值, 包括 `not_a_date_time`、`pos_infin` 等等, 同样也有类似的 `is_xxx()` 函数用于检测它是否为特殊时间, 用法与 `date`、`date_duration` 类似, 例如:

```
time_duration td1(not_a_date_time);
assert(td1.is_special() && td1.is_not_a_date_time());

time_duration td2(neg_infin);
assert(td2.is_negative() && td2.is_neg_infinity());
```

`time_duration` 支持完整的比较操作和四则运算, 因此它处理起来比 `date` 对象更加容易, 例如:

```
time_duration td1 = hours(1);
time_duration td2 = hours(2) + minutes(30);
assert(td1 < td2);
assert((td1+td2).hours() == 3);
assert((td1-td2).is_negative());
assert(td1 * 5 == td2 * 2);
assert((td1/2).minutes() == td2.minutes());
```

如果想要得到 `time_duration` 对象的字符串表示, 可以使用自由函数 `to_simple_string(time_duration)` 和 `to_iso_string(time_duration)`, 它们分别返回 `HH:MM:SS.fffffffffff` 和 `HHMMSS,ffffffffff` 格式的字符串。例如:

```
time_duration td(1,10,30,1000);
cout << to_simple_string(td) << endl;
cout << to_iso_string(td) << endl;
```

将输出:

```
01:10:30.001000
011030.001000
```

`time_duration` 也可以转换到 `tm` 结构, 同样使用 `to_tm()` 函数, 但不能进行反向转换。

2.7.3 时间长度的精确度

`date_time` 库默认时间的精确度是微秒, 纳秒相关的类和函数如 `nanosec` 和成员函数 `nanoseconds()`、`total_nanoseconds()` 都不可用, 秒以下的时间度量都使用微秒。

当定义了宏 `BOOST_DATE_TIME_POSIX_TIME_STD_CONFIG` 时, `time_duration` 的一些行为将发生变化, 它的时间分辨率将精确到纳秒, 构造函数中秒以下的时间度量单位也会变成纳秒。

请读者将这段代码与 2.7.2 节的 `time_duration` 构造进行比较:

```
#define BOOST_DATE_TIME_POSIX_TIME_STD_CONFIG    //定义纳秒精度宏
#define BOOST_DATE_TIME_SOURCE
#include <boost/date_time/posix_time/posix_time.hpp>

time_duration td(1,10,30,1000);                //1000 纳秒, 即 1 微秒
cout << td;
assert(td.total_milliseconds() == td.total_seconds()*1000);
//计算总毫秒数时微秒将被忽略
```

成员函数 `fractional_seconds()` 仍然返回秒的小数部分, 但单位是纳秒, 这也是它的名称不叫 `milli_seconds()` 或者 `nano_seconds()` 的原因:

```
assert(td.fractional_seconds() == 1000);
```

静态成员函数 `unit()` 返回一个 `time_duration` 对象, 它是 `time_duration` 计量的最小单位, 相当于 `time_duration(0,0,0,1)`, 默认情况下是微秒, 如果定义了 `BOOST_DATE_TIME_POSIX_TIME_STD_CONFIG` 则是纳秒:

```
assert(time_duration::unit()*1000*1000*1000 == seconds(1));
//1 秒等于 10 的 9 次方纳秒
```

`time_duration` 提供静态成员函数 `resolution()` 和 `num_fractional_digits()` 来检测当前的精度:

- `resolution()` 可以返回一个枚举值 `time_resolutions`, 表示时间长度的分辨率;
- 静态成员函数 `num_fractional_digits()` 返回秒的小数部分的位数 (微秒 6 位, 纳秒 9 位):

```
assert(td.resolution() == date_time::nano);    //如果是纳秒分辨率则断言成立
assert(td.num_fractional_digits() == 9);       //如果是纳秒分辨率则断言成立
```

`BOOST_DATE_TIME_POSIX_TIME_STD_CONFIG` 宏主要影响 `time_duration` 构造函数中小数秒的解释, 默认单位是微秒, 定义了纳秒精度则单位是纳秒。如果要编写分辨率无关的代码, 应尽量避免使用这种创建时间长度的方式, 可以使用 `millisec/milliseconds`、`microsec/microseconds` 这些预先定义好的时间单位来创建时间长度。

成员函数 `ticks_per_second()` 是 C 中的宏 `CLOCKS_PER_SEC` 的强化, 它返回每秒钟内的 tick 数, 也可以使用它来编写与时间精度无关的代码:

```
time_duration::tick_type my_millisec =          //自定义毫秒单位
    time_duration::ticks_per_second()/1000;
time_duration td(1,10,30,10 *my_millisec);     //10 毫秒, 即 0.01 秒
```

因为在大多数情况下程序都不会用到纳秒级别的计时精度，微秒已经非常足够了，故以下将不再对纳秒进行讨论。但除了时间分辨率的粒度，date_time 库的其他行为都是一致的。

2.7.4 时间点

在熟悉了时间长度类 time_duration 后，理解时间点概念就容易多了，它相当于一个日期再加上一个小于一天的时间长度。如果时间轴的基本单位是天，那么日期就相当于整数，时间点则是实数，定义了天之间的小数部分。

ptime 是 date_time 库处理时间的核心类，它使用 64 位（微秒级别）或者 96 位（纳秒级别）的整数在内部存储时间数据，依赖于 date 和 time_duration，因此接口很小。ptime 的类摘要如下：

```
class ptime
{
public:
    ptime(const date_type &, const time_duration_type &,
          dst_flags = not_dst);
    ptime(special_values);

    // public member functions
    date_type date() const;
    time_duration_type time_of_day() const;
    bool is_not_a_date_time() const;
    bool is_infinity() const;
    bool is_pos_infinity() const;
    bool is_neg_infinity() const;
    bool is_special() const;
    bool operator==(const time_type &) const;
    bool operator<(const time_type &) const;
};
```

ptime 同 date 一样，也是一个轻量级的对象，可以被高效地任意拷贝赋值，也支持全序比较和加减运算。

2.7.5 创建时间点对象

最基本的创建 ptime 的方式是在构造函数中同时指定 date 和 time_duration 对象，令 ptime 等于一个日期加当天的时间偏移量。如果不指定 time_duration，则默认为当天的零点。例如：

```
// posix_time 名字空间不包含 gregorian 名字空间，因此需要加上对它的引用
using namespace boost::gregorian;
ptime p(date(2010,3,5), hours(1));           //2010 年 3 月 5 日凌晨 1 时
```


`ptime` 也可以从字符串构造, 使用工厂函数 `time_from_string()` 和 `from_iso_string()`。前者使用分隔符分隔日期时间成分, 后者则是连续的数字, 日期与时间用字母 `T` 隔开:

```
ptime p1 = time_from_string("2010-3-5 01:00:00");
ptime p2 = from_iso_string("20100305T010000");
```

`date_time` 库为 `ptime` 也提供了时钟类, 可以从时钟产生当前时间。因为时间具有不同的分辨率, 有两个类 `second_clock` 和 `microsec_clock` 分别提供秒级和微秒级的分辨率, 它们的接口是相同的, `local_time()` 获得本地当前时间, `universal_time()` 获得 UTC 当前时间。例如:

```
ptime p1 = second_clock::local_time();           //秒精度
ptime p2 = microsec_clock::universal_time();     //微秒精度
cout << p1 << endl << p2;
```

`ptime` 可以构造为特殊时间值, 如无效时间、无限时间, 也同样有 `is_xxx()` 来检验特殊值。例如:

```
ptime p1(not_a_date_time);                       //无效时间
assert(p1.is_not_a_date_time());
ptime p2(pos_infin);                             //正无限时间
assert(p2.is_special() && p2.is_pos_infinity());
```

2.7.6 操作时间点对象

由于 `ptime` 相当于 `date+time_duration`, 因此对它的操作可以分解为对这两个组成部分的操作。

`ptime` 使用 `date()` 和 `time_of_day()` 两个成员函数获得时间点中的日期和时间长度, 然后就可以分别处理。例如:

```
//2010年3月20日中午12:30
ptime p(date(2010,3,20), hours(12)+minutes(30));

date d = p.date();
time_duration td = p.time_of_day();
assert(d.month() == 3 && d.day() == 20);
assert(td.total_seconds() == 12*3600 + 30*60);
```

`ptime` 支持比较操作与加减运算, 运算规则与日期类似:

```
ptime p1(date(2010,3,20), hours(12)+minutes(30));
ptime p2 = p1 + hours(3);                       //2010年3月20日15:30

assert(p1 < p2);
```

```
assert(p2 - p1 == hours(3));
p2 += months(1);
assert(p2.date().month() == 4);
```

ptime 提供了三个自由函数转换为字符串，分别是：

- to_simple_string(ptime)：转换为 YYYY-mm-DD HH:MM:SS.fffffffff 格式；
- to_iso_string(ptime)：转换为 YYYYMMDDTHHMMSS,fffffffff 格式；
- to_iso_extended_string(ptime)：转换为 YYYY-MM-DDTHH:MM:SS,fffffffff 格式。

其中的 ffffffffff 是秒的小数部分，如果为 0 则不显示，T 是日期与时间的分隔符。

示范这三个函数用法的代码如下：

```
ptime p(date(2010,2,14), hours(20));
cout << to_simple_string(p) << endl;
cout << to_iso_string(p) << endl;
cout << to_iso_extended_string(p) << endl;
```

程序运行结果如下：

```
2010-Feb-14 20:00:00
20100214T200000
2010-02-14T20:00:00
```

2.7.7 与 tm、time_t 等结构的转换

使用自由函数 to_tm()，ptime 可以单向转换到 tm 结构，转换规则是 date 和 time_duration 的组合。例如：

```
ptime p(date(2010,2,14), hours(20));
tm t = to_tm(p);
assert(t.tm_year == 110 && t.tm_hour == 20);
```

没有一个叫做 time_from_tm() 的函数可以把 tm 结构转换成 ptime，这与 date 对象的 date_from_tm() 是不同的。

如果想要把 tm 转换成 ptime，可以使用 date_from_tm() 得到 date 对象，然后手工操作 tm 结构得到 time_duration 对象，最后创建出 ptime。

另有两个函数：from_time_t(time_t) 和 from_ftime<ptime>(FILETIME)，它们可以从 time_t 和 FILETIME 结构创建出 ptime 对象。这种转换也是单向的，不存在逆向的转换。例如：

```
ptime p = from_time_t(std::time(0));
assert(p.date() == day_clock::local_day());

FILETIME ft;
ft.dwHighDateTime = 29715317;
ft.dwLowDateTime = 3865122988UL;
p = fromftime<ptime>(ft);
```

2.7.8 时间区间

与日期区间 `date_period` 对应，`date_time` 库也有时间区间的概念，使用类 `time_period`，使用 `ptime` 作为区间的两个端点，同样是左闭右开区间。

`time_period` 的用法与 `date_period` 基本相同，可以用 `begin()` 和 `last()` 返回区间的两个端点，`length()` 返回区间的长度，`shift()` 和 `expand()` 变动区间，也能计算时间区间的交集和并集——就像是 `date_period` 的一个时间分辨率的增强版。因此，`time_period` 的详细操作函数可参考 `date_period` (2.7.8 小节，51 页)。

示范 `time_period` 一些用法的代码如下：

```
ptime p(date(2010,1,1),hours(12)); //2010 年元旦中午
time_period tp1(p, hours(8)); //一个 8 小时的区间
time_period tp2(p + hours(8), hours(1)); //1 小时的区间
assert(tp1.end() == tp2.begin() && tp1.is_adjacent(tp2));
assert(!tp1.intersects(tp2)); //两个区间相邻但不相交

tp1.shift(hours(1)); //tp1 平移 1 小时
assert(tp1.is_after(p)); //tp1 在中午之后
assert(tp1.intersects(tp2)); //两个区间现在相交

tp2.expand(hours(10)); //tp2 向两端扩展 10 个小时
assert(tp2.contains(p) && tp2.contains(tp1));
```

2.7.9 时间迭代器

不同于日期迭代器，时间迭代器只有一个 `time_iterator`。它在构造时传入一个起始时间点 `ptime` 对象和一个步长 `time_duration` 对象，然后就同日期迭代器一样使用前置式 `operator++`、`operator--` 来递增或递减时间，解引用操作符返回一个 `ptime` 对象。

`time_iterator` 也可以直接与 `ptime` 比较，无须再使用解引用操作符。

下面的代码使用时间迭代器以 10 分钟为步长打印时间：

```
ptime p(date(2010,2,27),hours(10)) ;
for (time_iterator t_iter(p, minutes(10));
    t_iter < p + hours(1); ++ t_iter)
{
    cout << *t_iter << endl;
}
```



2.7.10 综合运用

本小节将综合运用 ptime 相关的所有类，给出一些具体的使用例子。

高精度计时器

我们首先利用 ptime 来实现一个高精度的计时器 ptimer，用来取代 2.2 小节介绍的计时器 timer 和 progress_timer。

ptimer 的原理与 timer 一样，在对象创建时就启动计时，然后可以用 elapsed() 函数返回流逝的时间，析构时也会自动输出时间。与 timer 不同的是，ptimer 利用了 date_time 库的高精度时间度量功能，总可以提供微秒级别的计时。如果需要，它还可以精确到纳秒级别。

ptimer 的唯一缺陷是它依赖于 date_time 库，需要编译 date_time 才能使用。

为了支持 date_time 库的两个时钟类：second_clock 和 microsec_clock，我们决定把 ptimer 实现为一个模板类 basic_ptimer，然后用定制时钟类的方式就可以实现两个精度分别为秒级别和微秒级别的计时器。

由于 date_time 库提供了大量的便利操作，因此 basic_ptimer 的实现代码相当简单，如下所示：

```
template<typename Clock = microsec_clock>
class basic_ptimer
{
public:
    basic_ptimer() //构造函数, 初始化起始时间
    { restart(); }
    void restart() //重新开始计时
    { _start_time = Clock::local_time(); }
    void elapsed() const //度量流逝的时间
    { cout << Clock::local_time() - _start_time; }
    ~basic_ptimer() //析构函数自动输出时间
    { elapsed(); }
private:
    ptime _start_time; //私有变量, 保存计时开始时间
};
typedef basic_ptimer<microsec_clock> ptimer; //typedef 具体类
```

```
typedef basic_ptimer<second_clock> sptimer;
```

ptimer 可以如 timer 一样使用, 例如:

```
int main()
{
    ptimer t;                                //创建 ptimer 计时器对象
    ...                                       //花费时间的任意操作
}                                           //退出作用域自动输出时间
```

输出如下所示:

```
00:00:02.828125
```

计算工作时间

接下来我们使用 `time_period` 类来表示工作日的工作时间, 并根据当前时间给用户一个友好的提示信息。

先不考虑节假日等特殊因素, 假设工作制为每天 8 小时, 早上 9:00 上班, 中午 12:30 到 13:30 为午餐时间, 下午 6:00 下班。为了便于处理, 我们把一天分成五个时间段, 分别表示上班前、上午、中午、下午和下班后。由于 `time_period` 支持比较操作, 故可以使用 `std::map` 来保存时间段与问候语的对应关系。

我们定义一个 `work_time` 类, 它封装了这些逻辑:

```
class work_time
{
```

接下来是关联容器 `map` 的类型定义, 用于简化代码:

```
public:
    typedef map<time_period, string> map_t;
```

成员变量 `map_ts` 保存了时间段与问候语的对应关系:

```
private:
    map_t map_ts;
```

成员函数 `init()` 是 `work_time` 类的核心, 初始化时间段与问候语, 向 `map_ts` 插入数据:

```
void init()
{
    ptime p(day_clock::local_day());           //获得当天的日期

    map_ts[time_period(p, hours(9))] = "It's too early, just relax.\n";
    p += hours(9);
    map_ts[time_period(p, hours(3)+minutes(30))] = "It's AM, please work hard.\n";
```

```

p += hours(3) + minutes(30);
map_ts[time_period(p, hours(1))] = "It's lunch time, are you hungry?\n";
p += hours(1);
map_ts[time_period(p, hours(4) + minutes(30))] = "It's PM, ready to go home.\n";
p += hours(4) + minutes(30);
map_ts[time_period(p, hours(6))] = "Are you still working? you do need a rest.\n";
}

```

work_time 的构造函数调用 init() 初始化 map_ts:

```

public:
    work_time()
    {        init(); }

```

最后是给出提示信息的 greeting() 函数, 它使用迭代器遍历 map 容器, 判断时间点是否在某个时间区间内, 如果是则输出对应的问候语:

```

void greeting(const ptime& t)
{
    map_t::iterator pos;
    for (pos = map_ts.begin(); pos != map_ts.end(); ++pos)
    {
        if (pos->first.contains(t))                //pair 的 first 成员是时间段
        {
            cout << pos->second << endl;
            break;
        }
    }
    //end if
    //end for
};
// work_time 类结束

```

work_time 类的使用如下:

```

int main()
{
    work_time wt;                                //创建一个 work_time 对象
    wt.greeting(second_clock::local_time());    //用当前时间致问候语
}

```

如果现在是上午, 那么程序的运行结果是:

```
It's AM, please work hard.
```

2.8 date_time 库的高级议题

date_time 库包含了丰富的内容, 涉及日期与时间处理的很多方面, 可以解决很多实际生活中遇到的问题。本小节简要讨论一些关于 date_time 库的高级议题。

2.8.1 编译配置宏

之前的章节中已经讨论了几个用于控制 date_time 库行为的宏，这里再做一个简短的归纳。

宏 BOOST_DATE_TIME_SOURCE 和 BOOST_DATE_TIME_NO_LIB 用来指定 date_time 库的编译设置，定义了它们将告诉编译器不使用自动链接库的功能，而是使用嵌入源码的方式。Boost 中许多需要编译的库也有类似名称的宏定义，请读者务必了解这个宏的用法。

宏 DATE_TIME_NO_DEFAULT_CONSTRUCTOR 可以禁止编译器创造出 date 和 ptime 的缺省构造函数，强制它们在构造时必须有一个有效的值，可以避免某些疏忽而导致的错误。

宏 BOOST_DATE_TIME_OPTIONAL_GREGORIAN_TYPES 启用了 weeks、months、years 等日期区间便捷类型，它们在处理日期时很有用，可以使代码更清晰易懂。但它们有时候也会在日期运算时产生非预期结果，如果不想使用它们，就 undef 这个宏，从而在程序中总使用 days 保证代码的正确性。

宏 BOOST_DATE_TIME_POSIX_TIME_STD_CONFIG 将启用 date_time 库更高的时间精确度，由微秒变为纳秒，同时纳秒相关的一些函数和类也会启用。缺省情况下它是关闭的，因为纳秒精度通常很依赖于操作系统，而且实际生活中很少用到这么高的精确度。

date_time 库编译的对象是格里高利历源码，不使用纳秒来处理日期，因此 BOOST_DATE_TIME_POSIX_TIME_STD_CONFIG 宏对库的编译没有任何影响，预编译源码文件 dateprebuild.cpp 不必为了支持纳秒精度而增加宏定义。

2.8.2 格式化时间

date_time 库默认的时间格式简单、标准且是英文，但并不是不可以被改变的。date_time 库提供了专门的格式化对象 date_facet、time_facet 等来搭配 IO 流，定制日期时间的表现形式。

这些格式化对象就像是 printf() 函数，使用一个格式化字符串来定制日期或时间的格式，也同样有大量的格式标志符。由于格式标志符非常多，本书不在此处列出，请参考 date_time 库的说明文档。

示范格式化的使用、把日期格式化为中文显示的代码如下：

```
date d(2010,3,6);
date_facet* dfacet = new date_facet("%Y年%m月%d日");
cout.imbue(locale(cout.getloc(), dfacet));
cout << d << endl;
```

```
time_facet *tfacet = new time_facet("%Y年%m月%d日%H点%M分%S%F秒");  
cout.imbue(locale(cout.getloc(), tfacet));  
cout << ptime(d, hours(21) + minutes(50) + millisec(100)) << endl;
```

程序的运行结果如下：

```
2010年03月06日  
2010年03月06日 21点50分00.100000秒
```

2.8.3 本地时间

之前对 `date_time` 库的讨论都基于简单的时间，通常对于日常工作和生活是足够的。但如果考虑到世界各地不同时区的因素，时间就变得复杂了，两个不同时区对同一个时间点的日期和时间表示可能会不同，如果再加上某些地区的夏令时因素就更加复杂了。^①

`date_time` 库使用 `time_zone_base`、`posix_time_zone`、`custom_time_zone`、`local_date_time` 等类和一个文本格式的时区数据库来解决本地时间中时区和夏令时的问题。

本地时间功能位于名字空间 `boost::local_time`，为了使用本地时间功能，需要包含头文件 `<boost/date_time/local_time/local_time.hpp>`，即：

```
#include <boost/date_time/local_time/local_time.hpp>  
using namespace boost::local_time;
```

`time_zone_base` 是时区表示的抽象类，通常我们使用一个 typedef: `time_zone_ptr`，它是一个指向 `time_zone_base` 的智能指针（参见 3.4 小节，69 页的 `shared_ptr`，）。

`local_date_time` 是一个含有时区信息的时间对象，它可以由 `date+time_duration+` 时区构造，构造时必须指定这个时间是否是夏令时，本地时间在内部以 UTC 的形式保存以方便计算。

为了便于时区编程，`date_time` 库附带了一个小型 CSV 格式的文本数据库 `date_time_zonespec.csv`，位于 `libs/date_time/data/` 下，可以自由使用。这个数据库包含了世界上几乎所有国家和地区的时区信息，`tz_database` 类专门管理这个数据库，只要指定时区名，就可以很方便地获得时区信息——一个 `time_zone_ptr`。

假设从北京时区飞往纽约时区，飞行时间为 15 个小时，示范跨时区的时间转换的代码如下：

```
#define BOOST_DATE_TIME_SOURCE
```

① 幸好，我国很早就取消了夏令时制度，统一使用的是北京时间。


```

//包含必要的头文件和声明名字空间
#include <boost/date_time/gregorian/gregorian.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/local_time/local_time.hpp>
using namespace boost::posix_time;
using namespace boost::gregorian;
using namespace boost::local_time;
int main()
{
    tz_database tz_db;                                //时区数据库对象
    {
        ptimer t;                                     //ptimer, 计算打开数据库的时间

        //假设文本数据库位于当前目录下
        tz_db.load_from_file("../date_time_zonespec.csv");
    }
    cout << endl;

    //使用字符串 Asia/Shanghai 获得上海时区, 即北京时间
    time_zone_ptr shz = tz_db.time_zone_from_region("Asia/Shanghai");

    //使用字符串 America/New_York 获得纽约时区
    time_zone_ptr nyz = tz_db.time_zone_from_region("America/New_York");

    cout << shz->has_dst() << endl;                    //上海时区无夏令时
    cout << shz->std_zone_name() << endl;                //上海时区的名称是 CST

    local_date_time dt_bj(date(2008,1,7),              //北京时间 2008,1,7
        hours(12),                                     //中午 12 点
        shz,                                           //上海时区
        false);                                       //没有夏令时
    cout << dt_bj << endl;

    time_duration flight_time = hours(15);             //飞行 15 小时
    dt_bj += flight_time;                             //到达的北京时间
    cout << dt_bj << endl;
    local_date_time dt_ny = dt_bj.local_time_in(nyz); //纽约当地时间
    cout << dt_ny;
}

```

程序的运行结果如下:

```

00:00:00.953125
0
2008-Jan-07 12:00:00 CST
2008-Jan-08 03:00:00 CST
2008-Jan-07 14:00:00 EST

```

2.8.4 序列化

`date_time` 库可以使用 `boost.serialization` 库的能力实现数据序列化, 把日期时间数据存入某个文件, 之后在任意的时刻读取恢复, 如同流操作一样简单方便。

由于 `serialization` 库需要编译才能使用, 而且本书不详细讨论 `serialization` 库, 故 `date_time` 库的序列化功能在这里不做详细介绍, 希望使用这一功能的读者可先阅读 14 章 14.11 (544 页) 小节, 了解 `serialization` 库的基本情况, 然后参考 Boost 的说明文档。

2.9 总结

本章是我们接触 Boost 程序库的第一章, 讨论了 Boost 在时间与日期领域的两个库: `timer` 和 `date_time`。

`timer` 库的实现原理很简单, 可能很多读者都曾经编写过类似的代码。但即使是这样小而简单的库也包含了深刻的设计理念。基于跨平台可移植的指导思想, `timer` 使用了 C 标准中的 `std::clock()`, 而没有使用操作系统可能提供的更精确的计时函数, `std::clock()` 也更加可靠。

`timer` 和 `progress_timer` 是用于计时的小工具, 精度不高但好用够用, 特别是 `progress_timer`, 它利用了 C++ 中析构函数会被自动调用的特点能够自动显示时间, 用起来更方便。

`progress_display` 是一个可以显示程序执行进度的工具, 在很耗时的应用程序中可以给出友好的进度提示, 使用也很容易。但它使用的是字符界面, 不够好看, 而且很容易被程序的其他输出打乱显示。使用它时我们必须保证自己的程序不能有任何可能的输出, 否则 `progress_display` 就完全失去了意义。

`progress_timer` 和 `progress_display` 还有一个有趣的特点是它们都私有继承了 `noncopyable` 类, 防止被无意的拷贝而破坏了正确的行为。`noncopyable` 是 Boost 库的一个实用工具类, 我们很快就会在 4.1 小节 (95 页) 看到它的详细用法。

时间与日期处理是生活中的常见问题, 但 C++98 标准中并没有制定这方面的规范, `date_time` 库填补了这个空白, 提供了基于格里高利历的日期时间处理, 功能非常全面。它支持时间点、时间长度和时间区间等基本概念, 提供了从年月日到时分秒、微秒乃至纳秒等许多级别的时间分辨率, 还重载了许多操作符, 可以进行比较、加减等运算, 能够满足绝大多数程序对时间处理的要求。

日期与时间的处理很复杂，`date_time` 库使用许多工具来简化工作，如日期、时间迭代器以指定的时间间隔遍历时间，时钟类获得所需精度的当前时间，时区与本地时间转换，定制日期时间的输出格式等等，它们比 C 语言中稀少且简陋的时间处理函数要强大很多，而且易于使用。

`date_time` 库也有小小的缺憾，它不能处理 1400 年以前的日期，因此无法用它来研究那之前的历史。但总的来说 `date_time` 的贡献是卓越的，它赋予了我们在 C++ 中自由操纵时间的能力，值得我们认真学习并掌握。

本章实现的工具类如下：

- `new_progress_timer`：主要是出于演示的目的，增强了 `progress_timer` 的能力，能够以指定的精确度自动计时；
- `ptimer`：它基于 `date_time` 库的能力，提供更高精度的计时器。相比 `new_progress_timer` 它更加精确可靠，最高可以精确到纳秒级别，而且可计量的时间跨度也非常的长。

第 3 章

内存管理

内存管理一直是令 C++ 程序员最头疼的工作，C++ 继承了 C 那高效而又灵活的指针，使用起来稍微不小心就会导致内存泄漏(memory leak)、“野”指针(wild pointer)、访问越界(access denied)等问题。曾几何时，C++ 程序员曾经无限地向往 Java、C# 等语言的垃圾回收机制。虽然 C++ 标准提供了智能指针 `std::auto_ptr`，但并没有解决所有问题。

阅读完本章，你会了解到高效的内存管理方法，彻底忘记“栈”(Stack)、“堆”(Heap)等内存分配相关的术语，并且还会发现，Boost 为 C++ 提供的解决方案可能要比 Java 和 C# 等其他语言更好。

3.1 smart_ptr 库概述

计算机系统中资源有很多种，内存是我们最常用到的，此外还有文件描述符、socket、操作系统 handle、数据库连接等等，程序中申请这些资源后必须及时归还系统，否则就会产生难以预料的后果。

3.1.1 RAII 机制

为了管理内存等资源，C++ 程序员通常采用 RAII 机制（资源获取即初始化，Resource Acquisition Is Initialization），在使用资源的类的构造函数中申请资源，然后使用，最后在析构函数中释放资源。

如果对象是用声明的方式在栈上创建的（一个局部对象），那么 RAII 机制会工作正常，当离开作用域时对象会自动销毁从而调用析构函数释放资源。但如果对象是用 `new` 操作符在堆上创建的，那么它的析构函数不会自动调用，程序员必须明确地用对应的 `delete` 操作符销毁它才能释

放资源。这就存在着资源泄漏的隐患，因为这时没有任何对象对已经获取的资源负责，如果因某些意外导致程序未能执行 `delete` 语句，那么内存等资源就永久地丢失了。例如：

```
int *p = new class_need_resource;    //对象创建，获取资源
..                                   //可能发生异常导致资源泄漏
delete p;                             //析构释放资源
```

`new`、`delete` 以及指针的不恰当运用是 C++ 中造成资源获取/释放问题的根源，能否正确而明智地运用 `delete` 是区分 C++ 新手与熟手的关键所在^①。但很多人——即使是熟练的 C++ 程序员，也经常会忘记调用 `delete`。

3.1.2 智能指针

智能指针 (smart pointer) 是 C++ 群体中热门的议题，围绕它有很多有价值的讨论和结论。它实践了推荐书目 [1] 中的代理模式，代理了原始“裸”指针的行为，为它添加了更多更有用的特性。

C++ 引入异常机制后，智能指针由一种技巧升级为一种非常重要的技术，因为如果没有智能指针，程序员必须保证 `new` 对象能在正确的时机 `delete`，四处编写异常捕获代码以释放资源，而智能指针则可以在退出作用域时——不管是正常流程离开或是因异常离开——总调用 `delete` 来析构在堆上动态分配的对象。

存在很多种智能指针，其中最有名的应该是 C++98 标准中的“自动指针” `std::auto_ptr`，它部分地解决了获取资源自动释放的问题，例如：

```
int main()
{
    auto_ptr< class_need_resource > p1(new class_need_resource);
    auto_ptr<demo_class> p2(factory.create());
    ...
}                                     //离开作用域，p1、p2 自动析构从而释放内存等资源
```

`auto_ptr` 的构造函数接受 `new` 操作符或者对象工厂创建出的对象指针作为参数，从而代理了原始指针。虽然它是一个对象，但因为重载了 `operator*` 和 `operator->`，其行为非常类似指针，可以把它用在大多数普通指针可用的地方。当退出作用域时（离开函数 `main()` 或者发生异常），C++ 语言会保证 `auto_ptr` 对象销毁，调用 `auto_ptr` 的析构函数，进而使用 `delete` 操作符删除原始指针释放资源。

① 常用的做法是在每个离开作用域的 `return` 语句前调用 `delete`，并且使用 `try` 捕获所有可能发生的异常，然后在 `catch` 块中调用 `delete`。

auto_ptr 很好用，被包含在 C++ 标准库中令它在世界范围内被广泛使用，使智能指针的思想、用法深入人心。但标准库并没有覆盖智能指针的全部领域，尤其是最重要的引用计数型智能指针。

boost.smart_ptr 库是对 C++98 标准的一个绝佳补充。它提供了六种智能指针，包括 scoped_ptr、scoped_array、shared_ptr、shared_array、weak_ptr 和 intrusive_ptr，从各个方面来增强 std::auto_ptr，而且是异常安全的。库中的两个类——shared_ptr 和 weak_ptr 已被收入到 C++ 新标准的 TR1 库中。

接下来的部分将详细介绍 scoped_ptr、scoped_array、shared_ptr 和 shared_array，简要介绍另两个组件 weak_ptr 和 intrusive_ptr。它们都是很轻量级的对象，速度与原始指针相差无几，对于所指的类型 T 也仅有一个很小且很合理的要求：类型 T 的析构函数不能抛出异常。

这些智能指针都位于名字空间 boost，为了使用 smart_ptr 组件，需要包含头文件 <boost/smart_ptr.hpp>，即：

```
#include <boost/smart_ptr.hpp>
using namespace boost;
```

3.2 scoped_ptr

scoped_ptr 是一个很类似 auto_ptr 的智能指针，它包装了 new 操作符在堆上分配的动态对象，能够保证动态创建的对象在任何时候都可以被正确地删除。但 scoped_ptr 的所有权更加严格，不能转让，一旦 scoped_ptr 获取了对象的管理权，你就无法再从它那里取回来。

scoped_ptr 拥有一个很好的名字，它向代码的读者传递了明确的信息：这个智能指针只能在本作用域里使用，不希望被转让。

3.2.1 类摘要

scoped_ptr 的类摘要如下：

```
template<class T>
class scoped_ptr {                                //noncopyable
private:
    T * px;
    scoped_ptr(scoped_ptr const &);
    scoped_ptr & operator=(scoped_ptr const &);
public:
    explicit scoped_ptr(T * p = 0);
```

```

~scoped_ptr();

void reset(T * p = 0);

T & operator*() const;
T * operator->() const;
T * get() const;

operator unspecified-bool-type() const;
void swap(scoped_ptr & b);
};

```

3.2.2 操作函数

`scoped_ptr` 的构造函数接受一个类型为 `T*` 的指针 `p`，创建一个 `scoped_ptr` 对象，并在内部保存指针参数 `p`。`p` 必须是一个 `new` 表达式动态分配的结果，或者是个空指针（0）。当 `scoped_ptr` 对象的生命期结束时，析构函数 `~scoped_ptr()` 会使用 `delete` 操作符自动销毁所保存的指针对象，从而正确地回收资源^①。

`scoped_ptr` 同时把拷贝构造函数和赋值操作符都声明为私有的，禁止对智能指针的复制操作（原理可参考 4.1 小节，95 页 `noncopyable`，），保证了被它管理的指针不能被转让所有权。

成员函数 `reset()` 的功能是重置 `scoped_ptr`：它删除原来保存的指针，再保存新的指针值 `p`。如果 `p` 是空指针，那么 `scoped_ptr` 将不持有任何指针。一般情况下 `reset()` 不应该被调用，因为它违背了 `scoped_ptr` 的本意——资源应该一直由 `scoped_ptr` 自己自动管理。

`scoped_ptr` 用 `operator*()` 和 `operator->()` 重载了解引用操作符 `*` 和箭头操作符 `->`，以模仿被代理的原始指针的行为，因此可以把 `scoped_ptr` 对象如同指针一样使用。如果 `scoped_ptr` 保存空指针，那么这两个操作的行为未定义。

`scoped_ptr` 不支持比较操作，不能在两个 `scoped_ptr` 之间、`scoped_ptr` 和原始指针或空指针之间进行相等或者不相等测试，我们也无法为它编写额外的比较函数，因为它已经将 `operator==` 和 `operator!=` 两个操作符重载都声明为私有的。但 `scoped_ptr` 提供了一个可以在 `bool` 语境（context）中自动转换成 `bool` 值（如 `if` 的条件表达式）的功能，用来测试 `scoped_ptr` 是否持有一个有效的指针（非空）。它可以代替与空指针的比较操作，而且写法更简单。

成员函数 `swap()` 可以交换两个 `scoped_ptr` 保存的原始指针。它是高效的操作，被用于实

① 根据 C++ 标准，对 0（空指针）执行 `delete` 操作是安全的。

现 `reset()` 函数，也可以被 `boost::swap` (4.5 小节, 113 页) 所利用。

最后是成员函数 `get()`，它返回 `scoped_ptr` 内部保存的原始指针，可以用在某些要求必须是原始指针的场景（如底层的 C 接口）。但使用时必须小心，这将使原始指针脱离 `scoped_ptr` 的控制！不能对这个指针做 `delete` 操作，否则 `scoped_ptr` 析构时会对已经删除的指针再进行删除操作，发生未定义行为（通常是程序崩溃，这可能是最好的结果，因为它说明你的程序存在 bug）。

3.2.3 用法

`scoped_ptr` 的用法很简单：在原本使用指针变量接受 `new` 表达式结果的地方改成用 `scoped_ptr` 对象，然后去掉哪些多余的 `try/catch` 和 `delete` 操作就可以了。像这样：

```
scoped_ptr<string> sp(new string("text"));
```

`scoped_ptr` 是一种“智能指针”，因此其行为与普通指针基本相同，可以使用非常熟悉的 `*` 和 `->` 操作符：

```
cout << *sp << endl;           //取字符串的内容
cout << sp->size() << endl;     //取字符串的长度
```

但记住：不再需要 `delete` 操作，`scoped_ptr` 会自动地帮助我们释放资源。如果我们对 `scoped_ptr` 执行 `delete` 会得到一个编译错误：因为 `scoped_ptr` 是一个行为类似指针的对象，而不是指针，对一个对象应用 `delete` 是不允许的。

`scoped_ptr` 不允许拷贝、赋值，只能在 `scoped_ptr` 被声明的作用域内使用。除了 `*` 和 `->` 外 `scoped_ptr` 也没有定义其他的操作符（不能对 `scoped_ptr` 进行 `++` 或者 `--` 等指针算术操作）。与普通指针相比它只有很小的接口，因此使指针的使用更加安全，更容易使用同时更不容易被误用。下面的代码都是 `scoped_ptr` 的错误用法：

```
sp++;           //错误，scoped_ptr 未定义递增操作符
scoped_ptr<string> sp2 = sp; //错误，scoped_ptr 不能拷贝构造
```

使用 `scoped_ptr` 会带来两个好处：一是使代码变得清晰简单，而简单意味着更少的错误；二是它并没有增加多余的操作，安全的同时保证了效率，可以获得与原始指针同样的速度。

示范 `scoped_ptr` 用法的另一段代码如下：

```
#include <boost/smart_ptr.hpp>
using namespace boost;
struct posix_file           //一个示范性质的文件类
{
    posix_file(const char * file_name) //构造函数打开文件
```

```

    {cout << "open file:" << file_name << endl;    }
    ~posix_file()                                //析构函数关闭文件
    {cout << "close file" << endl; }
};
int main()
{
    scoped_ptr<int> p(new int);                    //一个 int 指针的 scoped_ptr
    if (p)                                          //在 bool 语境中测试指针是否有效
    {
        *p = 100;                                //可以像普通指针一样使用解引用操作符*
        cout << *p << endl;
    }
    p.reset();                                    //reset() 置空 scoped_ptr, 仅仅是演示
    assert(p == 0);                               //p 不持有任何指针
    if (!p)                                        //在 bool 语境中测试, 可以用!操作符
    { cout << "scoped_ptr == null" << endl; }

    //文件类的 scoped_ptr,
    //将在离开作用域时自动析构, 从而关闭文件释放资源
    scoped_ptr<posix_file> fp(new posix_file("/tmp/a.txt"));
}                                                    //在这里发生 scoped_ptr 的析构,
                                                    //p 和 fp 管理的指针自动被删除

```

程序运行结果如下:

```

100
scoped_ptr == null
open file:/tmp/a.txt
close file

```

3.2.4 与 auto_ptr 的区别

scoped_ptr 的用法与 auto_ptr 几乎一样, 大多数情况下它可以与 auto_ptr 相互替换, 它也可以从一个 auto_ptr 获得指针的管理权 (同时 auto_ptr 失去管理权)。

scoped_ptr 也具有 auto_ptr 同样的“缺陷”——不能用作容器的元素, 但原因不同: auto_ptr 是因为它的转移语义, 而 scoped_ptr 则是因为不支持拷贝和赋值, 不符合容器对元素类型的要求。

scoped_ptr 与 auto_ptr 的根本性区别在于指针的所有权。auto_ptr 特意被设计为指针的所有权是可转移的, 可以在函数之间传递, 同一时刻只能有一个 auto_ptr 管理指针。它的用意是好的, 但转移语义太过于微妙, 不熟悉 auto_ptr 特性的初学者很容易误用引发错误。而 scoped_ptr 把拷贝构造函数和赋值函数都声明为私有的, 拒绝了指针所有权的转让——除了 scoped_ptr 自己, 其他任何人都无权访问被管理的指针, 从而保证了指针的绝对安全。

下面的代码清楚地演示了两者的区别:

```

auto_ptr<int> ap(new int(10));           //一个 int 自动指针
scoped_ptr<int> sp(ap);                 //从 auto_ptr 获得原始指针
assert(ap.get() == 0);                 //原 auto_ptr 不再拥有指针

ap.reset(new int(20));                 //auto_ptr 拥有新的指针
cout << *ap << ", " << *sp << endl;

auto_ptr<int> ap2;
ap2 = ap;                             //ap2 从 ap 获得原始指针, 发生所有权转移
assert(ap.get() == 0);                 //ap 不再拥有指针
scoped_ptr<int> sp2;                   //另一个 scoped_ptr
sp2 = sp;                             //赋值操作, 无法通过编译!!

```

如果代码编写者企图从一个 `scoped_ptr` 构造或赋值另一个 `scoped_ptr` (代码的最后一行), 那么编译器会报出一个错误, 阻止他这么做, 从而保护了你的代码, 而且是在编译期。

比起 `auto_ptr`, `scoped_ptr` 更明确地表明了代码原始编写者的意图: 只能在定义的作用域内使用, 不可转让, 这在代码后续的维护生命周期中很重要。

3.3 scoped_array

`scoped_array` 很像 `scoped_ptr`, 它包装了 `new[]` 操作符 (不是单纯的 `new`) 在堆上分配的动态数组, 为动态数组提供了一个代理, 保证可以正确地释放内存。

`scoped_array` 弥补了标准库中没有指向数组的智能指针的缺憾。

3.3.1 类摘要

`scoped_array` 的类摘要如下:

```

template<class T> class scoped_array    //noncopyable
{
public:
    explicit scoped_array(T * p = 0);
    ~scoped_array();

    void reset(T * p = 0);
    T & operator[](std::ptrdiff_t i) const;
    T * get() const;

    operator unspecified-bool-type() const;
    void swap(scoped_array & b);
};

```

`scoped_array` 的接口和功能几乎是与 `scoped_ptr` 是相同的 (甚至还要少一些), 主要特

点如下:

- 构造函数接受的指针 `p` 必须是 `new[]` 的结果, 而不能是 `new` 表达式的结果;
- 没有 `*`、`->` 操作符重载, 因为 `scoped_array` 持有的不是一个普通指针;
- 析构函数使用 `delete[]` 释放资源, 而不是 `delete`;
- 提供 `operator[]` 操作符重载, 可以像普通数组一样用下标访问元素;
- 没有 `begin()`、`end()` 等类似容器的迭代器操作函数。

3.3.2 用法

`scoped_array` 与 `scoped_ptr` 源于相同的设计思想, 故而用法非常相似: 它只能在被声明的作用域内使用, 不能拷贝、赋值。唯一不同的是 `scoped_array` 包装的是 `new[]` 产生的指针, 并在析构时调用 `delete[]`, 因为它管理的是动态数组, 而不是单个动态对象。

通常 `scoped_array` 的创建方式是这样的:

```
scoped_array<int> sa(new int[100]);           //包装动态数组
```

`scoped_array` 重载了 `operator[]`, 因此它用起来就像是一个普通的数组, 但因为它不提供指针运算, 所以不能用“数组首地址+N”的方式访问数组元素:

```
sa[0] = 10;                                   //正确用法, 使用 operator[]
*(sa + 1) = 20;                               //错误用法, 不能通过编译
```

在使用重载的 `operator[]` 时要小心, `scoped_array` 不提供数组索引的范围检查, 如果使用了超过动态数组大小的索引或者是负数索引将引发未定义行为。

下面的代码进一步示范了 `scoped_array` 的用法:

```
#include <boost/smart_ptr.hpp>
using namespace boost;
int main()
{
    int *arr = new int[100];                   //一个整数的动态数组
    scoped_array<int> sa(arr);                  //scoped_array 对象代理原始动态数组

    fill_n(&sa[0], 100, 5);                     //可以使用标准库算法赋值数据
    sa[10] = sa[20] + sa[30];                   //用起来就像是个普通数组
}                                                //这里 scoped_array 被自动析构,
                                                //释放动态数组资源
```

3.3.3 使用建议

scoped_array 没有给程序增加额外的负担,用起来很方便轻巧。它的速度与原始数组同样快,很适合那些习惯于用 new 操作符在堆上分配内存的程序员。但 scoped_array 的功能很有限,不能动态增长,也没有迭代器支持,不能搭配 STL 算法,仅有一个纯粹的“裸”数组接口。而且,我们应当尽量避免使用 new[] 操作符,它比 new 更可怕,是许多错误的来源,因为

```
int *p = new int[10];
delete p;
```

这样的代码完全可以通过编译,无论是编译器还是程序员都很难区分出 new[] 和 new 分配的空间,而错误地运用 delete 将导致资源异常。

在需要动态数组的情况下我们应该使用 std::vector,它比 scoped_array 提供了更多的灵活性,而只付出了很小的代价。使用 std::vector,之前的例子可以写成:

```
vector<int> sa(100, 5);
sa[10] = sa[20] + sa[30];
```

很明显, std::vector 只用一条语句就完成了 scoped_array 三条语句的初始化和赋值工作。因为 vector 有丰富的成员函数来操纵数据,能够使代码更加简单明了,易于维护。

除非对性能有非常苛刻的要求,或者编译器不支持标准库(比如某些嵌入式操作系统),否则本书不推荐使用 scoped_array,它只是为了与老式 C 风格代码兼容而使用的类,它的出现往往意味着你的代码中存在着隐患。

3.4 shared_ptr

shared_ptr 是一个最像指针的“智能指针”,是 boost.smart_ptr 库中最有价值、最重要的组成部分,也是最有用的,Boost 库的许多组件——甚至还包括其他一些领域的智能指针都使用了 shared_ptr。抱歉,我实在想不出什么更恰当的词汇来形容它在软件开发中的重要性。再强调一遍, shared_ptr 非常有价值、非常重要、非常有用。

shared_ptr 与 scoped_ptr 一样包装了 new 操作符在堆上分配的动态对象,但它实现的是引用计数型的智能指针^①,可以被自由地拷贝和赋值,在任意的地方共享它,当没有代码使用(引用计数为 0)它时才删除被包装的动态分配的对象。shared_ptr 也可以安全地放到标准容器中,

① shared_ptr 的早期名字就叫做 counted_ptr。

并弥补了 `auto_ptr` 因为转移语义而不能把指针作为 STL 容器元素的缺陷。

在 C++ 历史上曾经出现过无数的引用计数型智能指针实现，但没有一个比得上 `boost::shared_ptr`，在过去、现在和将来，它都是最好的。

3.4.1 类摘要

`shared_ptr` 要比同为智能指针的 `scoped_ptr` 复杂许多，它的类摘要如下：

```
template<class T> class shared_ptr
{
public:
    typedef T element_type;

    shared_ptr();
    template<class Y> explicit shared_ptr(Y * p);
    template<class Y, class D> shared_ptr(Y * p, D d);
    ~shared_ptr();

    shared_ptr(shared_ptr const & r);
    template<class Y> explicit shared_ptr(std::auto_ptr<Y> & r);

    shared_ptr & operator=(shared_ptr const & r);
    template<class Y> shared_ptr & operator=(shared_ptr<Y> const & r);
    template<class Y> shared_ptr & operator=(std::auto_ptr<Y> & r);

    void reset();
    template<class Y> void reset(Y * p);
    template<class Y, class D> void reset(Y * p, D d);

    T & operator*() const;
    T * operator->() const;
    T * get() const;

    bool unique() const;
    long use_count() const;

    operator unspecified-bool-type() const;
    void swap(shared_ptr & b);
};
```

3.4.2 操作函数

`shared_ptr` 与 `scoped_ptr` 同样是用于管理 `new` 动态分配对象的智能指针，因此功能上有很多相似之处：它们都重载了 `*` 和 `->` 操作符以模仿原始指针的行为，提供隐式 `bool` 类型转换以判断指针的有效性，`get()` 可以得到原始指针，并且没有提供指针算术操作。例如：

```

shared_ptr<int> spi(new int);           //一个 int 的 shared_ptr
assert(spi);                           //在 bool 语境中隐式转换为 bool 值
*spi = 253;                            //使用解引用操作符*
shared_ptr<string> sps(new string("smart")); //一个 string 的 shared_ptr
assert(sps->size() == 5);              //使用箭头操作符->

```

但 shared_ptr 的名字表明了它与 scoped_ptr 的主要不同：它是可以被安全共享的——shared_ptr 是一个“全功能”的类，有着正常的拷贝、赋值语义，也可以进行 shared_ptr 间的比较，是“最智能”的智能指针。

shared_ptr 有多种形式的构造函数，应用于各种可能的情形：

- 无参的 shared_ptr() 创建一个持有空指针的 shared_ptr；
- shared_ptr(Y * p) 获得指向类型 T 的指针 p 的管理权，同时引用计数置为 1。这个构造函数要求 Y 类型必须能够转换为 T 类型；
- shared_ptr(shared_ptr const & r) 从另外一个 shared_ptr 获得指针的管理权，同时引用计数加 1，结果是两个 shared_ptr 共享一个指针的管理权；
- shared_ptr(std::auto_ptr<Y> & r) 从一个 auto_ptr 获得指针的管理权，引用计数置为 1，同时 auto_ptr 自动失去管理权；
- operator= 赋值操作符可以从另外一个 shared_ptr 或 auto_ptr 获得指针的管理权，其行为同构造函数；
- shared_ptr(Y * p, D d) 行为类似 shared_ptr(Y * p)，但使用参数 d 指定了析构时的定制删除器，而不是简单的 delete。这部分将在 3.4.5 小节，75 页详述。

shared_ptr 的 reset() 函数的行为与 scoped_ptr 也不尽相同，它的作用是将引用计数减 1，停止对指针的共享，除非引用计数为 0，否则不会发生删除操作。带参数的 reset() 则类似相同形式的构造函数，原指针引用计数减 1 的同时改为管理另一个指针。

shared_ptr 有两个专门的函数来检查引用计数。unique() 在 shared_ptr 是指针的唯一所有者时返回 true（这时 shared_ptr 的行为类似 auto_ptr 或 scoped_ptr），use_count() 返回当前指针的引用计数。要小心，use_count() 应该仅仅用于测试或者调试，它不提供高效率的操作，而且有的时候可能是不可用的（极少数情形）。而 unique() 则是可靠的，任何时候都可用，而且比 use_count() == 1 速度更快。

shared_ptr 还支持比较运算，可以测试两个 shared_ptr 的相等或不相等，比较基于内部保存的指针，相当于 a.get() == b.get()。shared_ptr 还可以使用 operator< 比较大小，

同样基于内部保存的指针，但不提供除 `operator<` 以外的比较操作符，这使得 `shared_ptr` 可以被用于标准关联容器（`set` 和 `map`）：

```
typedef shared_ptr<string> sp_t;           //shared_ptr 类型定义
map<sp_t, int> m;                          //标准映射容器
sp_t sp(new string("one"));               //一个 shared_ptr 对象
m[sp] = 111;                              //关联数组用法
```

在编写基于虚函数的多态代码时指针的类型转换很有用，比如把一个基类指针转型为一个子类指针或者反过来。但对于 `shared_ptr` 不能使用诸如 `static_cast<T*>(p.get())` 的形式，这将导致转型后的指针无法再被 `shared_ptr` 正确管理。为了支持这样的用法，`shared_ptr` 提供了类似的转型函数 `static_pointer_cast<T>()`、`const_pointer_cast<T>()` 和 `dynamic_pointer_cast<T>()`，它们与标准的转型操作符 `static_cast<T>`、`const_cast<T>` 和 `dynamic_cast<T>` 类似，但返回的是转型后的 `shared_ptr`。

例如，下面的代码使用 `dynamic_pointer_cast` 把一个 `shared_ptr<std::exception>` 向下转型为一个 `shared_ptr<bad_exception>`，然后又用 `static_pointer_cast` 重新转为 `shared_ptr<std::exception>`：

```
shared_ptr<std::exception> sp1(new bad_exception("error"));
shared_ptr<bad_exception> sp2 = dynamic_pointer_cast<bad_exception>(sp1);
shared_ptr<std::exception> sp3 = static_pointer_cast<std::exception>(sp2);
assert(sp3 == sp1);
```

此外，`shared_ptr` 还支持流输出操作符 `operator<<`，输出内部的指针值，方便调试。

3.4.3 用法

`shared_ptr` 的智能使其行为最接近原始指针，因此它比 `auto_ptr` 和 `scoped_ptr` 的应用范围更广。几乎是 100% 可以在任何 `new` 出现的地方接受 `new` 的动态分配结果，然后被任意使用，从而完全消灭 `delete` 的使用和内存泄漏，而它的用法与 `auto_ptr` 和 `scoped_ptr` 一样的简单。

`shared_ptr` 也提供基本的线程安全保证，一个 `shared_ptr` 可以被多个线程安全读取，但其他的访问形式结果是未定义的。

示范 `shared_ptr` 基本用法的例子如下：

```
shared_ptr<int> sp(new int(10));           //一个指向整数的 shared_ptr
assert(sp.unique());                      //现在 shared_ptr 是指针的唯一持有者

shared_ptr<int> sp2 = sp;                  //第二个 shared_ptr, 拷贝构造函数
```



```
//两个 shared_ptr 相等,指向同一个对象,引用计数为 2
assert(sp == sp2 && sp.use_count() == 2);

*sp2 = 100; //使用解引用操作符修改被指对象
assert(*sp == 100); //另一个 shared_ptr 也同时被修改

sp.reset(); //停止 shared_ptr 的使用
assert(!sp); //sp 不再持有任何指针(空指针)
```

第二个例子示范了 shared_ptr 较复杂的用法:

```
class shared //一个拥有 shared_ptr 的类
{
private:
    shared_ptr<int> p; //shared_ptr 成员变量
public:
    shared(shared_ptr<int> p_):p(p_){} //构造函数初始化 shared_ptr
    void print() //输出 shared_ptr 的引用计数和指向的值
    {
        cout << "count:" << p.use_count()
              << "v =" <<*p << endl;
    }
};

void print_func(shared_ptr<int> p) //使用 shared_ptr 作为函数参数
{
    //同样输出 shared_ptr 的引用计数和指向的值
    cout << "count:" << p.use_count()
          << " v=" <<*p << endl;
}

int main()
{
    shared_ptr<int> p(new int(100));
    shared s1(p), s2(p); //构造两个自定义类

    s1.print();
    s2.print();

    *p = 20; //修改 shared_ptr 所指的值
    print_func(p);

    s1.print();
}
```

这段代码定义了一个类和一个函数,两者都接受 shared_ptr 对象作为参数,特别注意的是我们没有使用引用的方式传递参数,而是直接拷贝,就像是在使用一个原始指针,shared_ptr 支持这样的用法。

在声明了 `shared_ptr` 和两个 `shared` 类实例后，指针被它们所共享，因此引用计数为 3。`print_func()` 函数内部拷贝了一个 `shared_ptr` 对象，因此引用计数再增加 1，但当退出函数时拷贝自动析构，引用计数又恢复为 3。

程序的运行结果如下：

```
count:3 v=100
count:3 v=100
count:4 v=20
count:3 v=20
```

3.4.4 工厂函数

`shared_ptr` 很好地消除了显式的 `delete` 调用，如果读者掌握了它的用法，可以肯定 `delete` 将会在你的编程字典中彻底消失^①。

但这还不够，因为 `shared_ptr` 的构造还需要 `new` 调用，这导致了代码中的某种不对称性。虽然 `shared_ptr` 很好地包装了 `new` 表达式，但过多的显式 `new` 操作符也是个问题，它应该使用工厂模式来解决。

因此，`shared_ptr` 在头文件 `<boost/make_shared.hpp>`^② 中提供了一个自由工厂函数（位于 `boost` 名字空间）`make_shared<T>()`，来消除显式的 `new` 调用，它的名字模仿了标准库的 `make_pair()`，声明如下：

```
template<class T, class... Args>
    shared_ptr<T> make_shared( Args && ... args );
```

`make_shared()` 函数可以接受最多 10 个参数，然后把它们传递给类型 `T` 的构造函数，创建一个 `shared_ptr<T>` 的对象并返回。`make_shared()` 函数要比直接创建 `shared_ptr` 对象的方式快且高效，因为它内部仅分配一次内存，消除了 `shared_ptr` 构造时的开销。

下面的代码示范了 `make_shared()` 函数的用法：

```
#include <boost/make_shared.hpp>
int main()
{
    shared_ptr<string> sp =
        make_shared<string>("make_shared");    //创建 string 的共享指针
    shared_ptr<vector<int> > spv =
        make_shared<vector<int> >(10, 2);        //创建 vector 的共享指针
```

① 作者已经在数年的实践开发工作中很久没有写过这个单词了，一直到写作本书。

② 这个头文件不包含在 `<boost/smart_ptr.hpp>` 中，必须单独写出。

```
    assert(spv->size() == 10);
}
```

make_shared() 不能接受任意多数量的参数构造对象，一般情况下这不会成为问题。实际上，很少有如此多的参数的函数接口，即使有，那也会是一个设计的不够好的接口，应该被重构。

除了 make_shared(), smart_ptr 库还提供一个 allocate_shared(), 它比 make_shared() 多接受一个定制的内存分配器类型参数，其他方面都相同。

3.4.5 应用于标准容器

有两种方式可以将 shared_ptr 应用于标准容器（或者容器适配器等其他容器）。

一种用法是将容器作为 shared_ptr 管理的对象，如 shared_ptr<list<T> >, 使容器可以被安全地共享，用法与普通 shared_ptr 没有区别，我们不再讨论。

另一种用法是将 shared_ptr 作为容器的元素，如 vector<shared_ptr<T> >, 因为 shared_ptr 支持拷贝语义和比较操作，符合标准容器对元素的要求，所以可以实现在容器中安全地容纳元素的指针而不是拷贝。

标准容器不能容纳 auto_ptr, 这是 C++ 标准特别规定的（读者永远也不要再有这种想法）。标准容器也不能容纳 scoped_ptr, 因为 scoped_ptr 不能拷贝和赋值。标准容器可以容纳原始指针，但这就丧失了容器的许多好处，因为标准容器无法自动管理类型为指针的元素，必须编写额外的大量代码来保证指针最终被正确删除，这通常很麻烦很难实现。

存储 shared_ptr 的容器与存储原始指针的容器功能几乎一样，但 shared_ptr 为程序员做了指针的管理工作，可以任意使用 shared_ptr 而不用担心资源泄漏。

下面的代码示范了将 shared_ptr 应用于标准容器的用法：

```
#include <boost/make_shared.hpp>
int main()
{
    typedef vector<shared_ptr<int> > vs;    // 一个持有 shared_ptr 的标准容器类型
    vs v(10);                               // 声明一个拥有 10 个元素的容器，元素被
                                           // 初始化为空指针

    int i = 0;
    for (vs::iterator pos = v.begin(); pos != v.end(); ++pos)
    {
        (*pos) = make_shared<int>(++i);    // 使用工厂函数赋值
        cout << *(*pos) << ", ";          // 输出值
    }
    cout << endl;
```



```
shared_ptr<int> p = v[9];
*p = 100;
cout << *v[9] << endl;
}
```

这段代码需要注意的是迭代器和 operator[] 的用法，因为容器内存储的是 shared_ptr，我们必须对迭代器 pos 使用一次解引用操作符*以获得 shared_ptr，然后再对 shared_ptr 使用解引用操作符*才能操作真正的值。*(*pos) 也可以直接写成**pos，但前者更清晰，后者很容易让人迷惑。vector 的 operator[] 用法与迭代器类似，也需要使用*获取真正的值。

3.4.6 应用于桥接模式

桥接模式 (bridge) 是一种结构型设计模式，它把类的具体实现细节对用户隐藏起来，以达到类之间的最小耦合关系。在具体编程实践中桥接模式也被称为 pimpl 或者 handle/body 惯用法，它可以将头文件的依赖关系降到最小，减少编译时间，而且可以不使用虚函数实现多态。

scoped_ptr 和 shared_ptr 都可以用来实现桥接模式，但 shared_ptr 通常更合适，因为它支持拷贝和赋值，这在很多情况下都是有用的，比如可以配合容器工作。

本书不可能完整讲述桥接模式和 pimpl 惯用法的所有细节，仅通过一个小例子来说明 shared_ptr 如何用于 pimpl。

首先我们声明一个类 sample，它仅向外界暴露了最小的细节，真正的实现在内部类 impl，sample 用一个 shared_ptr 来保存它的指针：

```
class sample
{
private:
    class impl;                //不完整的内部类声明
    shared_ptr<impl> p;        //shared_ptr 成员变量
public:
    sample();                  //构造函数
    void print();              //提供给外界接口
};
```

在 sample 的 cpp 中完整定义 impl 类和其他功能：

```
class sample::impl            //内部类的实现
{
public:
    void print()
    { cout << "impl print" << endl; }
};
```

```

sample::sample():p(new impl){}           //构造函数初始化 shared_ptr
void sample::print()                      //调用 pimpl 实现 print()
{ p->print();}

```

最后是桥接模式的使用，很简单：

```

sample s;
s.print();

```

桥接模式非常有用，它可以任意改变具体的实现而外界对此一无所知，也减小了源文件之间的编译依赖，使程序获得了更多的灵活性。而 shared_ptr 是实现它的最佳工具之一，它解决了指针的共享和引用计数问题。（关于桥接模式更详细的讨论请见推荐书目 [1]）

9.4.10 小节（368 页）有另一个真实的 pimpl 用法例子，但很遗憾它没有用到智能指针。

3.4.7 应用于工厂模式

工厂模式是一种创建型设计模式，这个模式包装了 new 操作符的使用，使对象的创建工作集中在工厂类或者工厂函数中，从而更容易适应变化，make_shared() 就是工厂模式的一个很好的例子。

但由于 C++ 不能高效地返回一个对象，在程序中编写自己的工厂类或者工厂函数时通常需要在堆上使用 new 动态分配一个对象，然后返回对象的指针。这种做法很不安全，因为用户很容易忘记对指针调用 delete，存在资源泄漏的隐患。

使用 shared_ptr 可以解决这个问题，只需要修改工厂方法的接口，不再返回一个原始指针，而是返回一个被 shared_ptr 包装的智能指针，这样可以很好地保护系统资源，而且会更好地控制接口的使用。

接下来我们使用代码来解释 shared_ptr 应用于工厂模式的用法，首先实现一个纯抽象基类，也就是接口类：

```

class abstract                               //接口类定义
{
public:
    virtual void f() = 0;
    virtual void g() = 0;
protected:
    virtual ~abstract(){}                    //注意这里
};

```

注意 abstract 的析构函数，被定义为保护的，意味着除了它自己和它的子类，其他任何对象都无权调用 delete 来删除它。

然后我们再定义 `abstract` 的实现子类:

```
class impl:public abstract
{
public:
    virtual void f()
    { cout << "class impl f" << endl;}
    virtual void g()
    { cout << "class impl g" << endl;}
};
```

随后的工厂函数返回基类的 `shared_ptr`:

```
shared_ptr<abstract> create()
{ return shared_ptr<abstract>(new impl);}
```

这样我们就完成了全部工厂模式的实现, 现在可以把这些组合起来:

```
int main()
{
    shared_ptr<abstract> p = create(); //工厂函数创建对象
    p->f();                             //可以像普通指针一样使用
    p->g();                             //不必担心资源泄漏, shared_ptr 会自动管理指针
}
```

由于基类 `abstract` 的析构函数是保护的, 所以用户不能做出任何对指针的破坏行为, 即使是用 `get()` 获得了原始指针:

```
abstract *q = p.get();                //正确
delete q;                             //错误
```

这段代码不能通过编译, 因为无法访问 `abstract` 的保护析构函数。

但这不是绝对的, 使用“粗鲁”的方法也可以在 `shared_ptr` 外删除对象, 因为 `impl` 的析构函数是公开的, 所以:

```
impl *q = (impl*)(p.get());
delete q;                                //ok
```

这样就可以任意操作原本处于 `shared_ptr` 控制之下的原始指针了, 但最好永远也不要这样做, 因为这会使 `shared_ptr` 在析构时删除可能已经不存在的指针, 引发未定义行为。

3.4.8 定制删除器

在 3.4.2 小节, 我们特意没有讨论 `shared_ptr` 一种形式的构造函数 `shared_ptr(Y * p, D d)`, 它涉及 `shared_ptr` 的另一个重要概念: 删除器。

`shared_ptr(Y * p, D d)` 的第一个参数是要被管理的指针，它的含义与其他构造函数的参数相同。而第二个删除器参数 `d` 则告诉 `shared_ptr` 在析构时不是使用 `delete` 来操作指针 `p`，而要用 `d` 来操作，即把 `delete p` 换成 `d(p)`。

在这里删除器 `d` 可以是一个函数对象，也可以是一个函数指针，只要它能够像函数那样被调用，使得 `d(p)` 成立即可。对删除器的要求是它必须是可拷贝的，行为必须也像 `delete` 那样，不能抛出异常。

为了配合删除器的工作，`shared_ptr` 提供一个自由函数 `get_deleter(shared_ptr<T> const & p)`，它能够返回删除器的指针。

有了删除器的概念，我们就可以用 `shared_ptr` 实现管理任意资源。只要这种资源提供了它自己的释放操作，`shared_ptr` 就能够保证自动释放。

假设我们有一组操作 `socket` 的函数，使用一个 `socket_t` 类：

```
class socket_t {...};                //socket 类
socket_t* open_socket()              //打开 socket
{
    cout << "open_socket" << endl;
    return new socket_t;
}
void close_socket(socket_t * s)      //关闭 socket
{
    cout << "close_socket" << endl;

    //...其他操作，释放资源
}
```

那么，`socket` 资源对应的释放操作就是函数 `close_socket()`，它符合 `shared_ptr` 对删除器的定义，可以用 `shared_ptr` 这样管理 `socket` 资源：

```
socket_t *s = open_socket();
shared_ptr<socket_t> p(s, close_socket);    //传入删除器
```

在这里删除器 `close_socket()` 是一个自由函数，因此只需要把函数名传递给 `shared_ptr` 就可以了。在函数名前也可以加上取地址操作符 `&`，效果是等价的：

```
shared_ptr<socket_t> p(s, &close_socket);    //传入删除器
```

这样我们就使用 `shared_ptr` 配合定制的删除器管理了 `socket` 资源。当离开作用域时，`shared_ptr` 会自动调用 `close_socket()` 函数关闭 `socket`，再也不会会有资源遗失的担心。

再例如，对于传统的使用 `struct FILE` 的 C 文件操作，也可以使用 `shared_ptr` 配合定制

删除器自动管理，像这样：

```
shared_ptr<FILE> fp(fopen("./1.txt","r"), fclose);
```

当离开作用域时，`shared_ptr` 会自动调用 `fclose()` 函数关闭文件。

`shared_ptr` 的删除器在处理某些特殊资源时非常有用，它使得用户可以定制、扩展 `shared_ptr` 的行为，使 `shared_ptr` 不仅仅能够管理内存资源，而是成为一个“万能”的资源管理工具。

3.4.9 高级议题

本小节讨论关于 `shared_ptr` 的一些高级议题。

`shared_ptr<void>`

`shared_ptr<void>` 能够存储 `void*` 型的指针，而 `void*` 型指针可以指向任意类型，因此 `shared_ptr<void>` 就像是一个泛型的指针容器，拥有容纳任意类型的能力。

但将指针存储为 `void*` 同时也丧失了原来的类型信息，为了在需要的时候正确使用，可以用 `static_pointer_cast<T>` 等转型函数重新转为原来的指针。但这涉及到运行时动态类型转换，它会使代码不够安全，建议最好不要这样使用。

删除器的高级用法

基于 `shared_ptr<void>` 和定制删除器，`shared_ptr` 可以有更惊人的用法。由于空指针可以是任何指针类型，因此 `shared_ptr<void>` 还可以实现退出作用域时调用任意函数。例如：

```
void any_func(void* p)                                //一个可执行任意功能的函数
{ cout << "some operate" << endl;}
int main()
{
    shared_ptr<void> p((void*)0, any_func);            //容纳空指针，定制删除器
}                                                       //退出作用域时将执行 any_func()
```

`shared_ptr<void>` 存储了一个空指针，并指定了删除器是操作 `void*` 的一个函数，因此当它析构时会自动调用函数 `any_func()`，从而执行任意我们想做的工作。

其他高级用法

`shared_ptr` 的功能已经远远超出了智能指针的范围，除了以上的用法外它还有很多其他用途，如包装成员函数、延时释放等，限于篇幅本书不作详细介绍，读者可参考 Boost 说明文档。

3.5 shared_array

shared_array 类似 shared_ptr，它包装了 new[] 操作符在堆上分配的动态数组，同样使用引用计数机制为动态数组提供了一个代理，可以在程序的生命周期里长期存在，直到没有任何引用后才释放内存。

3.5.1 类摘要

shared_array 的类摘要如下：

```
template<class T> class shared_array {
public:
    explicit shared_array(T * p = 0);
    template<class D> shared_array(T * p, D d);
    ~shared_array();

    shared_array(shared_array const & r);

    shared_array & operator=(shared_array const & r);

    void reset(T * p = 0);
    template<class D> void reset(T * p, D d);

    T & operator[](std::ptrdiff_t i) const() const;
    T * get() const;

    bool unique() const;
    long use_count() const;

    void swap(shared_array<T> & b);
};
```

shared_array 的接口与功能几乎是与 shared_ptr 是相同的，主要区别如下：

- 构造函数接受的指针 p 必须是 new[] 的结果，而不能是 new 表达式的结果；
- 提供 operator[] 操作符重载，可以像普通数组一样用下标访问元素；
- 没有*、->操作符重载，因为 shared_array 持有的不是一个普通指针；
- 析构函数使用 delete[] 释放资源，而不是 delete。

3.5.2 用法

`shared_array`就像是`shared_ptr`和`scoped_array`的结合体——即具有`shared_ptr`的优点，也具有`scoped_array`的缺点。有关`shared_ptr`和`scoped_array`的讨论大都适合它，因此这里不再详细讲解，仅给出一个小例子说明：

```
#include <boost/smart_ptr.hpp>
using namespace boost;
int main()
{
    int *p = new int[100];           //一个动态数组
    shared_array<int> sa(p);          //shared_array 代理动态数组
    shared_array<int> sa2 = sa;       //共享数组，引用计数增加

    sa[0] = 10;                      //可以使用 operator[] 访问元素
    assert(sa2[0] == 10);

}                                     //离开作用域，自动删除动态数组
```

同样的，在使用`shared_array`重载的`operator[]`时要小心，`shared_array`不提供数组索引的范围检查，如果使用了超过动态数组大小的索引或者是负数索引将引发可怕的未定义行为。

`shared_array`能力有限，多数情况下它可以用`shared_ptr<std::vector>`或者`std::vector<shared_ptr>`来代替，这两个方案具有更好的安全性和更多的灵活性，而所付出的代价几乎可以忽略不计。

3.6 weak_ptr

`weak_ptr`是为配合`shared_ptr`而引入的一种智能指针，它更像是`shared_ptr`的一个助手而不是智能指针，因为它不具有普通指针的行为，没有重载`operator*`和`->`。它的最大作用在于协助`shared_ptr`工作，像旁观者那样观测资源的使用情况。

3.6.1 类摘要

`weak_ptr`的类摘要如下：

```
template<class T> class weak_ptr
{
public:
    weak_ptr();

    template<class Y> weak_ptr(shared_ptr<Y> const & r);
    weak_ptr(weak_ptr const & r);

    ~weak_ptr();
```

```

weak_ptr & operator=(weak_ptr const & r);

long use_count() const;
bool expired() const;
shared_ptr<T> lock() const;

void reset();
void swap(weak_ptr<T> & b);
};

```

weak_ptr 的接口很小，正如它的名字，是一个“弱”指针，但它能够完成一些特殊的工作，足以证明它的存在价值。

3.6.2 用法

weak_ptr 被设计为与 shared_ptr 共同工作，可以从一个 shared_ptr 或者另一个 weak_ptr 对象构造，获得资源的观测权。但 weak_ptr 没有共享资源，它的构造不会引起指针引用计数的增加。同样，在 weak_ptr 析构时也不会导致引用计数减少，它只是一个静静的观察者。

使用 weak_ptr 的成员函数 use_count() 可以观测资源的引用计数，另一个成员函数 expired() 的功能等价于 use_count() == 0，但更快，表示被观测的资源（也就是 shared_ptr 管理的资源）已经不复存在。

weak_ptr 没有重载 operator* 和 ->，这是特意的，因为它不共享指针，不能操作资源，这正是它“弱”的原因。但它可以使用一个非常重要的成员函数 lock() 从被观测的 shared_ptr 获得一个可用的 shared_ptr 对象，从而操作资源。但当 expired() == true 的时候，lock() 函数将返回一个存储空指针的 shared_ptr。

下面的代码示范了 weak_ptr 的用法：

```

shared_ptr<int> sp(new int(10));           //一个 shared_ptr
assert(sp.use_count() == 1);

weak_ptr<int> wp(sp);                      //从 shared_ptr 创建 weak_ptr
assert(wp.use_count() == 1);              //weak_ptr 不影响引用计数

if (!wp.expired())                        //判断 weak_ptr 观察的对象是否失效
{
    shared_ptr<int> sp2 = wp.lock();       //获得一个 shared_ptr
    *sp2 = 100;
    assert(wp.use_count() == 2);
}                                           //退出作用域, sp2 自动析构, 引用计数减 1

assert(wp.use_count() == 1);

```

```

sp.reset(); //shared_ptr 失效
assert(wp.expired());
assert(!wp.lock()); //weak_ptr 将获得一个空指针

```

3.6.3 获得 this 的 shared_ptr

weak_ptr 的一个重要用途是获得 this 指针的 shared_ptr，使对象自己能够生产 shared_ptr 管理自己：对象使用 weak_ptr 观测 this 指针，这并不影响引用计数，在需要的时候就调用 lock() 函数，返回一个符合要求的 shared_ptr 供外界使用。

这个解决方案被实现为一个惯用法，在头文件 <boost/enable_shared_from_this.hpp> 定义了一个助手类 enable_shared_from_this<T>，它的声明摘要如下：

```

template<class T>
class enable_shared_from_this
{
public:
    shared_ptr<T> shared_from_this();
}

```

使用的时候只需要让想被 shared_ptr 管理的类从它继承即可，成员函数 shared_from_this() 会返回 this 的 shared_ptr。例如：

```

#include <boost/enable_shared_from_this.hpp>
#include <boost/make_shared.hpp>
class self_shared: //一个需要用 shared_ptr 自我管理的类
{
public enable_shared_from_this<self_shared>
{
public:
    self_shared(int n):x(n){}
    int x;
    void print()
    { cout << "self_shared:" << x << endl; }
};
int main()
{
    shared_ptr<self_shared> sp = make_shared<self_shared>(314);
    sp->print();
    shared_ptr<self_shared> p = sp->shared_from_this();
    p->x = 1000;
    p->print();
}

```

需要注意的是千万不能从一个普通对象（非 shared_ptr）使用 shared_from_this() 获取 shared_ptr，例如：

```
self_shared ss;
```

```
shared_ptr<self_shared> p = ss.shared_from_this(); //错误!
```

这样虽然语法上正确，编译也无问题，但在运行时会导致 `shared_ptr` 析构时企图删除一个栈上分配的对象，发生未定义行为。

3.7 intrusive_ptr

`intrusive_ptr` 是一个侵入式的引用计数型指针，它可以用于以下两种情形：

- 对内存占用的要求非常严格，要求必须与原始指针一样；
- 现存代码已经有了引用计数机制管理的对象。

Boost 库不推荐使用 `intrusive_ptr`，因为 `shared_ptr` 已经非常强大且灵活，工作的足够好，可以满足绝大部分（99.99%）的需要。

因此本书不再介绍 `intrusive_ptr`，如果读者真的有非常特别的需求而且 `shared_ptr` 在性能、空间开销等方面影响了程序的运行（几乎是不可能的！），那么可以参考 Boost 文档以了解 `intrusive_ptr` 的详细用法。

3.8 pool 库概述

如果读者学习过操作系统相关的课程，学习过操作系统的内存管理机制和内存分配算法等知识，那么就可能了解“内存池”的概念。简单来说，内存池预先分配了一块大的内存空间，然后就可以在其中使用某种算法实现高效快速的自定义内存分配。

`boost.pool` 库基于简单分隔存储思想实现了一个快速、紧凑的内存池库，不仅能够管理大量的对象，还可以被用做 STL 的内存分配器。某种程度上讲，它近似于一个小型的垃圾回收机制，在需要大量地分配/释放小对象时很有效率，而且完全不需要考虑 `delete`。

`pool` 库包含四个组成部分：最简单的 `pool`、分配类实例的 `object_pool`、单件内存池 `singleton_pool` 和可用于标准库的 `pool_alloc`。

3.9 pool

`pool` 是最简单也最容易使用的内存池类，可以返回一个简单数据类型（POD）^①的内存指针。

① POD 是 C++ 标准中的技术术语，是“普通旧式数据”（Plain Old Data）的缩写。

它位于名字空间 boost，为了使用 pool 组件，需要包含头文件 <boost/pool/pool.hpp>，即：

```
#include <boost/pool/pool.hpp>
using namespace boost;
```

3.9.1 类摘要

pool 的类摘要如下：

```
template <typename UserAllocator = ... >
class pool
{
public:
    explicit pool(size_type requested_size);
    ~pool();
    size_type get_requested_size() const;

    void * malloc();
    void * ordered_malloc();
    void * ordered_malloc(size_type n);
    bool is_from(void * chunk) const;

    void free(void * chunk);
    void ordered_free(void * chunk);
    void free(void * chunks, size_type n);
    void ordered_free(void * chunks, size_type n);

    bool release_memory();
    bool purge_memory();
};
```

3.9.2 操作函数

pool 的模板类型参数 UserAllocator 是一个用户定义的内存分配器，它实现了特定的内存分配算法，通常可以直接用默认的 default_user_allocator_new_delete。

pool 的构造函数接受一个 size_type 类型的整数 requested_size，指示每次 pool 分配内存块的大小（而不是 pool 内存池的大小），这个值可以用 get_requested_size() 获得。pool 会根据需要自动地向系统申请或归还使用的内存，在析构时，pool 将自动释放它所持有的所有内存块。

成员函数 malloc() 和 ordered_malloc() 的行为很类似 C 中的全局函数 malloc()，用 void* 指针返回从内存池中分配的内存块，大小为构造函数中指定的 requested_size。如果内存分配失败，函数会返回 0，不会抛出异常。malloc() 从内存池中任意分配一个内存块，而

`ordered_malloc()`则在分配的同时合并空闲块链表。`ordered_malloc()`带参数的形式还可以连续分配 `n` 块的内存。分配后的内存块可以用 `is_from()` 函数测试是否是从这个内存池分配出去的。

与 `malloc()` 对应的一组函数是 `free()`，用来手工释放之前分配的内存块，这些内存块必须是从这个内存池分配出去的 (`is_from(chunk) == true`)。一般情况内存池会自动管理内存分配，不应该调用 `free()` 函数，除非你认为内存池的空间已经不足，必须释放已经分配的内存。

最后还有两个成员函数：`release_memory()` 让内存池释放所有未被分配的内存，但已分配的内存块不受影响；`purge_memory()` 则强制释放 `pool` 持有的所有内存，不管内存块是否被使用。实际上，`pool` 的析构函数就是调用的 `purge_memory()`。这两个函数一般情况下也不应该由程序员手工调用。

3.9.3 用法

`pool` 很容易使用，可以像 C 中的 `malloc()` 一样分配内存，然后随意使用。除非有特殊要求，否则不必对分配的内存调用 `free()` 释放，`pool` 会很好地管理内存。例如：

```
#include <boost/pool/pool.hpp>
using namespace boost;
int main()
{
    pool<> pl(sizeof(int));                //一个可分配 int 的内存池

    int *p = (int *)pl.malloc();           //必须把 void*转换成需要的类型
    assert(pl.is_from(p));

    pl.free(p);                            //释放内存池分配的内存块
    for (int i = 0; i < 100; ++i)          //连续分配大量的内存
    {   pl.ordered_malloc(10); }

    }                                       //内存池对象析构，所有分配的内存在这里都被释放
```

因为 `pool` 在分配内存失败的时候不会抛出异常，所以实际编写的代码应该检查 `malloc()` 函数返回的指针，以防止空指针错误，不过通常这种情况极少出现：

```
int *p = (int *)pl.malloc();
if ( p != NULL)
    ...
```

关于 `pool<>` 没有更多的解释，因为它真的很容易使用，只需要注意一点：它只能作为普通数据类型如 `int`、`double` 等的内存池，不能应用于复杂的类和对象，因为它只分配内存，不调用构造函数，这个时候我们需要用 `object_pool`。

3.10 object_pool

object_pool 是用于类实例（对象）的内存池，它的功能与 pool 类似，但会在析构时对所有已经分配的内存块调用析构函数，从而正确地释放资源。

object_pool 位于名字空间 boost，为了使用 object_pool 组件，需要包含头文件 <boost/pool/object_pool.hpp>，即：

```
#include <boost/pool/object_pool.hpp>
using namespace boost;
```

3.10.1 类摘要

object_pool 的类摘要如下：

```
template <typename ElementType >
class object_pool: protected pool
{
public:
    object_pool();
    ~object_pool();

    element_type * malloc();
    void free(element_type * p);
    bool is_from(element_type * p) const;

    element_type * construct(...);
    void destroy(element_type * p);
};
```

3.10.2 操作函数

object_pool 是 pool 的子类，但它使用的是保护继承，因此不能使用 pool 的接口，但基本操作还是很相似的。

object_pool 的模板类型参数 ElementType 指定了 object_pool 要分配的元素类型，要求其析构函数不能抛出异常。一旦在模板中指定了类型，object_pool 实例就不能再用于分配其他类型的对象。

malloc() 和 free() 函数分别分配和释放一块类型为 ElementType* 的内存块，同样，可以用 is_from() 来测试内存块的归属，只有是本内存池分配的内存才能被 free() 释放。但它们被调用时并不调用类的构造函数和析构函数，也就是说操作的是一块原始内存块，里面的值是未定义的，因此我们应当尽量少使用 malloc() 和 free()。

object_pool 的特殊之处是 construct() 和 destroy() 函数, 这两个函数是 object_pool 的真正价值所在。construct() 实际上是一组函数, 有多个参数的重载形式 (目前最多支持 3 个参数, 但可以扩展), 它先调用 malloc() 分配内存, 然后再在内存块上使用传入的参数调用类的构造函数, 返回的是一个已经初始化的对象指针。destroy() 则先调用对象的析构函数, 然后再用 free() 释放内存块。

这些函数都不会抛出异常, 如果内存分配失败, 将返回 0。

3.10.3 用法

object_pool 的用法也是很简单, 我们既可以像 pool 那样分配原始内存块, 也可以使用 construct() 来直接在内存池中创建对象。当然, 后一种使用方法是更方便的, 也是本书所推荐的。

下面的代码示范了 object_pool 的用法:

```
#include <boost/pool/object_pool.hpp>
using namespace boost;
struct demo_class                                //一个示范用的类
{
public:
    int a,b,c;
    demo_class(int x = 1, int y = 2, int z = 3):a(x),b(y),c(z){}
};
int main()
{
    object_pool<demo_class> pl;                    //对象内存池

    demo_class *p = pl.malloc();                   //分配一个原始内存块
    assert(pl.is_from(p));

    //p 指向的内存未经过初始化
    assert(p->a!=1 || p->b != 2 || p->c !=3);

    p = pl.construct(7, 8, 9);                     //构造一个对象,可以传递参数
    assert(p->a == 7);

    object_pool<string> pls;                         //定义一个分配 string 对象的内存池
    for (int i = 0; i < 10 ; ++i)                   //连续分配大量 string 对象
    {
        string *ps = pls.construct("hello object_pool");
        cout << *ps << endl;
    }
}                                                    //所有创建的对象在这里都被正确析构、释放内存
```

3.10.4 使用更多的构造参数

默认情况下, 在使用 `object_pool` 的 `construct()` 的时候我们只能最多使用 3 个参数来创建对象。大多数情况下这都是足够的, 但有的时候我们可能会定义 3 个以上参数的构造函数, 此时 `construct()` 的默认重载形式就不能用了。

但 `construct()` 被设计为是可以扩展的, 它基于宏预处理 `m4` (通常 Unix 和 Linux 系统自带, 也有 Windows 的版本) 实现了一个扩展机制, 可以自动生成接受任意数量参数的 `construct()` 函数。

`pool` 库在目录 `/boost/pool/detail` 下提供了一个名为 `pool_construct.m4` 和 `pool_construct_simple.m4` 的脚本, 并同时提供可在 Unix/Linux 和 Windows 下运行的同名 `sh` 和 `bat` 可执行脚本文件。只需要简单地向批处理脚本传递一个整数的参数 `N`, `m4` 就会自动生成能够创建具有 `N` 个参数的 `construct()` 函数源代码。

例如, 在 Linux 下, 执行命令:

```
./pool_construct_simple.sh 5; ./pool_construct.sh 5
```

将生成两个同名的 `.inc` 文件, 里面包含了新的 `construct()` 函数定义, 能够支持最多传递 5 个参数创建对象。由于 `m4` 生成的是 C++ 源代码, 因此 `.inc` 文件也可以拷贝到其他操作系统的 Boost 库中使用。

扩展 `construct()` 函数时请慎重, 数量过多的参数定义会导致程序的编译时间增加, 所以最好只定义最合适最需要的参数数量, 而不是一味地求多。

如果只是临时的需要增加 `construct()` 函数的参数数量, 或者工作的系统上 `m4` 不可用, 我们也可以简单地定义一个辅助模板函数。

下面的代码模仿 `construct()` 函数实现了一个可接受 4 个参数的创建函数:

```
template<typename P, typename T0, typename T1, typename T2, typename T3>
inline typename P::element_type*
construct(P& p, const T0& a0, const T1& a1, const T2& a2, const T3& a3)
{
    typename P::element_type* mem = p.malloc();
    assert(mem != 0);
    new (mem) P::element_type(a0, a1, a2, a3);
    return mem;
}
```

自由函数 `construct()` 接受 5 个参数, 第一个是 `object_pool` 对象, 其后是创建对象所

需的 4 个参数，要创建的对象类型可以使用 `object_pool` 的内部类型定义 `element_type` 来获得。函数中首先调用 `malloc()` 分配一块内存，然后调用不太常见的“定位 new 表达式” (`placement new expression`) 创建对象。

假设我们有如下的一个 4 参数构造函数的类和一个 `object_pool` 对象：

```
struct demo_class
{
    demo_class(int, int, int, int)           //构造函数接受 4 个参数
    {   cout << "demo_class ctor" << endl;}
    ~demo_class()
    {   cout << "demo_class dtor" << endl;}
};
object_pool<demo_class> pl;
```

那么使用 `m4` 和自定义的 `construct()` 创建对象的代码就是：

```
demo_class* d1 = pl.construct(1,2,3,4);      //使用 m4 扩展
demo_class* d2 = construct(pl, 1,2,3,4);     //使用自定义扩展
```

3.11 singleton_pool

`singleton_pool` 与 `pool` 的接口完全一致，可以分配简单数据类型 (POD) 的内存指针，但它是一个单件，并提供线程安全。

由于目前 Boost 还未提供标准的单件库，`singleton_pool` 在其内部实现了一个较简单、泛型的单件类，保证在 `main()` 函数运行之前就创建单件（详情可参考 4.6.1 小节，117 页）。

`singleton_pool` 位于名字空间 `boost`，为了使用 `singleton_pool` 组件，需要包含头文件 `<boost/pool/singleton_pool.hpp>`，即：

```
#include <boost/pool/singleton_pool.hpp>
using namespace boost;
```

3.11.1 类摘要

`singleton_pool` 的类摘要如下：

```
template <typename Tag, unsigned RequestedSize>
class singleton_pool
{
public:
    static bool is_from(void * ptr);
```



```
static void * malloc();
static void * ordered_malloc();
static void * ordered_malloc(size_type n);

static void free(void * ptr);
static void ordered_free(void * ptr);
static void free(void * ptr, std::size_t n);
static void ordered_free(void * ptr, size_type n);

static bool release_memory();
static bool purge_memory();
};
```

3.11.2 用法

singleton_pool 主要有两个模板类型参数（其余的可以使用缺省值）。第一个 Tag 仅仅是用于标记不同的单件，可以是空类，甚至是声明（这个用法还被用于 boost.exception，参见 4.9 小节，136 页）。第二个参数 RequestedSize 等同于 pool 构造函数中的整数 requested_size，指示 pool 分配内存块的大小。

singleton_pool 的接口与 pool 完全一致，但成员函数均是静态的，因此不需要声明 singleton_pool 的实例^①，直接用域操作符::来调用静态成员函数。因为 singleton_pool 是单件，所以它的生命周期与整个程序同样长，除非手动调用 release_memory() 或 purge_memory()，否则 singleton_pool 不会自动释放所占用的内存。除了这两点，singleton_pool 的用法与 pool 完全相同。

下面的代码示范了 singleton_pool 的用法：

```
#include <boost/pool/singleton_pool.hpp>
using namespace boost;
struct pool_tag{}; //仅仅用于标记的空类
typedef singleton_pool<pool_tag, sizeof(int)> spl; //内存池定义
int main()
{
    int *p = (int *)spl::malloc(); //分配一个整数内存块
    assert(spl::is_from(p));
    spl::release_memory(); //释放所有未被分配的内存
} //spl 的内存直到程序结束才完
//全释放，而不是退出作用域
```

singleton_pool 在使用时最好使用 typedef 来简化名称，否则会使得类型名过于冗长而

① 因为使用了单件模式，用户也无法创建 singleton_pool 的实例。

难以使用。如代码中所示：

```
typedef singleton_pool<pool_tag, sizeof(int)> spl;
```

用于标记的类 `pool_tag` 可以再进行简化，直接在模板参数列表中声明 `tag` 类，这样可以在一条语句中完成对 `singleton_pool` 的类型定义，例如：

```
typedef singleton_pool<struct pool_tag, sizeof(int)> spl;
```

3.12 pool_alloc

`pool_alloc` 提供了两个可以用于标准容器模板参数的内存分配器，分别是 `pool_alloc` 和 `fast_pool_allocator`，它们的行为与之前的内存池类有一点不同——当内存分配失败时会抛出异常 `std::bad_alloc`。它们位于名字空间 `boost`，需要包含头文件 `<boost/pool/pool_alloc.hpp>`。

除非有特别的需求，我们应该总使用 STL 实现自带的内存分配器，使用 `pool_alloc` 需要经过仔细的测试，以保证它与容器可以共同工作。

下面的代码示范了 `pool_alloc` 的用法：

```
#include <boost/pool/pool_alloc.hpp>
using namespace boost;
int main()
{
    vector<int, pool_allocator<int> > v;
                                //使用 pool_allocator 代替标准容器默认的内存分配器
    v.push_back(10);           //vector 将使用新的分配器良好工作
    cout << v.size();
}
```

3.13 总结

内存管理是 C++ 程序开发中永恒的话题，因为没有垃圾回收机制，小心谨慎地管理内存等系统资源是每一个 C++ 程序员都必须面对的问题。C++98 标准提供了 `auto_ptr`，可以自动释放资源，但没有解决所有问题。本章讨论了 Boost 关于内存管理的两个库：`smart_ptr` 和 `pool`，并对 `smart_ptr` 倾注了大量的篇幅。

`boost.smart_ptr` 库提供了数种新型智能指针，弥补了 `std::auto_ptr` 的不足，可以有效地消除 `new` 和 `delete` 的显示使用，减少甚至杜绝代码资源泄漏。

`scoped_ptr` 是 `smart_ptr` 库中最容易学习和使用的一个，它的行为类似 `auto_ptr`，但

所有权更明确，清晰地表明了这种智能指针只能在声明的作用域中使用，不能转让，任何对它的复制企图都会失败。这个特点对代码的后期维护工作非常有用。

`shared_ptr` 可能是最有用的智能指针，也是这些智能指针中最“智能”的一个，不仅可以管理内存，也可以管理其他系统资源，能够应用于许多场合。它可以自动地计算指针的引用计数，其行为最接近原始指针。几乎可以在任何可以使用原始指针的地方使用 `shared_ptr`，并且不用承担资源泄漏的风险。`shared_ptr` 不仅可以保存指针，通过配置删除器也可以自动释放指针关联的资源。

在基本的用法之外，我们还讨论了 `shared_ptr` 的很多其他用法，如实现 `pimpl` 惯用法、应用于工厂模式、持有任意对象的指针等，这些用法进一步展示了它的强大功能。为了方便 `shared_ptr` 的使用，`smart_ptr` 库还提供了工厂函数 `make_shared()`，进一步消除了代码中 `new` 操作符的使用。

`scoped_array` 和 `shared_array` 是 `scoped_ptr` 和 `shared_ptr` 对动态数组的扩展，它们为动态数组提供了可自动删除的代理，`shared_array` 比 `scoped_array` 有更多的用途。但我们更应该使用 `vector` 和 `shared_ptr<vector<>>`，除非程序对性能有非常苛刻的要求。

本章还简要讨论了 `smart_ptr` 的另两个组件：`weak_ptr` 能够“静态”地观察 `shared_ptr` 而不影响引用计数，`intrusive_ptr` 则为实现侵入式智能指针提供了技术方案。

`pool` 库是 Boost 程序库在内存管理方面提供的另一个有用工具，它实现了高效的内存池，用于管理内存资源。`pool` 库提供了 `pool`、`object_pool`、`singleton_pool` 和 `pool_alloc` 四种形式的内存池，适合于各种情形的应用。可以完全把它们当做是一个小型的垃圾回收机制，在内存池中随意地动态创建对象，而完全不用关心它的回收，也不用对原有类做任何形式的修改。

`pool` 库的四个内存池类中前三个都很有用，尤其是 `object_pool`，它可以统一地管理各种对象的创建与销毁，能够很好地应用在各种规模的面向对象软件系统中。至于 `pool_alloc`，它是符合 C++ 标准的一个内存分配器实现，快速且高效，但通常 STL 自带的内存分配器会更好地与容器配合工作，使用 `pool_alloc` 时需要仔细地评估可能带来的影响。

`pool` 库还提供一个底层的实现类 `simple_segregated_storage`，它实现了简单分隔存储的管理机制，是 `pool` 库其他类的基础。它不适合大多数库用户，但可以作为自行实现内存池类的一个很好的起点。

第 4 章

实用工具

本章详细讲述了 Boost 库中提供的十多个有用的小工具（不全是类，也有宏。是的，宏也可以很有用）。说它们小，是因为它们实现的功能比较单纯，代码也都比较简单（但也有例外），在实际产品代码中也往往处于不起眼的角落之中。

但这些 Boost 组件都非常非常有用。

正是因为小，它们几乎在程序中无处不在，就像轴承里的滚珠或者汽车引擎的润滑油，能够使程序运转的更加良好更加有效率。有了它们，会使你的编程工作更加轻松愉快。

你是否曾经反复地实现一个不可复制的类或者一个单件类？是否曾经为编写操作符重载而不停地敲打键盘？是否曾经为验证性测试而重复输入大量数据？是否……^①而这些实现仅仅是由于少量的代码不同。很多开发团队都曾经编写过大量用于自己项目的实用工具类，但因为接口不标准、文档不齐全等各种原因导致难以在更大的范围里复用。下面的 Boost 组件会把程序员从这种机械并且乏味的代码拷贝粘贴活动中彻底解放出来。

4.1 noncopyable

noncopyable 允许程序轻松地实现一个禁止复制的类。

noncopyable 位于名字空间 boost，为了使用 noncopyable 组件，需要包含头文件 `<boost/noncopyable.hpp>` 或者 `<boost/utility.hpp>`，后者包含了数个小工具的实现：

```
#include <boost/noncopyable.hpp> //或者
```

① 如果你没有，或许你的 C++ 知识还不够深入。

```
#include <boost/utility.hpp>
```

4.1.1 原理

在 C++ 中定义一个类时，如果不明确定义拷贝构造函数和拷贝赋值操作符，编译器会为我们自动生成这两个函数。

例如：

```
class empty_class {};
```

这样一个简单的“空”类，编译器在处理时会“默默地”为它增加拷贝构造函数和拷贝赋值操作符，真实代码类似于：

```
class empty_class
{
public:
    empty_class (const empty_class &){...}           //拷贝构造函数
    empty_class & operator=(const empty_class &){...} //拷贝赋值
};
```

一般情况下这是有用的，比如可以自动支持 `swap()`、符合容器的拷贝语义、可以放入标准容器处理，但有的时候我们不需要类的复制语义，希望禁止复制类的实例。

这是一个很经典的 C++ 惯用法，原理很好理解，只需要私有化拷贝构造函数和赋值操作符即可，手写代码也很简单（3.2 小节，63 页的 `scoped_ptr` 就使用了这个惯用法），例如：

```
class do_not_copy
{
private:
    //声明即可，不需要实现代码
    do_not_copy (const do_not_copy &);
    void operator=(const do_not_copy &);
};
```

但如果程序中有大量这样的类，重复写这样的代码是相当乏味的，而且代码出现的次数越多越容易增大手写出错的几率。

4.1.2 用法

`noncopyable` 为实现不可拷贝的类提供了简单清晰的解决方案：从 `boost::noncopyable` 派生即可。

使用 `noncopyable`，上面的例子可简化为：


```
#include <boost/utility.hpp>
class do_not_copy: boost::noncopyable
{...};
```

注意，这里使用默认的私有继承是允许的。我们也可以显式写出 `private` 或者 `public` 修饰词，但效果是相同的。因此直接这样写少输入了一些代码，也更清晰，并且表明了 HAS-A 关系（而不是 IS-A）。

如果有其他人误写了代码（很可能是没有仔细阅读接口文档），企图拷贝构造或者赋值 `do_not_copy`，那么将不能通过编译器的审查：

```
do_not_copy d1;
do_not_copy d2(d1);           //编译出错!
do_not_copy d3;
d3 = d1;                      //编译出错!
```

在 VC8 下编译，会报出类似下面的错误提示：

```
boost::noncopyable_::noncopyable::noncopyable' : cannot access private member
declared in class 'boost::noncopyable_::noncopyable'
```

这条错误信息以数个很明显的“`noncopyable`”告诉我们，类是禁止拷贝的。

只要有可能，就使用 `boost::noncopyable`，它明确无误地表达了类设计者的意图，对用户更加友好，而且与其他 Boost 库也配合得很好。

4.2 typedef

`typedef` 库使用宏模拟了 C++0x 新增加的 `typedef` 和 `auto` 关键字，可以减轻书写烦琐的变量类型声明的工作，简化代码。

为使用 `typedef` 库，需要包含头文件 `<boost/typedef/typedef.hpp>`，即：

```
#include <boost/typedef/typedef.hpp>
```

4.2.1 动机

C++ 是一种静态强类型语言，所有变量在使用前都必须声明其类型，这使得 C++ 具有运行速度快、代码规范等很多优点。但有的时候，这个优点却是麻烦的来源——尤其是 C++ 引入名字空间特性后——会导致烦琐的类型声明。

考虑如下 STL 迭代器的声明：

```
std::map<std::string, std::string>::iterator pos = ...;
```

仅仅声明一个用于遍历 map 容器的迭代器，变量名不过三个字符，而类型声明却长达 44 个字符，是变量名的十多倍。

再看使用 Boost 库的情况，以下冗长的代码声明了一个 date 的 shared_ptr：

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <boost/shared_ptr.hpp>
boost::shared_ptr<boost::gregorian::date> pd = ...;
```

使用 using 关键字打开名字空间可以部分减少类型声明的长度，但没有从根本上解决这个问题，某些泛型编程技术的运用会产生难以想象的类型。下面的例子摘自 Boost 说明文档，没有使用名字空间修饰：

```
lambda_functor<
    lambda_functor_base<
        logical_action<and_action>,
        tuple<
            lambda_functor<
                lambda_functor_base<
                    relational_action<greater_action>,
                    tuple<
                        lambda_functor<placeholder<1> >,
                        int const
                    >
                >
            >,
            lambda_functor<
                lambda_functor_base<
                    relational_action<less_action>,
                    tuple<
                        lambda_functor<placeholder<2> >,
                        int const
                    >
                >
            >
        >
    >
>
f = _1 > 15 && _2 < 20;
```

这串“可怕的”代码定义了一个简单的 boost.lambda^①函数对象（function object），几乎可以断定，没有人能够一次性写出正确的类型声明。

随着编程技术的精进，将来读者也很有可能不得不写出这样的代码。C++ 静态强类型的优点

① boost.lambda 库可以就地定义匿名函数对象。

在这时已经成为了阻碍程序员生产力的“缺陷”。

有鉴于此,C++标准委员会考虑在新标准中增加关键字 `typedef`、重新定义来自 C 语言的 `auto` 关键字以解决上述问题。

使用 `typedef` 关键字,可以让 C++编译器在编译时自动推导表达式的类型,之前的三个例子可以改写成如下形式:

```
typedef(...) pos = ...;
typedef(...) pd = ...;
typedef(_1 > 15 && _2 < 20) f = _1 > 15 && _2 < 20;
```

使用 `auto` 关键字则进一步简化了写法,更加简单:

```
auto pos = ...;
auto pd = ...;
auto f = _1 > 15 && _2 < 20;
```

但在 C++0x 真正到来之前,我们还不得不使用烦琐的声明语法。为此 `typedef` 提供了使用库的解决方案,无论编译器是否支持新的 `typedef` 或者 `auto` 关键字,它都可以正常工作。

4.2.2 用法

头文件 `<boost/typedef.hpp>` 里定义了两个宏: `BOOST_TYPEDEF` 和 `BOOST_AUTO`,分别用于仿真 C++新标准的 `typedef` 和 `auto` 关键字,可以在编译期自动推导表达式的类型。它们不仅能够推导 C++语言内建的 `int`、`double`、数组、函数指针等等类型,也支持标准库中的容器类型,使程序员再也不需要写复杂的类型定义就能够轻松声明变量。

这两个宏完全模仿了 `typedef` 和 `auto` 关键字的用法,除了因为宏的语法限制而不得不使用逗号分隔参数,宏的简要声明如下:

```
#define BOOST_TYPEDEF(Expr) \
    boost::type_of::decode_begin<BOOST_TYPEDEF_ENCODED_VECTOR(Expr) >::type
#define BOOST_AUTO(Var, Expr) BOOST_TYPEDEF(Expr) Var = Expr
```

`BOOST_TYPEDEF` 使用一个表达式作为宏的参数,它可以如 `typedef` 那样推导出表达式 `Expr` 的类型。`BOOST_AUTO` 的功能则与 `auto` 类似,使用 `BOOST_TYPEDEF` 推导表达式 `Expr` 类型,然后用这个类型声明变量 `Var`,并将表达式的结果赋值给 `Var`。由于 `BOOST_AUTO` 实际上调用了 `BOOST_TYPEDEF`,因此它不仅能够声明普通变量,也能够在变量名前加上 `&` 或者 `*` 修饰,声明引用或者指针变量。

示范 `BOOST_TYPEDEF` 和 `BOOST_AUTO` 用法的代码如下:

```
#include <boost/typedef/typedef.hpp>
vector<string> func() //一个返回 vector<string>的函数
```



```

{
    vector<string> v(10);
    return v;
}
int main()
{
    BOOST_TYPEOF(2.0*3) x = 2.0 * 3;           //推导类型为 double
    BOOST_AUTO(y, 2+3);                         //推导类型为 int

    BOOST_AUTO(&a, new double[20]);             //推导类型为 double*的引用
    BOOST_AUTO(p, make_pair(1, "string"));       //推导类型为 pair<int,const char*>

    BOOST_AUTO(v, func());                     //推导类型为 vector<string>
}

```

4.2.3 向 typeof 库注册自定义类

typeof 库支持 C++ 内置的基本类型和 STL 中的大多数类型，但它没有智能到支持任何类型的程度。如果想让用户自己定义的类型能够应用于 typeof 库，则需要使用库提供的一组宏注册后才能使用。

要向 typeof 库注册类型，首先要以如下的语句开始：

```
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()
```

之后的注册类型使用宏 BOOST_TYPEOF_REGISTER_TYPE，它必须在全局名字空间使用，宏的参数是类型的完全名称，包括名字空间限定。

示范如何注册类型的代码如下：

```

#include <boost/typeof/typeof.hpp>
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()
namespace ex
{
    struct demo_class{int a,b;};                //一个简单的类
}
BOOST_TYPEOF_REGISTER_TYPE(ex::demo_class)      //向 typeof 库注册类

int main()
{
    BOOST_AUTO(x, make_pair("test", ex::demo_class()));
    cout << typeid(x).name() << endl;
    x.second.a = 10;
    x.second.b = 20;
    cout << x.second.a << x.second.b << endl;
}

```

4.2.4 高级议题

本小节讨论关于 `typeof` 的一些高级议题。

为 BOOST_AUTO 更名

`BOOST_AUTO` 是一个非常有用的宏，它能够使变量的赋值工作更加轻松容易。但它的名字有点过长，书写起来不太方便，虽然这完全是为了符合 Boost 库的命名规范。

如果想要让使用 `BOOST_AUTO` 的代码看起来更漂亮优雅一些，可以再使用宏定义给它换个名字，比如：

```
#define    auto_t BOOST_AUTO           //使用 auto_t 来代替 BOOST_AUTO
int main()
{
    auto_t(p, make_shared<int>(10));
    assert(*p == 10);
}
```

这个小小的技巧会使代码看起来更美观，更类似 C++0x 风格。

模板类型自动推导

`typeof` 库不仅能够推导一般表达式的类型，也能够推导带有模板的复杂表达式，在模板语境中时需要使用 `BOOST_TYPEOF_TPL` 和 `BOOST_AUTO_TPL`。对于泛型编程的讨论已超出了本书的范围，故在此不做更深入的介绍。

4.3 optional

`optional` 库使用“容器”语义，包装了“可能产生无效值”的对象，实现了“未初始化”的概念。

`optional` 位于名字空间 `boost`，为了使用 `optional`，需要包含头文件 `<boost/optional.hpp>`，即：

```
#include <boost/optional.hpp>
using namespace boost;
```

4.3.1 “无意义”的值

函数并不是总能返回有效的返回值，很多时候函数可能返回“无意义”的值，这不意味着函数执行失败，而是表明函数正确执行了，但结果却不是有用的值。如果用数学语言来解释，就是

返回值位于函数解空间之外。

例如，求一个数的倒数，在实数域内开平方，在字符串中查找子串，它们都可能返回无效的值。有些无效返回的情况可以用抛出异常的方式来通知用户，但有的情况下这样代价很高，或者不允许异常，这时必须要以某种合理的高效的方式通知用户。

表示返回值无意义最常用的做法是增加一个“哨兵”的角色，它位于解空间之外，如 `NULL`、`-1`、`EOF`、`string::npos`、`vector::end()` 等。但这些做法不够通用，而且很多时候不存在解空间之外的“哨兵”。

`optional` 使用“容器”语义，为这种“无效值”的情形提供了一个较好的解决方案。

4.3.2 类摘要

`optional` 很像一个仅能存放一个元素的容器，它实现了“未初始化”的概念：如果元素未初始化，那么容器就是空的，否则，容器内就是有效的、已经初始化的值。

`optional` 的类简要声明如下：

```
template<class T>
class optional
{
public :

    optional () ;
    optional ( T const& v ) ;
    optional ( bool condition, T v ) ;
    optional& operator= ( T const& rhs ) ;

    T*      operator ->() ;
    T&      operator *() ;
    T&      get() ;
    T*      get_ptr() ;
    T const& get_value_or( T const& default ) const ;

    bool operator!() const ;
};
```

`optional` 的真实接口很复杂，因为它要能够包装任何的类型，但实际的接口还是比较简单并且易于理解的，下面的章节将进行详细说明。

4.3.3 操作函数

`optional` 的模板类型参数 `T` 可以是任何类型，就如同一个标准容器对元素的要求，并不需

要 `T` 具有缺省构造函数，但必须是可拷贝构造的。

可以有很多方式创建 `optional` 对象，例如：

- 无参的 `optional()` 或者 `optional(boost::none)` 构造一个未初始化 `optional` 对象。参数 `boost::none` 是一个类似空指针的 `none_t` 类型常量，表示未初始化；
- `optional(v)` 构造一个已初始化的 `optional` 对象，其值为 `v` 的拷贝。如果模板类型为 `T&`，那么 `optional` 内部持有对引用的包装；
- `optional(condition, v)` 根据条件 `condition` 来构造 `optional` 对象，如果条件成立 (`true`) 则初始化为 `v`，否则为未初始化；
- 此外 `optional` 还支持拷贝构造和赋值操作，可以从另一个 `optional` 对象构造。当想让一个 `optional` 对象重新恢复到未初始化状态时，可以向对象赋 `none` 值。

`optional` 采用了指针语义来访问内部保存的元素，这使得 `optional` 未初始化时的行为就像一个空指针。它重载了 `operator*` 和 `operator->` 以实现与指针相同的操作，`get()` 和 `get_ptr()` 可以以函数的操作形式获得元素的引用和指针。

成员函数 `get_value_or(default)` 是一个特别的访问函数，可以保证返回一个有效的值，如果 `optional` 已初始化，那么返回内部的元素，否则返回 `default`。

`optional` 也可以用隐式类型转换进行 `bool` 测试(用于条件判断)，就像一个对指针的判断。

`optional` 还全面支持比较运算，包括 `==`、`!=`、`<`、`<=`、`>`、`>=`。与普通指针比较的“浅比较”(仅比较指针值)不同，`optional` 的比较是“深比较”，同时加入了对未初始化情况的判断。

4.3.4 用法

`optional` 的接口简单明了，把它认为是一个大小为 1 并且行为类似指针的容器就可以了，或者把它想象成是一个类似 `scoped_ptr`、`shared_ptr` 的智能指针(但要小心，`optional` 不是智能指针，用法类似但用途不同)。

示范 `optional` 基本用法的代码如下：

```
#include <boost/optional.hpp>
using namespace boost;
int main()
{
    optional<int> op0;                                //一个未初始化的 optional 对象
    optional<int> opl(none);                          //同上,使用 none 赋予未初始化值
    assert(!op0);
```

```

    assert(op0 == op1);
    assert(op1.get_value_or(253) == 253);           //获取可选值

    optional<string> ops("test");                  //初始化为字符串 test
    cout << *ops << endl;                          //用解引用操作符获取值

    vector<int> v(10);
    optional<vector<int>& > opv(v);                  //容纳一个容器的引用
    assert(opv);

    opv->push_back(5);                              //使用箭头操作符操纵容器
    assert(opv->size() == 11);

    opv = none;                                     //置为未初始化状态
    assert(!opv);
}

```

这段代码演示了 `optional` 的一些基本操作，接下来我们再看一个略微复杂的例子，代码使用 `optional` 作为函数的返回值，解决了本节一开始提出的几个问题：

```

#include <boost/optional.hpp>

using namespace boost;
optional<double> calc(int x)                        //计算倒数
{
    return optional<double>( x != 0, 1.0 / x); //条件构造函数
}
optional<double> sqrt_op(double x)                 //计算实数的平方根
{
    return optional<double>(x > 0, sqrt(x)); //条件构造函数
}
int main()
{
    optional<double> d = calc(10);
    if (d)                                           //bool 语境测试 optional 的有效性
    {
        out << *d << endl; }

    d = sqrt_op(-10);
    if (!d)                                         //使用重载的逻辑非操作符
    {
        cout << "no result"<< endl; }
}

```

4.3.5 工厂函数

`optional` 提供一个类似 `make_pair()`、`make_shared()` 的工厂函数 `make_optional()`，可以根据参数类型自动推导 `optional` 的类型，用来辅助创建 `optional` 对象。它的声明如下：

```
optional<T> make_optional( T const& v );
```



```
optional<T> make_optional( bool condition, T const& v );
```

但 `make_optional()` 无法推导出 `T` 引用类型的 `optional` 对象，因此如果需要一个 `optional<T&>` 的对象，就不能使用 `make_optional()` 函数。

示范 `make_optional()` 函数用法的代码如下：

```
#include <boost/optional.hpp>

using namespace boost;
int main()
{
    BOOST_AUTO(x, make_optional(5));
    assert(*x == 5);

    BOOST_AUTO(y, make_optional<double>((*x > 10), 1.0));
    assert(!y);
}
```

4.3.6 高级议题

本小节讨论关于 `optional` 的一些高级议题。

异常

`optional<T>` 同 STL 容器一样，只提供基本的异常保证，不会超过被包装的类型 `T`，它自身不抛出任何异常，只有在 `T` 构造时可能会抛出异常。

就地创建

`optional<T>` 要求类型 `T` 具有拷贝语义，因为它内部会保存值的拷贝，但很多时候复杂对象的拷贝代价很高，而且这个值仅仅作作为拷贝的临时用途，是一种浪费。因此 `optional` 库提出了“就地创建”的概念，可以不要求类型具有拷贝语义，直接用构造函数所需的参数创建对象。这导致发展出了另一个 Boost 库——`in_place_factory`。

`in_place_factory` 库本书不作详细介绍，示范 `in_place()` 用于 `optional` 用法的代码如下：

```
#include <boost/optional.hpp>
#include <boost/utility/in_place_factory.hpp>
using namespace boost;
int main()
{
    //就地创建 string 对象，不需要临时对象 string("...")
    optional<string> ops(in_place("test in_place_factory"));
```

```
cout << *ops;

//就地创建 std::vector 对象, 不需要临时对象 vector(10, 3)
optional<vector<int> > opp(in_place(10, 3));
assert(opp->size() == 10);
assert((*opp)[0] == 3);
}
```

引用类型

`optional` 的模板参数类型可以是引用 (`T&`), 它在很多方面与原始类型 `T` 有不同, 比如无法使用就地创建、就地赋值。与 C++ 语言内置的引用类型不同的是, 它可以声明时不指定初值, 并且在赋值时转移包装的对象, 而不是对原包装的对象赋值。

4.4 assign

许多情况下我们都需要为容器初始化或者赋值, 填入大量的数据, 比如初始错误代码和错误信息, 或者是一些测试用的数据。STL 容器仅提供了容纳这些数据的方法, 但填充的步骤却是相当地麻烦, 必须重复调用 `insert()` 或者 `push_back()` 等成员函数, 这正是 `boost.assign` 出现的理由。

`assign` 库重载了赋值操作符 `operator+=`、逗号操作符 `operator,` 和括号操作符 `operator()`, 可以用难以想象的简洁语法非常方便地对 STL 容器赋值或者初始化。在需要填入大量初值的地方很有用, 本书 8.1 小节 (327 页) 介绍的 `foreach` 库和其他很多地方都大量使用了 `assign`, 可以做进一步的参考。

`assign` 库位于名字空间 `boost::assign`, 为了使用 `assign` 库, 需要包含头文件 `<boost/assign.hpp>`, 它包含了大部分 `assign` 库的工具, 即:

```
#include <boost/assign.hpp>
using namespace boost::assign;
```

4.4.1 使用操作符+=向容器增加元素

`boost.assign` 的用法非常简单, 由于重载了操作符 `+=` 和逗号, 可以用简洁到令人震惊的语法完成原来用许多代码才能完成的工作, 如果不熟悉 C++ 操作符重载的原理你甚至都不会意识到在简洁语法下的复杂工作。

使用 `assign` 库时必须使用 `using` 指示符, 只有这样才能让重载的 `+=`, 等操作符在作用域内生效。例如:

```
#include <boost/assign.hpp>
```

```

int main()
{
    using namespace boost::assign;           //很重要, 启用 assign 库的功能
    vector<int> v;                             //标准向量容器
    v += 1,2,3,4,5, 6*6;                       //用 operator+=和, 填入数据

    set<string> s;                             //标准集合容器
    s += "cpp", "java", "c#", "python";

    map<int, string> m;                         //标准映射容器
    m += make_pair(1, "one"), make_pair(2, "two");
}

```

上面的代码示范了 assign 库操作标准容器的能力。+=操作符后可以接若干个可被容器容纳的元素，元素之间使用逗号分隔。元素不一定是常量，表达式或者函数调用也是可以接受的，只要其结果能够转换成容器可容纳的类型。比较特别的是 map 容器，必须使用 make_pair() 辅助函数来生成容器元素，单纯地用括号把 pair 的两个成员括起来是无效的。

operator+=很好用，但有一点遗憾，它仅限应用于 STL 中定义的标准容器（vector、list、set 等），对于其他类型的容器（如 Boost 新容器）则无能为力。

4.4.2 使用操作符()向容器增加元素

因为 operator+=使用上有些小的限制，而且在处理 map 容器时也显得有些麻烦，assign 库使用操作符 operator() 提供更通用的解决方案。

我们不能直接使用 operator()，而应当使用 assign 库提供三个辅助函数 insert()、push_front()、push_back()。这些函数可作用于拥有同名成员函数的容器，接受容器变量作为参数，返回一个代理对象 list_inserter，它重载了 operator(), = 等操作符用来实现向容器填入数据的功能。

示范辅助函数 insert()、push_front()、push_back() 用法的代码如下：

```

#include <boost/assign.hpp>
int main()
{
    using namespace boost::assign;
    vector<int> v;
    push_back(v) (1) (2) (3) (4) (5);           //使用 push_back 辅助函数

    list<string> l;
    push_front(l) ("cpp") ("java") ("c#") ("python"); //使用 push_front 辅助函数

    set<double> s;
}

```

```
insert(s)(3.14)(0.618)(1.732);           //使用 insert 辅助函数

map<int, string> m;
insert(m)(1, "one")(2, "two");             //使用 insert 辅助函数
}
```

这段代码与使用 `operator+=` 没有太多的不同。对于拥有 `push_back()` 或 `push_front()` 成员函数的容器（`vector`、`list`），可以调用 `assign::push_back()` 或 `assign::push_front()`，而对于 `set` 和 `map`，则只能使用 `assign::insert()`。

`operator()` 的好处是可以在括号中使用多个参数，这对于 `map` 这样的元素是由多个值组成类型非常方便，避免了 `make_pair()` 函数的使用。而且，如果括号中没有参数，那么将调用容器元素的缺省构造函数填入一个缺省值，逗号操作符则不能这样做。

括号操作符也可以与逗号等操作符配合使用，写法更简单，有时甚至看起来不像是合法的 C++ 代码（但的确是完全正确的 C++ 代码），例如：

```
using namespace boost::assign;
vector<int> v;
push_back(v), 1, 2, 3, 4, 5;
push_back(v)(6), 7, 64 / 8, (9), 10;
// v = [ 1 2 3 4 5 6 7 8 9 10]
deque<string> d;
push_front(d)() = "cpp", "java", "c#", "python";
assert(d.size()==5);
// d = ['python', 'c#', 'java', 'cpp', '']
```

4.4.3 初始化容器元素

操作符 `+=` 和 `()` 解决了对容器的赋值问题，但有的时候我们需要在容器构造的时候就完成数据的填充，这种方式较赋值更为高效。C++ 内建的数组和标准字符串类 `string` 支持这样做，但其他 STL 容器则不行。

`assign` 库使用 `list_of()`、`map_list_of()` / `pair_list_of()` 和 `tuple_list_of()` 三个函数解决了这个问题。

`list_of`

`list_of()` 函数的用法与之前的 `insert()`、`push_back()` 等函数很相似，也重载了括号、逗号操作符。它很智能，返回一个匿名的列表，可以赋值给任意容器。

演示 `list_of` 用法的代码如下：

```
#include <boost/assign.hpp>
```

```

int main()
{
    using namespace boost::assign;

    vector<int> v = list_of(1) (2) (3) (4) (5);
    // v = [1, 2, 3, 4, 5]

    deque<string> d =
    (list_of("power") ("bomb"), "phazon", "suit");    //注意括号的使用
    // d = [power bomb phazon suit]

    set<int> s = (list_of(10), 20, 30, 40, 50);        //注意括号的使用
    // s = {10 20 30 40 50}

    map<int, string> m = list_of(make_pair(1, "one")) (make_pair(2, "two"));
    // m = [(1, "one") (2, "two")]
}

```

`list_of()` 函数可以全部使用括号操作符，也可以把括号与逗号结合起来，但使用后者时需要将整个 `list_of` 表达式用括号括起来，否则会使编译器无法推导出 `list_of` 的类型而无法赋值。

map_list_of/pair_list_of

使用 `list_of()` 处理 `map` 容器不是很方便，于是 `map_list_of()` / `pair_list_of()` 应运而生，`map_list_of()` 可以接受两个参数，然后自动构造 `std::pair` 对象插入 `map` 容器，`pair_list_of()` 则纯粹是 `map_list_of` 的同义词，两者的用法功能完全相同。

`map_list_of()` 和 `pair_list_of()` 的基本形式如下：

```

template< class Key, class T >
map_list_of( const Key& k, const T& t );           //key,value

template< class F, class S >
pair_list_of( const F& f, const S& s )           //first,second
{
    return map_list_of( f, s );
}

```

演示 `map_list_of` 用法的代码如下：

```

#include <boost/assign.hpp>
int main()
{
    using namespace boost::assign;

    map<int, int> m1 = map_list_of(1, 2) (3, 4) (5, 6);
    //m1 = [(1, 2) (3, 4) (5, 6)]
}

```

```
map<int, string> m2 = map_list_of(1, "one")(2, "two");
//m2 = [(1, "one")(2, "two")]
}
```

tuple_list_of

tuple_list_of 用于初始化元素类型为 tuple 的容器, tuple 是 Boost 引入的一种新的容器/数据结构, 关于 tuple 和的 tuple_list_of 的用法请参见 7.6.8 小节 (282 页)。

4.4.4 减少重复输入

在填充数据时有的时候要输入重复数据, 如果用之前的方法要写大量的重复代码, 很麻烦也容易造成多写或少写的错误。assign 库提供 repeat()、repeat_fun() 和 range() 三个函数来减轻工作量。

这三个函数的简要声明如下:

```
template< class U >
generic_list& repeat( std::size_t sz, U u );

template< class Nullary_function >
generic_list& repeat_fun( std::size_t sz, Nullary_function fun );

template< class SinglePassIterator >
generic_list& range( SinglePassIterator first,
                    SinglePassIterator last );

template< class SinglePassRange >
generic_list& range( const SinglePassRange& r )
```

repeat() 函数把第二个参数作为要填入的值, 重复第一个参数指定的次数, 与 vector、deque 等容器的构造函数很相似; repeat_fun() 函数同样重复第一个参数的次数, 但第二个参数是个无参的函数或函数对象, 它返回填入的数值; range() 函数则可以把一个序列全部或者部分元素插入到另一个序列里。

示范它们用法的代码如下:

```
#include <boost/assign.hpp>
#include <cstdlib> //for rand()
int main()
{
    using namespace boost::assign;

    vector<int> v = list_of(1).repeat(3, 2)(3)(4)(5);
    //v = 1,2,2,2,3,4,5

    multiset<int> ms ;
```

```

insert(ms).repeat_fun(5, &rand).repeat(2, 1), 10;
//ms = x,x,x,x,x,1,1,10

deque<int> d;
push_front(d).range(v.begin(), v.begin() + 5);
//d = 3,2,2,2,1
}

```

4.4.5 与非标准容器工作

assign 库不仅支持全部八个 STL 标准容器 (vector、string、deque、list、set、multiset、map、multimap)，也对 STL 中的容器适配器提供了适当的支持，包括 stack、queue 和 priority_queue。

因为 stack 等容器适配器不符合容器的定义，没有 insert、push_back 等成员函数，所以不能使用赋值的方式填入元素，只能使用初始化的方式，并在 list_of 表达式最后使用 to_adapter() 成员函数来适配到非标准容器。如果使用逗号操作符还需要把整个表达式用括号括起来，才能使用点号调用 to_adapter()。

示范 assign 库应用于容器适配器的代码如下：

```

#include <boost/assign.hpp>
#include <stack>
#include <queue>

int main()
{
    using namespace boost::assign;

    stack<int> stk = (list_of(1), 2, 3).to_adapter();
    while(!stk.empty())                //输出 stack 的内容
    {
        cout << stk.top() << " ";
        stk.pop();
    }
    cout << endl;

    queue<string> q = (list_of("china") ("us") ("uk")).repeat(2, "russia").to_adapter();
    while(!q.empty())                //输出 queue 的内容
    {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl;

    priority_queue<double> pq = (list_of(1.414), 1.732, 2.236).to_adapter();
}

```

```

while(!pq.empty())                //输出优先队列的内容
{
    cout << pq.top() << " ";
    pq.pop();
}

```

程序运行结果如下:

```

3 2 1
china us uk russia russia
2.236 1.732 1.414

```

assign 库也支持部分不在 STL 中定义的非标准容器, 如 STLport 中的 slist 和 hash_set/hash_map, 因为它们符合容器的定义, 故用法与标准容器没有什么区别。

示范 assign 库应用于非标准容器的代码如下:

```

#include <boost/assign.hpp>
#include <hash_map>                //非标准的散列映射容器
#include <slist>                    //非标准的单向链表
int main()
{
    using namespace boost::assign;

    slist<int> sl;
    push_front(sl, 1,2,3,4,5;      //不能用 push_back 和 insert

    hash_map<string, int> hm = map_list_of("one", 1)("two", 2);
}

```

此外, assign 库还支持大部分 Boost 库容器, 如 array、circular_buffer、unordered 等, 用法同样与标准容器基本类似, 在第 7 章可以找到更多的示例代码。

4.4.6 高级用法

本小节讨论关于 assign 库的一些高级用法。

list_of()的嵌套使用

list_of() 可以就地创建匿名列表, 这一点很有用, 它可以嵌套在 assign 库用法中, 创建复杂的数据结构。

下面的代码使用 vector 构造了一个二维数组, 使用 list_of(list_of()) 的嵌套形式来初始化:


```
using namespace boost::assign;
vector< vector<int> > v = list_of( list_of(1)(2))(list_of(3)(4));

v += list_of(5)(6),list_of(7)(8);
```

引用初始化列表

在 `list_of` 之外 `assign` 库还有两个类似功能的 `ref_list_of()` 和 `cref_list_of()`，这两个函数接受变量的引用作为参数来创建初始化匿名列表，较 `list_of()` 的效率更高，例如：

```
using namespace boost::assign;
int a = 1, b = 2, c = 3;
vector<int> v = ref_list_of<3>(a)(b)(c);
assert(v.size() == 3);
```

`assign` 库还特别支持 Boost 中的指针容器，提供 `ptr_push_back()`、`ptr_list_of()` 等函数。由于本书未涉及指针容器，故在这里不做介绍，感兴趣的读者请阅读 Boost 说明文档中 `assign` 库和 `pointer container` 相关部分。

4.5 swap

`boost::swap` 是对标准库提供的 `std::swap` 的增强和泛化，为交换两个变量（可以是 `int` 等内置数据类型，或者是类实例、容器）的值提供了便捷的方法。

使用 `boost::swap`，需要包含头文件 `<boost/swap.hpp>`（真正实现在 `<boost/utility/swap.hpp>`），即：

```
#include <boost/swap.hpp>
```

4.5.1 原理

先让我们来复习一下 C++ 标准中的 `std::swap()`。

```
///std::swap 的典型实现，具体代码依 STL 版本而不同
template<typename T>
void swap(T& a, T& b)
{
    T tmp(a);
    a = b;
    b = tmp;
}
```

从代码中可以看出，`std::swap()` 要求交换的对象必须是可拷贝构造和可拷贝赋值的（即重载 `operator=`），它提供的是最通用同时也是效率最低的方法，需要进行一次拷贝构造和两次赋

值操作，如果交换的对象很大，那么运行代价会相当昂贵。所以，对于我们自己写的类，最好能够实现优化的 `swap()`。

解决方案有两种。第一种方案是直接利用函数重载，编写一个同名的 `swap` 函数，这个 `swap()` 再调用类内部的高效成员交换函数，这样编译器在编译时就不会使用 `std::swap()`。第二种方案是使用 ADL (argument dependent lookup, 参数依赖查找，又称 Koenig Lookup) 查找模板特化的 `swap`。这两种方案就是 `boost::swap` 的工作原理：

它查找有无针对类型 `T` 的 `std::swap()` 的特化 `swap()` 或者通过 ADL 查找模板特化的 `swap()`，如果有则调用，如果两种查找都失败时则退化为 `std::swap()`。此外，`boost::swap` 还增加了一个很实用的功能——对 C++ 内建数组交换的支持（已经被接纳入 C++ 新标准），

`boost::swap()` 函数的声明如下：

```
template<class T1, class T2>
void swap(T1& left, T2& right);
```

4.5.2 交换数组

我们先来看一下 `boost::swap` 最简单的用法——交换两个数组的内容，它要求参与交换的两个数组必须具有相同的长度。下面的代码使用标准库算法 `fill_n` 将两个数组分别赋值为 5 和 20，然后调用 `boost::swap()` 交换：

```
int a1[10]; //两个数组
int a2[10];

std::fill_n(a1, 10, 5); //fill_n 赋初始值
std::fill_n(a2, 10, 20);

boost::swap(a1, a2); //交换两个数组的内容
```

`boost::swap` 交换数组内容的实现很简单，它使用了一个 `for` 循环，对数组中的每个元素调用单个元素版的 `boost::swap` 完成整个数组内容的交换。在上面的代码执行后 `a1` 中元素的值将为 20，而 `a2` 中元素的值将为 5。

如果企图用 `boost::swap` 交换两个长度不相同的数组，那么将无法通过编译：

```
int a1[10], a2[12]; //两个长度不相同的数组
boost::swap(a1, a2); //发生编译错误
```

4.5.3 特化 `std::swap`

接下来我们示范利用模板特化的方法使用 `boost::swap`。用一个简单的三维空间的点

point 作为例子，它实现了内部高效的交换函数：

```
class point
{
    int x, y, z;
public:
    explicit point(int a=0, int b=0, int c=0):x(a),y(b),z(c){}
    void print()const
    {
        cout << x << ", "<< y << ", "<< z << endl; }
    void swap(point &p)                                //内置高效交换函数
    {
        std::swap(x, p.x);
        std::swap(y, p.y);
        std::swap(z, p.z);
        cout << "inner swap" << endl;
    }
};
```

特化 `std::swap()` 的方法需要向 `std` 名字空间添加自定义函数（虽然原则上是不允许变动 `std` 名字空间的），代码如下：

```
namespace std
{
    template<>
    void swap(point &x, point &y)                    //模板特化 swap 函数
    {x.swap(y);}
}
int main()
{
    point a(1,2,3), b(4,5,6);
    cout << "std::swap" << endl;
    std::swap(a,b);                                    //调用 std::swap
    cout << "boost::swap" << endl;
    boost::swap(a, b);                                //调用 boost::swap
}
```

程序运行结果如下：

```
std::swap
inner swap
boost::swap
inner swap
```

由于我们在名字空间特化了 `std::swap`，因此，`boost::swap` 与 `std::swap` 的效果相同，都使用了特化后的 `swap` 函数。

4.5.4 特化 ADL 可找到的 swap

依然使用刚才的 `point` 类，但这次我们不变动 `std` 名字空间，而是在全局名字空间内实现



swap 函数:

```
//global namespace
void swap(point &x, point &y)
{ x.swap(y); }
int main()
{
    point a(1,2,3), b(4,5,6);
    cout << "std::swap" << endl;
    std::swap(a,b);
    cout << "boost::swap" << endl;
    boost::swap(a, b);
}
```

程序运行结果如下:

```
std::swap
boost::swap
inner swap
```

这段代码的运行结果与之前的特化 `std::swap` 有明显不同, `std::swap` 使用了标准的交换操作, 而 `boost::swap` 通过 ADL 规则找到了全局名字空间的特化交换函数, 实现了高效的交换。

如果读者担心在全局名字空间编写自由函数 `swap` 会造成名字“污染”, 也可以把特化的 `swap` 加入到 `boost` 名字空间, 或者其他 ADL 可以找到的名字空间。

4.5.5 使用建议

我们应该尽量使用 `boost::swap`, 它提供了比 `std::swap` 更好的优化策略。如果你自己写的类实现了高效的交换(应该总这样), 或者想交换两个数组的内容, 那么就使用 `boost::swap`, 它可以保证总能够找到最恰当的交换方法。

变量值交换是一个基础但很重要的操作, 几乎所有 Boost 库组件都实现了自己的 `swap` 成员函数, 并且用 `boost::swap` 来提高交换的效率, 可以在很多代码中找到 `swap` 的实现范例。

在推荐书目 [5] 的条款 25 有关于 `swap` 的详细论述, 读者可以参考。

4.6 singleton

singleton 即单件模式, 实现这种模式的类在程序生命周期里只能有一个且仅有一个实例。单件模式是一个很有用的创建型模式, 有许多实际的应用, 并被广泛且深入地研究。如果读者还不熟悉这个模式, 最好先阅读推荐书目 [1], 它提供了对 singleton 模式完整深刻的表述。

虽然单件模式非常重要，但很遗憾目前 Boost 中并没有专门的单件库，而仅是在其他库中有并不十分完善的实现。虽然不够完善（但比普通实现还是要好很多），但总比没有强。

4.6.1 boost.pool 的单件实现

在之前介绍 pool 库的时候，里面有一个组件名字是 singleton_pool，它包含了一个泛型的单件类 singleton_default。

singleton_default 位于名字空间 boost::details::pool，为了使用 singleton_default，需要包含头文件 <boost/pool/detail/singleton.hpp>，即：

```
#include <boost/pool/detail/singleton.hpp>
using boost::details::pool::singleton_default;
```

类摘要

singleton_default 的简要声明如下：

```
template <typename T>
struct singleton_default
{
public:
    typedef T object_type;
    static object_type & instance();
};
```

singleton_default 把模板参数 T 实现为一个单件类，唯一实例只能通过静态成员函数 instance() 访问。它运用了很巧妙的技术，可以在 main() 运行之前就创建单件。对类型 T 的要求是有缺省（无参）构造函数，而且在构造和析构时不能抛出异常，因为单件在 main() 前后构造和析构，如果发生异常会无法捕获。

用法

在 4.5 小节（113 页）我们定义了一个三维点 point，它具有缺省构造函数，因此可以应用于 singleton_default，实现一个唯一的原点实例。为了演示 singleton_default<> 的创建与销毁，下面的代码增加了构造函数与析构函数：

```
#include <boost/pool/detail/singleton.hpp>
using boost::details::pool::singleton_default;           // 单件名字空间
class point
{public:
    point(int a=0, int b=0, int c=0):x(a),y(b),z(c)      // 构造函数
    {cout << "point ctor" << endl;}
    ~point()                                              // 析构函数
    {cout << "point dtor" << endl;}
```

```

...
};

int main()
{
    cout << "main() start" << endl;
    typedef singleton_default<point> origin;           //定义单件类

    origin::instance().print();                        //使用 instance() 获得单件对象
    cout << "main() finish" << endl;
}

```

程序运行结果如下:

```

point ctor
main() start
0,0,0
main() finish
point dtor

```

这段代码示范了如何使用 `singleton_default`, 用法非常简单, 只需要把想成为单件的类作为它的模板参数就可以了, 接下来的工作由 `singleton_default` 自动完成。可能唯一的一点不方便之处是过长的类型名, 但可以用 `typedef` 来简化。

`point` 的例子可能过于简单, 下面再用一个比较接近实际的例子。假设我们要实现一个数据库访问类, 它负责程序中所有涉及数据库的操作。很自然, 这样的类在程序中只能有一个实例存在, 它可以实现为单件。

```

class SqlDB_t
{
public:
    void connect()
    {cout << "connect db" << endl; }
    void exec(const char *sqlstr)
    {cout << "exec insert/update/delete : " << sqlstr << endl;}
    void query(const char *sqlstr)
    {cout << "exec select " << sqlstr << endl;}
};
typedef singleton_default<SqlDB_t> SqlDB;

int main()
{
    cout << "main() start" << endl;

    SqlDB::instance().connect();
    SqlDB::instance().exec("create table goods(int id, varchar name(20));");
    SqlDB::instance().exec("insert into goods values(101, 'wii')");
    SqlDB::instance().query("select * from goods");
}

```

```
    cout << "main() finish" << endl;
}
```

程序运行结果如下:

```
main() start
connect db
exec insert/update/delete :create table goods(int id, varchar name(20)
exec insert/update/delete :insert into goods values(101, 'wii')
exec select select * from goods
main() finish
```

4.6.2 boost.serialization 的单件实现

在序列化库 `serialization` 中有另一个单件实现类: `singleton`, 它位于名字空间 `boost::serialization`, 为了使用 `singleton`, 需要包含头文件 `<boost/serialization/singleton.hpp>`, 即:

```
#include <boost/serialization/singleton.hpp>
using boost::serialization::singleton;
```

类摘要

`singleton` 的类简要声明如下:

```
template <typename T>
class singleton : public boost::noncopyable
{
public:
    static const T & get_const_instance();
    static T & get_mutable_instance();
};
```

`singleton` 与 `pool.singleton_default` 基本相同, 对模板参数 `T` 也有同样的要求 (具有缺省构造函数, 构造析构不抛异常), 但访问单件实例的成员函数却有区分, 分为了常对象和可变对象两个函数。这种区分是出于线程安全的考虑, 常对象单件总是线程安全的, 因为它不会改变内部状态, 而可变对象单件则不是线程安全的, 可能会发生线程竞争问题。

用法

`singleton` 可以如 `pool.singleton_default` 一样使用模板参数的方式来使用:

```
#include <boost/serialization/singleton.hpp>
using boost::serialization::singleton;

class point{...}
```

```
int main()
{
    cout << "main() start" << endl;
    typedef singleton<point> origin;                //单件类定义

    origin::get_const_instance().print();            //常对象
    origin::get_mutable_instance().print();          //可变对象

    cout << "main() finish" << endl;
}
```

代码中需要注意的是类名 `singleton`，还有就是访问实例的函数不是 `instance()`，而是 `get_const_instance()` 和 `get_mutable_instance()`，程序运行结果与 `pool.singleton_default` 完全相同。

除了这种模板方式，`serialization.singleton` 还可以通过继承的方式来使用，把 `singleton` 作为单件类的基类，并把单件类作为 `singleton` 的模板参数：

```
#include <boost/serialization/singleton.hpp>
using boost::serialization::singleton;

class point:public singleton<point>                //注意这里
{...}

int main()
{
    cout << "main() start" << endl;

    point::get_const_instance().print();
    point::get_mutable_instance().print();

    cout << "main() finish" << endl;
}
```

这段代码与模板参数的用法仅有很小的不同，最重要的变化在于 `point` 类的基类声明处 `public 继承 singleton<point>`（如果读者不熟悉泛型编程，可能会对这种父类中使用子类的形式有些困惑），因此 `point` 继承了 `singleton` 的所有能力，包括不可拷贝和单件。

与模板参数方式相比，继承方式实现单件模式更为彻底一些（很像 `noncopyable` 的用法），使被单件化的类成为了一个真正的单件类，而模板参数方式则更“温和”一些，它包装（wrap）了被单件化的类，对原始类没有任何影响。

4.7 tribool

`boost.tribool` 类似 C++ 内置的 `bool` 类型，但基于三态的布尔逻辑：在 `true`（真）和

false (假) 之外还有一个 indeterminate 状态 (未知、不确定)。

tribool 位于名字空间 boost::logic, 但为了方便使用被 using 语句引入了 boost 名字空间, 为使用 tribool 组件, 需要包含头文件 <boost/logic/tribool.hpp>, 即:

```
#include <boost/logic/tribool.hpp>
using namespace boost;
```

4.7.1 类摘要

tribool 的类摘要如下:

```
class tribool
{
public:
    tribool(bool value);
    ...                               //其他构造函数
    enum value_t { false_value, true_value, indeterminate_value } value;
};
bool indeterminate(tribool x);
tribool operator!(tribool x);
...                               //其他逻辑运算符和比较运算符重载
```

tribool 类很简单, 它内部实现了三态 bool 值的表示, 除了构造函数没有什么其他成员函数。可以在创建 tribool 对象的同时传入三态 bool 值对它进行初始化, 如果使用无参的缺省构造函数, 那么 tribool 默认值是 false。

对 tribool 的操作都是通过逻辑运算符和比较运算符的重载来完成的, 支持的逻辑运算包括 ||、&& 和 !, 比较运算支持 == 和 !=, 这些操作都可以任意混合 bool 和 tribool 一起运算。

不确定状态 indeterminate 是一个特殊的 tribool 值, 它与 bool 值 true、false 的运算遵循三态布尔逻辑:

- 任何与 indeterminate 的比较操作结果都是 indeterminate;
- 与 indeterminate 的逻辑或运算 (||) 只有与 true 运算结果为 true, 其他均为 indeterminate;
- 与 indeterminate 的逻辑与运算 (&&) 只有与 false 运算结果为 false, 其他均为 indeterminate;
- indeterminate 的逻辑非操作 (!) 结果仍为 indeterminate。

自由函数 indeterminate() 可以判断一个 tribool 是否处于不确定状态。

4.7.2 用法

tribool 可以像普通的 bool 类型一样使用，只是多出了一个 indeterminate（未知、不确定）的取值。如果仅使用 true 和 false 两个值，那么 tribool 与 bool 的用法是完全相同的，但如果需要使用 indeterminate 值，就必须遵循三态布尔的逻辑运算规则。

示范 tribool 基本用法的代码如下：

```
#include <boost/logic/tribool.hpp>
using namespace boost;
int main()
{
    tribool tb(true);                //值为 true 的 tribool
    tribool tb2(!tb);                //值为 false

    if (tb)                          //tb==true
    {   cout << "true" << endl; }

    tb2 = indeterminate;              //tb2 是不确定状态
    assert(indeterminate(tb2));        //用 indeterminate 函数检测状态
    cout << tb2 << endl;
    if (tb2 == indeterminate)          //可以直接与 indeterminate 值比较
    {   cout << "indeterminate" << endl;   }

    cout << (tb2 || true) << endl;      //逻辑或运算，输出 true
    cout << (tb2 && false) << endl;     //逻辑与运算，输出 false
}
```

在处理 tribool 的不确定状态时必须要小心，因为它既不是 true 也不是 false，使用它进行条件判断永远都不会成立，判断不确定状态必须要与 indeterminate 值比较或者使用 indeterminate() 函数：

```
tribool tb(indeterminate);
if (tb)                          //永远不成立
    cout << "never reach here" << endl;
if (!tb)                          //永远不成立
    cout << "never reach here" << endl;
if (indeterminate(tb))            //必须使用 indeterminate
    cout << "indeterminate" << endl;
```

4.7.3 为第三态更名

tribool 库默认采用 indeterminate 作为第三态的名字，很清晰明确但可能有些长。因此库允许把 indeterminate 改变成任意用户喜欢的名字，常用的名字可以是 unknown、maybe、true_or_false 等等。

只需要在全局域内使用宏 `BOOST_TRIBOOL_THIRD_STATE` 就可以为第三态更名，像这样：

```
BOOST_TRIBOOL_THIRD_STATE(unknown)
```

然后我们就可以随意使用这个新名字代替原来的 `indeterminate`：

```
tribool tb(unknown);           //可以作为不确定值
assert(unknown(tb));           //可以作为检测函数
assert(unknown(tb || false));
```

改名的原理很简单：

`tribool` 默认的第三态 `indeterminate` 不是一个真正的类型值（`typedef` 或者 `enum`），而是一个函数，它被用来判断 `tribool` 对象内部值是否是第三态，宏 `BOOST_TRIBOOL_THIRD_STATE` 只是定义了一个新的等价函数而已，类似这样：

```
inline bool some_name(tribool x)
{ return x.value == tribool::indeterminate_value; }
```

因为宏 `BOOST_TRIBOOL_THIRD_STATE` 实质上定义了一个函数，而 C++ 不允许函数嵌套，所以这个宏最好在全局域使用，它将在定义后的整个源代码中都生效。

如果把 `BOOST_TRIBOOL_THIRD_STATE` 用在一个名字空间里，那么新的第三态名字将成为名字空间的一个成员，使用时需加上名字空间限定，例如：

```
namespace tmp_ns                //一个临时名字空间
{
    BOOST_TRIBOOL_THIRD_STATE(unknown)
};
tribool tb(tmp_ns::unknown);    //使用名字空间限定
```

4.7.4 输入输出

`tribool` 可以如 `bool` 类型一样进行流操作，但需要包含另外一个头文件 `<boost/logic/tribool_io.hpp>`。

只要包含了这个头文件，就可以使用 `>>`、`<<` 操作符向 `cin`、`cout` 等流对象输入输出，`false`、`true` 和 `indeterminate` 分别对应整数 0、1 和 2。如果设置了流的 `boolalpha` 标志，则对应字符串 `"false"`、`"true"` 和 `"indeterminate"`。

例如：

```
#include <boost/logic/tribool_io.hpp>
using namespace boost;
int main()
```

```

{
    tribool tb1(true), tb2(false), tb3(indeterminate);
    cout << tb1 << ", "           //输出 1
         << tb2 << ", "           //输出 0
         << tb3 << endl;          //输出 2
}

```

4.7.5 与 optional<bool>的区别

optional<bool>在功能上有些类似 tribool, 一个未初始化的 optional<bool>同样可以表示不确定的 bool 值, 例如:

```

optional<bool> b;
if (!b)                               //b 未初始化, 既不是 true 也不是 false
{ cout << "indeterminate" << endl; }
b = false;
if (b)                                 //b 有值 false
{ cout << "b=" << *b << endl; }

```

但 optional<bool>的语义是未初始化的 bool, 是无意义的值, 而 tribool 的 indeterminate 是已经初始化的有意义值, 它表示 bool 值不确定。这两者存在着细微但十分重要的差别。

此外, 两者的使用方法也存在差别。optional<bool>需要使用类似指针的方式访问容器内的 bool 值, 而 tribool 可以如普通的 bool 类型一样直接使用, 并且支持各种逻辑运算。

由于 optional 存在一个隐式 bool 转换, 用于检测 optional 是否已经初始化, 因此在 bool 语境下如果不注意 optional 的这个特性很容易导致意外的错误。例如, 下面的代码本意是想使用 optional 内的 bool 值作为 if 语句的判断条件, 但实际上条件判断的是 optional 未初始化:

```

optional<bool> b(false);
if (!b)                               //optional 的隐式 bool 转换
{ cout << "false" << endl; }

```

正确的写法应该是:

```

if (b && !*b)                         //b 已经初始化且值为 false
{ cout << "false" << endl; }

```

选择 optional<bool>还是 tribool 需要由代码的具体需求来决定。如果返回值可能是无效的 (不存在有效的返回值), 那么就是 optional<bool>; 如果返回值总是确定的, 但可能无法确定其意义, 那么就用 tribool。

4.8 operators

C++提供了操作符重载的能力，可以把大多数操作符重新定义为函数，使对对象的操作更加简单直观。这方面很好的例子就是标准库中的 `string` 和 `complex`，可以像操作内置类型 `int`、`double` 那样对它们进行算术运算和比较运算，非常方便。

但实现重载操作符却比使用它要麻烦许多，因为很多运算具有对称性，如果定义了 `operator+`，那么很自然需要 `operator-`，如果有小于比较，那么也应该有小于等于、大于、大于等于比较。完全实现这些操作符的重载工作是单调乏味的，而且增加的代码量也增加了出错的可能性，还必须保证这些操作符都实现了正确的语义。

实际上，很多操作符可以从其他的操作符自动推导出来，例如 `a!=b` 可以是 `!(a==b)`，`a>=b` 可以是 `!(a<b)`。因此原则上只需要定义少量的基本操作符，其他的操作符就可以用逻辑组合实现。

在 C++98 标准的 `std::rel_ops` 名字空间里提供了四个模板比较操作符 `!=`、`>`、`<=`、`>=`，只需要为类定义了 `==` 和 `<` 操作符，那么这四个操作符就可以自动实现。

例如：

```
#include <utility>
class demo_class                                //一个定义 operator< 的类
{
public:
    demo_class(int n):x(n){}
    int x;
    friend bool operator<(const demo_class& l, const demo_class&r)
    {    return l.x < r.x;    }
};
int main()
{
    demo_class a(10), b(20);
    using namespace std::rel_ops;                //打开 std::rel_ops 名字空间
    cout << (a<b) << endl;                        //自定义的<操作符
    cout << (b>=a) << endl;                        //>=等操作符被自动实现
}
```

但 `std::rel_ops` 的解决方案过于简单，还很不够。除了比较操作符，还有很多其他的操作符重载标准库没有给出解决方案，而且使用这些操作符需要用 `using` 语句导入 `std::rel_ops` 名字空间，不方便，也会带来潜在的冲突风险。

`boost.operators` 库因此应运而生。它采用类似 `std::rel_ops` 的实现手法，允许用户在自己的类里仅定义少量的操作符（如 `<`），就可方便地自动生成其他操作符重载，而且保证正确

的语义实现。

operators 位于名字空间 boost，为了使用 operators 组件，需要包含头文件<boost/operators.hpp>，即：

```
#include <boost/operators.hpp>
using namespace boost;
```

4.8.1 基本运算概念

由于 C++ 可重载的操作符很多，因此 operators 库是由多个类组成的，分别用来实现不同的运算概念，比如 less_than_comparable 定义了<系列操作符，left_shiftable 定义了<<系列操作符。operators 中的概念非常多，囊括了 C++ 中的大部分操作符重载，在这里我们先介绍一些最常用的算术操作符：

- equality_comparable : 要求提供==，可自动实现!=，相等语义；
- less_than_comparable : 要求提供<，可自动实现>、<=、>=；
- addable : 要求提供+=，可自动实现+；
- subtractable : 要求提供-=，可自动实现-；
- incrementable : 要求提供前置++，可自动实现后置++；
- decrementable : 要求提供前置--，可自动实现后置--。
- equivalent : 要求提供<，可自动实现==，等价语义，它与 equality_comparable 的区别请参见 4.8.5 小节。

这些概念在库中以同名类的形式提供，用户需要以继承的方式来使用它们。继承的修饰符并不重要（private、public 都可以），因为 operators 库里的类都是空类，没有成员变量和成员函数，仅定义了数个友元操作符函数。

例如，less_than_comparable 的形式是^①：

```
template <class T>
struct less_than_comparable {
    friend bool operator>(const T& x, const T& y);
    friend bool operator<=(const T& x, const T& y);
```

① 这段代码不是 Boost 程序的真实代码，实际上 less_than_comparable 是 less_than_comparable1 的子类。

```
friend bool operator>=(const T& x, const T& y);
};
```

如果要同时实现多个运算概念则可以使用多重继承技术，把自定义类作为多个概念的子类，但多重继承在使用时存在很多问题，稍后将看到 operators 库使用了特别的技巧来解决这个问题。

4.8.2 算术操作符的用法

我们使用之前 4.5.3 小节定义的三维空间的点 point 作为 operators 库的示范类，在此把它重新定义如下（去掉了 swap 函数）：

```
class point
{
    int x, y, z;
public:
    explicit point(int a=0, int b=0, int c=0):x(a),y(b),z(c){}
    void print()const
    { cout << x << ", "<< y << ", "<< z << endl; }
};
```

我们先来实现 less_than_comparable，它要求 point 类提供<操作符，并由它继承。假定 point 的小于关系是三个坐标值的平方和决定的，下面的代码示范了 less_than_comparable 的用法，只需要为 point 增加父类，并定义 less_than_comparable 概念所要求的 operator<：

```
#include <boost/operators.hpp>
class point:
    boost::less_than_comparable<point>           //小于关系，私有继承
{
public:
    friend bool operator<(const point& l, const point& r)
    {
        return (l.x*l.x + l.y*l.y + l.z*l.z <
                r.x*r.x + r.y*r.y + r.z*r.z);
    }
    ...
};                                           //其他成员函数
```

less_than_comparable 作为基类的用法可能稍微有点奇怪，它把子类 point 作为了父类的模板参数：less_than_comparable<point>，看起来好像是个“循环继承”。实际上，point 类作为 less_than_comparable 的模板类型参数，只是用来实现内部的比较操作符，用作操作符函数的类型，没有任何继承关系。less_than_comparable 生成的代码可以理解成这样^①：

① 如果把这种用法按照模板的方式读成“less_than_comparable of point class”就比较好理解了。

```
//template<T = point>
struct less_than_comparable
{
    friend bool operator>=(const point& x, const point& y)
    { return !(x < y); }
}
```

明白了 `less_than_comparable` 的继承用法，剩下的就很简单了：`point` 类定义了一个友元 `operator<` 操作符，然后其余的 `>`、`<=`、`>=` 就由 `less_than_comparable` 自动生成。几乎不费什么力气，在没有污染名字空间的情况下我们就获得了四个操作符的能力：

```
int main()
{
    point p0, p1(1,2,3), p2(3,0,5), p3(3,2,1);

    assert(p0 < p1 && p1 < p2);
    assert(p2 > p0);
    assert(p1 <= p3);
    assert(!(p1<p3)&&!(p1>p3) );
}
```

同样我们可以定义相等关系，使用 `equality_comparable`，规则是 `point` 的三个坐标值完全相等，需要自行实现 `operator==`：

```
class point:boost::less_than_comparable<point>    //使用多重继承
,boost::equality_comparable<point>              //新增相等关系
{
public:
    friend bool operator<(const point& l, const point& r)
    { /*同前*/ }
    friend bool operator==(const point& l, const point& r)
    { return r.x == l.x && r.y == l.y && r.z == l.z; }
};
```

然后我们就自动获得了 `operator!=` 的定义：

```
point p0, p1(1,2,3), p2(p1), p3(3,2,1);
assert(p1 == p2);
assert(p1 != p3);
```

在使用 `operators` 库时要注意一点，模板类型参数必须是子类自身，特别是当子类本身也是个模板类的时候，不要错写成子类的模板参数或者子类不带模板参数的名称，否则会造成编译错误。假如我们改写 `point` 类为一个模板类：

```
template<typename T> class point {...}
```

那么如下的形式都是错误的：


```
template<typename T> class point: boost::less_than_comparable<T>
template<typename T> class point: boost::less_than_comparable<point>
```

正确的写法应该是：

```
template<typename T> class point: boost::less_than_comparable< point<T> >
```

因为只有 `point<T>` 才是模板类 `point` 的全名。

4.8.3 基类链

多重继承一直是 C++ 中引发争论的话题，喜欢它的人和讨厌它的人几乎同样多。总的来说，多重继承是一种强大的面向对象技术，但使用不当也很容易引发诸多问题，比如难以优化和经典的“钻石型”继承。

`operators` 库使用泛型编程的“基类链”技术解决了多重继承的问题，这种技术通过模板把多继承转换为链式的单继承。

前面当讨论到 `less_than_comparable<point>` 这种用法时，我们说它不是继承，然而，现在，我们将看到它居然真的可以实现继承的功能，这从一个方面展示了泛型编程的强大威力。

`operators` 库的操作符模板类除了接受子类作为比较类型外，还可以接受另外一个类，作为它的父类，由此可以无限串联链接在一起（但要受编译器的模板编译能力限制），像这样：

```
demo: x< demo, y<demo, z<demo,...> > >
```

使用基类链技术，`point` 类的基类部分可以是这样：

```
boost::less_than_comparable<point,           //注意这里
boost::equality_comparable<point> >         //是一个有很大模板参数列表的类
```

对比一下多重继承的写法：

```
boost::less_than_comparable<point>,          //注意这里
boost::equality_comparable<point>            //有两个类
```

代码非常相似，区别仅仅在于模板参数列表结束符号（>）的位置，如果不仔细看可能根本察觉不出差距。但正是这个小小的差距，使基类链通过模板组成了一连串的单继承链表，而不是多个父类的多重继承。

例如，如果为 `point` 类再增加加法和减法定义，继承列表就是：

```
class point:
    less_than_comparable<point,           //小于操作
```

```

equality_comparable<point,           //相等操作
addable<point,                       //加法操作
subtractable<point                   //减法操作
> > > >
{...}

```

基类链技术会导致代码出现一个有趣的形式：在派生类的基类声明末尾处出现一长串的>（模板声明的结束符），在编写代码时需要小心谨慎以保证尖括号的匹配，使用良好的代码缩进和换行可以减少错误的发生。

4.8.4 复合运算概念

基类链技术解决了多重继承的效率问题，但它也带来了新的问题，为了使用操作符概念需要写出很长的基类链代码。因此 operators 库使用基类链把一些简单的运算概念组合成了复杂的概念，即复合运算。这是个很自然的要求，如果有<，当然会需要==，如果有了+，可能还需要-、*和/。复合运算不仅进一步简化了代码的编写，给出了更明确的语义，它也可以避免用户代码中基类链过长的问题。

operators 库提供的常用复合运算概念如下：

- totally_ordered: 全序概念，组合了 equality_comparable 和 less_than_comparable;
- additive : 可加减概念，组合了 addable 和 subtractable;
- multiplicative : 可乘除概念，组合了 multipliable 和 dividable;
- arithmetic : 算术运算概念，组合了 additive 和 multiplicative
- unit_stoppable : 可步进概念，组合了 incrementable 和 decrementable;

使用复合运算概念，point 类只需要很少的代码就可以很容易地获得完全的算术运算能力：

```

class point:
    totally_ordered<point,           //全序比较运算
    additive<point> >               //可加减运算
{
public:
    friend bool operator<(const point& l, const point& r)
    {...}
    friend bool operator==(const point& l, const point& r)
    {...}
    point& operator+=(const point& r)    //支持 addable 概念
    {

```

```

        x += r.x;
        y += r.y;
        z += r.z;
        return *this;
    }
    point& operator+=(const point& r)        //支持 subtractable 概念
    {
        x += r.x;
        y += r.y;
        z += r.z;
        return *this;
    }
};

```

point 的操作符重载验证代码如下:

```

point p0, p1(1,2,3), p2(5,6,7), p3(3,2,1);

using namespace boost::assign;
vector<point> v = (list_of(p0), p1, p2, p3);

BOOST_AUTO(pos, std::find(v.begin(), v.end(), point(1,2,3)));
//find 算法使用 operator==
pos->print();

(p1 + p2).print();           //operator+
(p3 - p1).print();           //operator-

assert((p2 - p2) == p0);     //operator-和 operator==

```

point 类很好地示范了复合运算操作符的用法: 一般情况下, 类型 T 继承 boost::totally_ordered<T>, 再定义<、==操作符即可获得完全的比较运算功能, 能够用于标准容器和算法。

operators 库另一个例子是将在 9.2 小节 (344 页) 介绍的 rational 类, 它实现了有理数, 支持全序和算术运算, 不过很可惜的是它虽然使用了 operators 库, 但没有用到复合运算概念, 从而导致基类声明有 16 个 “>”!

4.8.5 相等与等价

相等 (equality) 与等价 (equivalent) 是两个极易被混淆的概念。一个简单快速的解释是: 相等基于操作符==, 即 x==y; 而等价基于<, 即!(x<y)&&!(x>y), 两者在语义上有很大的差别。

对于简单类型 (如 int), 相等和等价两者是一致的, 例如 5==10/2 和!(5<10/2)&&!

($5 > 10/2$)。但对于大多数复杂类型和自定义类型，由于`==`和`<`操作符是两个不同的运算，比较原则可能不同，从而两者具有不同的意义。

之前的 `point` 类是一个很好的例子。`p1(1,2,3)` 和 `p3(3,2,1)` 两者完全不相等，但等价，因为等价运算使用的是 `operator<`，它比较依据的是成员变量的平方和：`1+2*2+3*3 == 3*3+2*2+1`。

`operators` 库使用 `equality_comparable` 和 `equivalent` 明确地区分了相等与等价这两个概念。`equality_comparable` 基于`==`，`equivalent` 则基于`<`。但让人困扰的是它们最终都提供了操作符`==`，表现相同但涵义非常不同。

了解相等与等价的区分非常重要，特别是当自定义类被用作容器的元素的时候。标准库中的关联容器（`set`、`map`）和排序算法使用的是等价关系的`<`操作符，而各种查找算法 `find` 使用的是相等关系的`==`操作符。

`point` 类使用不同的规则定义了`==`和`<`，可以获得正确的比较和相等语义，下面是应用于标准容器的示范代码：

```
point p0, p1(1,2,3), p2(5,6,7), p3(3,2,1);

using namespace boost::assign;
vector<point> v = (list_of(p0), p1, p2, p3);

BOOST_AUTO(pos, std::find(v.begin(), v.end(), point(1,2,3)));
//使用相等语义查找元素
for (; pos != v.end(); //查找下一个相等的元素
    pos = std::find(pos + 1, v.end(), point(1,2,3)))
{ pos->print(); //1, 2, 3 }

pos = std::find(v.begin(), v.end(), point(2,1,3));
assert(pos == v.end());
```

这段代码将只找到 `p1(1,2,3)`，并且最后一个 `assert` 断言成立，找不到值为 `(2,1,3)` 的 `point` 对象。

如果我们改变 `point` 的定义，不使用 `equality_comparable`，而改用 `equivalent` 来实现`==`操作符（等价语义），那么我们不必单独定义`==`操作符，它将由 `equivalent` 自动用`<`操作符来实现。上面的测试代码的行为将完全不同，它会输出两个点：`p1(1,2,3)` 和 `p3(3,2,1)`，并且断言失败。

在使用关联容器 `set` 和 `map` 时更需要留意，它们仅要求`<`操作符，因此是基于等价语义的，即使使用 `equality_comparable` 定义了`==`操作符，把 `point` 对象放入 `set` 或 `map` 也会产生

equivalent 的效果。

请读者谨慎地考虑自定义类需要什么样的语义，如果只关心类的等价语义，那么就用 equivalent，如果想要精确地比较两个对象的值，就使用 equality_comparable。

4.8.6 解引用操作符

operators 库使用 dereferenceable 提供了对解引用操作符*、->的支持，它的用法与之前介绍的算术操作符不太相同。

dereferenceable 的类摘要如下：

```
template <class T, class P, class B = ...>
struct dereferenceable : B
{
    P operator->() const;
};
```

dereferenceable 有三个模板参数：

- 第一个参数 T 是要实现 operator->的子类，它的含义与算术操作符类相同；
- 第二个参数 P 是 operator->所返回的类型，也就是指针类型，通常应该是 T*；
- 最后一个参数 B 是用于基类链技术的父类，实际使用时我们不需要关心。

dereferenceable 类要求子类提供 operator*，会自动实现 operator->。注意它的 operator->函数的定义，不是如其他算术操作符类那样的友元函数，因此在使用 dereferenceable 时必须使用 public 继承，否则 operator->将会成为类的私有成员函数，外界无法访问。

由于 dereferenceable 实现了解引用操作符的语义，因此它可以用于实现自定义的智能指针类，或者是实现代理模式，包装代理某些对象。

例如，下面的代码实现了一个简单的智能指针类 my_smart_ptr，它 public 继承了 dereferenceable，重载 operator* 并自动获得了 operator->的定义：

```
template<typename T>
class my_smart_ptr:
    public dereferenceable<my_smart_ptr<T>, T* >    //必须 public 继承
{
    T *p;                                           //内部保存的指针
public:
    my_smart_ptr(T *x):p(x) {}                    //构造函数
```

```

~my_smart_ptr(){delete p;} //析构函数
T& operator*() const      //operator*定义,必须是常函数
{
    return *p;
};

```

my_smart_ptr 的用法就像是 scoped_ptr:

```

my_smart_ptr<string > p(new string("123"));
assert(p->size() == 3);

```

4.8.7 下标操作符

operators 库使用 indexable 提供了下标操作符[]的支持,它也属于解引用的范畴,用法与 dereferenceable 很相似,类摘要如下:

```

template <class T, class I, class R, class B >
struct indexable : B
{
    R operator[](I n) const;
};

```

indexable 模板参数列表中的 T 和 B 含义与 dereferenceable 的 T 和 B 含义相同,分别是子类型和基类链的父类类型。

参数 I 是下标操作符的值类型,通常应该是整数,但也可以是其他类型,只要它能够与类型 T 做加法操作。参数 R 是 operator[] 的返回值类型,通常应该是一个类型的引用。

indexable 要求子类提供一个 operator+(T, I) 的操作定义,类似于一个指针的算术运算,它应该返回一个迭代器类型,能够使用 operator* 解引用得到 R 类型的值。

与 dereferenceable 的例子类似,我们使用一个 my_smart_array 类来模仿实现 scoped_array。my_smart_array 从 indexable 公开继承,下标操作符类型是 int, operator[] 的返回值类型是 T&:

```

template<typename T>
class my_smart_array:
    public indexable<my_smart_array<T>, int, T& >
{
    T *p; //保存动态数组指针
public:
    typedef my_smart_array<T> this_type;
    typedef T* iter_type; //迭代器类型
    my_smart_array(T *x):p(x){} //构造函数
    ~my_smart_array(){delete[] p;} //析构函数
    friend iter_type operator+(const this_type& a, int n)
    {

```

```

        return a.p + n;                //返回一个迭代器，可以使用 operator*操作
    }
};

```

由于 `my_smart_array` 实现了 `operator+`，因此它支持算术运算，又由于它继承自 `indexable`，所以它还自动获得了 `operator[]` 的定义：

```

my_smart_array<double> ma(new double[10]);
ma[0] = 1.0;                        //operator[]
*(ma + 1) = 2.0;                    //指针算术运算
cout << ma[1] << endl;              //输出 2.0

```

4.8.8 高级议题

本小节讨论关于 `operators` 库的一些高级议题。

二元操作符

二元操作符（如`-`、`+`）的两个参数一般是同类型的，但有时候也可能是不同类型的。比如 `point`，可以允许它与一个整型的标量做加减法，其效果是每个坐标值对标量做加减法。

`operators` 库提供了使用两个模板类型参数的概念类，用来支持这种用法，例如 `less_than_comparable < T, U>`。对于不支持模板偏特化的编译器，`operators` 为每个操作符提供额外的两种形式，增加后缀“1”和“2”。如果程序可能在不同的编译器上编译，为了兼容请使用带后缀的模板。例如：

```

class point:
{
public:
    ...
    point& operator+=(const int& r)
    {
        x += r;
        y += r;
        z += r;
        return *this;
    }
};

```

//一个模板参数的概念类
//两个模板参数的概念类

operators 复合运算概念

`operators` 库里有一个特别的复合运算概念：`operators`，它提供了所有数学操作符的重载，包括比较运算、算术运算、位运算和步进运算。但建议读者最好不要用它，它是历史遗留产

物，其模板声明中没有 Base 类，不能被用于基类链与其他概念组合。而且 operators 库没有对它的改进打算。

其他操作符重载

除了基本的算术操作符、比较操作符和解引用操作符，operators 库还实现了更多的操作符，包括许多复杂的复合运算概念，还有迭代器操作符和一些辅助模板类。

限于篇幅本书不对它们进行详细解释，但如果读者理解了 operators 库的基本原理和用法，掌握它们不会是一件困难的事情。

更多的例子

operators 库非常实用，本书在以下的章节也用到了它来简化操作符重载定义，读者可进一步参考：

- 5.1.4 小节（166 页）“应用于自己的类”，依据 operator 的设计原理，实现了流操作符的重载；
- 7.1.6 小节（243 页）“实现 ref_array”和附录，有 operators 的更详细的使用例子。
- 9.2.10 小节（348 页）“实现无限精度的整数类型”，实现了一个具有无限精度整数类 iint，用到了 totally_ordered 和 addable。

4.9 exception

异常是 C++ 错误处理的重要机制，它改变了传统的使用错误返回值的处理模式，简化了函数的接口和调用代码，有助于编写整洁、优雅、健壮的程序。C++98 标准定义了标准异常类 std::exception 及一系列子类，是整个 C++ 语言错误处理的基础。

boost.exception 库针对标准库中异常类的缺陷进行了强化，提供 << 操作符重载，可以向异常传入任意数据，有助于增加异常的信息和表达力。

exception 位于名字空间 boost，为使用 exception，需要包含头文件 <boost/exception/all.hpp>^①，即：

```
#include <boost/exception/all.hpp>
```

① exception 库原本位于头文件 <boost/exception.hpp>，但这个头文件自 boost1.42 版开始被正式废弃，包含它会导致一个编译错误。如果读者曾经使用过旧版的 exception 库，又不想改动已有的代码，可以编辑 <boost/exception.hpp>，去掉 #error 预处理指令，改为包含 <boost/exception/all.hpp>。


```
using namespace boost;
```

4.9.1 标准库中的异常

为了使用 `boost.exception`，我们需要先了解 C++98 标准规定的异常体系。

C++98 标准中定义了一个异常基类 `std::exception` 和 `try/catch/throw` 异常处理机制，`std::exception` 又派生出若干子类，用以描述不同种类的异常，如 `bad_alloc`、`bad_cast`、`out_of_range` 等等，共同构建了 C++ 异常处理框架。

C++ 允许任何类型作为异常抛出，但在 `std::exception` 出现后，我们应该尽量使用它，因为 `std::exception` 提供了一个很有用的成员函数 `what()`，可以返回异常所携带的信息，这比简单地抛出一个整数错误值或者字符串更好、更安全。

如果 `std::exception` 及其子类不能满足程序对异常处理的要求，我们也可以继承它，为它添加更多的异常诊断信息。例如，想要为异常增加错误码信息可以这样：

```
class my_exception : public std::exception           //继承标准库异常类
{
private:
    int err_no;                                       //错误码信息
public:
    my_exception(const char* msg, int err):           //构造函数
        std::exception(msg), err_no(err) {}         //初始化父类和错误码信息
    int get_err_no()                                 //获取自定义的错误码信息
    { return err_no; }
};
```

但当系统中需要很多不同种类的异常时，这种实现方法很快就成为了程序员的沉重负担——为了容纳不同的信息需要编写大量极其相似的代码。

而且，这种解法还存在一个问题：很多时候当发生异常时不能获得有关异常的完全诊断信息，而标准库的异常类一旦被抛出，它就成为了一个“死”对象，程序失去了对它的控制能力，只能使用它或者再抛出一个新的异常。

C++98 标准的异常处理机制还不够完美，需要 `boost.exception` 来完善。

接下来的讨论在说到 `exception` 时，如果不特别指明，均是指 `boost::exception` 类，标准库的异常使用 `std::exception`。

4.9.2 类摘要

`exception` 库提供两个类：`exception` 和 `error_info`，它们是 `exception` 库的基础。

exception 的类摘要如下:

```
class exception
{
protected:
    exception();
    exception( exception const & x );
    ~exception();
    template <class E,class Tag,class T>
private:
    friend E const & operator<<( E const &, error_info<Tag,T> const & );
};
template <class ErrorInfo,class E>
    typename ErrorInfo::error_info::value_type * get_error_info( E & x );
```

exception 类几乎没有公开的成员函数 (但有大量用于内部实现的私有函数和变量), 被保护的构造函数表明了它的设计意图: 它是一个抽象类, 除了它的子类, 任何人都不能创建或者销毁它, 这保证了 exception 不会被误用。

exception 的重要能力在于其友元操作符<<, 可以存储 error_info 对象的信息, 存入的信息可以用自由函数 get_error_info<>() 随时再取出来。这个函数返回一个存储数据的指针, 如果 exception 里没有这种类型的信息则返回空指针。

exception 特意没有从 std::exception 继承, 因为现实中已存的大量代码已经有很多 std::exception 的子类, 而如果 exception 也是 std::exception 的子类, 对 exception 再使用继承可能会引发“钻石型”多重继承问题。^①

error_info 的类摘要如下:

```
template <class Tag,class T>
class error_info
{
public:
    typedef T value_type;

    error_info( value_type const & v );
    value_type & value();
};
```

error_info 提供了向异常类型添加信息的通用解法。第一个模板类型参数 Tag 是一个标记, 我们已经在 3.11 小节 (91 页) 见过了类似的用法。它通常 (最好) 是一个空类, 仅用来标记 error_info

① 应该总对异常类使用虚继承, 这是由于异常的特殊处理机制决定的。但现实通常不那么理想, 但从现在开始, 对 boost::exception 应该总使用虚继承。

类，使它在模板实例化时生成不同的类型。第二个模板类型参数 `T` 是真正存储的信息数据，可以用成员函数 `value()` 访问。

4.9.3 向异常传递信息

`exception` 和 `error_info` 被设计为配合 `std::exception` 一起工作，自定义的异常类可以安全地从 `exception` 和 `std::exception` 多重继承，从而获得两者的能力。

因为 `exception` 被定义为抽象类，因此我们的程序必须定义它的子类才能使用它，如前所述，`exception` 必须使用虚继承的方式。通常，继承完成后自定义异常类的实现也就结束了，不需要“画蛇添足”：向它增加成员变量或者成员函数，这些工作都已经由 `exception` 完成了。例如：

```
struct my_exception :                //自定义异常类
    virtual std::exception,          //虚继承
    virtual boost::exception        //虚继承
{};                                  //空实现，不需要实现代码
```

接下来我们需要定义异常需要存储的信息——使用模板类 `error_info`。用一个 `struct` 作为第一个模板参数来标记信息类型，再用第二个模板参数指定信息的数据类型。由于 `error_info<>` 的类型定义较长，为了使用方便起见，通常需要使用 `typedef`。

下面的代码使用 `error_info` 定义了两个存储 `int` 和 `string` 的信息类：

```
//异常信息的类型
typedef boost::error_info<struct tag_err_no, int>    err_no;
typedef boost::error_info<struct tag_err_str, string> err_str;
```

当发生异常时，我们就可以创建一个自定义异常类，并用 `<<` 操作符向它存储任意信息，这些信息可以在任何时候使用 `get_error_info()` 函数提取。

示范 `exception` 用法的代码如下：

```
#include <boost/exception/all.hpp>
...
int main()
try
{
    using namespace boost;
    try
    {
        //抛出异常，存储错误码
        throw my_exception() << err_no(10);
    }
    catch (my_exception& e)
    //之前的异常类和信息类定义
    //function-try 块
    //捕获异常，使用引用形式
```



```

{
    //获得异常内存存储的信息
    cout << *get_error_info<err_no>(e)<<endl;
    cout << e.what()<<endl;
    e << err_str("other info");
    throw;
}
}
catch(my_exception& e)
{
    cout << *get_error_info<err_str>(e)<<endl;
}

```

//向异常追加信息
//再次抛出异常
//function-try 的捕获代码
//获得异常信息

代码里的注释已经解释了很多东西，我们再来看一下。

程序首先定义了一个异常类 `my_exception`，然后使用 `typedef` 定义了两种异常信息：`err_no` 和 `err_str`，用 `int` 和 `string` 分别存储错误码和错误信息。`main()` 函数使用 `function-try` 块来捕获异常，它把整个函数体都包含在 `try` 块中，可以更好地把异常处理代码与正常流程代码分离（较老的编译器可能不支持 `function-try` 的用法）。

`throw my_exception()` 语句创建了一个 `my_exception` 异常类的临时对象，并立刻使用 `<<` 向它传递了 `err_no` 对象，存入错误码 10。随后，异常被 `catch` 块捕获，自由函数 `get_error_info<err_no>(e)` 可以获得异常内部保存的信息值的指针，所以需要解引用操作符 `*` 访问。

异常还可以被追加信息，同样使用操作符 `<<`，最后在 `function-try` 的 `catch` 块部分，异常被最终处理，程序结束。

4.9.4 更进一步的用法

通过例子我们基本了解了 `exception` 的用法，可以看到 `exception` 的使用是相当清晰简单的，同时又提供了灵活强大的功能，使异常类有了更多的用途。

由于从 `exception` 派生的异常类定义非常简单，没有实现代码，因此，可以很容易地建立起一个适合自己程序的、精细完整的异常类体系，使每个异常类只对应一种错误类型。只要都使用虚继承，类体系可以任意复杂，充分表达错误的含义。

处理异常的另一个重要工作是定义错误信息类型，基本方法是使用 `typedef` 来具体化 `error_info` 模板类。这通常比较麻烦，特别是有大量信息类型的时候。因此 `exception` 库特意提供若干预先定义好的错误信息类，如同标准库定义的 `logic_err` 等类型，使程序员用起来更轻松：

```

typedef error_info<struct errinfo_api_function_,char const *>
    errinfo_api_function;
typedef error_info<struct errinfo_at_line_,int> errinfo_at_line;
typedef error_info<struct errinfo_errno_,int> errinfo_errno;
typedef error_info<struct errinfo_file_handle_,weak_ptr<FILE> >
    errinfo_file_handle;
typedef error_info<struct errinfo_file_name_,std::string> errinfo_file_name;
typedef error_info<struct errinfo_file_open_mode_,std::string>
    errinfo_file_open_mode;
typedef error_info<struct errinfo_type_info_name_,std::string>
    errinfo_type_info_name;

```

它们可以用于常见的调用 API、行号、错误代码、文件 handle、文件名等错误信息的处理。

例如：

```

try
{
    throw my_exception() << errinfo_api_function("call api")
        << errinfo_errno(101);
}
catch (boost::exception& e)
{
    cout << *get_error_info<errinfo_api_function>(e);
    cout << *get_error_info<errinfo_errno>(e);
}

```

另外，exception 库还提供三个预定义错误信息类型，但命名规则略有不同：

```

typedef error_info<struct throw_function_,char const *> throw_function;
typedef error_info<struct throw_file_,char const *> throw_file;
typedef error_info<struct throw_line_,int> throw_line;

```

这三个错误信息类型主要用于存储源代码的信息，配合宏 `BOOST_CURRENT_FUNCTION`（参见 4.12.2 小节，158 页）、`__FILE__` 和 `__LINE__` 使用，可以获得调用函数名、源文件名和源代码行号。

但如果这些预定义类不能满足要求，我们还要再使用 `typedef`。为解决这个不大不小的麻烦，我们可以自定义一个辅助工具宏 `DEFINE_ERROR_INFO`，它可以方便快捷地实现 `error_info` 的定义：

```

#define DEFINE_ERROR_INFO(type, name) \
    typedef boost::error_info<struct tag_##name, type> name

```

宏 `DEFINE_ERROR_INFO` 接受两个参数，`type` 是它要存储的类型，`name` 是所需要的错误信息类型名，使用预处理命令 `##` 创建了 `error_info` 所需要的标签类。它的使用方法很简单，就像是声明一个变量：

```

DEFINE_ERROR_INFO(int, err_no);

```

在宏展开后它相当于：

```
typedef boost::error_info<struct tag_err_no, int> err_no;
```

如果你担心 tag_ 前缀太简单，可能会与其他类名发生冲突（通常几率很小），也可以自己定制特殊的标记字符串，比如采用公司名+团队名的方式。但如果这两者发生变化（公司改制、团队合并等），标记的含义就会变得难以维护。本书建议用内置的 `__FILE__` 和 `__LINE__` 宏，像这样：

```
tag_##__FILE__##_LINE_##name
```

这几乎不会造成任何可能的冲突。

4.9.5 包装标准异常

exception 库提供一个模板函数 `enable_error_info<T>(T &e)`，其中 T 是标准异常类或者其他自定义类型。它可以包装类型 T，产生一个从 `boost::exception` 和 T 派生的类，从而在不修改原异常处理体系的前提下获得 `boost::exception` 的所有好处。如果类型 T 已经是 `boost::exception` 的子类，那么 `enable_error_info` 将返回 e 的一个拷贝。

`enable_error_info()` 通常用在程序中已经存在异常类的场合，对这些异常类的修改很困难甚至不可能（比如已经编译成库）。这时候 `enable_error_info()` 就可以包装原有的异常类，从而很容易地在不变动任何已有代码的基础上把 `boost::exception` 集成到原有异常体系中。

示范 `enable_error_info()` 用法的代码如下：

```
#include <boost/exception/all.hpp>
struct my_err{}; //某个自定义的异常类,未使用 boost::exception
int main()
{
    using namespace boost;
    try
    {
        //使用 enable_error_info 包装自定义异常
        throw enable_error_info(my_err()) << errinfo_errno(10);
    }
    catch (boost::exception& e) //这里必须使用 boost::exception 来捕获
    {
        cout << *get_error_info<errinfo_errno>(e)<<endl;
    }
}
```

注意代码中 `catch` 的用法，`enable_error_info()` 返回的对象是 `boost::exception` 和 `my_err` 的子类，`catch` 的参数可以是这两者中的任意一个，但如果要使用 `boost::exception` 所存储的信息，就必须用 `boost::exception` 来捕获异常。

`enable_error_info()` 也可以包装标准库的异常:

```
throw enable_error_info(std::runtime_error("runtime"))
    << errinfo_at_line(__ LINE __);           //包装标准异常
```

4.9.6 使用函数抛出异常

`exception` 库为异常增加了许多新的功能, 带来了很多好处, 但由于这样那样的原因, 程序中的异常类并不能总是从 `boost::exception` 继承, 必须使用 `enable_error_info()` 来包装。

`exception` 库提供 `throw_exception()` 函数来简化 `enable_error_info()` 的调用, 它可以代替原始的 `throw` 语句来抛出异常, 会自动使用 `enable_error_info()` 来包装异常对象, 而且支持线程安全, 比直接使用 `throw` 更好, 相当于:

```
throw (boost::enable_error_info(e))
```

从而确保抛出的异常是 `boost::exception` 的子类, 可以追加异常信息。例如:

```
throw_exception(std::runtime_error("runtime"));
```

在 `throw_exception()` 的基础上 `exception` 库又提供了一个非常有用的宏 `BOOST_THROW_EXCEPTION`, 它调用了 `boost::throw_exception()` 和 `enable_error_info()`, 因而可以接受任意的异常类型, 同时又使用 `throw_function`、`throw_file` 和 `throw_line` 自动向异常添加了发生异常的函数名、文件名和行号等信息。

但需要注意一点, 为了保证与配置宏 `BOOST_NO_EXCEPTIONS` 兼容(这个宏本书不做介绍), `throw_exception()` 函数和 `BOOST_THROW_EXCEPTION` 宏都要求参数 `e` 必须是 `std::exception` 的子类。

对于新写的异常类来说这通常不是问题, 因为它们总是从 `std::exception` 和 `boost::exception` 继承, 但假如旧代码中有非 `std::exception` 的派生异常就无法使用, 这在一定程度上限制了 `throw_exception()` 和 `BOOST_NO_EXCEPTIONS` 的应用范围。

如果确保在程序中总使用 `boost::exception`, 不会去定义配置宏 `BOOST_NO_EXCEPTIONS` (这应该是大多数情况), 那么可以修改 Boost 源代码, 在 `<boost/throw_exception.hpp>` 里注释掉 `throw_exception_assert_compatibility(e)` 这条语句, 以取消这个限制。^①

`throw_exception()` 和 `BOOST_THROW_EXCEPTION` 的示范代码请参见 4.9.7 小节。

① 但要小心, 修改 boost 程序库的代码属于 “hack” 行为, 后果需要自行承担。

4.9.7 获得更多的调试信息

exception 提供了方便的存储信息能力，可以向它添加任意数量的信息，但当异常对象被用 operator<< 多次追加数据时，会导致它存储有大量的信息（BOOST_THROW_EXCEPTION 就是个很好的例子）。如果还是采用自由函数 get_error_info 来逐项检索的话可能会很麻烦甚至不可能，这时我们需要另外一个函数：diagnostic_information()。

diagnostic_information() 可以输出异常包含的所有信息，如果异常是由宏 BOOST_THROW_EXCEPTION 抛出的，则可能相当多并且不是用户友好的，但对于程序开发者可以提供很好的诊断错误的信息。

示范 BOOST_THROW_EXCEPTION 和 diagnostic_information() 用法的代码如下：

```
#include <boost/exception/all.hpp>
using namespace boost;
struct my_err{}; //自定义的异常类
int main()
{
    using namespace boost;
    try
    {
        //使用 enable_error_info 包装自定义异常
        throw enable_error_info(my_err())
            << errinfo_errno(101)
            << errinfo_api_function("fopen");
    }
    catch (boost::exception& e)
    {
        cout << diagnostic_information(e)<<endl;
    }

    try
    {
        BOOST_THROW_EXCEPTION(std::logic_error("logic")); //必须是标准异常
    }
    catch (boost::exception& e)
    {
        cout << diagnostic_information(e)<<endl;
    }
}
```

程序运行结果可能如下：

```
Throw in function (unknown)
```



```
Dynamic exception type: struct boost::exception_detail::error_info_injector<struct my_err>
[struct boost::errinfo_api_function_*] = fopen
[struct boost::errinfo_errno_*] = 101, "Unknown error"

xxx.cpp(65): Throw in function int __cdecl main(void)
Dynamic exception type: class boost::exception_detail::clone_impl<struct boost::exception_detail::error_info_injector<class stlpd_std::logic_error> >
std::exception::what: logic
```

运行结果显示了普通的 throw 语句与 BOOST_THROW_EXCEPTION 的不同, 后者可以显示出更多对调试有用的信息。

exception 库里还有一个更方便的函数 current_exception_diagnostic_information(), 它只能在 catch 块内部使用, 以 std::string 返回异常的诊断字符串, 这免去了指定异常参数的小麻烦。

4.9.8 高级议题

本小节讨论关于 exception 库的一些高级议题。

对异常信息打包

exception 支持使用 boost::tuple (参见 7.5 小节, 271 页) 对异常信息进行组合打包, 当异常信息类型很多又经常成组出现时可以简化抛出异常的编写。例如:

```
//tuple 类型定义
typedef tuple<errinfo_api_function, errinfo_errno > err_group;
try
{
    //使用 enable_error_info 包装自定义异常
    throw enable_error_info(std::out_of_range("out"))
        << err_group("syslogd", 874);
}
catch (boost::exception& )
{
    cout << current_exception_diagnostic_information()<<endl;
}
```

有关 tuple 的更多信息, 请参考 7.5 小节, 271 页。

类型转换

模板函数 current_exception_cast<E>() 提供类似标准库的转型操作, 它类似于 current_exception_diagnostic_information(), 只能在 catch 块内部使用, 可以把异

常对象转型为指向 E 类型的指针，如果异常对象无法转换成 E*，则返回空指针。

例如下面的代码把 `boost::exception` 转型成为了 `std::exception`，前提是异常必须是 `std::exception` 的子类：

```
catch (boost::exception& )
{
    cout << current_exception_cast<std::exception>()->what();
}
```

线程间传递异常

`exception` 库支持在线程间传递异常，这需要使用 `boost::exception` 的 `clone` 能力。使用 `enable_current_exception()` 包装异常对象或者使用 `throw_exception()` 都能够包装异常对象使之可以被 `clone`。

当发生异常时，线程在需要 `catch` 块调用函数 `current_exception()` 得到当前异常对象的指针 `exception_ptr` 对象，它指向异常对象的拷贝，是线程安全的，可以被多个线程同时拥有并发修改。`rethrow_exception()` 可以重新抛出异常。

示范在线程中处理异常的代码如下：

```
void thread_work()                                //线程工作函数
{ throw_exception(std::exception("test")); }
int main()
{
    using namespace boost;
    try
    {
        thread_work();                             //启动一个线程，可能抛出异常
    }
    catch (...)
    {
        exception_ptr e = current_exception();
        cout << current_exception_diagnostic_information();
    }
}
```

4.10 uuid

`uuid` 库是一个小的实用工具，可以表示和生成 UUID。

UUID 是 Universally Unique Identifier 的缩写，它是一个 128 位的数字（16 字节），不需要有一个中央认证机构就可以创建全球唯一的标识符。例如 “E4A0D7CE-9E6D-4E74-

9E6D-7E749E6D7E74” 就是一个 UUID。

UUID 的另一个别名是 GUID，在微软的 COM 中被广泛使用，用于标识 COM 组件接口。UUID 还可以用在很多地方，比如用于数据库记录的 RowID、标识某个系统的用户、标识网络传输消息等等……只要你想唯一地标识一个实体，就可以使用 UUID。

uuid 位于名字空间 boost::uuids，但它没有一个集中的头文件，而是把功能分散在了若干小文件中，因此为了使用 uuid 组件，需要包含数个头文件^①，即：

```
//uuids.hpp 可以自定义一个头文件，包含 uuid 的所有声明头文件
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
using namespace boost::uuids;
```

4.10.1 类摘要

uuid 库使用类 uuid 来表示 UUID，它的类摘要如下：

```
class uuid {
public:
    static size_type static_size();
    size_type size() const;
    uint8_t data[static_size()]; //内部实现

    iterator begin();
    iterator end();

    bool is_nil() const;

    enum variant_type {
        variant_ncs,           //NCS backward compatibility
        variant_rfc_4122,      //defined in RFC 4122 document
        variant_microsoft,     //Microsoft Corporation backward compatibility
        variant_future         //future definition
    };
    variant_type variant() const;

    enum version_type {
        version_unknown = -1,
        version_time_based = 1,
        version_dce_security = 2,
```

① Boost1.43 版中 uuid 库修正了一个小小的 bug，并将生成器分散在了数个头文件中，但仍然使用 uuid_generators.hpp。



```

    version_name_based_md5 = 3,
    version_random_number_based = 4,
    version_name_based_shal = 5
};
version_type version() const;

void swap(uuid& rhs);
};

```

在以上列出的成员函数之外, uuid 还全面支持比较操作和流输入输出。两个 uuid 值的比较是基于字典序的, 分别使用了标准算法 `std::equal()` 和 `std::lexicographical_compare()`。

4.10.2 用法

uuid 是一个很小的类, 目标是尽量简单, 便于高效率地操作。因此它特意被设计为没有构造函数, 可以像 POD 数据类型一样使用。

uuid 内部使用一个 16 字节的数组 `data` 作为 UUID 值的存储, 这个数组是 `public` 的, 因此可以任意访问, 比如拷贝或者赋值。基于 `data` 数组, uuid 提供了 `begin()` 和 `end()` 的迭代器支持, 可以像一个容器一样操作 UUID 值的每个字节。成员函数 `size()` 和静态成员函数 `static_size()` 可以获得 UUID 的长度, 是一个固定值, 大小总为 16。因此, 某种程度上可以把 uuid 看作是一个容量固定为 16、元素类型为 `unsigned char` 的容器。

示范 uuid 容器用法的代码如下:

```

#include "uuids.hpp"                                //自定义头文件
using namespace boost::uuids;                       //名字空间
int main()
{
    uuid u ;                                         //声明一个 uuid 对象
    assert(uuid::static_size() == 16);             //uuid 的长度总是 16
    assert(u.size() == 16);

    vector<unsigned char> v(16, 7);                 //一个 vector 对象
    std::copy(v.begin(), v.end(), u.begin());       //使用标准拷贝算法
    assert(u.data[0] == u.data[1]                  //数组方式访问
           && u.data[15] == 7);

    //流输出:07070707-0707-0707-0707-070707070707
    cout << u << endl;
    std::fill_n(u.data+ 10,6, 8);                   //标准算法 fill_n 直接操纵数组

    //流输出:07070707-0707-0707-0707-080808080808
    cout << u << endl;
}

```

uuid 内部定义的枚举类型 `variant_type` 标识了 UUID 的变体号, 表示了 UUID 的布局类型, 成员函数 `variant()` 可以获得这个 UUID 的变体号。

UUID 的生成有不同的算法, 这些算法使用枚举 `version_type` 来标识, `version()` 函数可以获得 UUID 的算法版本。uuid 类可以识别现有的五种生成算法, 分别是:

- 基于时间和 MAC 的算法 (`version_time_based`);
- 分布计算环境算法 (`dce_security`);
- MD5 摘要算法 (`version_name_based_md5`);
- 随机数算法 (`version_random_number_based`);
- SHA1 摘要算法 (`version_name_based_shal`)。

在数量庞大的 UUID 中有一个特殊的全零值 `nil`, 它表示一个无效的 UUID, 成员函数 `is_nil()` 可以检测 uuid 是否是 `nil`。

示范 uuid 用于表示 UUID 用法的代码如下:

```
uuid u ;
std::fill_n(u.begin(), u.size(), 0xab);           //直接填入 0xab
assert(!u.is_nil());                               //不是无效 UUID
assert(u.variant() == u.variant_rfc_4122);        //4122 变体类型
assert(u.version() == u.version_unknown);         //生成算法未知
cout << u << endl;                                //一串 ab 值

std::memset(u.data, 0, 16);                         //可以使用 memset 操纵数组
assert(u.is_nil());                                 //此时是一个无效的 UUID
```

uuid 还支持各种比较操作, 例如:

```
uuid u1 ,u2;                                       //两个 uuid 变量

std::fill_n(u1.begin(), u1.size(), 0xab);         //全 0xab
std::fill_n(u2.begin(), u2.size(), 0x10);         //全 0x10
assert(u1 != u2 && u1 > u2);                      // u1 > u2

u2.data[0] = 0xff;                                //u2 的第一个字节改为 0xff
assert( u1 < u2);                                  //此时 u1 < u2

std::memset(u1.data, 0, 16);                       //两个 uuid 都置为全 0
std::memset(u2.data, 0, 16);
assert(u1 == u2);                                  //两者相等
```

4.10.3 生成器

使用 uuid 提供的数组和迭代器接口，我们可以直接写出任意的 UUID 值，但因为不使用规定的算法，手工创建的 UUID 值很容易重复。这种方式只适合于从其他地方获得的原始 UUID 值导入到 uuid 对象中，如果要创建属于自己的 UUID，我们需要使用 UUID 生成器。

uuid 库提供了四种生成器，分别是 Nil 生成器、字符串生成器、名字生成器和随机生成器。它们都是函数对象，重载了 operator()，可以直接调用生成 uuid 对象。

Nil 生成器

Nil 生成器是最简单的 UUID 生成器，只能生成一个无效的 UUID 值（即全零的 UUID），它的存在只是为了方便地表示无效 UUID。

Nil 生成器的类名是 nil_generator，另外有一个内联函数 nil_uuid()，相当于直接调用了 Nil 生成器。

示范 Nil 生成器用法的代码如下：

```
uuid u = nil_generator()();  
assert(u.is_nil());  
  
u = nil_uuid();  
assert(u.is_nil());
```

注意：代码的第一行，在 nil_generator 类名后面出现了两对圆括号，不熟悉函数对象的读者可能会不太理解。它的语义解析是：第一对圆括号与 nil_generator 结合，结果是调用 nil_generator 的构造函数，生成一个临时对象，然后第二对圆括号是 nil_generator 对象的 operator() 操作符重载，就像是一个函数调用，产生了一个 nil uuid 对象。

字符串生成器

字符串生成器 string_generator 可以从一个字符串创建出 uuid 对象，字符串可以是 C 数组（NULL 结尾）、string、wstring，或者是一对迭代器指定的字符序列的区间。

string_generator 对字符串的格式有严格的要求，有两种格式是可接受的。一种是没有连字符的全 16 进制数字，另一种是使用连字符，但必须符合 UUID 的定义，在第 5、7、9、11 字节前使用连字符，其他位置出现连字符都不允许。UUID 字符串也可以使用花括号括起来，除了花括号不能出现 16 进制数以外的任何字符，否则会抛出 runtime_error 异常。

示范 string_generator 用法的代码如下：

```

string_generator sgen;                                //声明字符串生成器对象

uuid u1 = sgen("0123456789abcdef0123456789abcdef");
cout << u1 << endl;
uuid u2 = sgen("01234567-89ab-cdef-0123-456789abcdef");
cout << u2 << endl;
uuid u3 = sgen(L"01234567-89ab-cdef-0123-456789abcdef");
cout << u3 << endl;

```

名字生成器

名字生成器 `name_generator` 使用基于名字的 SHA1 摘要算法，它需要先指定一个基准 UUID，然后使用字符串名字派生出基于这个 UUID 的一系列 UUID。名字生成器的典型的应用场景是为一个组织内的所有成员创建 UUID 标识，只要基准 UUID 不变，那么相同的名字总会产生相同的 UUID。

示范 `name_generator` 用法的代码如下：

```

//首先生成一个组织的 UUID
uuid www_xxx_com= string_generator()
    ("0123456789abcdef0123456789abcdef");
name_generator ngen(www_xxx_com);                    //构造名字生成器

uuid u1 = ngen("mario");                             //为名字 mario 生成 UUID
assert(!u1.is_nil() &&                               //version 是 sha1 算法
    u1.version() == uuid::version_name_based_shal);
uuid u2 = ngen("link");                             //为名字 link 生成 UUID
cout << u2 << endl;

```

随机生成器

随机生成器采用随机数生成 UUID，`uuid` 库使用 Boost 库的另一个组件 `random`（参见 9.4 小节，357 页）作为随机数的发生源，它可以产生高质量的伪随机数，保证生成的随机 UUID 不会重复。

随机生成器 `basic_random_generator` 是一个模板类，可以用模板参数指定要使用的随机数发生器，具体的随机数类可以参考 `random` 库。为了方便使用，`uuid` 定义了一个常用的生成器 `random_generator`，它使用 `mt19937` 作为随机数发生器。

示范随机生成器用法的代码如下：

```

random_generator rgen;                                //随机生成器

uuid u = rgen();                                       //生成一个随机的 UUID
assert(u.version() == uuid::version_random_number_based);
cout << u << endl;

```

4.10.4 增强的 uuid 类

uuid 类为了追求效率而没有提供构造函数,要生成一个 UUID 值必须要使用生成器。但有的时候这个操作步骤显得有些麻烦,因此我们可以从 uuid 类派生一个可以自动产生 UUID 值的增强类,以简化 UUID 的使用。

下面的代码实现了这个增强类 uuid_t,它是 uuid 的子类,因而具有 uuid 的全部能力。它在内部定义了两个生成器的静态成员变量^①,分别用来产生随机 UUID 和字符串 UUID,对应地也提供了两种重载形式的构造函数。对于 Nil 生成器,uuid_t 使用带 int 参数的构造函数来调用实现,而名字生成器则使用了接受 uuid 和字符串参数的构造函数。

uuid_t 还实现了两个类型转换操作符重载,可以隐式地转换成 uuid 对象,方便被应用在其他使用 uuid 类型的场景。

uuid_t 的全部实现代码如下:

```
using namespace boost::uuids;
class uuid_t : public uuid
{
private:
    static random_generator rgen;           //随机生成器
    static string_generator sgen;          //字符串生成器
public:
    uuid_t(): uuid(rgen()){}                //缺省构造函数,生成随机 UUID
    uuid_t(int): uuid(nil_uuid()){}         //0 值的 UUID 构造函数
    uuid_t(const char* str): uuid(sgen(str)) {} //字符串构造函数
    uuid_t(const uuid& u, const char* str):  //名字生成器构造函数
        uuid(name_generator(u)(str)) {}
    explicit uuid_t(const uuid& u): uuid(u){} //拷贝构造函数
    operator uuid()                        //转换到 uuid 类型
    { return static_cast<uuid&>(*this); }
    operator uuid() const                  //常函数,转换到 const uuid 类型
    { return static_cast<const uuid&>(*this); }
};
random_generator uuid_t::rgen;            //静态成员变量的定义
string_generator uuid_t::sgen;
```

由于 uuid_t 类封装了 uuid 的所有生成器,故它比 uuid 用起来更加方便容易,例如:

```
uuid_t u0 = 0;                            //0 值的 UUID
assert(u0.is_nil());
```

① 也可以不用静态成员变量,直接在构造函数中使用 UUID 生成器临时对象。


```

uuid_t u1,u2; //创建两个随机 UUID 值
cout << u1 << endl;
cout << u2 << endl;

uuid_t u3("{01234567-89ab-cdef-0123-456789abcdef}"); //字符串构造
cout << u3 << endl;

cout << uuid_t(u3, "test name gen") << endl; //通过名字构造

```

4.10.5 与字符串的转换

uuid 可以使用字符串生成器从字符串生成,但却没有提供一个反向的操作,不能直接获得一个 uuid 的字符串表示。因为 uuid 支持流输入输出,故可以使用 `std::stringstream` 转换为字符串,例如:

```

uuid u = string_generator() //字符串生成器
    ("01234567-89ab-cdef-0123-456789abcdef");
stringstream ss; //字符串流
ss << u; //uuid 输出到字符串流
string str; //字符串对象
ss >> str; //字符串流输出到字符串对象
cout << str << endl;

```

这种方法虽然可行,但要写很多代码,比较麻烦,我们可以用另一个 Boost 库组件 `lexical_cast` (将在 5.1 小节, 163 页介绍),它可以非常方便地实现字符串与 uuid 的双向转换(得益于 uuid 的流输入输出功能):

```

#include <boost/lexical_cast.hpp> // lexical_cast 头文件
using namespace boost;
using namespace boost::uuids;
int main()
{
    uuid u = lexical_cast<uuid> //字符串转换到 uuid
        ("01234567-89ab-cdef-0123-456789abcdef");
    cout << u << endl;
    string str = lexical_cast<string>(u); //uuid 转换到字符串
    cout << str << endl;
};

```

`lexical_cast` 的字符串转换 uuid 的用法很类似字符串生成器 `string_generator`,但功能要弱很多,因为 `lexical_cast` 的转换功能是基于 uuid 的流输入能力,因此只能接受连字符格式的字符串,而且不能有花括号。

4.10.6 SHA1 摘要算法

uuid 的名字生成器使用了 SHA1 摘要算法,这是一个很重要的密码学算法,可以将任意长度的文本压缩成一个只有 20 字节(160 位)的独一无二的摘要,被广泛地应用于防篡改、身份鉴定等安全认证领域。

sha1 算法位于名字空间 `boost::uuids::detail`,使用时需要包含头文件 `<boost/uuid/shal.hpp>`, 它的类摘要如下。

```
class sha1
{
public:
    typedef unsigned int(digest_type)[5];
    sha1();
    void reset();

    void process_byte(unsigned char byte);
    void process_block(void const* bytes_begin, void const* bytes_end);
    void process_bytes(void const* buffer, size_t byte_count);

    void get_digest(digest_type digest);
};
```

sha1 类的用法很简单,使用成员函数 `process_byte()`、`process_block()` 和 `process_bytes()` 以不同的方式把数据“喂”给 sha1 对象,当输入所有数据后用 `get_digest()` 获得计算出的摘要值。

示范 sha1 用法的代码如下:

```
#include <boost/uuid/shal.hpp>
using namespace boost::uuids::detail;
int main()
{
    sha1 sha; //sha1 对象

    char *szMsg = "a short message"; //用于摘要的消息
    sha.process_byte(0x10); //处理一个字节
    sha.process_bytes(szMsg, strlen(szMsg)); //处理多个字节
    sha.process_block(szMsg, szMsg + strlen(szMsg));

    unsigned int digest[5]; //摘要的返回值
    sha.get_digest(digest); //获得摘要
    for (int i = 0; i < 5 ; ++i)
    {
        cout << hex << digest[i]; //16 进制输出
    }
}
```

4.11 config

config 库主要是提供给 Boost 库开发者（而不是库用户）使用，它将程序的编译配置分解为三个正交的部分：平台、编译器和标准库，帮助他们解决特定平台特定编译器的兼容问题。

一般来说，config 库不应该被库用户使用，不过在这个库里，我们也可以找到几个有用的小工具。

4.11.1 BOOST_STRINGIZE

宏 BOOST_STRINGIZE 可以将任意字面量转换为字符串。为了使用 BOOST_STRINGIZE 宏，需要包含头文件<boost/config/suffix.hpp>，即：

```
#include <boost/config/suffix.hpp>
```

BOOST_STRINGIZE 是一个宏，这一点很重要，意味着它仅能用于编译期（准确地说是编译前预处理时），不支持运行时转换。如果要在运行时转换数字或者其他变量到字符串，请使用 5.1 小节的 lexical_cast（163 页）。

示范 BOOST_STRINGIZE 用法的代码如下所示，其中的 __LINE__ 是 C 标准中定义的内置宏，表示程序的行号，是一个 long 型整数：

```
cout << BOOST_STRINGIZE(__LINE__) << endl;
cout << std::boolalpha << (string("22") == BOOST_STRINGIZE(__LINE__)) << endl;
int x = 255;
cout << BOOST_STRINGIZE(x) << endl;
```

程序的运行结果如下：

```
21
true
x
```

代码的前两条语句运用了 BOOST_STRINGIZE，把编译期的整数转换成了字符串，而第三个 BOOST_STRINGIZE 因无法处理运行时的整数变量，只能把它转换成字面意义的字符串。

BOOST_STRINGIZE 最常见的用途是在泛型编程中将编译期常数转换成字符串，在调试或者输出日志时相当方便，如 boost.exception 中就用到了 BOOST_STRINGIZE。

4.11.2 BOOST_STATIC_CONSTANT

C++98 标准允许直接在类声明中为静态整型成员变量赋初始值，但并不是所有编译器都实现

了这个特性。如果使用的编译器缺乏这个能力，那么只能使用 `enum` 来变通实现。

下面的代码定义了一个类 `static_int`，它同时使用了这两种方法：

```
struct static_int
{
    static const int v1 = 10;           //静态整型变量赋初值
    enum { v2 = 20};                   //枚举得到编译期整数值
    int a[v2];
};
cout << static_int::v1 << endl;
cout << static_int::v2 << endl;
```

这段代码在 VC8 下编译运行正常，但在较老的 VC6 下编译时 `v1` 的声明不被认可，`v2` 的 `enum` 方式编译正常。

`BOOST_STATIC_CONSTANT` 宏为这个解决问题提供了一个可移植的简便语法：

```
BOOST_STATIC_CONSTANT(type, assignment)
```

使用 `BOOST_STATIC_CONSTANT` 宏，我们无须担心编译器的能力限制，代码会更具可移植性。它的用法就像是一个普通的变量赋值语句，只是使用了略微有些变形的宏形式，有点类似 `BOOST_AUTO`。

之前的例子可以用 `BOOST_STATIC_CONSTANT` 宏改写如下：

```
struct static_int
{
    BOOST_STATIC_CONSTANT(int, v1 = 10);
    BOOST_STATIC_CONSTANT(int, v2 = 20);
    ...
};
```

这样的代码比单纯的静态成员变量声明或者 `enum` 更清晰，具可读性。

4.11.3 禁止编译器警告

头文件 `<boost/config/warning_disable.hpp>` 可用于禁止特定编译器（目前有微软的 VC 和 intel 编译器）的警告，尤其是 VC 中“讨厌”的 C4996 警告。

C4996 警告表明 VC 编译器遇到了某个被标记为 `deprecated` 的函数调用，但大多数情况下而这些函数可能是标准的 C/C++ 库函数，用法完全正确——甚至在微软自己的某些库中也使用了这些函数。

例如下面的代码，它调用了 C 的标准字符串拷贝函数：

```
char s1[] = "teststr";
char s2[20];
strcpy(s2, s1);
```

将导致 VC8 报 C4996 警告: 'strcpy' was declared deprecated.

如果代码编译时出现了大量的 C4996 警告, 而你确信函数调用完全没有问题, 为了避免使重要的错误信息淹没在无聊的警告中, 可以考虑使用 `<boost/config/warning_disable.hpp>`。

使用该头文件应保证它出现在所有头文件之前, 即:

```
#include <boost/config/warning_disable.hpp>
#include <...>
```

最后提一点忠告, 不是所有的警告都可以忽略的, 有的警告预示着可能潜在的错误, 这也是为什么 Boost 仅仅禁止了 C4996 这一个警告的原因。

4.11.4 其他工具

除了 `BOOST_STRINGIZE`、`BOOST_STATIC_CONSTANT` 和头文件 `<boost/config/warning_disable.hpp>`, `config` 库还有很多用于解决编译器能力差异的有用的宏和工具, 如 `BOOST_JOIN`、`BOOST_EXPLICIT_TEMPLATE_TYPE` 等等, 本书在这里不做详细介绍, 读者可参考说明文档以获得更多的信息。

4.12 utility

`utility` 库不是一个有统一主题的 Boost 库, 而是包含了若干个很小但有用的工具。

本章开头介绍的 `noncopyable`、`swap` 都被归类在 `utility` 库里, 此外 `utility` 还包括其他很多个实用类, 如 `checked_delete`、`compressed_pair`、`base_from_member` 等。本小节详细阐述其中的两个组件, 其余的组件将在第 14 章“其他 Boost 组件”, 537 页介绍。

4.12.1 BOOST_BINARY

`BOOST_BINARY` 提供一组宏, 用于实现简单的二进制常量表示。

`BOOST_BINARY` 的定义位于 `<boost/utility/binary.hpp>` 中, 也可以通过 `<boost/utility.hpp>` 间接包含, 即:

```
#include <boost/utility/binary.hpp>           //或者
#include <boost/utility.hpp>
```

它使用 `boost.preprocessor` 预处理元编程工具将一组或多组 01 数字在编译期展开成为一个八进制数字。每个数字组之间可以用空格分隔，每组可以容纳 1 个到 8 个 0/1 数字。

这里特别要注意的是，数字组的长度一定不能超过八个，由于预处理器宏展开的限制，嵌套层次太深会导致无法通过编译，报出一大堆错误。

示范 `BOOST_BINARY` 用法的例子如下：

```
#include <boost/utility.hpp>
...
cout << hex << showbase;
cout << BOOST_BINARY(0110) << endl;
cout << BOOST_BINARY(0110 1101) << endl;
cout << BOOST_BINARY(10110110 01) << endl;
cout << bitset<5>(BOOST_BINARY(0110)) << endl;
```

运行输出结果如下：

```
0x6
0x6d
0x2d9
00110
```

除了最基本最通用的 `BOOST_BINARY` 宏之外，本组件还包含形如 `BOOST_BINARY_XXX` 的宏，其中的 `XXX` 是标准的整数后缀，如 `U(unsigned int)`、`UL(unsigned long)` 等，用以支持需要特定整数类型的地方。例如，如果编译器支持 `long long` (64 位整数)，则对应的宏是 `BOOST_BINARY_LL`。

示范这些宏用法的代码如下：

```
cout << BOOST_BINARY_UL(101 1001) << endl;
long long x = BOOST_BINARY_LL(1101);
cout << x << endl;
```

`BOOST_BINARY` 宏提供了很好的初始化操作方法，在某些需要按位操作的情况下特别有用，比如使用 `std::bitset`^①。而且 `BOOST_BINARY` 宏都是在编译期展开的，没有任何运行时开销。

4.12.2 BOOST_CURRENT_FUNCTION

微软编译器 (VC) 在 C89 的 `__FILE__` 和 `__LINE__` 之外定义了一些扩展宏，其中的 `__FUNCTION__` 宏可以表示函数名称，GCC、intel C 等编译器也定义有类似的宏，而 C99 标

① 在 STL 中，`std::bitset` 不能直接使用字符串构造，必须使用 `bitset<5>(string("10110"))` 这样的形式，导致生成一个 `string` 临时变量，效率较低。

准则定义了 `__func__` 宏以实现同样的功能。但目前的 C++98 标准不能这样做。

`BOOST_CURRENT_FUNCTION` 宏为 C++ 补充了这个功能，而且功能更强大，更具可移植性。

用法

为了使用 `BOOST_CURRENT_FUNCTION` 宏，需要包含 `<boost/current_function.hpp>`，即：

```
#include <boost/current_function.hpp>
```

只需要在代码中使用 `BOOST_CURRENT_FUNCTION` 宏，就可获得包含该宏的外围函数名称，它表现为一个包含完整函数声明的编译期字符串。如果 `BOOST_CURRENT_FUNCTION` 宏不处于任何函数作用域之内，则行为依编译器而不同（在 VC 上会报出编译错误）。

示范 `BOOST_CURRENT_FUNCTION` 宏用法的代码如下：

```
double func()
{
    cout << BOOST_CURRENT_FUNCTION << endl;
    return 0.0;
}
//string str = BOOST_CURRENT_FUNCTION; //err
int main()
{
    cout << __FUNCTION__ << endl;
    cout << BOOST_CURRENT_FUNCTION << endl;
    func();
}
```

程序在 VC8 编译后运行结果如下：

```
main
int __cdecl main(void)
double __cdecl func(void)
```

实现原理

读者可能会惊讶于 `BOOST_CURRENT_FUNCTION` 宏的奇特魔力，其实 `BOOST_CURRENT_FUNCTION` 宏并不神秘。它的实现代码实际上相当简单，仅仅是针对各种编译器把编译器特定的宏定义为 `BOOST_CURRENT_FUNCTION`，因此它的能力完全依赖于编译器。

例如对于 VC：

```
# define BOOST_CURRENT_FUNCTION __FUNCSIG__
```

而 GCC 则是：

```
# define BOOST_CURRENT_FUNCTION __PRETTY_FUNCTION__
```

虽然 `BOOST_CURRENT_FUNCTION` 的功能和实现都很简单，但它的确为函数名称的显示提供了一个通用的解决办法，在配合抛出异常或者输出诊断日志时非常有用。

4.13 总结

本章讨论了很多个有用的工具，涉及实际开发的方方面面，显得有些杂乱，但可以肯定，使用它们会极大地提高开发效率并减轻代码维护成本。本小节将对这些组件进行简要的总结，并提出一些使用建议或忠告。

本章首先介绍了 `noncopyable`，它为定义一个不允许复制的类提供了简单而清晰的解决方法，可以节省很多代码和时间。应当总用 `boost::noncopyable` 的名字空间域限定形式来使用它，而不是用 `using` 语句，避免在头文件打开 `boost` 名字空间。

`typeof` 库模仿了 C++0x 的 `typeof` 和 `auto` 关键字，在表达式赋值时不再需要写出类型定义，使得 C++ 具有了类似动态语言的特性。在 C++0x 标准可用之前，它是减轻程序员变量类型定义工作量的最好选择，本书也在多处都使用了它。不过 `typeof` 库不是万能的，毕竟它不被语言内建支持，因此偶尔会失效（引起编译器错误），但这只是极少数的情况。

`optional` 库使用容器语义为“无效值”提供了一个很好的解决方案。它类似容器与智能指针的混合体，可以容纳任意类型的元素，同时又提供 `operator*` 和 `operator->` 重载，可以像指针一样访问元素。在使用 `optional` 存储的元素之前，必须测试它的有效性。

`assign` 库演示了重载逗号、括号操作符的“神奇”用法，利用 C++ 的“语法糖”特性，我们可以写出近乎不可思议的代码，轻松完成原本要一大堆代码才能完成的初始化或者赋值任务，在构建原型或者测试程序时非常有用。`assign` 库对标准容器、容器适配器和 Boost 容器都提供了很全面的支持，这使得它具有很高的使用价值。

变量值的交换是一个看似简单实则复杂的操作，它被用来处理异常安全性和类的自我赋值，如果想要自己的类安全高效，那么应该提供一个好的 `swap` 函数，它是很多实用功能的基础，Boost 库中几乎所有的类都有 `swap()` 成员函数就清楚地说明了这一点。`boost::swap` 是对 `std::swap` 的泛化，可以通过各种方式找到最佳的对象交换方法，而 `std::swap` 只能提供最通用的交换。如果你自己的类实现了高效的交换方法，那么 `boost::swap` 就会自动调用它。

`singleton` 是一个很重要的设计模式，它让类在整个程序生命周期中只能有一个实例。Boost 库提供了两种实现的方式：`pool.singleton` 和 `serialization.singleton`。两者都要求模板类型参数 `T` 具有缺省构造函数，而且构造和析构时不能抛出异常，通常情况下这两个

要求都很容易满足。两者的区别是 `pool::singleton_default` 的用法比较简单，而 `serialization.singleton` 不仅支持模板参数方式，还支持继承方式，可以返回 `mutable` 或 `const` 实例。相反的，`pool::singleton_default` 的构造函数是私有的，就不能使用继承的方式。

`tribool` 实现了三态布尔逻辑，如同语言内置的 `bool` 类型一样简单好用，只需要稍微学习一下它的运算规则就能够完全掌握。我们还讨论了它与 `optional<bool>` 的细微区别。在本书的 9.2.10 小节（348 页）有它的一个使用例子，读者可以参考。

操作符重载简化了代码的编写，提供了清晰易懂的语法，但实现操作符重载是件烦琐的工作。`operators` 库提供了很多定义良好的模板类，可以极大地简化操作符重载的实现工作。程序员只需要用基类链技术继承它们，再实现少量必须的操作符，就能够轻松获得其他操作符的功能。

`operators` 库很强大易用，但保证操作符具有正确的语义还是程序员自己的责任，操作符的重载应该与它原来的含义基本一致——至少不能违背大多数人的常识。`equality_comparable`、`less_than_comparable` 和 `totally_ordered` 是最常用的操作符重载类，它们提供比较运算，被用于支持标准容器。其他的操作符应当慎重使用，只有当类具有明显的运算语义，操作符重载能够极大简化语法时才应该使用。读者可以参考 `assign` 库、`exception` 库、`format` 库和 `rational` 库，它们对操作符的重载恰到好处，保持了清晰自然的语义又没有给用户带来过多的语法学习困难。

`exception` 库是对 C++98 标准的异常体系的补充和完善，它可以在异常对象中存储任意数量任意类型的数据，有助于增强异常的表达能力。`boost::excepton` 被特意设计为不从 `std::exception` 派生，这使得它可以很好地配合已有代码工作，但我们最好是用虚继承的方式来使用它。`exception` 库还有一系列的辅助函数、类和宏，帮助我们更好地处理异常相关的问题。我们还实现了一个很有用的工具宏 `DEFINE_ERROR_INFO`，它可以简化错误信息的定义工作。

`uuid` 组件实现了对 UUID 的表示和处理，提供了基于名字和随机数的生成算法生成全球唯一的标识符，可以用在很多地方来唯一地标识对象。`uuid` 库里还附带了一个 SHA1 算法的实现，能够为任意长度的数据生成 SHA1 摘要。

最后我们讨论了 `config` 和 `utility` 库，它们提供了几个很有用的宏：`BOOST_STRINGIZE` 实现编译期字符串转换；`BOOST_STATIC_CONSTANT` 可以定义类的静态整型成员常量；`BOOST_BINARY` 便利了二进制数字的书写方法；`BOOST_CURRENT_FUNCTION` 能够输出函数名称字符串，它被 `exception` 库所使用。还有头文件 `<boost/config/warning_disable.hpp>`，可以禁止编译器的烦人警告。

本书在随后的部分将多次使用这些实用工具，读者将看到更多的使用例子，也应该尽快把它们应用到具体开发工作中去。

本章实现的实用函数和类如下：

- `DEFINE_ERROR_INFO`：它是一个简化异常信息定义的宏；
- `uuid_t`：它封装了 `boost::uuid` 及相关的生成器，可以非常容易地生成各种版本的 UUID。

第 5 章

字符串与文本处理

字符串与文本处理一直是 C++ 的弱项，虽然 C++98 标准提供了一个标准字符串类 `std::string`，暂解燃眉之急，但 C++ 仍然缺乏很多文本处理的高级特性，如正则表达式、分词等等，使得不少 C++ 程序员不得不转向其他语言（如 Perl、Python）。

Boost 填补了 C++ 这方面的空白，为整个 C++ 社区做出了重要的贡献。

本章将讨论 Boost 中五个字符串与文本处理领域的程序库。首先是两个与 C 标准库函数功能类似的 `lexical_cast` 和 `format`，它们关注于字符串的表示，可以将数值转化为字符串，对输出做精确的格式化。`string_algo` 库提供了大量常用的字符串处理函数，可以满足 90% 以上的应用需求，剩下的 10% 可以使用 `tokenizer` 和 `xpressive`，前者是一个分词器，而后者则是一个灵活且功能强大的正则表达式分析器，同时也是一个语法分析器。

使用 Boost，C++ 中文本处理的一切问题将迎刃而解。^①

5.1 lexical_cast

顾名思义，`lexical_cast` 库进行“字面量”的转换，类似 C 中的 `atoi` 函数，可以进行字

① 随着软件越来越国际化，Unicode 和区域字符编码转换逐渐成为了软件开发中无法回避的问题。C++98 标准定义了 `wchart_t` 和 `locale` 等概念，C++0x 还专门引入了 `<codecvt>`，但 Boost 库却还没有对字符国际化给出解决方案。因此，除了 5.4.7 小节，本章所讨论的字符串和文本处理均使用“窄字符” `char` 和 `std::string`，但大部分讨论均适用于 `wstring`。

另外，Boost 的很多字符串的处理都需要标准库的 `<locale>`，如果读者所使用的编译器或标准库实现没有这个头文件，那么本章的很多功能都无法使用。

字符串、整数/浮点数之间的字面转换。

`lexical_cast` 位于名字空间 `boost`，为了使用 `lexical_cast` 组件，需要包含头文件 `<boost/lexical_cast.hpp>`，即：

```
#include <boost/lexical_cast.hpp>
using namespace boost;
```

5.1.1 用法

我们都很熟悉 C 语言中的 `atoi()`、`atof()` 系列函数，它们可以把字符串转换成数值，但这种转换是不对称的，不存在如 `itoa()` 这样的反向转换（C 语言标准未提供，但有的编译器厂商会提供非标准函数，如 `_itoa()`），要想把数值转换为字符串，只能使用不安全的 `printf()`。

`lexical_cast` 使用类似 C++98 标准转型操作符（`xxx_cast`）的形式给出了通用、一致、可理解的语法。

使用 `lexical_cast` 可以很容易地在数值与字符串之间转换，只需要在模板参数中指定转换的目标类型即可，例如：

```
#include <boost/lexical_cast.hpp>
int main()
{
    using namespace boost;
    int x = lexical_cast<int>("100");           //字符串->整数
    long y = lexical_cast<long>("2000");        //字符串->长整数
    float pai = lexical_cast<float>("3.14159e5"); //字符串->float
    double e = lexical_cast<double>("2.71828"); //字符串->double

    cout << x << y << pai << e << endl;

    string str = lexical_cast<string>(456);      //整数->字符串
    cout << str << endl;
    cout << lexical_cast<string>(0.618) << endl; //float->字符串
    cout << lexical_cast<string>(0x10) << endl;  //16 进制整数->字符串
}
```

程序的运行结果如下：

```
100 2000 314159 2.71828
456
0.6179999999999999
16
```

使用 `lexical_cast` 时要注意，要转换成数字的字符串中只能有数字和小数点，不能出现

字母（表示指数的 e/E 除外）或其他非数字字符，也就是说，lexical_cast 不能转换如“123L”、“0x100”这样 C++ 语法许可的数字字面量字符串。而且 lexical_cast 不支持高级的格式控制，不能把数字转换成指定格式的字符串，如果需要更高级的格式控制，可使用 std::stringstream 或者 boost::format。

除了转换数值与字符串，lexical_cast 也可以转换 bool 类型，但功能很有限，不能使用 true/false 字面量，只能使用 1 或 0，例如：

```
cout << lexical_cast<bool>("1") << endl;
```

5.1.2 异常 bad_lexical_cast

当 lexical_cast 无法执行转换操作时会抛出异常 bad_lexical_cast，它是 std::bad_cast 的派生类。为了使程序更加健壮，我们需要使用 try/catch 块来保护转换代码，例如：

```
try //try 块里的每条语句都会引发异常
{
    cout << lexical_cast<int>("0x100");
    cout << lexical_cast<double>("HelloWorld");
    cout << lexical_cast<long>("1000L");
    cout << lexical_cast<bool>("false") << endl;
}
catch (bad_lexical_cast& e) //捕获异常
{
    cout << "error:" << e.what() << endl;
}
```

代码运行结果如下：

```
error:bad lexical cast: source type value could not be interpreted as target
```

我们也可以利用 bad_lexical_cast 来验证数字字符串的合法性，实现一个模板函数 num_valid() 的代码如下：

```
template<typename T>
bool num_valid(const char *str)
try
{
    lexical_cast<T>(str); //尝试转换数字
    return true;
}
catch(bad_lexical_cast&)
{
    return false;
};
```

函数 `num_valid()` 使用了一个 `function-try` 块捕获 `bad_lexical_cast` 异常, 如果对字符串调用 `lexical_cast` 成功则返回 `true`, 否则返回 `false`。它的使用方法如下:

```
int main()
{
    assert(num_valid<double>("3.14"));
    assert(!num_valid<int>("3.14"));
    assert(num_valid<int>("65535"));
}
```

这个小函数在验证用户输入有效性时会很有用。

5.1.3 对转换对象的要求

虽然 `lexical_cast` 的用法非常像转型操作符, 但它仅仅是在用法上模仿了转型操作符而已, 它实际上是一个模板函数。`lexical_cast` 内部使用了标准库的流操作, 因此, 对于它的转换对象有如下要求:

- 转换起点对象是可流输出的, 即定义了 `operator<<`;
- 转换终点对象是可流输入的, 即定义了 `operator>>`;
- 转换终点对象必须是可缺省构造和可拷贝构造的。

C++语言中的内建类型 (`int`、`double` 等) 和 `std::string` 都满足以上的三个条件, 它们也是 `lexical_cast` 最常用的工作搭档。

对于 STL 中的容器和其他用户自定义类型, 这些条件一般都不满足, 不能使用 `lexical_cast`。

5.1.4 应用于自己的类

如果想要将 `lexical_cast` 应用于自定义的类, 把类转换为可理解的字符串描述 (类似于 Java 语言中的 `Object.toString()` 用法), 只需要满足 `lexical_cast` 的要求即可。准确地说, 需要实现流输出操作符 `operator<<`。

示范重载流操作符的代码如下:

```
class demo_class
{
    friend std::ostream& operator<<(std::ostream& os, const demo_class& x)
    {
        os << "demo_class's Name";
        return os;
    }
}
```

```

    }
};
int main()
{
    cout << lexical_cast<string>(demo_class()) << endl;
}

```

代码的运行结果如下：

```
class demo_class
```

这段代码具有通用性，值得把它提取为一个模板类。可以仿造 boost.operator 库，定义一个模板类 outable，以简化流输出操作符<<的重载。注意，这里没有使用基类链技术，不能用于 operator 的基类串联，但可以很容易添加这个功能：

```

template<typename T>
struct outable
{
    friend std::ostream& operator<<(std::ostream& os, const T& x)
    {
        os << typeid(T).name();           //使用 typeid 操作符输出类型名称
        return os;
    }
};

```

这样，任何继承 outable 的类，都会自动获得流输出操作符<<或者 lexical_cast 的支持。使用 outable<T>，之前的例子就可以简化成：

```

class demo_class: outable<demo_class>{};
int main()
{
    cout << lexical_cast<string>(demo_class()) << endl;
}

```

outable<T>的实现只是简单地运用 typeid 操作符输出了类的名字，如果想扩展 outable<T>的功能，可以考虑要求模板类型 T 提供 to_string() 或者 print() 之类的函数来输出特定信息。

依据同样的原理，读者也可以实现流输入操作符>>的重载。

5.2 format

C++标准库提供了强大的、富有弹性的 IO 流处理，使用流可以对输出的格式做各种精确的控制，如宽度、精度、进制、填充字符、对齐等等，新式流输出操作符<<可以串联起任意数量的参数，非常自由。

但 C++ IO 流也不是完美无瑕的，精确输出的格式控制要写大量的操控函数，而且还会改变流的状态，用完后还需要及时恢复，有时候会显得十分烦琐（2.3.3 节就是一个很好的例子）。因此，还是有很多程序员怀念 C 语言中经典的 `printf()`，虽然它缺乏类型安全检查，还有其他的一些缺点，但它语法简单高效，并且被广泛地接受和使用，影响深远。许多其他的编程语言也都受 `printf()` 的影响提供类似的格式化输出机制，比如 Python、C#。

`boost.format` 库“扬弃”了 `printf`，实现了类似于 `printf()` 的格式化对象，可以把参数格式化到一个字符串，而且是完全类型安全的。

`format` 组件位于名字空间 `boost`，为了使用 `format` 组件，需要包含头文件 `<boost/format.hpp>`，即：

```
#include <boost/format.hpp>
using namespace boost;
```

5.2.1 简单的例子

我们先通过一个简单的例子来了解 `format`，看看它与 `printf()` 有什么相似和不同：

```
#include <boost/format.hpp>
using namespace boost;
int main()
{
    cout << format("%s:%d+%d=%d\n")%"sum" % 1 % 2 % (1+2);

    format fmt("(%1% + %2%) * %2% = %3%\n");
    fmt % 2 % 5 ;
    fmt % ((2+5)*5);
    cout << fmt.str();
}
```

程序运行结果如下：

```
sum:1+2=3
(2 + 5) * 5 = 35
```

大多数 C/C++ 程序员都应该对这段代码有似曾相识的感觉，`format` 的设计在很大程度上参照了 `printf()`，但用法上却有很大的不同^①。

程序的第一条语句演示了 `format` 的最简单用法，使用 `format(...)` 构造了一个 `format` 临时（匿名）对象。构造函数的参数是格式化字符串，其语法是我们非常熟悉的标准 `printf()`

① 如果读者了解 Python 或者 C# 语言，那么可能对 `format` 的格式就不会太陌生。

语法, 使用%x 来指定参数的格式。

因为要被格式化的参数个数是不确定的, printf() 使用了 C 语言里的可变参数 (即参数声明中的省略号), 但它是不安全的。format 模仿了流操作符<<, 重载了二元操作符 operator% 作为参数输入符, 它同样可以串联任意数量的参数, 因此,

```
format(...) % a % b %c
```

可以理解成:

```
format(...) << a << b << c
```

操作符%把参数逐个地“喂”给 format 对象, 完成对参数的格式化^①。

最后, format 对象支持流输出, 可以直接向输出流 cout 输出内部保存的已格式化好的字符串。

第一条 format 语句的等价 printf() 调用是:

```
printf("%s:%d+%d=%d\n", "sum", 1, 2, (1+2));
```

程序后面的三行代码演示了 format 的另一种用法, 预先创建一个 format 格式化对象, 它可以被后面的代码多次用于格式化操作。format 对象仍然用操作符%来接受被格式化的参数, 可以分多次输入 (不必一次给全), 但参数的数量必须满足格式化字符串的要求。最后, 使用 format 对象的 str() 成员函数获得已格式好的字符串向 cout 流输出。

第二个 format 用了略不同于 printf() 的格式化语法: "(%1% + %2%) * %2% = %3%\n", 有点类似 C# 语言, %X 可以指示参数的位置, 减少参数输入的工作, 是对 printf() 语法的一个改进。

第二个 format 对象的等价 printf() 调用是:

```
printf("(%d + %d) * %d = %d\n", 2, 5, 5, (2+5)*5);
```

5.2.2 输入操作符%

刚才的例子基本解释清楚了 format 组件的基本用法。大部分读者都会很容易理解它的格式化字符串, 而对于操作符%可能就比较难理解。但是, 选择 operator% 是经过了许多讨论而最终确定的, 有很多的理由支持它。

^① 实际上, format 内部实现中就有有一个函数名为 feed。

为什么使用重载操作符

`format` 库在形成过程中考虑过许多其他形式的实现，比如完全仿造 `printf()`，在函数里接受所有待格式化参数，像这样：

```
format("%s,%d...", x0,x1,...);
```

基于类型安全的考虑，`format` 不能使用省略号来实现可变参数。如果使用函数的调用形式，那么就需要定义不同参数数量的模板函数，如：

```
template <class T1, class T2, ..., class TN>
string format(string s, const T1& x1, ..., const TN& xN);
```

在 C++ 没有实现可变数量模板参数之前，无论定义多少个这样的重载形式，都无法满足格式化“无限”个参数的需求。

因此，必须使用操作符的方式来接受参数，而且 IO 流的 `operator<<` 已经应用了许多年，证明这种方式的确有效。

二元操作符 `operator%` 的声明如下：

```
format& operator%(T& x);
```

它接受 `format` 对象和任意值作为参数，然后返回 `format` 对象的引用，因此可以再对返回的 `format` 对象实施 `%` 操作符，所以

```
format("...") %x %y
```

被编译器解释为：

```
operator%(format("..."), x) %y =>
operator% (operator%(format("..."), x), y)
```

以次类推，从而能够接受无限个待格式化的参数。

为什么使用 `operator%`

接下来的问题是为什么不使用 `operator<<`，它可能比 `%` 更为广大程序员所熟悉，或者是 `operator[]`、`()`、……

`operator<<` 已经被定义为用于流输出，如果 `format` 也使用 `operator<<`，那么在编写代码的时候很容易造成概念上的混乱，编译器无法区分哪些 `<<` 是用于格式化，哪些 `<<` 是用于流输出，需要使用括号来限定 `format` 表达式。而且 `format` 本身也支持流输出，还有与其他算术运算符的优先级问题，都加重了混乱程度。因此，使用 `<<` 不是一个好的选择。

`operator[]`、`()` 和 `,` 也是可以采用的，比如 `assign` 库（4.4 小节，106 页）就重载了括

号和逗号操作符，同样串联了多个参数，可以是这样：

```
format(...)[x][y][z]
format(...)(x)(y)(z)
format(...),x,y,z
```

这些操作符在技术上是完全可行的，但代码风格不如%简洁、好看，也不能清晰地表明格式化操作的意图。

最后是一些语言习惯方面的原因。原 printf() 用户都很熟悉格式化字符串中的%，这能够给他们带来亲切感，认识到这是一个格式化操作，可以降低学习门槛。而且，有的语言（如 Python）的 print 语句就已经使用了%来分隔参数。

5.2.3 类摘要

format 并不是一个真正的类，而是一个 typedef，真正的实现是 basic_format，它们的声明如下：

```
template<class charT, class Traits=std::char_traits<charT> >
    class basic_format;
typedef basic_format<char >      format;
```

basic_format 的类摘要如下：

```
template<class charT, class Traits=std::char_traits<charT> >
class basic_format
{
public:
    explicit basic_format(const charT* str);
    explicit basic_format(const string_t& s);
    basic_format& operator= (const basic_format& x);

    string_t str() const;
    size_type size() const;
    void clear();
    basic_format& parse(const string_t&);

    //pass arguments through those operators :
    template<class T> basic_format& operator%(T& x);
    friend std::basic_ostream& operator<< (...)
}; //basic_format

typedef basic_format<char >      format;
typedef basic_format<wchar_t >  wformat;

string str(const format& );
```

`basic_format` 构造函数可以接受 C 字符串（以 0 结尾的字符数组）、`std::string` 作为格式化字符串，格式化字符串使用类 `printf` 的格式规则。构造函数都被声明为 `explicit`，因此我们只能使用显式构造。

成员函数 `str()` 返回 `format` 对象内部已经格式化好的字符串（不清空），如果没有得到所有格式化字符串要求的参数则会抛出异常。`format` 库还同时提供一个同名的自由函数 `str()`，它位于 `boost` 名字空间，返回 `format` 对象内部已格式化好的字符串。

成员函数 `size()` 可以获得已格式化好的字符串的长度，相当于 `str().size()`；同样，如果没有得到所有格式化字符串要求的参数则会抛出异常。

成员函数 `parse()` 清空 `format` 对象内部缓存，并改使用一个新的格式化字符串。如果仅仅想清空缓存，可以使用 `clear()`，它把 `format` 对象恢复到初始化状态。这两个函数执行后调用 `str()` 或 `size()` 都会抛出异常。

`format` 重载了 `operator%`，可以接受待格式化的任意参数。`%` 输入的参数个数必须恰好等于格式化字符串要求的数量，过多或过少在 `format` 对象输出时都会导致抛出异常。当调用 `str()` 输出字符串或 `clear()` 清空缓冲区之后，则可以再次使用 `%`。

`format` 还重载流输出操作符，因此可以直接向 IO 流输出已格式化好的字符串，相当于向流输出 `str()`。

5.2.4 格式化语法

`format` 基本继承了 `printf` 的格式化语法，以便老程序员能够尽快掌握，它仅对 `printf` 语法有少量的不兼容，一般情况下我们很难遇到。

每个 `printf` 格式化选项以 `%` 开始，后面是格式规则，规定了输出的对齐、宽度、精度、字符类型，我们都非常熟悉，例如：

- `%05d` : 输出宽度为 5 的整数，不足位用 0 填充；
- `%-8.3f` : 输出左对齐，总宽度为 8，小数位 3 位的浮点数；
- `% 10s` : 输出 10 位的字符串，不足位用空格填充；
- `%05X` : 输出宽度为 5 的大写 16 进制整数，不足位用 0 填充。

示范这几个格式化选项的代码如下：

```
format fmt("%05d\n%-8.3f\n% 10s\n%05X\n");  
cout << fmt %62 % 2.236 % "123456789" % 48;
```

运行结果如下：

```
00062
2.236
 123456789
00030
```

在经典的 `printf` 式格式化外，`format` 还增加了新的格式：

- `%|spec|`：与 `printf` 格式选项功能相同，但两边增加了竖线分隔，可以更好地区分格式化选项与普通字符；
- `%N%`：标记第 `N` 个参数，相当于占位符，不带任何其他的格式化选项。

使用 `%|spec|` 的形式，刚才的例子可以写成：

```
format fmt("%|05d|\n%|-8.3f|\n%| 10s|\n%|05X|\n");
```

它的输出与 `printf` 格式的 `format` 完全相同，但看起来更清楚，尤其是对于 `%| 10s|`，可以很明显地看出有格式化参数“空格”。

`%N%` 格式的用法请读者参考 5.2.1 小节（168 页）的例子。

其他使用 `format` 的例子可以参见第 6 章“正确性与测试”（213 页）和本书的其余章节。

5.2.5 format 的性能

`printf()` 不进行类型检查，直接向 `stdout` 输出，因此它的速度非常快，而 `format` 较 `printf()` 做了很多安全检查的工作，因此性能略差，速度上要慢很多，总的来说要比 `printf()` 至少慢 2 到 5 倍。

不过如今很多程序的运行环境都不是很苛刻，而且字符串的格式化一般并不是应用程序的性能瓶颈，对于一个直接与用户交互的人机界面来说，大部分时间都花费在对用户输入的等待上，`format` 的性能损失完全是可以忽略不计的。

如果真的很在意 `format` 的性能，那么可以先建立 `const format` 对象，然后拷贝这个对象进行格式化操作，这样比直接使用 `format` 对象能够提高一些速度，像这样：

```
const format fmt("%|10d %|20.8f %|10X %|10.5e|\n");
cout << format(fmt) %62 % 2.236 % 255 % 0.618;
```

5.2.6 高级用法

`format` 提供了类似 `printf` 的功能，但它并不等同于 `printf` 函数，这就是面向对象的好

处。在通常的格式化字符串之外，`format` 类还拥有几个高级功能，可以在运行时修改格式化选项、绑定输入参数。

这些高级功能用到的函数如下：

■ `basic_format& bind_arg(int argN, const T& val)`

把格式化字符串第 `argN` 位置的输入参数固定为 `val`，即使调用 `clear()` 也保持不变，除非调用 `clear_bind()` 或 `clear_binds()`。

■ `basic_format& clear_bind(int argN)`

取消格式化字符串第 `argN` 位置的参数绑定。

■ `basic_format& clear_binds()`

取消格式化字符串所有位置的参数绑定，并调用 `clear()`。

■ `basic_format& modify_item(int itemN, T manipulator)`

设置格式化字符串第 `itemN` 位置的格式化选项，`manipulator` 是一个 `boost::io::group()` 返回的对象。

■ `boost::io::group(T1 al, ..., Var const& var)`

它是一个模板函数，最多支持 10 个参数（10 个重载形式），可以设置 IO 流操纵器以指定格式或输入参数值，IO 流操纵器位于头文件 `<iomanip>`。

我们使用例子来说明这些函数的用法，为求简化，参数都是 `int`：

```
#include <boost/format.hpp>
#include <iomanip>
using namespace boost;
using boost::io::group;
int main()
{
    //声明 format 对象,有三个输入参数,五个格式化选项
    format fmt("%1% %2% %3% %2% %1% \n");
    cout << fmt %1 % 2 % 3;

    fmt.bind_arg(2, 10);           //将第二个输入参数固定为数字 10
    cout << fmt %1 % 3;           //输入其余两个参数

    fmt.clear();                  //清空缓冲,但绑定的参数不变
}
```

```
//在%操作符中使用 group(),指定 IO 流操纵符第一个参数显示为八进制
cout << fmt % group(showbase,oct, 111) % 333;

fmt.clear_binds(); //清除所有绑定参数

//设置第一个格式化项,十六进制,宽度为 8,右对齐,不足位用*填充
fmt.modify_item(1, group(hex, right, showbase,setw(8), setfill('*')));
cout << fmt % 49 % 20 % 100;
}
```

程序运行结果如下:

```
1 2 3 2 1
1 10 3 10 1
0157 10 333 10 0157
****0x31 20 100 20 49
```

5.3 string_algo

C++98 在标准库中提供了字符串标准类 `std::string`, 它有一些成员函数可以查找子串, 访问字符, 可以执行基本的字符串处理功能。由于 `std::string` 符合容器的定义, 也可以把它看做是元素类型为 `char` (或 `wchar_t`) 的序列容器, 可以使用标准算法来对它进行运算, 但标准算法并不是为字符串处理定制的, 很多时候会显得有些“笨拙”。

`string_algo` 库的出现改变了这个尴尬的局面。它是一个非常全面的字符串算法库, 提供了大量的字符串操作函数, 如大小写无关比较、修剪、特定模式的子串查找等, 可以在不使用正则表达式的情况下处理大多数字符串相关问题。

`string_algo` 库位于名字空间 `boost::algorithm`, 但被 `using` 语句引入到了名字空间 `boost`, 为了使用 `string_algo` 组件, 需要包含头文件 `<boost/algorithm/string.hpp>`, 即

```
#include <boost/algorithm/string.hpp>
using namespace boost;
```

5.3.1 简单的例子

我们通过一个简单的例子来了解 `string_algo` 库中一些算法:

```
#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    string str("readme.txt");
```

```

if (ends_with(str, "txt")) //判断后缀
{
    cout << to_upper_copy(str) + " UPPER" << endl; //大写
    assert(ends_with(str, "txt"));
}

replace_first(str, "readme", "followme"); //替换
cout << str << endl;

vector<char> v(str.begin(), str.end()); //一个字符的 vector
vector<char> v2 = to_upper_copy( //大写
    erase_first_copy(v, "txt")); //删除字符串
for (int i = 0; i < v2.size(); ++i)
{
    cout << v2[i]; }
}

```

程序运行结果如下：

```

README.TXT UPPER
followme.txt
FOLLOWME.

```

这段代码示范了 `string_algo` 库中 `ends_with()`、`to_upper_copy()`、`replace_first()` 和 `erase_first_copy()` 等函数的基本用法，它们名称的含义都是自说明的，不言自明。

值得注意的是代码后面对 `std::vector` 的操作，它说明了 `string_algo` 库的一个重要特征：库里包含的是泛型算法，因此可作用于容器上（但需符合一些要求），不必是 `string` 或者 `basic_string<T>`。

5.3.2 string_algo 概述

`string_algo` 被设计用于处理字符串，然而它的处理对象并不一定是 `string` 或 `basic_string<T>`，可以是任何符合 `boost.range`^① 要求的容器。容器内的元素也不一定是 `char` 或 `wchar_t`，任何可拷贝构造和赋值的类型均可，但如果类型的拷贝赋值代价很高，那么 `string_algo` 的性能会下降。

当然，我们使用 `string_algo` 库主要的工作对象还是字符串 `string` 和 `wstring`，它也可以工作在标准容器 `vector`、`deque`、`list` 和非标准容器 `slist`、`rope`^② 上。为求简便，接下来

① `boost.range` 基于 STL 迭代器提出了“范围”的概念，是一个容器的半开空间，本书并没有对它进行详细论述。

② `rope` 是 SGISTL 实现的高效字符串类，STLport 中也同样提供，但它是非标准的，可能会有移植问题。

的讨论我们都基于 `std::string`。

`string_algo` 库基于 `boost.range`，因此避免了标准库算法必须提供 `begin()` 和 `end()` 迭代器的麻烦，也使得算法可以嵌套在一起串联处理字符串。

`string_algo` 库中的算法命名遵循了标准库的惯例，算法名均为小写形式，并使用不同的前缀或者后缀来区分不同的版本，命名规则如下：

- 前缀 `i` ：有这个前缀表明算法是大小写不敏感的，否则是大小写敏感的；
- 后缀 `_copy`：有这个后缀表明算法不变动输入，返回处理结果的拷贝，否则算法原地处理，输入即输出；
- 后缀 `_if` ：有这个后缀表明算法需要一个作为判断式的谓词函数对象，否则使用默认的判断准则。

`string_algo` 库提供的算法共分五大类，如下：

- 大小写转换；
- 判断式与分类；
- 修剪；
- 查找与替换；
- 分割与合并。

接下来将对这些算法进行详细介绍（但不包含 `regex` 库相关的算法）。

5.3.3 大小写转换

`string_algo` 库可以高效地实现字符串的大小写转换，包括两组算法：`to_upper()` 和 `to_lower()`。

这两个算法的声明如下：

```
template<typename T> void to_upper(T & Input);  
template<typename T> void to_lower(T & Input);
```

（由于附加了前缀 `i` 和后缀 `_copy` 的算法声明的声明形式与原地处理的算法差距不大，为节省篇幅不再列出，以下同。）

这两个算法具有后缀为 `_copy` 的版本。基本算法直接把输入在原地改变大小写，而 `copy` 版

则不变动输入，返回变动后的一个拷贝，用法如下：

```
#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    string str("I Don't Know.\n");
    cout << to_upper_copy(str);           //返回大写拷贝
    cout << str;                          //原字符串不改变
    to_lower(str);                        //字符串小写
    cout << str;                          //原字符串被改变
}
```

程序运行结果如下：

```
I DON'T KNOW.
I Don't Know.
i don't know.
```

5.3.4 判断式（算法）

判断式算法可以检测两个字符串之间的关系，包括：

- `starts_with` : 检测一个字符串是否是另一个的前缀；
- `ends_with` : 检测一个字符串是否是另一个的后缀；
- `contains` : 检测一个字符串是否被另一个包含；
- `equals` : 检测两个字符串是否相等；
- `lexicographical_compare`: 根据字典顺序检测一个字符串是否小于另一个；
- `all` : 检测一个字符串中的所有元素是否满足指定的判断式。

除了 `all`，这些算法都有另一个 `i` 前缀的版本。由于它们不变动字符串，因此没有 `_copy` 版本。

使用这些判断算法，可以很容易地解决困扰 C++ 程序员多年的大小写无关字符串比较的问题；只需要使用带 `i` 前缀的算法，如 `ilexicographical_compare`。

这些算法的声明如下：

```
template<typename Range1T, typename Range2T>
    bool starts_with(const Range1T & Input, const Range2T & Test);
template<typename Range1T, typename Range2T>
    bool ends_with(const Range1T & Input, const Range2T & Test);
template<typename Range1T, typename Range2T>
```

```

bool contains(const Range1T & Input, const Range2T & Test);
template<typename Range1T, typename Range2T>
bool equals(const Range1T & Input, const Range2T & Test);
template<typename Range1T, typename Range2T>
bool lexicographical_compare(const Range1T & Arg1, const Range2T & Arg2);
template<typename RangeT, typename PredicateT>
bool all(const RangeT & Input, PredicateT Pred);

```

另外要注意的是，它们的名称略微违背了 `string_algo` 库的命名原则，它们同时还有一种接受比较谓词函数对象的三参数版本，而没有使用 `_if` 后缀。谓词用于逐个地对字符串元素进行比较，相当于 `Pred(a[i],b[i])`。

示范这些算法的用法的代码如下：

```

#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    string str("Power Bomb");

    assert(iends_with(str, "bomb"));           //大小写无关检测后缀
    assert(!ends_with(str, "bomb"));           //大小写敏感检测后缀

    assert(starts_with(str, "Pow"));            //检测前缀

    assert(contains(str, "er"));                //测试包含关系

    string str2 = to_lower_copy(str);           //转换小写
    assert(iequals(str, str2));                 //大小写无关判断相等

    string str3("power suit");
    assert(ilexicographical_compare(str, str3)); //大小写无关比较

    assert(all(str2.substr(0, 5), is_lower())); //检测字符串均小写
}

```

这段代码大部分都很容易理解，最后一句话需要进行解释。`str2.substr(0, 5)` 返回 `str2` 前五个字符组成的子串，也就是“power”，`is_lower()` 是接下来就要讲到的一个分类算法，它可以检测字符是否是小写字母。算法 `all` 把这两个参数组合起来，检测“power”是否全为小写字母组成。

5.3.5 判断式（函数对象）

`string_algo` 增强了标准库中的 `equal_to<>` 和 `less<>` 函数对象，允许对不同类型的参数进行比较，并提供大小写无关的形式。这些函数对象包括：

- `is_equal` : 类似 `equals` 算法, 比较两个对象是否相等;
- `is_less` : 比较两个对象是否具有小于关系;
- `is_not_greater` : 比较两个对象是否具有不大于关系。

示范这些函数对象用法的代码如下:

```
#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    string str1("Samus"), str2("samus");

    assert(!is_equal()(str1, str2));
    assert(is_less()(str1, str2));
}
```

注意函数对象名称后的两个括号, 第一个括号调用了函数对象的构造函数, 产生一个临时对象, 第二个括号才是真正的函数调用操作符 `operator()`。

5.3.6 分类

`string_algo` 提供一组分类函数, 可以用于检测一个字符是否符合某种特性, 主要用于搭配其他算法, 比如刚才的 `all` 算法。

`string_algo` 库的分类函数如下:

- `is_space`: 字符是否为空格;
- `is_alnum`: 字符是否为字母和数字字符;
- `is_alpha`: 字符是否为字母;
- `is_cntrl`: 字符是否为控制字符;
- `is_digit`: 字符是否为十进制数字;
- `is_graph`: 字符是否为图形字符;
- `is_lower`: 字符是否为小写字符;
- `is_print`: 字符是否为可打印字符;
- `is_punct`: 字符是否为标点符号字符;

- `is_upper` : 字符是否为大写字符;
- `is_xdigit` : 字符是否为十六进制数字;
- `is_any_of` : 字符是否是参数字符序列中的任意字符;
- `if_from_range` : 字符是否位于指定区间内, 即 `from <= ch <= to`。

需要注意的是, 这些函数并不真正地检测字符, 而是返回一个类型为 `detail::is_classifiedF` 的函数对象, 这个函数对象的 `operator()` 才是真正的分类函数 (因此, 这些函数都属于工厂函数)。

函数对象 `is_classifiedF` 重载了逻辑运算符 `||`、`&&` 和 `!`, 可以使用逻辑运算符把它们组合成逻辑表达式, 以实现更复杂的条件判断。

如果 `string_algo` 提供的这些分类判断式不能满足要求, 我们也可以自己实现专用的判断式。方法很简单, 定义一个返回值为 `bool` 的函数对象就可以了。例如, 判断字符是否为 0 或 1 的代码如下, 它等价于 `is_any_of("01")`:

```
struct is_zero_or_one
{
    bool operator()(char &x)
    { return x == '0' || x == '1'; }
};
```

分类函数将结合其他算法示范其用法。

5.3.7 修剪

`string_algo` 提供 3 个修剪算法: `trim_left`、`trim_right` 和 `trim`。

修剪算法可以删除字符串开头或结尾部分的空格, 它有 `_if` 和 `_copy` 两种后缀, 因此每个算法都有四个版本。`_if` 版本接受一个判断式 `IsSpace`, 将所有被判定为空格 (`IsSpace(c) == true`) 的字符删除。

这些算法的声明如下:

```
template<typename SequenceT>
    void trim_left(SequenceT & Input);
template<typename SequenceT>
    void trim_right(SequenceT & Input);
template<typename SequenceT>
    void trim(SequenceT & Input);
```

示范修剪算法用法的代码如下，使用 `format` 库（参见 5.2 小节）进行输出格式化：

```
#include <boost/format.hpp>
#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    format fmt("|%s|\n");

    string str = " samus aran ";
    cout << fmt % trim_copy(str);           //删除两端的空格
    cout << fmt % trim_left_copy(str);      //删除左端空格

    trim_right(str);                        //原地删除右端的空格
    cout << fmt % str;

    string str2 = "2010 Happy new Year!!!";
    cout << fmt % trim_left_copy_if(str2, is_digit()); //删除左端的数字
    cout << fmt % trim_right_copy_if(str2, is_punct()); //删除右端的标点
    cout << fmt % trim_copy_if(str2, is_punct() || is_digit() || is_space()); //删除两端的标点、数字和空格
}
```

这段代码值得注意的是最后一行，它使用逻辑运算符组合了三个分类判断式，修剪了字符串两端所有的标点、数字和空格。

程序运行结果如下：

```
|samus aran|
|samus aran |
| samus aran|
| Happy new Year!!!|
|2010 Happy new Year|
|Happy new Year|
```

5.3.8 查找

`string_algo` 的查找算法提供与 `std::search()` 类似的功能，但接口不太一样。它不是返回一个迭代器（查找到的位置），而使用了 `boost.range` 库的 `iterator_range` 返回查找到的整个区间，获得了更多的信息，便于算法串联和其他处理（比如：根据 `iterator_range` 的两个迭代器将原字符串拆成三份）。

在这里简单介绍一下 `iterator_range`。它概念上类似 `std::pair<iterator, iterator>`，包装了两个迭代器，可以用 `begin()` 和 `end()` 访问，相当于定义了一个容器的子区间，并可以像

原容器一样使用。

string_algo 提供的查找算法包括：

- find_first : 查找字符串在输入中第一次出现的位置；
- find_last : 查找字符串在输入中最后一次出现的位置；
- find_nth : 查找字符串在输入中的第 n 次（从 0 开始计数）出现的位置；
- find_head : 取一个字符串开头 N 个字符的子串，相当于 substr(0,n)；
- find_tail : 取一个字符串末尾 N 个字符的子串。

它们的声明如下：

```
template<typename Range1T, typename Range2T>
    iterator_range find_first(Range1T & Input, const Range2T & Search);
template<typename Range1T, typename Range2T>
    iterator_range find_last(Range1T & Input, const Range2T & Search);
template<typename Range1T, typename Range2T>
    iterator_range find_nth(Range1T & Input, const Range2T & Search, int Nth);
template<typename RangeT>
    find_head(RangeT & Input, int N);
template<typename RangeT>
    find_tail(RangeT & Input, int N);
```

这些算法都不变动字符串，因此没有_copy 后缀版本，但其中前三个算法有 i 前缀的版本。示范这些算法用法的代码如下：

```
#include <boost/format.hpp>
#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    format fmt("|%s|. pos = %d\n");

    string str = "Long long ago, there was a king.";

    iterator_range<string::iterator> rge; //迭代器区间

    rge = find_first(str, "long"); //找第一次出现
    cout << fmt % rge % (rge.begin() - str.begin());

    rge = ifind_first(str, "long"); //大小写无关找第一次出现
    cout << fmt % rge % (rge.begin() - str.begin());

    rge = find_nth(str, "ng", 2); //找第三次出现
```

```

cout << fmt % rge % (rge.begin() - str.begin());

rge = find_head(str, 4);                                //取前 4 个字符
cout << fmt % rge % (rge.begin() - str.begin());

rge = find_tail(str, 5);                                //取末尾 5 个字符
cout << fmt % rge % (rge.begin() - str.begin());

rge = find_first(str, "samus");                          //找不到
assert(rge.empty() && !rge);
}

```

这段代码的最后两行说明 `iterator_range` 可以像标准容器一样判断是否为空，也可以隐式转换为 `bool` 值。

`string_algo` 库还有另外三个查找算法：`find_token`、`find_regex` 和通用的 `find` 算法，本书不做介绍，读者可阅读 Boost 文档了解其用法。

5.3.9 替换与删除

替换、删除操作与查找算法非常接近，是在查找到结果后再对字符串进行处理，因此它们的算法名称很相似。这些算法包括：

- `replace/erase_first`： 替换/删除一个字符串在输入中的第一次出现；
- `replace/erase_last`： 替换/删除一个字符串在输入中的最后一次出现；
- `replace/erase_nth`： 替换/删除一个字符串在输入中的第 *n* 次(从 0 开始)出现；
- `replace/erase_all`： 替换/删除一个字符串在输入中的所有出现；
- `replace/erase_head`： 替换/删除输入的头；
- `replace/erase_tail`： 替换/删除输入的末尾。

这些算法是一个相当大的家族。前八个算法每个都有前缀 `i_`、后缀 `_copy` 和组合，有四个版本，后四个则只有后缀 `_copy` 的两个版本。

`replace` 算法的声明如下：

```

template<typename SequenceT, typename Range1T, typename Range2T>
void replace_first(SequenceT & Input, const Range1T & Search,
                  const Range2T & Format);
template<typename SequenceT, typename Range1T, typename Range2T>
void replace_last(SequenceT & Input, const Range1T & Search,

```



```

        const Range2T & Format);
template<typename SequenceT, typename Range1T, typename Range2T>
    void replace_all(SequenceT & Input, const Range1T & Search,
        const Range2T & Format);
template<typename SequenceT, typename Range1T, typename Range2T>
    void replace_nth(SequenceT & Input, const Range1T & Search, int Nth,
        const Range2T & Format);
template<typename SequenceT, typename RangeT>
    void replace_head(SequenceT & Input, int N, const RangeT & Format);
template<typename SequenceT, typename RangeT>
    void replace_tail(SequenceT & Input, int N, const RangeT & Format);

```

erase 算法的声明如下:

```

template<typename SequenceT, typename RangeT>
    void erase_first(SequenceT & Input, const RangeT & Search);
template<typename SequenceT, typename RangeT>
    void erase_last(SequenceT & Input, const RangeT & Search);
template<typename SequenceT, typename RangeT>
    void erase_nth(SequenceT & Input, const RangeT & Search, int Nth);
template<typename SequenceT, typename RangeT>
    void erase_all(SequenceT & Input, const RangeT & Search);
template<typename SequenceT> void erase_head(SequenceT & Input, int N);
template<typename SequenceT> void erase_tail(SequenceT & Input, int N);

```

示范替换与删除算法用法的代码如下:

```

#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    string str = "Samus beat the monster.\n";

    cout << replace_first_copy(str, "Samus", "samus");

    replace_last(str, "beat", "kill");
    cout << str;

    replace_tail(str, 9, "ridley.\n");
    cout << str;

    cout << ierase_all_copy(str, "samus");
    cout << replace_nth_copy(str, "l", 1, "L");
    cout << erase_tail_copy(str, 8);
}

```

程序运行结果如下:

```

samus beat the monster.
Samus kill the monster.

```

```
Samus kill the ridley.
kill the ridley.
Samus kill the ridley.
Samus kill the
```

5.3.10 分割

`string_algo` 提供了两个字符串分割算法: `find_all` (虽然它的名称有 `find`, 但因为其功能而被归类为分割算法) 和 `split`, 可以使用某种策略把字符串分割成若干部分, 并将分割后的字符串拷贝存入指定的容器。

分割算法对容器类型的要求是必须能够持有查找到结果的拷贝或者引用, 因此容器的元素类型必须是 `string` 或者 `iterator_range< string::iterator>`, 容器则可以是 `vector`、`list`、`deque` 等标准容器。

这两个算法的声明如下:

```
template<typename SequenceSequenceT, typename Range1T, typename Range2T>
SequenceSequenceT &
find_all(SequenceSequenceT & Result, Range1T & Input,
         const Range2T & Search);
```

`find_all` 算法类似于普通的查找算法, 它搜索所有匹配的字符串, 加入到容器中, 有一个忽略大小写的前缀 `i` 版本。

```
template<typename SequenceSequenceT, typename RangeT, typename PredicateT>
SequenceSequenceT &
split(SequenceSequenceT & Result, RangeT & Input, PredicateT Pred, token_
compress_mode_type eCompress = token_compress_off);
```

`split` 算法使用判断式 `Pred` 来确定分割的依据, 如果字符 `ch` 满足判断式 `Pred (Pred(ch) == true)`, 那么它就是一个分割符, 将字符串从这里分割。

参数 `eCompress` 可以取值为 `token_compress_on` 或 `token_compress_off`, 如果值为前者, 那么当两个分隔符连续出现时将被视为一个, 如果为 `token_compress_off` 则两个连续的分隔符标记了一个空字符串。参数 `eCompress` 默认取值为 `token_compress_off`。

示范分割算法用的代码法如下:

```
#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    string str = "Samus,Link.Zelda::Mario-Luigi+zelda";
```

```

deque<string> d;
ifind_all(d, str, "zELDA"); //大小写无关分割字符串
assert(d.size() == 2);
for (BOOST_AUTO(pos, d.begin()); pos !=d.end(); ++pos)
{
    cout << "["<< *pos << "]" "; }
cout << endl;

list<iterator_range<string::iterator> > l;
split(l, str, is_any_of(",.:-+")); //使用标点分割
for (BOOST_AUTO(pos, l.begin()); pos !=l.end(); ++pos)
{
    cout << "["<< *pos << "]" "; }
cout << endl;

l.clear();
split(l, str, is_any_of(",.:-"), token_compress_on);
for (BOOST_AUTO(pos, l.begin()); pos !=l.end(); ++pos)
{
    cout << "["<< *pos << "]" "; }
cout << endl;
}

```

程序首先示范了 `find_all` 算法，使用 `deque<string>` 接收忽略大小写查找到的所有字符串。接着我们使用 `split` 算法，使用 `",.:-+"` 中的任意字符分割字符串，由于原串中有两个 `::`，因此分割的结果会有一个空串。最后的 `split` 算法使用了 `token_compress_on` 参数，并缩小了分隔符的取值范围。

程序的运行结果如下：

```

[Zelda] [zelda]
[Samus][Link][Zelda][][Mario][Luigi][zelda]
[Samus,Link][Zelda][Mario][Luigi+zelda]

```

5.3.11 合并

合并算法 `join` 是分割算法的逆运算，它把存储在容器中的字符串连接成一个新的字符串，并且可以指定连接的分隔符。

合并算法的声明如下：

```

template<typename SequenceSequenceT, typename RangeIT>
range_value< SequenceSequenceT >::type
join(const SequenceSequenceT & Input, const RangeIT & Separator);

```


`join` 还有一个后缀 `_if` 的版本，它接受一个判断式，只有满足判断式的字符串才能参与合并。

示范合并算法用法的代码如下：

```

#include <boost/assign.hpp>

```



```

#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    using namespace boost::assign;
    vector<string> v = list_of("Samus")("Link")("Zelda")("Mario");
    cout << join(v, "+") << endl;           //合并

    //定义一个简单的函数对象
    struct is_contains_a
    {
        bool operator()(const string &x)
        {    return contains(x, "a");}
    };
    cout << join_if(v, "**", is_contains_a());    //合并
}

```

程序首先使用 `assign` 库向 `vector` 添加了四个字符串，然后以“+”合并它们。随后定义了一个简单的函数对象，它包装了算法 `contains`，判断字符串是否包含字符 `a`，最后把这个函数对象作为参数传递给 `join_if` 算法。

程序运行结果如下：

```

Samus+Link+Zelda+Mario
Samus**Zelda**Mario

```

5.3.12 查找（分割）迭代器

在通用的 `find_all` 或 `split` 外，`string_algo` 库还提供两个查找迭代器 `find_iterator` 和 `split_iterator`，它们可以在字符串中像迭代器那样遍历匹配，进行查找或者分割，无需使用容器来容纳。

使用查找迭代器，可以实现某些简单算法无法实现的功能。示范它们用法的代码如下：

```

#include <boost/algorithm/string.hpp>
using namespace boost;
int main()
{
    string str("Samus||samus||mario||||Link");

    typedef find_iterator<string::iterator> string_find_iterator;
    //查找迭代器类型定义

    string_find_iterator pos, end;           //声明查找迭代器变量

    for (pos = make_find_iterator(str, first_finder("samus", is_iequal()));

```

```

        pos != end; ++pos)
    {
        cout << "[" << *pos << "]" ;    }
    cout << endl;

    typedef split_iterator<string::iterator> string_split_iterator;
                                                //分割迭代器类型定义

    string_split_iterator p, endp;                //声明分割迭代器变量
    for (p = make_split_iterator(str, first_finder("||", is_iequal()));
        p != endp; ++p)
    {
        cout << "[" << *p << "]" ;    }
    cout << endl;
}

```

使用查找迭代器首先要声明迭代器对象 `find_iterator` 或 `split_iterator`，它们的模板类型参数是一个迭代器类型 `a`，例如 `string::iterator` 或者 `char*`。

为了获得迭代器的起始位置，我们需要调用 `first_finder()` 函数，它用于判断匹配的对象，然后再用 `make_find_iterator` 或 `make_split_iterator` 来真正创建迭代器。同族的查找函数还有 `last_finder`、`nth_finder`、`token_finder` 等，它们的含义与查找算法类似，从不同的位置开始查找返回迭代器。

初始化工作完成后，我们就可以像使用标准迭代器或者指针那样，不断地遍历迭代器对象，使用解引用操作符获取查找的内容，直到找不到匹配的对象。

这里要特别注意一下分割迭代器的运用，它可以以任意长度的字符串作为分隔符进行分割，而普通的 `split` 算法则只能以字符作为分隔符。

查找（分割）迭代器的用法与下面的 `tokenizer` 库用法很接近，故本书不再详述，请读者参考 `tokenizer` 库。

5.4 tokenizer

`tokenizer` 库是一个专门用于分词（token）的字符串处理库，可以使用简单易用的方法把一个字符串分解成若干个单词。它与 `string_algo` 库的分割算法很类似，但有更多的变化。

`tokenizer` 位于名字空间 `boost`，为了使用 `tokenzier` 组件，需要包含头文件 `<boost/tokenizer.hpp>`，即：

```

#include <boost/tokenizer.hpp>
using namespace boost;

```

5.4.1 类摘要

`tokenizer` 类是 `tokenizer` 库的核心，它以容器的外观提供分词的序列，类摘要如下：

```
template <
    typename TokenizerFunc = char_delimiters_separator<char>,
    typename Iterator = std::string::const_iterator,
    typename Type = std::string
>
class tokenizer
{
    tokenizer(Iterator first, Iterator last, const TokenizerFunc& f);
    tokenizer(const Container& c, const TokenizerFunc& f);

    void assign(Iterator first, Iterator last);
    void assign(const Container& c);
    void assign(const Container& c, const TokenizerFunc& f)

    iterator begin() const ;
    iterator end() const;
}
```

`tokenizer` 接受三个模板类型参数，分别是：

- `TokenizerFunc` : `tokenizer` 库专门的分词函数对象，默认是使用空格和标点分词；
- `Iterator` : 字符序列的迭代器类型；
- `Type` : 保存分词结果的类型。

这三个模板类型都提供了默认值，但通常只有前两个模板参数可以变化，第三个类型一般只能选择 `std::string` 或者 `std::wstring`，这也是它位于模板参数列表最后的原因。

`tokenizer` 的构造函数接受要进行分词的字符串，可以以迭代器的区间形式给出，也可以是一个有 `begin()` 和 `end()` 成员函数的容器。

`assign()` 函数可以重新指定要分词的字符串，用于再利用 `tokenizer`。

`tokenizer` 具有类似标准容器的接口，`begin()` 函数使 `tokenizer` 开始执行分词功能，返回第一个分词的迭代器，`end()` 函数表明迭代器已经到达分词序列的末尾，分词结束。

5.4.2 用法

`tokenizer` 的用法很像 `string_algo` 的分割迭代器，但要简单一些。可以像使用一个容器那样使用它，向 `tokenizer` 传入一个欲分词的字符串构造，然后用 `begin()` 获得迭代器反复迭代，就可以轻松完成分词功能。

由于 tokenizer 具有类似容器的接口，所以可以使用 BOOST_AUTO。

示范 tokenizer 用法的代码如下：

```
#include <boost/tokenizer.hpp>
using namespace boost;
int main()
{
    string str("Link raise the master-sword.");

    tokenizer<> tok(str);                                //使用缺省模板参数创建分词对象

    //可以像遍历一个容器一样使用 for 循环获得分词结果
    for(BOOST_AUTO(pos, tok.begin()); pos != tok.end(); ++pos)
    {
        cout << "[" << *pos << " ";
    }
}
```

程序运行结果如下：

```
[Link] [raise] [the] [master] [sword]
```

注意一下这个分词结果，tokenizer 默认把所有的空格和标点符号作为分隔符，因此分割出的只是单词，这与 `string_algo::split` 的算法含义（分割字符串）有所差别。

5.4.3 分词函数对象

tokenizer 作为分词的容器本身的用法很简单，它的真正威力在于第一个模板类型参数 `TokenizerFunc`。`TokenizerFunc` 是一个函数对象，它决定如何进行分词处理。`TokenizerFunc` 同时也是一个规范，只要具有合适的 `operator()` 和 `reset()` 语言的函数对象都可以用于 tokenizer 分词。

tokenizer 库提供预定义好的四个分词对象，它们是：

- `char_delimiters_separator`: 使用标点符号分词，是 tokenizer 默认使用的分词函数对象。但它已经被声明废弃，应当尽量不使用它；
- `char_separator`: 它支持一个字符集作为分隔符，默认的行为与 `char_delimiters_separator` 类似；
- `escaped_list_separator`: 用于 CSV 格式（逗号分隔）的分词；
- `offset_separator`: 使用偏移量来分词，在分解平文件格式的字符串时很有用。

下面我们分别来介绍后三个分词函数对象。为了简化程序输出代码，接下来的代码使用了一个辅助模板函数 `print()`，它遍历输出分词的结果，如下：

```
template<typename T>
void print(T &tok)
{
    for (BOOST_AUTO(pos, tok.begin()); pos != tok.end(); ++pos)
    {
        cout << "[" << *pos << " ";
    }
    cout << endl;
}
```

5.4.4 char_separator

char_separator 使用一个字符集作为分词依据, 行为很类似 split 算法(参见 5.3.10 小节), 它的构造函数声明是:

```
char_separator(const Char* dropped_delims,
               const Char* kept_delims = 0,
               empty_token_policy empty_tokens = drop_empty_tokens);
```

构造函数中的参数含义如下:

- 第一个参数 dropped_delims 是分隔符集合, 这个集合中的字符不会作为分词的结果出现;
- 第二个参数 kept_delims 也是分隔符集合, 但其中的字符会保留在分词结果中;
- 第三个参数 empty_tokens 类似 split 算法的 eCompress 参数, 处理两个连续出现的分隔符。如为 keep_empty_tokens 则表示连续出现的分隔符标识了一个空字符串, 相当于 split 算法的 token_compress_off 值; 如为 drop_empty_tokens, 则空白单词不会作为分词的结果。

如果使用默认的构造函数, 不传入任何参数, 则其行为等同于 char_separator(" ", 标点符号字符, drop_empty_tokens), 以空格和标点符号分词, 保留标点符号, 不输出空白单词。

示范 char_separator 用法的代码如下:

```
char *str = "Link ;; <master-sword> zelda";

char_separator<char> sep; //一个 char_separator 对象
tokenizer<char_separator<char>, char*> tok(str, str + strlen(str), sep);
//传入 char_separator 构造分词对象
print(tok); //分词并输出

tok.assign(str, str + strlen(str), //重新分词
           char_separator<char>(" ;->", "<>"));
print(tok);

tok.assign(str, str + strlen(str), //重新分词
           char_separator<char>(" ;-><>", "", keep_empty_tokens));
print(tok);
```


这段代码我们对 tokenizer 的模板参数稍微做了一下改变，第二个参数改为 `char*`，这使得 tokenizer 可以分析 C 风格的字符串数组。同时构造函数也必须变为传入字符串的首末位置，不能仅传递一个字符串首地址，因为字符串数组不符合容器的概念。

第一次分词我们使用 `char_separator` 的缺省构造，以空格和标点分词，保留标点作为单词的一部分，并抛弃空白单词；第二次分词我们使用 `" ; -"` 和 `"<>"` 共 5 个字符分词，保留 `<>` 作为单词的一部分，同样抛弃空白单词；最后一次分词我们同样使用 `" ; -<>"` 分词，但它们都不作为单词的一部分，并且我们保留空白单词。

程序运行结果如下：

```
[Link][;][;][<][master][-][sword][>][zelda]
[Link][<][master][sword][>][zelda]
[Link][][][][][master][sword][][zelda]
```

5.4.5 escaped_list_separator

`escaped_list_separator` 是专门处理 CSV 格式 (Comma Split Value, 逗号分隔值) 的分词对象，它的构造函数声明是：

```
escaped_list_separator(Char e = '\\', Char c = ',', Char q = '\"')
```

`escaped_list_separator` 的构造函数参数一般都取默认值，含义如下：

- 第一个参数 `e` 指定了字符串中的转义字符，默认是斜杠 `\`；
- 第二个参数是分隔符，默认是逗号；
- 第三个参数是引号字符，默认是 `"`；

示范 `escaped_list_separator` 用法的代码如下：

```
string str = "id,100,name,\"mario\"";

escaped_list_separator<char> sep;
tokenizer<escaped_list_separator<char>> > tok(str, sep);
print(tok);
```

程序运行结果如下：

```
[id][100][name][mario]
```

5.4.6 offset_separator

`offset_separator` 与前两种分词函数对象不同，它分词的功能不基于查找分隔符，而是

使用偏移量的概念，在处理某些不使用分隔符而使用固定字段宽度的文本时很有用。它的构造函数声明如下：

```
template <typename Iter>
offset_separator(Iter begin, Iter end, bool wrap_offsets = true, bool return_
    partial_last = true);
```

`offset_separator` 的构造函数接收两个迭代器参数（也可以是数组指针）`begin` 和 `end`，指定分词用的整数偏移量序列，整数序列的每个元素是分词字段的宽度。

`bool` 参数 `bwrapoffsets`，决定是否在偏移量用完后继续分词。`bool` 参数 `return_partial_last` 决定在偏移量序列最后是否返回分词不足的部分。这两个附加参数的默认值都是 `true`。

示范 `offset_separator` 用法的代码如下：

```
string str = "2233344445";
int offsets[] = {2,3,4};
offset_separator sep(offsets, offsets + 3, true, false);
tokenizer<offset_separator> tok(str, sep);
print(tok);

tok.assign(str, offset_separator(offsets, offsets + 3, false));
print(tok);

str += "56667";
tok.assign(str, offset_separator(offsets, offsets + 3, true, false));
print(tok);
```

代码中我们用一个数组 `offsets` 指定了 3 个偏移量，要求分割出 3 个长度分别为 2、3、4 的单词。程序运行结果如下：

```
[22][333][4444][5]
[22][333][4444]
[22][333][4444][55][666]
```

请读者注意输出的第一行和最后一行，两者的分词都设置了 `return_partial_last` 为 `false`，但输出却略有不同。

这是因为 `return_partial_last` 只影响偏移量序列的最后一个元素（即代码中的 `offsets[2]=4`），对于序列中的其他元素则不起作用，无论是否分词不足均输出。所以输出中第一行的 `[5]` 因为它对应偏移量序列的第一个元素，所以总能输出，而最后一行字符串末尾原本应该有输出 `[7]`，但因为 `return_partial_last` 为 `false` 且它对应偏移量序列的最后一个元素，所以不被输出。

5.4.7 tokenizer 库的缺陷

tokenizer 库可以很容易地执行分词操作，但它存在一些固有的缺陷。

缺陷之一是它只支持使用单个字符进行分词，如果要分解如“||”等多个字符组成的分隔符则无能为力，只能自己定义分词函数对象，或者使用 `string_algo`、正则表达式等其他字符串功能库。

它的另一个小的（但很麻烦的）缺陷是对 `wstring` (unicode) 缺乏完善的考虑，也没有像 `string_algo` 那样使用 `std::locale()`，不方便使用。

例如，如果我们使用 `wstring`，`string_algo` 库可以很简单地分词：

```
wstring str(L"Link mario samus");

typedef split_iterator<wstring::iterator> string_split_iterator;

string_split_iterator p, endp;
for (p = make_split_iterator(str, first_finder(L" ", is_iequal()));
     p != endp; ++p)
{
    wcout << L "[" << *p << L "]" ; }
}
```

为了使用 `wstring`，我们需要把字符串类型改为 `wstring`，字符串常量用 `L` 标记是宽字符，再使用 `wcout` 输出。处理字符串用的 `string_algo` 算法只需要变动 `split_iterator` 的一处模板参数就可以了。

而使用 `tokenizer`，除了以上的操作，我们必须完整无误地写出它的全部模板参数，像这样：

```
char_separator<wchar_t> sep(L" ");
tokenizer<char_separator<wchar_t>,wstring::const_iterator,
        wstring > tok(str, sep);
for(BOOST_AUTO(pos, tok.begin()); pos != tok.end(); ++pos)
{
    wcout << L "[" << *pos << L "]" ; }
}
```

使用 `escaped_list_separator` 等其他分词函数对象也是如此，如：

```
escaped_list_separator<wchar_t> sep;
tokenizer<escaped_list_separator<wchar_t>,wstring::const_iterator, wstring >
    tok(str, sep);
```

这显得过于烦琐。如果我们将来某一天必须要支持 `wstring`，对这些代码的修改和维护将会是个不小的工作量。

针对这个问题，本书提供了一个包装类，它能够部分解决这个问题：

```
template<typename Func, typename String = std::string>
```

```
struct tokenizer_wrapper
{
//typedef typename Func::string_type String;
typedef tokenizer<Func, typename String::const_iterator, String > type;
};
```

tokenizer_wrapper 有两个模板参数，第一个 Func 是分词函数对象，第二个是分词所使用的字符串类型，内部用这两个模板类型 typedef 简化了 tokenizer 的模板声明。使用这个包装类，上面的声明可以简化为：

```
tokenizer_wrapper<char_separator<wchar_t>,wstring >::type tok(str, sep);
tokenizer_wrapper<escaped_list_separator<wchar_t>,wstring >::type
    tok(str, sep);
```

这样看起来就好多了。

请读者注意 tokenizer_wrapper 类内部的那行注释，很遗憾 tokenizer 库的分词函数对象均把字符串类型作为它内部的 typedef，不能被外界使用，并且 offset_separator 不提供这个 typedef，否则包装类可以节省一个模板参数，像这样：

```
template<typename Func >
struct tokenizer_wrapper
{
    //不能通过编译！string_type 是私有 typedef
    typedef typename Func::string_type String;
    typedef tokenizer<Func, typename String::const_iterator, String > type;
};
```

但现实不允许我们这么做，tokenizer 库设计之初就没有对这些问题做很好的考虑，除非改动源代码。但本书不建议修改，因为正则表达式和 string_algo 通常都比 tokenizer 工作得更好。

5.5 xpressive

正则表达式是处理文本强有力的工具，它使用一套复杂的语法规则，能够解决文本处理领域的绝大多数问题，诸如验证、匹配、查找、替换等等，而这些问题用通常的字符串算法是很难甚至无法解决的。

许多编程语言都提供了内置的正则表达式支持，如 Perl、Python、Java，但 C++98 标准中没有，这使得 C++ 程序员失去了一个得力的助手，不得不将文本处理工作转交给其他语言。

xpressive 是一个先进的、灵活的、功能强大的正则表达式库，提供了对正则表达式的全面

支持，而且比原正则表达式库 `boost.regex` 要好的是它不需要编译，速度快，同时语法又很类似，大大降低了学习的难度。

5.5.1 两种使用方式

`xpressive` 不仅是一个类似 `boost.regex` 的正则表达式解析器，同时它还是一个类似于 `boost.spirit` 的语法分析器，并且将这两种完全不相交的文本处理方式完美地融合在了一起。

因此 `xpressive` 提供动态和静态两种使用方式。静态方式类似于 `spirit`，使用操作符重载生成编译期的表达式对象，可以在编译期进行正则表达式的语法检查。动态方式则是较传统的用法，与 `boost.regex` 或 Python 中的 `re` 模块很相似，以字符串作为一个表达式对象，在运行时进行语法检查和处理。

这两种方式不是对立的，因为 `xpressive` 把它们融合成了一个整体，两者可以在程序中以任意的形式组合混用。

- 如果想混用两种方式或者不关心这两种方式，可以包含头文件 `<boost/xpressive/xpressive.hpp>`。
- 如果仅想使用静态方式，可以只包含头文件 `<boost/xpressive/xpressive_static.hpp>`。
- 如果仅想使用动态方式，可以只包含头文件 `<boost/xpressive/xpressive_dynamic.hpp>`。

`xpressive` 的静态方式使用了大量的操作符重载，相当于一套全新的语法，理解上有一定难度。而动态用法很接近原始正则表达式、`boost.regex` 和 C++0x TR1 草案，熟悉了动态用法将对以后的学习有很大的辅助作用，因此本书接下来的内容将重点介绍动态用法。

`xpressive` 库位于名字空间 `boost::xpressive`，为了使用 `xpressive` 的动态用法，需要包含头文件 `<boost/xpressive/xpressive_dynamic.hpp>`，即：

```
#include <boost/xpressive/xpressive_dynamic.hpp>
using namespace boost::xpressive;
```

5.5.2 正则表达式语法简介

作为 `xpressive` 库的学习准备，本小节介绍一些正则表达式的基本语法知识。这些知识没有完全覆盖所有正则表达式的内容，但用于本书已经足够了。如果读者已经对正则表达式很熟悉，那么可以跳过；想了解更多有关正则表达式的使用方法可以参阅其他相关书籍。

正则表达式定义了一套完善而复杂的语法规则，用于匹配有特定模式的字符串。在正则表达式中，大部分字符都匹配自己（即普通字符），只有少量的字符被用于定义特殊的匹配模式语法，它们是：`.^$()*+?{}[]\|`。

1. 点号（`.`）可以匹配任意的单个字符，是单字符的通配符。

2. `^`匹配行的开头。

3. `$`匹配行的末尾。

4. `()`用于定义一个正则表达式匹配子元素（子表达式），可以被引用或者重复。

5. `*`表示前面的元素可以重复任意多次（ $n \geq 0$ ）。

6. `+`表示前面的元素可以重复一次或多次（ $n > 0$ ）。

7. `?`表示前面的元素可以重复 0 次或 1 次（ $n = 0, 1$ ）。

8. `{}`可以手工指定元素重复的次数。`{n}`重复 $x=n$ 次，`{n,}`重复 $x \geq n$ 次，`{n,m}`重复 n 次到 m 次之间的次数，即 $n \leq x \leq m$ 。

9. `[]`用于定义字符集合，可以列出单个字符，也可以定义范围，或者是集合的补集。

10. `\`是转义字符（与 C/C++ 类似），特殊字符经转义后与自身匹配。

11. `|`表示逻辑或的概念，匹配它两侧的元素之一。

下面我们用几个例子来理解正则表达式。

例 1 “`a.`”表示字符 `a` 和任意一个字符构成的字符串，如 `ab`、`a0`，但不是 `abb`。

例 2 “`(a*)(b|c+)`”用括号定义了两个子表达式，表示任意数量的 `a`，然后接一个 `b` 或者一个以上的 `c`，如 `aaab`、`aaccc`，但不是 `bcc`、`abccc`。

例 3 “`[abc]?.\{\}`”表示 `abc` 三个字符之一可以出现 0 次或一次，然后是一个任意字符，再接一个 `{}`，如 `a={}`、`c2{}`，但不是 `ua{}`、`ab`。

正则表达式还使用转义字符斜杠预定义了一些匹配概念，如 `\d` 匹配数字（相当于 `[0-9]`），`\w` 匹配字母（相当于 `[a-z]`）、`\s` 匹配空格等。

使用以上的基本规则，再进行组合，就可以定义复杂的字符串正则表达式匹配模式。但需要注意一点：

由于 C++ 没有如 C# 和 Python 那样提供“原始字符串”的表达方式^①，而正则表达式里会经常使用斜杠，在 C++ 代码中就要变成双斜杠，无论是写还是阅读都会有不小的麻烦。本书建议在使用正则表达式前可先在普通的文本编辑器（UltraEdit、Editplus、Notepad++ 均可）里写好，然后用替换功能把所有的单斜杠替换为双斜杠。在代码中使用正则表达式时，一定要在语句前用注释说明其原始表达式，以方便将来的调试和维护。

接下来我们将使用 xpressive 来体验正则表达式的强大威力。

5.5.3 类摘要

本小节介绍 xpressive 库提供的 3 个重要的类，分别是：basic_regex、match_results 和 sub_match。

basic_regex

模板类 basic_regex 是 xpressive 库的核心，它封装了正则表达式的解析和编译，它的类摘要如下：

```
template<typename BidIter>
struct basic_regex
{
    basic_regex();
    basic_regex(basic_regex< BidIter > const &);

    regex_id_type regex_id() const;
    std::size_t mark_count() const;
    void swap(basic_regex< BidIter > &) ;

    static basic_regex< BidIter >
    compile(InputRange const & pat) ;
};

typedef basic_regex<std::string::const_iterator> sregex;
typedef basic_regex<char const *> cregex;
```

basic_regex 是正则表达式的的基本类，它常用的是两个 typedef：sregex 和 cregex。sregex 用于操作标准字符串类 std::string，cregex 用于操作字符数组（C 风格字符串）。

basic_regex 的构造函数很简单，默认构造一个空的正则表达式，支持从另一个 basic_regex 拷贝构造和赋值。成员函数 regex_id() 返回正则表达式的唯一标志，mark_count() 可以返回表达式中匹配的子表达式的个数，对于空的正则表达式对象，这两个函

① C# 和 Python 都可以表示字符串的“原始形态”，C# 用前缀 @，Python 用 r。

数都会返回 0。

`basic_regex` 的核心功能是静态成员函数 `compile()`，它是一个工厂方法，可以根据正则表达式参数创建出一个 `basic_regex` 对象。`compile()` 有多个重载形式，最常用的是传入一个字符串，它必须是合法有效的正则表达式，否则会运行时发生错误。

match_results

`match_results` 是 `xpressive` 库里另一个重要的类，保存了正则表达式匹配的结果，它的类摘要如下：

```
template<typename BidiIter>
struct match_results
{
    size_type size() const;
    bool empty() const;
    template<typename Sub>
    const_reference operator[](Sub const & i) const;
};
typedef match_results<std::string::const_iterator> smatch;
typedef match_results<char const *> cmatch;
```

`match_results` 为正则表达式的匹配结果提供一个类似容器的视图，可以用 `size()` 和 `empty()` 判断匹配结果中子表达式的数量，`operator[]` 返回第 `i` 个子表达式。如果 `i==0`，则返回整个表达式的匹配对象。

同样，为了支持不同的字符串类型，`match_results` 有两个方便的 `typedef`，分别是 `smatch` 和 `cmatch`，用于支持 `std::string` 和字符数组，它们的命名风格与 `sregex`、`cregex` 是相同的，使用同样的前缀。

sub_match

最后一个模板类 `sub_match` 是一个类似迭代器对的对象，继承自 `std::pair`，可以把它当作一个字符串的区间表示，它的类摘要如下：

```
template<typename BidiIter>
struct sub_match : public std::pair< BidiIter, BidiIter >
{
    string_type str() const;
    difference_type length() const;
    bool operator!() const;
    int compare(string_type const &) const;
    bool matched;
};
```


5.5.4 匹配

从最简单最基础的正则表达式匹配开始学习 xpressive 的用法

自由函数 `regex_match()` 用来检查一个字符串是否完全匹配一个正则表达式，返回一个 `bool` 结果。它有很多重载形式以支持各种应用情况，基本的声明大概是这样：

```
bool regex_match(String, basic_regex const & re);
bool regex_match(String, match_results& what, basic_regex const & re);
```

`regex_match()` 的最简单用法接受两个参数，它的第一个参数是要被匹配检查的字符串，第二个参数是正则表达式对象（`sregex` 或 `cregex`）。

`regex_match()` 的第二种重载形式多了一个 `match_results` 输出参数，可以返回查找到的字符串。

示范字符串匹配的简单用法的代码如下：

```
#include <boost/xpressive/xpressive_dynamic.hpp>
int main()
{
    using namespace boost::xpressive;          //打开名字空间

    cregex reg = cregex::compile("a.c");       //C 字符串正则表达式对象
    assert(regex_match("abc", reg));           //匹配
    assert(regex_match("a+c", reg));           //匹配

    assert(!regex_match("ac", reg));           //不匹配
    assert(!regex_match("abd", reg));          //不匹配
}
```

我们来仔细研究一下这第一个正则表达式程序。

字符串 `"a.c"` 定义了一个正则表达式，它有三个字符，两端是 `a` 和 `c`，中间可以是任意字符。接下来的

```
cregex reg = cregex::compile("a.c");
```

使用 `cregex` 类的工厂方法 `compile()` 创建了一个正则表达式对象 `reg`，它可以对字符数组进行正则表达式匹配。

这之后是四个 `assert()` 宏，使用 `regex_match()` 检验四个字符串：`"abc"`、`"a+c"`、`"ac"` 和 `"abd"`。前两个匹配模式，返回 `true`，而后两个不匹配，返回 `false`。

看一个比较复杂的正则表达式应用——匹配身份证号码

首先要了解身份证的基本编码规则（为简单起见，我们只考虑新式身份证），它是由 18 位数字组成的，前 6 位是地区编码，中间 8 位是年月日，最后 4 位数字可能有一个是 x。

我们使用 `\d` 来表示数字，那么前 6 位的正则表达式是：`\d{6}`，后 4 位的正则表达式是：`\d{3}(X|\d)`。

中间的年月日的检验要稍微复杂一些，为了演示做适当的简化，不检查闰年和月份天数的有效性，因此可以写成：`(1|2)\d{3}(0|1)\d[0-3]\d`。其中的 `(1|2)\d{3}` 检查 4 位的年份，第一个数字必须是 1 或 2，`(0|1)\d` 检查月份，`[0-3]\d` 检查天。

把这些组合起来我们得到：`\d{6}(1|2)\d{3}(0|1)\d[0-3]\d\d{3}(X|\d)`，再转换成 C 字符串就是：`"\\d{6}(1|2)\\d{3}(0|1)\\d[0-3]\\d\\d{3}(X|\\d)"`。

示范身份证验证的正则表达式使用的代码如下：

```
#include <boost/xpressive/xpressive_dynamic.hpp>
int main()
{
    using namespace boost::xpressive;

    cregex reg = cregex::compile("\\d{6}(1|2)\\d{3}(0|1)\\d[0-3]\\d\\d{3}(X|\\d)", icafe);
    assert(regex_match("999555197001019999", reg));
    assert(regex_match("99955519700101999X", reg));
    assert(regex_match("99955520100101999x", reg));

    assert(!regex_match("99955520100101999Z", reg));
    assert(!regex_match("99955530100101999X", reg));
    assert(!regex_match("999555201099019998", reg));
    assert(!regex_match("999555201012419991", reg));
}
```

这段代码中 `cregex` 的创建略有不同，除了正则表达式字符串较长外，它还使用了一个 `icafe` 标志，用于指示匹配时忽略大小写。此外，整个程序与第一个没有什么不同。

我们把这个程序稍微修改一下，增加子表达式，使用 `match_results` 来提取匹配结果中的年月日信息。

首先是修改正则表达式，改为：

```
"\\d{6}((1|2)\\d{3})((0|1)\\d)(([0-3]\\d)(\\d{3}(X|\\d)))"
```

这里我们增加了四对括号，分别定义了四个子表达式，它们是：`((1|2)\\d{3})`、`((0|1)\\d)`、

([0-3]\\d) 和 (\\d{3}(X|\\d))。注意，子表达式里的括号也仍然是一个子表达式，因此这个正则表达式里共有七个子表达式，按照出现的顺序从 1 至 7 编号。

示范代码如下：

```
#include <boost/xpressive/xpressive_dynamic.hpp>
int main()
{
    using namespace boost::xpressive;

    cregex reg = cregex::compile("\\d{6}((1|2)\\d{3})((0|1)\\d)([0-3]\\d)(\\d{3}(X|\\d))", icase);
    cmatch what; // 匹配结果对象
    assert(regex_match("999555197001019999", what, reg));
    for (BOOST_AUTO(pos, what.begin()); pos != what.end(); ++pos)
    {
        cout << "[" << *pos << " ";
    }
    cout << endl;

    cout << "date:" << what[1] << what[3] << what[5] << endl;
}
```

程序运行结果如下：

```
[999555197001019999] [1970] [1] [01] [0] [01] [9999] [9]
date:19700101
```

regex_match() 还可以替代部分 string_algo 库的判断式算法的功能，例如下面的代码使用 sregex 实现了 starts_with 和 ends_with 算法：

```
string str("readme.txt");

sregex start_reg = sregex::compile("^re.*"); // 匹配开头
sregex end_reg = sregex::compile(".*txt$"); // 匹配末尾

assert(regex_match(str, start_reg));
assert(regex_match(str, end_reg));
```

5.5.5 查找

为了使用正则表达式的查找功能，我们要用到另外一个自由函数 regex_search()。

它与 regex_match() 的区别是：

regex_match() 要求输入字符串必须要与正则表达式完全匹配，而 regex_search() 则检测输入表达式中是否包含正则表达式，即存在一个匹配正则表达式的子串。

因此 regex_search() 比 regex_match() 使用起来更加灵活。

`regex_search()` 与 `regex_match()` 相似，也有很多重载形式以支持各种应用情况，基本的声明大概是这样：

```
bool regex_search(String, basic_regex const & re);
bool regex_search(String, match_results& what, basic_regex const & re);
```

`regex_search()` 的调用形式与 `regex_match()` 完全相同，有一点除外，即它不要求完全匹配，只要找到有匹配的子串就返回 `true`。

示范了 `regex_search()` 用法的代码如下，其中，使用了正则表达式 “(power)-(.{4})” 来搜索 power+连字符+4 个任意字符：

```
#include <boost/xpressive/xpressive_dynamic.hpp>
int main()
{
    using namespace boost::xpressive;

    char* str = "there is a POWER-suit item";
    regex reg =cregex::compile("(power)-(.{4})", icase);

    assert(regex_search(str, reg)); //可搜索到字符串

    cmatch what;
    regex_search(str, what, reg); //保存搜索结果
    assert(what.size() == 3); //共 3 个子表达式

    cout << what[1] << what[2] << endl;
    assert(!regex_search("error message", reg));
}
```

`regex_search()` 的用法很简单，如果读者已经掌握了 `regex_match()`，那么将很快学会 `regex_search()`。

`regex_search()` 可以替代 `string_algo` 的 `contains`、`starts_with`、`ends_with` 和查找算法，例如：

```
string str("readme.TXT");

sregex start_reg = sregex::compile("^re");
sregex end_reg = sregex::compile("txt$", icase);

assert(regex_search(str, start_reg)); //starts_with
assert(regex_search(str, end_reg)); //ends_with
assert(regex_search(str, cregex::compile("me"))); //contains
```

5.5.6 替换

`xpressive` 的替换功能使用的函数是 `regex_replace()`，它先使用正则表达式查找匹配

的字符串，然后再用指定的格式替换，基本声明如下：

```
String regex_replace(String, basic_regex const & re, Format);
```

前两个参数与 `regex_match()`、`regex_search()` 相同，第三个参数 `Format` 可以是一个简单字符串，也可以是一个符合 ECMA-262 定义的带格式的字符串，它可以用 `$N` 引用正则表达式匹配的子表达式，`$&` 引用全匹配。在替换完成后，`regex_replace()` 返回一个字符串的拷贝。

示范 `regex_replace()` 用法的代码如下：

```
#include <boost/xpressive/xpressive_dynamic.hpp>
int main()
{
    using namespace boost::xpressive;

    string str("readme.txt");

    sregex reg1 = sregex::compile("(.) (me)");
    sregex reg2 = sregex::compile("(t) (.) (t)");

    cout << regex_replace(str, reg1, "manual") << endl;
    cout << regex_replace(str, reg1, "$1you") << endl;
    cout << regex_replace(str, reg1, "$&$&") << endl;
    cout << regex_replace(str, reg2, "$1N$3") << endl;

    str = regex_replace(str, reg2, "$1$3");
    cout << str << endl;
}
```

程序中定义了两个正则表达式对象，为了方便进行 ECMA-262 格式替换，使用括号定义了子表达式。运行结果如下：

```
manual.txt
readyou.txt
readmereadme.txt
readme.tNt
readme.tt
```

`regex_replace()` 可以替代 `string_algo` 库中的修剪和删除算法，例如：

```
string str("2010 Happy new Year!!!");

sregex reg1 = sregex::compile("^(\\d| )*"); //查找开头的数字和空格
sregex reg2 = sregex::compile("!*$");      //查找末尾的标点

cout << regex_replace(str, reg1, "") << endl; //trim_left
cout << regex_replace(str, reg2, "") << endl; //trim_right
```

```
str = regex_replace(str, reg1, "Y2000 ");           //replace_all
cout << str << endl;
```

5.5.7 迭代

xpressive 库提供一个强大的迭代器模板类 `regex_iterator<>`，它类似 `string_algo` 的查找迭代器和 `tokenizer`，提供一个类似迭代器的视图（不是容器！），可以遍历正则表达式的匹配结果。在具体应用时我们应当使用它的 `typedef`，如 `sregex_iterator` 和 `cregex_iterator`。

`regex_iterator<>` 的类摘要如下：

```
template<typename BidIter>
struct regex_iterator
{
    typedef match_results< BidIter > value_type;
    regex_iterator(BidIter, BidIter, basic_regex const &);
    value_type const & operator*() const;
    value_type const * operator->() const;
    regex_iterator< BidIter > & operator++() ;
    regex_iterator< BidIter > operator++(int) ;
}
```

`regex_iterator<>` 的构造函数完成迭代初始化工作，要求传入进行分析的容器区间和正则表达式对象，之后就可以对它反复调用 `operator++`，使用 `*` 或者 `->` 获取匹配的结果 `match_results` 对象。

`regex_iterator<>` 可以替代 `string_algo` 库的查找算法和查找迭代器。

示范 `regex_iterator<>` 用法的代码如下：

```
#include <boost/xpressive/xpressive_dynamic.hpp>
int main()
{
    using namespace boost::xpressive;

    string str("Power-bomb, power-suit, pOWER-beam all items\n");

    sregex reg = sregex::compile("power-(\\w{4})", icase);

    regex_iterator pos(str.begin(), str.end(), reg);
    sregex_iterator end;
    while(pos != end)
    {
        cout << "[" << (*pos)[0] << "]\n";
        ++pos;
    }
```

```

    }
    cout << endl;
}

```

代码中需要注意的是迭代器的解引用操作, `operator*` 返回一个 `match_results` 对象, 因为操作符优先级的原因, 我们必须用括号把 `operator*` 括起来, 然后才能使用 `operator[]`。

程序运行结果如下:

```
[Power-bomb] [power-suit] [pOWER-beam]
```

5.5.8 分词

`xpressive` 库使用模板类 `regex_token_iterator<>` 提供了强大的分词迭代器, 要比 `string_algo` 和 `tokenizer` 的分词能力强大、灵活得多, 读者一旦掌握了它, 就会深深地为它的功能所折服。

`regex_token_iterator<>` 的类摘要如下:

```

template<typename BidIter>
struct regex_token_iterator
{
    typedef sub_match< BidIter > value_type;
    regex_token_iterator(BidIter, BidIter, basic_regex const &,
                        match_flag_type args);
    value_type const & operator*() const;
    value_type const * operator->() const;
    regex_token_iterator< BidIter > & operator++();
    regex_token_iterator< BidIter > operator++(int);
}

```

`regex_token_iterator<>` 同样不能直接使用, 你需要用它的两个 typedef: `sregex_token_iterator` 和 `cregex_token_iterator`。

`regex_token_iterator<>` 的用法大致与 `regex_iterator<>` 相同, 但它的变化在于匹配对象的使用方式。

默认情况下 `regex_token_iterator<>` 同 `regex_iterator<>`, 返回匹配的字符串。如果构造时传入 -1 作为最后一个 `args` 参数的值, 它将把匹配的字符串视为分隔符。如果 `args` 是一个正数, 则返回匹配结果的第 `args` 个字符串。

还有一点与 `regex_iterator<>` 不同的是它解引用返回的是一个 `sub_match` 对象, 而不是 `match_results` 对象。

示范 `regex_token_iterator<>` 用法的代码如下:

```

#include <boost/xpressive/xpressive_dynamic.hpp>
int main()
{
    using namespace boost::xpressive;

    char* str = "*Link*||+Mario+||Zelda!!!|Metroid";

    //查找所有的单词，无视标点符号
    cregex reg = sregex::compile("\\w+", icase);

    cregex_token_iterator pos(str, str + strlen(str), reg);
    while(pos != cregex_token_iterator())
    {
        cout << "[" << *pos << " ";
        ++pos;
    }
    cout << endl;

    //使用分隔符正则表达式，分隔符是“||”
    cregex split_reg = cregex::compile("\\|\\|");
    pos = cregex_token_iterator(str, str + strlen(str), split_reg, -1);
    //重用原来的分词迭代器变量
    while(pos != cregex_token_iterator())
    {
        cout << "[" << *pos << " ";
        ++pos;
    }
    cout << endl;
}

```

程序运行结果如下：

```

[Link][Mario][Zelda][Metroid]
[*Link*][+Mario+][Zelda!!!][Metroid]

```

5.5.9 与 regex 的区别

boost.regex 是 Boost 库中另一个正则表达式解析库，xpressive 的形成受到了它的很大影响，接口几乎与 boost.regex 完全相同。但因为设计思想有根本的差异，两者之间存在如下的重要区别：

- xpressive 的 basic_regex<>的模板参数是迭代器类型，而 boost.regex 是字符类型；
- xpressive 的 basic_regex<>必须使用工厂方法 compile() 创建，不能直接从一个字符串创建；
- xpressive 的 basic_regex<>不具有类似 std::string 的操作接口；

■ xpressive 对 `basic_regex<>` 的一些定制功能位于工厂方法 `compile()` 中。

了解这些差异可以更深刻地理解 xpressive 和正则表达式概念，特别是在 `boost.regex` 被纳入 C++ 0x TR1 标准草案之后，可以为程序今后的移植提供帮助。

5.5.10 高级议题

本小节讨论关于 xpressive 库的一些高级议题。

工厂类

除了可以使用 `sregex::compile()` 创建正则表达式对象外，xpressive 还提供一个专门的工厂类 `regex_compiler<>`，它的类摘要如下：

```
template<typename BidiIter, typename RegexTraits, typename CompilerTraits>
struct regex_compiler {
    locale_type imbue(locale_type) ;
    locale_type getloc() const;
    basic_regex< BidiIter > compile(InputIter, InputIter) ;
    basic_regex< BidiIter > & operator[](string_type const &) ;
};
```

通常我们不直接使用 `regex_compiler`，而是使用它预定义的 `typedef`，如 `sregex_compiler` 和 `cregex_compiler`。

`regex_compiler` 也可以创建正则表达式对象，但比 `regex::compile()` 有更多的功能，可以传入特定的 `locale` 支持本地化，并重载 `operator[]` 实现了 `flyweight` 模式，可以把它当作一个正则表达式对象池。

示范 `regex_compiler` 用法的代码如下，它使用 `regex_compiler` 生产所有的正则表达式对象并统一管理：

```
cregex_compiler rc; //一个正则表达式工厂
rc["reg1"] = rc.compile("a|b|c"); // "reg1" 关联一个正则表达式
rc["reg2"] = rc.compile("\\d*"); // "reg2" 关联一个正则表达式
assert(!regex_match("abc", rc["reg1"]));
assert(regex_match("123", rc["reg2"]));
```

在 10.3.11 小节(394 页)另有一个 `regex_compiler` 的具体例子，它还示范了 `replace_all_copy()` 算法的使用。

异常

当 xpressive 在编译不正确的正则表达式或者执行其他操作出错时会抛出 `regex_error` 异常。

`regex_error` 是 `std::runtime_error` 和 `boost::exception` (参见 4.9 小节, 136 页) 的子类, 因此具有 `boost::exception` 的所有能力, 可以任意追加错误信息。

在使用 `xpressive` 时应当总使用 `try-catch` 块, 以防止异常。

格式化器

在使用 `regex_replace()` 进行替换时, 除了使用简单字符串和格式字符串, 还可以使用格式化器。格式化器是一个具有 `operator()` 的可调用对象, 函数指针、函数对象都可以, 它必须能够处理查找到的 `match_results` 对象。

下面的代码定义了一个函数对象 `formater`, 它将查找到的 `cmatch` 对象全部改为大写, 使用了 `string_algo` 库的 `to_upper_copy` 算法:

```
struct formater
{
    string operator()(cmatch const &m) const
    {
        return boost::to_upper_copy(m[0].str());
    }
};

char* str = "Link*||+Mario+||Zelda!!!|Metroid";

cregex reg = sregex::compile("\\w+", icase);
cout << regex_replace(str, reg, formater()) << endl;
```

让正则表达式执行得更快

这里我们需要区分一下“编译”这个词, 因为使用了正则表达式, 因此在程序中就出现了两种语言: C++和正则表达式。`xpressive` 的静态方式在程序的编译期同时进行 C++和正则表达式的编译处理, 而动态方式则在编译期只编译 C++, 而在运行时编译正则表达式。

与 `format` 库 (参见 5.2 小节, 167 页) 类似, 编译一个正则表达式需要很多的运行时开销, 应当少做 `sregex/cregex` 对象的创建工作, 尽量重用。

同样, `smatch/cmatch` 也应当尽量重用, 它们缓存了动态分配的内存, 不至于每次重新分配内存 (这个操作代价很高)。

静态正则表达式

静态正则表达式具有与动态正则表达式同样的功能, 但对表达式的编译是在程序的编译期, 因此省去了运行时的编译开销。据 `Boost` 库的测试, 静态方式要比动态方式平均快 10%到 15%, 这是一个不小的性能提升。如果程序对运行速度要求较高, 那么可以考虑静态正则表达式, 它的语法并不是很复杂、深奥, 只是要比动态方式多学习大约 30%到 50%的时间。

下面的代码定义了一个静态正则表达式，它等价于 “`^\d*\w+`”：

```
char* str = "123abc";
cregex reg = '^'>>*_d>>+_w;           //使用 operator>>连接表达式
cout << regex_match(str, reg) << endl;
```

其他高级特性

此外 `xpressive` 还有很多高级特性，如支持 `perl/sed` 的正则表达式语法、正则表达式语法嵌套、动态的命名捕获、构建自定义语法等等。有的用法相当复杂，本书不做进一步的介绍，感兴趣的读者可参考 `Boost` 自带的说明文档。

5.6 总结

本章讨论 `Boost` 库在字符串和文本处理方面的应用，这些工具大大增强了 C++ 语言在文本处理领域的能力。

我们首先看到的是 `lexical_cast`，一个方便的用于转换数字和字符串的小工具，它替代了 C 中的 `atoi()` 系列函数，而且功能更强大，在小的转换工作中很方便实用，但如果需要更精细的格式控制或者高级功能，就应该转向 `format` 库。

`format` 库为 C++ 提供了类似 `printf()` 的能力，可以任意格式化字符串。它使用 `%` 操作符代替了 `printf` 的可变参数，是类型安全的，还增加了新的格式化选项、语法和其他高级功能，可以比 `printf` 更容易、更灵活地输出格式化字符串。`format` 库的唯一缺点是性能不如 `printf`，但这个缺点完全不能与它所带来的巨大优势相比。

`string_algo` 是一个算法库，为字符串与文本处理提供了大量有用的算法，其他编程语言（如 Java、Python）中的 `trim`、`join`、`split` 等字符串算法在这里都可以找到，可以轻松解决大小写敏感的字符串处理问题。和标准库的算法一样，`string_algo` 的算法也是复杂而多变的，为了适合各种应用情况而有不同的形式，学习并掌握它们需要一定的时间，但肯定会带来不菲的回报。

`tokenizer` 库专注于文本处理领域中的分词功能，以类似容器的形式迭代分词结果，比较简单方便。但它的能力比较有限，预定义的分词函数对象较少，如果需要更多的功能需要自己定义，而且它对 `wstring` 的支持存在缺陷。在只需要简单的分词功能时可以使用，如果进行复杂的操作请转向 `string_algo` 和正则表达式。

`xpressive` 是一个灵活且功能强大的正则表达式解析库，可以构建小型文法，同时提供动态与静态两种用法。借助正则表达式的强大能力，`xpressive` 能够轻松解决文本处理领域绝大多数

的问题，可以基本替代 `string_algo` 库和 `tokenizer` 库。

但正则表达式本身是一门复杂的技术，要考虑它的调试和维护成本。解决同样的问题，`string_algo` 和 `tokenizer` 可能速度更快，编写的代码也易于理解，只有当它们确实无法解决你的问题时才应该使用正则表达式。

本章实现的工具类如下。

- `num_valid()`: 利用 `lexical_cast` 实现了简易验证数字有效性。
- `outable<>`: 为重载流输出操作符提供了方便，是对 `boost.operators` 库的一个有用的补充。
- `tokenizer_wrapper`: 包装了 `tokenizer` 的类型声明，可以较容易地支持 `string` 和 `wstring`，有利于增强代码的可移植性。

第 6 章

正确性与测试

测试对于软件开发是非常重要的，程序员——尤其是 C++ 程序员更应该认识到这一点。

但 C/C++ 只提供了很有限的正确性验证/测试支持——`assert` 宏（没错，它是一个宏，虽然它违背常识使用了小写的形式），这是很不够的。C++98 标准中的 `std::exception` 能够处理运行时异常、但并不能检查代码的逻辑，故 C/C++ 都缺乏足够的、语言级别的工具来保证软件的正确性，使程序员很容易陷入与 bug 搏斗的泥沼中。

Boost 在这方面前进了一大步：`boost.assert` 库增强了原始的运行时 `assert` 宏，`static_assert` 库提供了静态断言（编译期诊断），而 `boost.test` 库则构建了完整的单元测试框架。

6.1 assert

`boost.assert` 提供的主要工具是 `BOOST_ASSERT` 宏，它类似于 C 标准中的 `assert` 宏，提供运行时的断言，但功能有所增强。

为使用 `boost.assert` 需要包含头文件 `<boost/assert.hpp>`，即

```
#include <boost/assert.hpp>
```

6.1.1 基本用法

默认情况下 `BOOST_ASSERT` 宏等同于 `assert` 宏，断言表达式为真，即：

```
# define BOOST_ASSERT(expr) assert(expr)
```

宏的参数 `expr` 表达式可以是任意（合法的）C++ 表达式，从简单的关系比较到复杂的函数嵌

套调用都允许。如果表达式值为 `true`，那么断言成立，程序会继续向下执行，否则断言会引发一个异常，在终端上输出调试信息并终止程序的执行。例如：

```
BOOST_ASSERT(16 == 0x10);           //断言成立
BOOST_ASSERT(string().size() == 1);  //断言失败，抛出异常
```

`BOOST_ASSERT` 宏仅会在 `debug` 模式下生效，在 `release` 模式下不会进行编译，不会影响到运行效率，所以可以放心大胆地在代码中使用 `BOOST_ASSERT` 断言。它不仅能够诊断错误、保证程序正确运行，而且其规范的大写形式也能够起到类似代码注释的作用，提醒代码维护人员什么该做、什么不该做。

下面的例子演示了 `BOOST_ASSERT` 宏的一般用法，程序定义了一个取倒数函数 `func`，使用 `BOOST_ASSERT` 确保不会出现除以 0 的错误：

```
#include <boost/assert.hpp>
double func(int x)           //取倒数的函数
{
    BOOST_ASSERT(x != 0 && "divided by zero"); //断言参数非 0
    return 1.0 / x;
}
```

请读者注意：代码中 `BOOST_ASSERT` 宏表达式的用法，除了要检查的表达式“`x != 0`”，宏还使用逻辑与操作符 `&&` 向表达式增加了断言的描述信息。当断言失败时，可以给出更具描述性的文字，有助于排错。

在 `debug` 模式下以参数 0 调用函数 `func()`

```
int main()
{
    func(0);           //error
}
```

会导致程序异常终止报出如下的错误消息：

```
Assertion failed: x != 0 && "divided by zero", file d:\vc\main.cpp, line 48
```

6.1.2 禁用断言

`BOOST_ASSERT` 是标准断言宏 `assert` 的增强版，因此它有更多的使用灵活性。

如果在头文件 `<boost/assert.hpp>` 之前定义了宏 `BOOST_DISABLE_ASSERTS`，那么 `BOOST_ASSERT` 将会定义为 `((void)0)`，自动失效。但标准的 `assert` 宏并不会受影响，这可以让程序员有选择地关闭 `BOOST_ASSERT`。

修改一下之前的例子：

```
#define BOOST_DISABLE_ASSERTS           //注意这里
#include <cassert>                       //标准断言宏头文件
#include <boost/assert.hpp>             //boost 断言宏头文件
double func(int x)
{
    BOOST_ASSERT(x != 0 && "divided by zero");    //失效
    cout << "after BOOST_ASSERT" << endl;
    assert(x != 0 && "divided by zero");          //有效
    cout << "after" << endl;
    return 1.0 / x;
}
```

仍然调用 func(0)，程序运行结果将会是如下的样子：

```
after BOOST_ASSERT
Assertion failed: x != 0 && "divided by zero", file d:\vc\ main.cpp, line 52
```

6.1.3 扩展用法

如果在头文件<boost/assert.hpp>之前定义了宏 BOOST_ENABLE_ASSERT_HANDLER，这将导致 BOOST_ASSERT 的行为发生改变：

它将不再等同于 assert 宏，断言的表达式无论是在 debug 还是 release 模式下都将被求值。如果断言失败，会发生一个断言失败的函数调用 boost::assertion_failed()——这相当于提供了一个错误处理 handle。

函数 assertion_failed() 声明在 boost 名字空间里，但特意被设计为没有具体实现，其声明如下：

```
namespace boost
{
    void assertion_failed(char const * expr, char const * function, char const * file,
        long line);
}
```

当断言失败时，BOOST_ASSERT 宏会把断言表达式字符串、调用函数名（使用 BOOST_CURRENT_FUNCTION，参见 4.12.2 小节，158 页）、所在源文件名和行号都传递给 assertion_failed() 函数处理。用户需要自己实现 assertion_failed() 函数，以恰当的方式处理错误——通常是记录日志或者抛出异常。

演示 assertion_failed() 函数基本用法的代码如下，其中用到了 5.2 小节（167 页）的 format 库来代替 C 语言中的 printf() 函数：

```

#include <boost/format.hpp>
namespace boost {
void assertion_failed(char const * expr, char const * function,
                      char const * file, long line)
{
    boost::format fmt("Assertion failed!\n Expression: %s\nFunction: %s\nFile:
        %s\nLine: %ld\n\n");
    fmt % expr % function % file % line;
    cout << fmt;
}
} //namespace boost
#define BOOST_ENABLE_ASSERT_HANDLER
#include <boost/assert.hpp>
double func(int x){...}
int main()
{
    func(0); //error
}

```

程序的运行结果如下:

```

Assertion failed!
Expression: x != 0 && "divided by zero"
Function: double __cdecl func(int)
File: d:\vc\main.cpp
Line: 61

```

BOOST_ASSERT 的这种错误 handle 的用法很有用, 适合那些需要有统一错误处理方式的地方, 最常见的就是函数入口参数检查: 在函数入口断言参数, 当参数出错时拼错误字符串, 抛出参数异常类, 同时终止函数的流程。

如果担心 BOOST_ASSERT 提供的详细诊断信息有泄漏源代码的危险, 可以有选择地输出错误消息, 或者对字符串加密。

抛出异常的 assertion_failed 函数的实现可以是这样的:

```

void assertion_failed(...)
{
    string str;
    ... //拼字符串
    throw std::invalid_argument(str);
}

```

6.1.4 BOOST_VERIFY

BOOST_VERIFY 宏是 assert 库提供的另一个工具。它具有与 BOOST_ASSERT 一样的行为, 之前对 BOOST_ASSERT 的讨论对 BOOST_VERIFY 也同样适用。仅有一点区别: 断言的表达式一

定会被求值。在运用断言运算（而不仅仅是错误检查）及验证函数返回值时很有用。

示范 BOOST_VERIFY 宏的求值用法的代码如下：

```
#include <boost/assert.hpp>
...
int len;
BOOST_VERIFY(len = strlen("123"));           //len 被求值
```

使用 BOOST_VERIFY 需要注意的是，它在 release 模式下同样会失效，程序最好不应该依赖于它的副作用。

6.2 static_assert

assert 宏和 BOOST_ASSERT 宏是用于运行时的断言，它们是程序员的好帮手，应该在程序中被广泛应用。但有的时候，运行时的断言已经太晚了，程序已经发生了无可挽回的错误。

static_assert 库把断言的诊断时刻由运行期提前到编译期，让编译器检查可能发生的错误，能够更好地增加程序的健壮性。

为了使用 static_assert 组件，需要包含头文件<boost/static_assert.hpp>，即：

```
#include < boost/static_assert.hpp>
```

6.2.1 用法

与 BOOST_ASSERT 类似，static_assert 库定义了宏 BOOST_STATIC_ASSERT，用来进行编译期断言，使用方法也比较相似。例如：

```
BOOST_STATIC_ASSERT(2 == sizeof(short));
BOOST_STATIC_ASSERT(true);
BOOST_STATIC_ASSERT(16 == 0x10);
```

BOOST_STATIC_ASSERT 使用了较复杂的技术，但简单来理解，它实际上最终是一个 typedef，因此在编译时同样不会产生任何代码和数据，对运行效率不会有任何影响——不论是 debug 模式还是 release 模式。

BOOST_STATIC_ASSERT 是一个编译期断言，使用了 typedef 和模板元技术实现，虽然在很多方面它都与 BOOST_ASSERT 很相似，但用法还是有所不同的。最重要的区别是使用范围，BOOST_ASSERT(assert) 必须是一个能够执行的语句，它只能在函数域里出现，而 BOOST_STATIC_ASSERT 则可以出现在程序的任何位置：名字空间域、类域或函数域。

下面的代码定义了一个简单的模板函数 `my_min`, 出于某种目的, 它仅支持 `short` 或者 `char` 的类型:

```
#include <boost/static_assert.hpp>
template<typename T>
T my_min(T a, T b)
{
    BOOST_STATIC_ASSERT(sizeof(T) < sizeof(int)); //静态断言
    return a < b? a: b;
}
int main()
{
    cout << my_min((short)1, (short)3); //OK
    cout << my_min(1L, 3L); //编译期错误
}
```

静态断言的具体错误信息依据编译器而不同, 在 VC8 下, `main()` 的第二条语句会产生如下的提示:

```
error C2027: use of undefined type 'boost::STATIC_ASSERTION_FAILURE<x>'
```

断言的错误信息可能不够明显 (“未定义的类型错误”, 这是 `BOOST_STATIC_ASSERT` 的实现方式), 但它能够指明错误的位置, 并为解决错误指出了基本的方向。

`BOOST_STATIC_ASSERT` 在类域和名字空间域的使用方式与在函数域的方式相同, 例如:

```
namespace my_space
{
    class empty_class //一个“空”类
    {
        //在类域中静态断言, 要求 int 至少 4 字节
        BOOST_STATIC_ASSERT(sizeof(int)>=4);
    };

    //名字空间域静态断言, 是一个“空类”
    BOOST_STATIC_ASSERT(sizeof(empty_class) == 1);
}
```

这段代码同时展示了一个“有趣”的事实, “空类”其实并不空, 因为 C++ 不允许大小为 0 的类或对象的存在, 通常的“空类”会由编译器在里面安插一个类型为 `char` 的成员变量, 令它有一个确定的大小。

6.2.2 使用建议

`BOOST_STATIC_ASSERT` 主要在泛型编程或模板元编程中用于验证编译期常数或者模板类型

参数,可能读者暂时不会用到它,但随着对 C++认识的深入,也许不久的将来你就会发现它的好处。

使用 `BOOST_STATIC_ASSERT` 时还需要小心,断言的表达式必须能够在编译期求值,可能会让很多人不太适应,很正常,因为这正是迈向 C++泛型编程和模板元编程的第一步。

在 2.3 节 `progress_timer` 有另一个 `static_assert` 的具体使用例子,读者可参考。

6.3 test

`test` 库提供了一个用于单元测试的基于命令行界面的测试套件 Unit Test Framework,简称 UTF^①,还附带有检测内存泄漏的功能,比其他的单元测试库更强大更方便好用。它不仅能够支持简单的测试,也能够支持全面的单元测试,并且还具有程序运行监控功能,是一个用于保证程序正确性的强大工具。

为了使用 `test` 库需要包含头文件 `<boost/test/unit_test.hpp>`,即:

```
#include <boost/test/unit_test.hpp>
```

6.3.1 编译 test 库

`test` 库需要编译才能使用, `bjam` 命令如下:

```
bjam -toolset=msvc -with-test -build-type=complete stdlib=stlport stage
```

如果不想编译 `test` 库,同样可以使用工程中嵌入实现代码的方式,从而享受与操作系统、编译器、Boost 库版本无关的好处。

`test` 库非常体贴地提供了预编译源码文件,不需要如 `date_time` 库那样自己动手实现。头文件 `<boost/test/included/unit_test.hpp>` 包含了 `test` 库的所有实现代码,因此我们需要在工程中加入 `cpp` 文件如下:

```
//test_main.cpp
#define BOOST_TEST_MAIN //定义主测试套件,是测试 main 函数入口
#include <boost/test/included/unit_test.hpp>
```

这样,将导致把 `test` 的所有实现代码编译进测试程序。其他测试套件的源代码文件仍然需要包含 `<boost/test/unit_test.hpp>`,但前面应加入宏定义 `BOOST_TEST_INCLUDED`,告诉 `test` 库我们使用嵌入源码的使用方式,即:

```
//test_suitel.cpp
```

① 这可能会令有的读者联想到 Unicode 甚至 XML,抱歉,在阅读本章时请完全忘记它们。

```
#define BOOST_TEST_INCLUDED
#include <boost/test/unit_test.hpp>
```

这样就相当于在文件 `test_main.cpp` 中编译了 `test` 的静态库, 而其他文件则仅包含 `test` 库的声明, 直接使用编译好的静态库。

6.3.2 最小化的测试套件

`test` 库提供一个最小化的测试套件 `minimal test`, 不需要对 `test` 库做任何形式的编译, 只需要包含头文件 `<boost/test/minimal.hpp>` 就可以使用, 即:

```
#include <boost/test/minimal.hpp>
```

它只提供最基本的单元测试功能, 没有 `UTF` 那么强大, 不支持多个测试用例, 能够使用的测试断言也很少。但因为功能简单小巧, 适合入门和简单的测试。

头文件 `<boost/test/minimal.hpp>` 中已经实现了一个 `main()`, 因此我们不必再定义自己的 `main()`, 只需要实现一个 `test_main()` 函数, 它是 `minimal test` 的真正功能函数。

`test_main()` 函数的声明与标准的 `main()` 很相似:

```
int test_main( int argc, char* argv[] )
```

在 `test_main()` 的函数体内, 我们可以使用四个测试断言宏:

- `BOOST_CHECK(predicate)` : 断言测试通过, 如不通过不影响程序执行;
- `BOOST_REQUIRE(predicate)` : 要求测试必须通过, 否则程序停止执行;
- `BOOST_ERROR(message)` : 给出一个错误信息, 程序继续执行;
- `BOOST_FAIL(message)` : 给出一个错误信息, 程序运行终止。

示范 `minimal test` 用法的代码如下, 我们用它来简单测试一下 `format` 库 (参见 5.2 小节, 167 页):

```
#include <boost/test/minimal.hpp> //最小化测试套件头文件
#include <boost/format.hpp>
#include <iostream>
int test_main( int argc, char* argv[] ) //测试主函数
{
    using namespace boost;
    format fmt("%d-%d");

    BOOST_CHECK(fmt.size() != 0); //断言 format 对象已经初始化
    fmt % 12 % 34;
```

```

BOOST_REQUIRE(fmt.str() == "12-34");           //验证格式化结果

BOOST_ERROR("演示一条错误消息");             //不影响程序的执行

fmt.clear();
fmt % 12;
try
{
    std::cout << fmt;                         //输入参数不完整, 抛出异常
}
catch (...)
{
    BOOST_FAIL("致命错误, 测试终止");
}
return 0;
}

```

这段代码虽然很短, 但具备了单元测试的各个基本要素, 说明了单元测试的基本步骤。

我们用 `test_main()` 函数建立了这个程序中唯一的一个测试用例、同时也是唯一的一个测试套件, 测试的对象是 `boost::format`。然后我们可以使用各种方法操作测试对象, 并用类似 `BOOST_ASSERT` 的测试断言宏来验证操作结果。如果操作结果如预期, 那么一切都好; 否则, 单元测试框架会记录下断言失败的位置和数量。当遇到致命的错误, 可能导致无法继续运行测试时, 我们可以使用 `BOOST_FAIL` 来终止测试的运行。

单元测试程序运行结束后, `minimal test` 会在控制台给出一个本次测试的总结, 列出所有的失败错误断言和错误总数, 可以据此跟踪错误发源地, 进而纠正错误。

程序的运行结果大致是这样:

```

xxx.cpp(15): 演示一条错误消息 in function: 'int __cdecl test_main(int, char *[])'
xxx.cpp(25): 致命错误, 测试终止 in function: 'int __cdecl test_main(int, char *[])'

**** 2 errors detected

```

`minimal test` 简单好用, 但它的功能非常有限, 无法把它应用于大中型软件项目, 因为那经常需要很多的测试用例和测试套件, 而 `minimal test` 不能组织起复杂的测试结构。

`minimal test` 仅适用于单元测试的演示, 或者规模较小的程序, 通常这样的程序是由一个人在几天之内完成的, 要测试的接口不超过 10 个。

6.3.3 单元测试框架简介

`test` 库提供了强有力的单元测试框架 (UTF), 它为软件开发的基本领域——单元测试提供了简单而富有弹性的解决方案, 可以满足开发人员从高到低的各种需求, 它的优点包括:

- 易于理解，任何人都可以很容易地构建单元测试模块；
- 提供测试用例、测试套件的概念，并能够以任意的复杂度组织它们；
- 提供丰富的测试断言，能够处理各种情况，包括 C++ 异常；
- 可以很容易地初始化测试用例、测试套件或者整个测试程序；
- 可以显示测试进度，这对于大型测试是非常有用的；
- 测试信息可以显示为多种格式，如平文件或者 XML 格式；
- 支持命令行，可以指定运行任意一个测试套件或测试用例；
- 还有许多更高级的用法。

接下来我们将详细介绍 UTF 的各个组成部分，首先是测试断言，它是单元测试的基本工具。

6.3.4 测试断言

在 `test` 库中，测试断言是一组命名清楚的宏，它们的用法类似 `BOOST_ASSERT`，断言测试通过。如果测试失败，则会记录出错的文件名和行号以及错误信息。

`test` 库中一个典型的测试断言是 `BOOST_CHECK_EQUAL`，形式是 `BOOST_XXX_YYY`，具体命名规则如下：

- `BOOST_`：遵循 Boost 库的命名规则，宏一律以大写的 `BOOST` 开头；
- `XXX`：断言的级别。`WARN` 是警告级，不影响程序运行，也不增加错误数量；`CHECK` 是检查级别，如果断言失败增加错误数量，但不影响程序运行；`REQUIRE` 是最高的级别，如果断言失败将增加错误数量并终止程序运行。最常用的断言级别是 `CHECK`，`WARN` 可以用于不涉及程序关键功能的测试，只有当断言失败会导致无法继续进行测试时才能够使用 `REQUIRE`；
- `YYY`：各种具体的测试断言，如断言相等/不等、抛出/不抛出异常、大于或小于等等。

在 6.3.2 小节（220 页）我们已经见到了四个基本的测试断言：`BOOST_CHECK`、`BOOST_REQUIRE`、`BOOST_ERROR` 和 `BOOST_FAIL`。它们是最基本的测试断言，能够在任何地方使用，但同时为了通用性也不具有其他断言的好处，我们应当尽量少使用它们。

`test` 库中最常用的几个测试断言如下：

- `BOOST_XXX_EQUAL(l, r)`: 检查 `l==r`, 当测试失败时会给出详细的信息。它不能用于浮点数的比较, 浮点数的相等比较应使用 `BOOST_XXX_CLOSE`;
- `BOOST_XXX_GE(l, r)`: 检查 `l>=r`, 同样的还有 `GT (l>r)`、`LT (l<r)`、`LE (l<=r)` 和 `NE (l!=r)`, 它们用于测试各种不等性;
- `BOOST_XXX_THROW(expr, exception)`: 检测表达式 `expr` 抛出指定的 `exception` 异常;
- `BOOST_XXX_NO_THROW(expr, exception)`: 检测表达式 `expr` 不抛出指定的 `exception` 异常;
- `BOOST_XXX_MESSAGE(expr, message)`: 它与不带 `MESSAGE` 后缀的断言功能相同, 但测试失败时给出指定的消息;
- `BOOST_TEST_MESSAGE(message)`: 它仅输出通知用的信息, 不含有任何警告或者错误, 默认情况不会显示。

之后的几个小节将会看到这些测试断言的用法。

6.3.5 测试用例与套件

单元测试领域有很多专有概念, 本小节仅介绍 `test` 库中最重要的几个。

`test` 库将测试程序定义为一个测试模块, 由测试安装、测试主体、测试清理和测试运行器四个部分组成。测试主体是测试模块的实际运行部分, 由测试用例和测试套件组织成测试树的形式。

测试用例是一个包含多个测试断言的函数, 它可以被独立执行测试的最小单元, 各个测试用例之间是无关的, 发生的错误不会影响到其他测试用例。

要添加测试用例, 需要向 UTF 注册。在 `test` 库中, 可以采用手工或者自动两种形式, 通常自动的方式更加简单易用, 可以简化测试代码的编写。我们使用宏 `BOOST_AUTO_TEST_CASE` 像声明函数一样创建测试用例, 它的定义是:

```
#define BOOST_AUTO_TEST_CASE( test_name )
```

宏的参数 `test_name` 是测试用例的名字, 一般以 `t` 开头, 表明整个名字是一个测试用例, 例如:

```
BOOST_AUTO_TEST_CASE(t_case1)                //测试用例声明
{
    BOOST_CHECK_EQUAL(1, 1);                  //测试 1==1
```

```

    ...                               //其他测试断言
}

```

测试套件是测试用例的容器，它包含一个或多个测试用例，可以将繁多的测试用例分组管理，共享安装/清理代码，更好地组织测试用例。测试套件可以嵌套，并且没有嵌套层数的限制。

测试套件同样可以有手工和自动两种使用方式，自动方式使用两个宏 `BOOST_AUTO_TEST_SUITE` 和 `BOOST_AUTO_TEST_SUITE_END`，它们的定义是：

```

#define BOOST_AUTO_TEST_SUITE( suite_name )
#define BOOST_AUTO_TEST_SUITE_END()

```

这两个宏必须成对使用，宏之间的所有测试用例都属于这个测试套件。一个 C++ 源文件中可以有任意多个测试套件，测试套件也可以任意嵌套，没有深度的限制。测试套件的名字一般以 `s` 开头，例如：

```

BOOST_AUTO_TEST_SUITE(s_suite1)           //测试套件开始

BOOST_AUTO_TEST_CASE(t_case1)             //测试用例 1
{
    BOOST_CHECK_EQUAL(1, 1);
    ...                                   //其他测试断言
}
BOOST_AUTO_TEST_CASE(t_case2)             //测试用例 2
{
    BOOST_CHECK_EQUAL(5, 10/2);
    ...                                   //其他测试断言
}

BOOST_AUTO_TEST_SUITE_END()               //测试套件结束

```

任何一个 UTF 单元测试程序都必须存在一个主测试套件，它是整个测试树的根节点，其他的测试套件都是它的子节点。

主测试套件的定义可以使用宏 `BOOST_TEST_MAIN` 或者 `BOOST_TEST_MODULE`，定义了这个宏的源文件中不需要再有宏 `BOOST_AUTO_TEST_SUITE` 和 `BOOST_AUTO_TEST_SUITE_END`，所有测试用例都自动属于主测试套件。

因此，6.3.1 小节（219 页）的预编译文件 `test_main.cpp` 实际上不仅是预编译 `test` 库，它还定义了一个不包含任何测试用例的空主测试套件。

6.3.6 测试实例

了解了 `test` 库中测试用例和测试套件的概念，我们就可以进行真正的单元测试了。

首先要保证我们已经编译了 test 库，方法如 6.3.1 小节（219 页）所述，建立一个 test_main.cpp，包含有 test 库的编译源码。这个文件就是 test 库的预编译源程序，含有一个空的主测试套件。一旦写好，不应该再变动它。

接下来我们新建一个 cpp 文件，它包含有我们的第一个测试套件，这里我们选择 smart_ptr（参见 3.1 小节，61 页）作为我们的测试对象。测试套件的声明是：

```
BOOST_AUTO_TEST_SUITE(s_smart_ptr)
BOOST_AUTO_TEST_SUITE_END()
```

测试用例针对 scoped_ptr 和 shared_ptr，像这样：

```
BOOST_AUTO_TEST_CASE(t_scoped_ptr)
{
    //...
}
```

完整的单元测试程序如下：

```
#define BOOST_TEST_INCLUDED
#include <boost/test/unit_test.hpp>
#include <boost/smart_ptr.hpp>
using namespace boost;

//开始测试套件 s_smart_ptr
BOOST_AUTO_TEST_SUITE(s_smart_ptr)

//测试用例 1: t_scoped_ptr
BOOST_AUTO_TEST_CASE(t_scoped_ptr)
{
    scoped_ptr<int> p(new int(874));
    BOOST_CHECK(p); //p 不是空指针
    BOOST_CHECK_EQUAL(*p, 874); //测试解引用的值

    p.reset(); //scoped_ptr 复位
    BOOST_CHECK(p == 0); //为空指针
}

//测试用例 2: t_shared_ptr
BOOST_AUTO_TEST_CASE(t_shared_ptr)
{
    shared_ptr<int> p(new int(100));

    BOOST_CHECK(p); //p 不是空指针
    BOOST_CHECK_EQUAL(*p, 100); //测试解引用的值
    BOOST_CHECK_EQUAL(p.use_count(), 1); //引用计数为 1
}
```

```

shared_ptr<int> p2 = p;           //拷贝构造另一个 shared_ptr
BOOST_CHECK_EQUAL(p, p2);       //两个 shared_ptr 必定相等
BOOST_CHECK_EQUAL(p2.use_count(), 2); //引用计数为 2

*p2 = 255;                       //改变第二个 shared_ptr 所指的内容
BOOST_CHECK_EQUAL(*p, 255);      //第一个所指的内容也同时改变
BOOST_CHECK_GT(*p, 200);
}

//结束测试套件
BOOST_AUTO_TEST_SUITE_END()

```

单元测试程序的运行结果如下：

```

Running 2 test cases...

*** No errors detected

```

6.3.7 测试夹具

测试用例和测试套件构成了单元测试的主体，可以满足大部分单元测试的功能需求，但有的时候这些还不够，因为它们不能完成测试安装和测试清理的任务。

测试安装执行测试前的准备工作，初始化测试用例或测试套件所需的数据。测试清理是测试安装的反向操作，执行必要的清理工作。

测试安装和测试清理的动作很像 C++ 中的构造函数和析构函数，因此，可以定义一个辅助类，它的构造函数和析构函数分别执行测试安装和测试清理，之后我们就可以在每个测试用例最开始声明一个对象，它将自动完成测试安装和测试清理。

基于这个基本原理，UTF 中定义了“测试夹具”的概念，它实现了自动的测试安装和测试清理，就像是一个夹在测试用例和测试套件两端的夹子。测试夹具不仅可以用于测试用例，也可以用于测试套件和单元测试全局。

使用测试夹具，必须要定义一个夹具类，它只有构造函数和析构函数，用于执行测试安装和测试清理，基本形式是：

```

struct test_fixture_name           //测试夹具类
{
    test_fixture_name(){}          //测试安装工作
    ~test_fixture_name(){}         //测试清理工作
};

```

夹具类通常是个 struct，因为它被 UTF 用于继承，测试套件可以使用它的所有成员。当然

夹具类也可以是一个标准的 class，有私有、保护和公开成员，但这样测试套件就只能访问夹具的保护和公开成员。

指定测试用例和测试套件的夹具类需要使用另外两个宏：

```
#define BOOST_FIXTURE_TEST_SUITE( suite_name, F )
#define BOOST_FIXTURE_TEST_CASE( test_name, F )
```

它们替代了之前的 BOOST_AUTO_TEST_CASE 和 BOOST_AUTO_TEST_SUITE 宏，第二个参数指定了要使用的夹具类。

可以在测试套件级别指定夹具，这样套件内的所有子套件和测试用例都自动使用夹具类提供的安装和清理功能，但子套件和测试用例也可以另外自己指定其他夹具，不会受上层测试套件的影响。

全局测试夹具需要使用另一个宏 BOOST_GLOBAL_FIXTURE，它定义的夹具类被应用于整个测试模块的所有测试套件——包括主测试套件。

示范测试夹具用法的代码如下，测试对象是 assign 库（参见 4.4 小节，106 页）：

```
#define BOOST_TEST_INCLUDED
#include <boost/test/unit_test.hpp>
#include <boost/assign.hpp>
using namespace boost;

//全局测试夹具类
struct global_fixture
{
    global_fixture(){cout << ("global setup\n");}
    ~global_fixture(){cout << ("global teardown\n");}
};

//定义全局夹具
BOOST_GLOBAL_FIXTURE(global_fixture);

//测试套件夹具类
struct assign_fixture
{
    assign_fixture()
    {cout << ("suit setup\n");}
    ~assign_fixture()
    {cout << ("suit teardown\n");}

    vector<int> v; //所有测试用例都可用的成员变量
};

//定义测试套件级别的夹具
```

```

BOOST_FIXTURE_TEST_SUITE(s_assign, assign_fixture)

BOOST_AUTO_TEST_CASE(t_assign1)                                //测试+=操作符
{
    using namespace boost::assign;

    v += 1,2,3,4;
    BOOST_CHECK_EQUAL(v.size(), 4);
    BOOST_CHECK_EQUAL(v[2], 3);
}

BOOST_AUTO_TEST_CASE(t_assign2)                                //测试push_back函数
{
    using namespace boost::assign;

    push_back(v) (10) (20) (30);

    BOOST_CHECK_EQUAL(v.empty(), false);
    BOOST_CHECK_LT(v[0], v[1]);
}

BOOST_AUTO_TEST_SUITE_END()                                    //测试套件结束

```

单元测试程序运行结果如下:

```

global setup
Running 2 test cases...
suit setup
suit teardown
suit setup
suit teardown
global teardown

*** No errors detected

```

6.3.8 测试日志

测试日志是单元测试在运行过程中产生的各种文本信息,包括警告、错误和基本信息,默认情况下这些测试日志都被定向到标准输出(stdout)。测试日志不同于测试报告,后者是对测试日志的总结。

每条测试日志都有一个级别,只有超过允许级别的日志才能被输出。UTF 的日志的级别从低到高,高级别禁止了低级别的许可,但比它更高级别的日志则不受限制。

这些日志级别如下:

- `all` : 输出所有的测试日志;
- `success` : 相当于 `all`;
- `test_suite` : 仅允许运行测试套件的信息;
- `message` : 仅允许输出用户测试信息 (`BOOST_TEST_MESSAGE`);
- `warning` : 仅允许输出警告断言信息 (`BOOST_WARN_XXX`);
- `error` : 仅允许输出 `CHECK`、`REQUIRE` 断言信息 (`BOOST_CHECK_XXX`);
- `cpp_exception` : 仅允许输出未被捕获的 C++ 异常信息;
- `system_error` : 仅允许非致命的系统错误;
- `fatal_error` : 仅允许输出致命的系统错误;
- `nothing` : 禁止任何信息输出。

默认情况下, UTF 的日志级别是 `warning`, 会输出大部分单元测试相关的诊断信息, 但 `BOOST_TEST_MESSAGE` 宏由于是 `message` 级别, 它的信息不会输出。

日志级别可以通过接下来介绍的单元测试程序的命令行参数改变。

6.3.9 运行参数

基于 UTF 的单元测试程序在编译完成后可以独立运行, `test` 库提供了许多运行时的命令行参数, 可以调整程序的运行状态, 在测试大型程序时非常有用。

UTF 的命令行参数基本格式是:

```
--arg_name=arg_value
```

参数名称和参数值都是大小写敏感的, 并且 `==` 两边和 `--` 右边不能有空格。

常用的命令行参数如下:

- `run_test`: 可以指定要运行的测试用例或测试套件, 用斜杠 (/) 来访问测试树的任意节点, 支持使用通配符 *;
- `build_info`: 单元测试时输出编译器、STL、Boost 等系统信息, 取值为 `yes/no`;
- `output_format`: 指定输出信息的格式, 取值为 `hrf` (可读格式) / `xml`;

- `log_format`: 指定日志信息的格式, 取值为 `hrf` (可读格式) / `xml`;
- `log_level`: 允许输出的日志级别。取值为 `all`、`success`、`test_suite`、`message`、`warning`、`error`、`cpp_exception`、`system_error`、`fatal_error`、`nothing`, 默认是 `warning`;
- `show_progress`: 基于 `progress_display` 组件 (参见 2.4 小节, 23 页), 显示测试的进度, 取值为 `yes/no`。UTF 不能显示测试用例内部的进度, 只能显示已经完成的测试用例与测试用例总数的比例。

例如, 对于 6.3.7 (226 页) 小节的单元测试程序, 如果使用命令行参数:

```
--build_info=yes --run_test=s_assign/* --output_format=xml
```

程序运行结果可能是这样:

```
global setup
<TestLog>
<BuildInfo platform="Win32" compiler="Microsoft Visual C++ version 8.0" stl=
  "STLPort standard library version 0x521" boost="1.42.0"/>
  suit setup
  suit teardown
  suit setup
  suit teardown
global teardown
</TestLog>
<TestResult>
<TestSuite name="Master Test Suite" result="passed" assertions_passed="4" assertions_failed="0" expected_failures="0" test_cases_passed="2" test_cases_failed="0" test_cases_skipped="0" test_cases_aborted="0">
</TestSuite>
</TestResult>
```

这段稍显“凌乱”的测试输出, 以 XML 格式 (`--output_format=xml`) 显示了单元测试的运行系统信息 (`--build_info=yes`), 并运行了测试套件 `s_assign` 下的所有测试用例 (`--run_test=s_assign/*`)。

6.3.10 函数执行监视器

`test` 库在 UTF 框架底层提供一个函数执行监视器类 `execution_monitor`, 它被 UTF 用于单元测试, 但也可以被用于生产代码。`execution_monitor` 可以监控某个函数的执行, 即使函数发生预想以外的异常, 也能够保证程序不受影响地正常运行, 异常将会以一致的方式被 `execution_monitor` 处理。

如果编译了 test 库, 那么 execution_monitor 可以直接使用。如果不想仅仅为了使用 execution_monitor 就编译庞大的 test 库, 那么也可以单独编译 execution_monitor 部分。使用嵌入编译的方式需要定义如下的源文件:

```
//emprebuild.cpp
#include <boost/test/impl/execution_monitor.ipp>
#include <boost/test/impl/debug.ipp>
```

execution_monitor 位于名字空间 boost, 在需要使用 execution_monitor 的地方应加入如下头文件声明:

```
#define BOOST_TEST_INCLUDED
#include <boost/test/execution_monitor.hpp>
using namespace boost;
```

用法

execution_monitor 目前可以监控返回值为 int 或者可转换为 int 的函数, 并需要使用 unit_test::callback0<int>函数对象来包装^①, 之后成员函数 execute() 就可以监控执行被包装的函数。

execution_monitor 的监控执行语句需要嵌入在一个 try-catch 块里。如果一切正常, 那么被 execution_monitor 监控执行的函数就像未被监控一样运行并返回。否则, 如果发生了未捕获的异常、软硬件 signal 或 trap、以及 VC 下的 assert 断言, 那么 execution_monitor 就会捕获这个异常, 重新抛出一个 execution_exception 异常, 它存储了异常相关的信息。

execution_exception 不是标准库异常 std::exception 的子类, 必须在 catch 块明确地写出它的类型, 否则 execution_monitor 抛出的异常不会被捕获。

示范 execution_monitor 用法的代码如下:

```
#define BOOST_TEST_INCLUDED
#include <boost/test/execution_monitor.hpp>
using namespace boost;
int f() //一个简单的测试函数, 必须是无参返回值为 int
{
    cout << "f execute." << endl;
    throw "a error accoured"; //抛出一个未捕获的异常
    return 10;
}
```

① 很遗憾, 虽然 execution_monitor 也定义了用于包装多个参数和不同返回值类型的函数对象, 但 execute() 成员函数只提供了参数为 unit_test::callback0<int>的一种形式。

```

int main()
{
    execution_monitor em;                //声明一个监视器对象
    try                                  //开始 try-catch 块
    {
        em.execute(unit_test::callback0<int>(f));    //监控执行 f
    }
    catch (execution_exception& e)        //捕获异常
    {
        cout << "execution_exception" << endl;
        cout << e.what().begin()<< endl;          //输出异常信息
    }
}

```

请读者注意在上面的代码中对 `execution_exception` 异常的使用，在输出它存储的信息时我们使用了 `e.what().begin()` 的方式。这是因为 `execution_monitor` 被设计为在很少或者没有内存的情况下也可以使用，故它没有使用 `std::string` 来表示字符串，而是使用了一个内部类 `const_string`。`const_string` 默认不支持流输出，如果要使用流输出功能，需要包含头文件 `<boost/test/utlis/basic_cstring/io.hpp>`。

程序运行结果如下：

```

f execute.
execution_exception
C string: a error accoured

```

其他用法

除了基本的监控函数执行，`execution_monitor` 还有其他的用法。它提供了 `p_timeout`、`p_auto_start_dbg` 等读写属性用来设置监控器的行为，或者检测内存泄露，但这些功能不是完全可移植的，有的功能仅限于某些编译器或操作系统。

`execution_monitor` 也可以用于统一处理程序的异常，使用户不必自己编写错误处理代码。在这种情况下，程序抛出的异常类型必须是 C 字符串、`std::string` 或者 `std::exception` 三者之一，才能够被 `execution_exception` 所处理，通常这能够适合大多数应用。

如果因为某些原因必须使用自定义的异常类，而又想利用执行监控器的监控功能，那么可以定义异常类的翻译函数，并注册到 `execution_monitor` 中。例如：

```

struct my_error                                //一个自定义异常类
{
    int err_code;                                //错误代码
    my_error(int ec):err_code(ec){}              //构造函数
};
void translate_my_err(const my_error& e)        //翻译函数

```



```

{
    cout << "my err = " << e.err_code << endl; //输出到 cout
}
int f()                                     //被监控函数
{
    cout << "f execute." << endl;
    throw my_error(100);                   //抛出自定义异常
    return 0;
}
int main()
{
    execution_monitor em;

    //使用 register_exception_translator 函数注册异常翻译函数
    em.register_exception_translator<my_error>(&translate_my_err);
    try                                     //开始 try-catch 块, 监控函数的执行
    {
        em.execute(unit_test::callback0<int>(f));
    }
    catch (const execution_exception& e)
    {
        cout << "execution_exception" << endl;
        cout << e.what().begin();
    }
}

```

程序的运行结果如下:

```

f execute.
my err = 100

```

6.3.11 程序执行监视器

test 库在函数执行监视器 `execution_monitor` 的基础上提供程序执行监视器, 它的目的与 `execution_monitor` 相似, 监控整个程序的执行, 把程序中的异常统一转换为标准的操作系统可用的错误返回码。

程序执行监视器的用法很像 `minimal test`, 只需要包含一个头文件, 并实现与 `main()` 具有相同签名的 `cpp_main()`:

```

#include <boost/test/included/prg_exec_monitor.hpp>
int cpp_main( int argc, char* argv[] ){...}

```

注意: `cpp_main()` 必须要返回一个整数, 它不同于 `main()`, 不具有默认返回 0 值的能力。

程序执行监视器使用一个函数对象包装了 `cpp_main()`, 将它转换成一个返回 `int` 的无参函

数对象，然后使用 `execution_monitor` 监控执行。因此它的行为基本上与 `execution_monitor` 相同，当 `cpp_main()` 发生异常或者返回非 0 值时就会被捕获，并把异常信息输出到屏幕上。

6.3.12 高级议题

`boost.test` 库是一个复杂的系统，很难在这短短的几个小节中就讲述清楚，下面简单介绍一下 `test` 库的若干高级议题，供读者参考。

超轻量级测试

头文件 `<boost/detail/lightweight_test.hpp>` 包含一个未文档化的超轻量级单元测试工具 `lightweight_test`，它甚至比 `minimal_test` 还要小，功能也更少。

`lightweight_test` 不是一个单元测试框架，它提供三个测试断言：

- `BOOST_TEST` : 相当于 `BOOST_CHECK`，断言表达式成立；
- `BOOST_ERROR` : 直接断言失败，输出一条错误消息；
- `BOOST_TEST_EQ` : 相当于 `BOOST_CHECK_EQUAL`，断言两个表达式相等。

当临时要写测试代码、简单地验证程序正确性或者编译器不符合标准（无法使用 `test` 库的高级特性）时，`lightweight_test` 特别有用，但它的局限性也很大，如果要进行正式的单元测试，则应该使用 `minimal_test` 或者 `UTF`。

`lightweight_test` 不需要编译，也不需要特定的入口函数，测试断言可以用在程序的任何地方，就像使用 `assert` 一样。简单示范 `lightweight_test` 用法的代码如下：

```
#include <boost/smart_ptr.hpp>
#include <boost/detail/lightweight_test.hpp>
int main()
{
    shared_ptr<int> p(new int(10));
    BOOST_TEST(*p == 10);
    BOOST_TEST_EQ(p.use_count(), 1);

    BOOST_ERROR("error accored!!");
}
```

因为 `lightweight_test` 很小，因此简单地拷贝它所属的头文件 `lightweight_test.hpp` 及所需的 `current_function.hpp`，再修改一下源代码中的 `#include` 语句，就可以构造出一个便携的单元测试环境，在没有 Boost 程序库的环境下也可以使用。

期望测试失败

通常情况下所有的测试断言都应当通过，但有的时候需要特定的测试失败，允许有少量的断言不通过。这时我们可以使用宏 `BOOST_AUTO_TEST_CASE_EXPECTED_FAILURES`，它的声明如下：

```
#define BOOST_AUTO_TEST_CASE_EXPECTED_FAILURES( test_name, n )
```

在测试套件内部使用这个宏，指定测试用例名和允许失败的数量，例如：

```
BOOST_FIXTURE_TEST_SUITE(test_suite, fixture)

// 允许出现两个断言失败
BOOST_AUTO_TEST_CASE_EXPECTED_FAILURES(t_case1, 2)
BOOST_AUTO_TEST_CASE(t_case1)
{...}
BOOST_AUTO_TEST_SUITE_END()
```

这个功能在程序还没有完成所有模块的功能时很有用，可以先写好单元测试代码，使用 `BOOST_AUTO_TEST_CASE_EXPECTED_FAILURES` 忽略未完成的功能代码测试。

手工注册测试用例

使用测试用例最简单方便的方式是使用宏 `BOOST_AUTO_TEST_CASE`，它使用宏参数生成一个同名的无参函数，可以在里面编写测试代码。但手工注册也有好处，可以任意编写测试用函数或者接受一些测试用的数据，最后只需要写少量的代码就可以注册到 UTF，实现测试代码与注册的分离，有利于代码的维护。

手工注册需要使用宏 `BOOST_TEST_CASE` 和 `BOOST_PARAM_TEST_CASE` 生成测试用例类，并调用 `framework::xxx_test_suite().add()` 方法，本书不作详细的介绍，读者可参考 Boost 库文档。

测试泛型代码

UTF 也能够测试模板函数和模板类，这对于泛型库开发是非常有用的。手工注册测试用例需要使用两个宏：`BOOST_TEST_CASE_TEMPLATE_FUNCTION`，用于定义测试用例模板主体；`BOOST_TEST_CASE_TEMPLATE`，基于前者创建并注册测试用例。

如果采用自动测试方式，可使用 `BOOST_AUTO_TEST_CASE_TEMPLATE`，它需要使用模板元编程库 `mpl`，声明如下：

```
#define BOOST_AUTO_TEST_CASE_TEMPLATE( test_name, type_name, TL )
```

简单示范泛型单元测试用法的代码如下，测试对象是 `lexical_cast` (参见 5.1 小节, 163 页)：



```
#define BOOST_TEST_INCLUDED
#include <boost/test/unit_test.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/mpl/list.hpp>
using namespace boost;

BOOST_AUTO_TEST_SUITE(s_lexical_cast)

typedef mpl::list<short, int, long> types;           //模板元编程类型容器
BOOST_AUTO_TEST_CASE_TEMPLATE(t_lexical_cast, T, types)
{
    T n(20);
    BOOST_CHECK_EQUAL(lexical_cast<string>(n), "20");
}

BOOST_AUTO_TEST_SUITE_END()
```

VC 下使用 test 库的一个小技巧

test 库在文档中向库用户提供了一个在 VC 系列开发环境使用 test 库的一个小技巧，可以在工程选项 build-event 中设置 post-build，加入命令：

```
"$(TargetDir)\$(TargetName).exe" --result_code=no --report_level=no
```

这样在 VC 编译完成后可以立刻运行单元测试，并且在 Output 窗口显示出未通过的测试断言，可以用双击的方式快速跳转到断言的位置，提高测试的工作效率。

6.4 总结

很多软件方法学如敏捷开发、极限编程都非常强调测试的重要性，使用 Boost 测试库，我们就可以实践这些方法。

assert 库提供一个 assert 宏的增强版本 BOOST_ASSERT，它默认的行为与 assert 相同，但其规范的大写形式有助于代码的维护。如果定义了配置宏 BOOST_ENABLE_ASSERT_HANDLER，BOOST_ASSERT 就给断言安装了一个处理 handle，能够以任意方式处理断言失败的情形。

static_assert 是编译期的断言，计算编译期表达式的值，在泛型编程和模板元编程领域非常有用。如果读者正在使用模板技术编写泛型算法或者泛型类，那么应该仔细地研究一下它的使用法，它可以保证程序中的模板函数或者模板类如预期那样工作。

本章的重点内容是 test 库，它实现了一个完整的单元测试框架 UTF，不仅可以测试普通的函数和类，也可以测试模板函数和模板类，这是其他类似功能的单元测试工具所不具备的。test

库必须编译后才能使用，但可以采用嵌入工程编译的方法以获得系统独立性。

最简单最容易使用的单元测试是 `minimal test`，它不需要编译就可以使用，提供 UTF 最基本的测试功能，很容易使用，但不适合大多数实际的软件开发项目。

UTF 提供了完整的单元测试框架，提供各种测试概念，如测试安装/清理、测试套件、测试用例、测试断言，用法非常自由，支持手工和自动两种向框架注册的方式，自动方式在通常情况下是最佳的选择，可以简化测试代码的编写工作。

UTF 在测试的输出方面也很灵活，输出日志有详细的级别设定，通过运行参数可以指定允许输出的级别，日志格式也可以选择普通的 HRF (Human Readable Format) 或 XML 格式。运行参数还能够控制单元测试的其他方面，如输出系统消息、显示测试进度等等。

除了单元测试，`test` 库还提供了用于生产环境的执行监视器，保证不抛出异常，并对外提供统一的异常处理方式。

第 7 章

容器与数据结构

容器是 STL 中最引人注意的部分，vector、deque、list、set 和 map 分别实现了最常用的动态数组（向量）、双端队列、双向链表、集合和映射五种数据结构，它们以泛型的形式提供，可以容纳任意的类型（但也有例外，比如 auto_ptr），极大地方便了 C++ 程序员的工作。

Boost 程序库基于 STL 同样的设计理念，实现了数个新型容器（数据结构），它们都是很常用而 STL 没有来得及实现的，如散列容器、循环队列、多维数组等，大大扩展了“容器”概念的内涵和外延。

本章总共介绍 10 个容器（数据结构），首先是 array、dynamic_bitset、unordered_bimap 和 circular_buffer，它们都是对 STL 原有容器在概念上的扩展，与 STL 容器的接口非常相似，比较容易学习。随后是 tuple、any、variant，它们三个既是数据结构也是特殊的容器，展示了泛型编程的精妙用法。最后介绍新型的容器 multi_array 和 property_tree。

这些容器都对容纳的元素类型有一个基本要求：析构函数不能抛出异常，有的容器还要求元素具有缺省构造函数或者拷贝构造函数。

7.1 array

array 包装了 C++ 语言内建的数组，为其提供标准的 STL 容器接口，如 begin()、front() 等。速度、性能上与原始数组相差无几，在程序对性能要求很高，或者不需要动态数组的情况下可以使用。

本库已被收入 C++ 新标准的 TR1 库。

array 位于名字空间 boost，为使用 array 组件，需要包含头文件 <boost/array.hpp>，

即:

```
#include <boost/array.hpp>
using namespace boost;
```

7.1.1 类摘要

array 的类摘要如下:

```
template<class T, std::size_t N>
class array {
public:
    T elems[N];

    iterator begin() ;
    iterator end() ;

    reference operator[](size_type i) ;
    reference at(size_type i);

    reference front();
    reference back();

    static size_type size() ;
    static bool empty() ;
    static size_type max_size() ;

    T* data() ;
    T* c_array() ;
    void swap(array<T, N>& y);
    void assign (const T& value);
}
```

array 的大部分成员函数都是仅有一行的内联代码,这使得它很容易被编译器优化。它的接口比较简单也易于理解,如果读者熟悉 `std::vector`,可以很快地掌握它的用法。

7.1.2 操作函数

array 的模板参数指明了 array 的元素类型和大小, `array<int, 5>` 相当于声明了一个普通数组 `int a[5]`。

array 包装了普通数组,并提供类似 STL 容器的接口:

- `begin()` 和 `end()` 函数可以返回容器区间的开始和结束迭代器,相当于数组的首指针和末指针+1;

- `front()` 和 `back()` 返回容器首末元素的引用，相当于 `elems[0]` 和 `elems[N-1]`；
- `size()` 和 `max_size()` 返回容器的大小，因为是静态数组，故两者返回值相同，都是模板参数 `N`；
- `empty()` 判断容器是否为空，因为是静态数组，该函数总返回 `false`。

`array` 重载了 `[]` 操作符，因而可以像普通数组和 `std::vector` 一样用下标访问内部元素，并支持左值和右值。它的目标是提供高效的操作，仅使用 `BOOST_ASSERT` 断言索引是有效的，如果在 `debug` 模式下越界访问会抛出异常，但 `release` 模式下不提供检查。

成员函数 `at()` 类似 `operator[]`，根据索引值访问内部元素。但 `at()` 有范围检查的功能，如果索引值超过容器大小会抛出异常。

`array` 的 `data()` 和 `c_array()` 以 C 数组的形式返回内部数组指针，用于需要原始指针的场合。两者的功能基本相同，但 `data()` 在需要的时候可以返回一个数组的常量指针，不允许变动容器元素。

为了方便操作 `array` 对象，`array` 还提供了 `operator=`、`swap()` 和 `assign()` 函数^①：

- `operator=` 使用 `std::copy` 算法实现了赋值操作；
- `swap()` 使用 `boost::swap` 可以高效地交换两个 `array` 对象；
- `assign()` 使用标准算法 `std::fill_n` 将容器内所有元素赋值为 `value`。

此外，`array` 还重载了 `==`、`<`、`>` 等比较操作符，使用字典序比较两个 `array` 对象（但没有使用 `operators` 库来简化操作符重载定义）。

7.1.3 用法

`array` 需要在模板参数列表中指明数组元素的类型和大小，声明的形式与 `vector` 和普通数组都很相像，它的使用方法与普通数组没有太大的区别，一个很小的例子就可以演示清楚：

```
#include <boost/array.hpp>
using namespace boost;
int main()
{
    array<int, 10> ar;           // 一个大小为 10 的 int 数组
```

① Boost1.43 版中 `array` 增加了 `fill()` 成员函数，功能与 `assign()` 完全相同。

```

ar[0] = 1; //使用 operator[]
ar.back() = 10; //back() 访问最后一个元素
assert(ar[ar.max_size() - 1] == 10);

ar.assign(777); //数组所有元素赋值为 777
for (BOOST_AUTO(pos, ar.begin()); pos != ar.end(); ++pos)
{ cout << *pos << ", "; }

int *p = ar.c_array(); //获得原始数组指针
*(p + 5) = 253; //指针算术运算
cout << ar[5] << endl;

ar.at(8) = 666; //使用 at 函数访问元素
sort(ar.begin(), ar.end()); //可以使用标准算法排序
}

```

这段代码示范了 `array` 的大部分功能，如 `operator[]`、`assign()`、`c_array()` 和迭代器。

基本上，读者可以把 `array` 看做是普通数组，只是多了一些 STL 风格的成员函数，可以方便地搭配 STL 算法，是一个“更好的”数组。

7.1.4 能力限制

模板类 `array` 本质上是一个对静态数组的包装，因此它不完全符合标准容器的定义，不具有标准容器的许多功能，虽然它在很多方面很像标准容器。

`array` 的能力缺陷主要是：

- 没有构造函数，不能指定大小和初始值（只能用模板参数指定大小）；
- 没有 `push_back()` 和 `push_front()`，因为它不能动态增长；
- 不能搭配插入迭代器适配器功能，同样因为它不能动态增长。

因此，`array` 的功能相当有限，只能应用在已知数组大小，或者对运行速度要求很高的场合。如果需要可动态变动容量的数组，请使用 `std::vector` 或者 `boost::scoped_array`。

7.1.5 `array` 的初始化

虽然 `array` 没有构造函数，但它可以使用时与普通数组一样的风格进行初始化，把数组元素以逗号分隔放进花括号中，例如：

```
array<string, 3> ar = {"alice", "bob", "carl"};
```

也可以在花括号中初始化部分元素，剩余的元素将调用缺省构造函数完成初始化：

```
int a[10] = {0}; //普通数组初始化
array<int, 10> ar = {0}; //array 初始化
assert(std::equal(ar.begin(), ar.end(), a)); //两者的值同为全 0
array<string, 3> ar2 = {"racer"}; //初始化第一个元素
assert(ar2.at(1).empty()); //第二个元素缺省构造,是个空字符串
```

array 也可以与 assign 库(参见 4.4 小节)的 list_of 配合工作。但因为没有 insert()、push_back()、push_front() 等成员函数,不能对它调用这些同名的 assign 库函数。例如:

```
using namespace boost::assign;
array<int, 3> arr(list_of(2)(4)(6)); //只能用 list_of
for (int i = 0; i < arr.size(); ++i)
{ cout << arr[i] << ", "; }
```

如果使用 VC 较老版本自带的 Dinkumware STL 或者其他不太符合标准的 STL 实现,那么上面的写法可能会失效,需要改用另一种形式:

```
array<int, 2> arr;
arr = list_of(3)(4).to_array(arr);
assert(arr[0] == 3);
```

7.1.6 实现 ref_array

array 对 C++ 内建数组实现了较完善的包装,但还不够。在实际开发工作中我们经常需要与老式的 C 接口打交道,比如很多 API 都有类似的形式 some_func(char* buf, int *buflen), array 无法操作这种形式的数组。这种情况下,我们需要类似 scope_array 和 shared_array 那样的代理类,但可惜的是,目前 Boost 库还没有这种对静态数组的代理类。

本书提供一个可用的数组代理实现,名字叫 ref_array。

ref_array 的设计思想与 array 很相似,基于对内建数组的包装,但 array 是在编译期指定数组的维数,而 ref_array 则使用构造函数在运行时指定。因此,ref_array 内部除了使用一个指针来存储数组地址外,还要使用一个整型变量来保存数组的大小。

在容器功能方面 ref_array 与 array 基本相同,也不能动态增长(那应该是 vector 的职责)。在重载操作符时,ref_array 使用了 boost.operators 库,简化了代码。

ref_array 的类摘要如下:

```
template<typename T>
class ref_array:
private boost::totally_ordered<ref_array<T> >
{
private:
    T* elems; //数组指针
```

```

std::size_t N;                                //数组大小

public:
explicit ref_array(T *arr, std::size_t n):      //构造函数
    elems(arr), N(n) { BOOST_ASSERT(arr != NULL); }
~ref_array() { }                               //析构函数

//迭代器
iterator begin() { return elems; }
iterator end() { return elems+N; }

//容器容量操作
size_type size() const { return N; }
bool empty() const { return false; }
size_type max_size() const { return N; }

T* data() { return elems; }                   //提供原始数组访问
T* c_array() { return elems; }
void assign (const T& value);                  //赋值操作
void assign (const Cont& from);

void swap (ref_array<T>& y);                    //交换数组内容
};

```

ref_array 的完全实现代码请参考本书附录，那里还同时实现了 cref_array，一个对常数组的代理。

7.1.7 ref_array 的用法

ref_array 的用法与 scoped_array/shared_array 类似，都需要有一个原始数组作为它代理的对象，不同的是它只能代理静态数组，如果把它应用于动态数组，那么它不会自动释放内存，会导致内存泄漏。

ref_array 的接口几乎与 array 一样，区别仅在于构造函数，它的模板参数仅接受元素类型，而元素的数量则在构造函数中指定。

为了支持与其他容器互操作，ref_array 增加了新的 assign() 函数，可以从另一个容器那里拷贝元素。

示范 ref_array 用法的代码如下：

```

//一个C风格接口的异或处理函数
void xor(unsigned char *buf, int buf_len)
{
    ref_array<unsigned char> ra(buf, buf_len);
}

```

```
    for (BOOST_AUTO(pos, ra.begin()); pos != ra.end(); ++pos)
    {
        *pos ^= 0x5a;
    }
}

int main()
{
    unsigned char szInt[10];
    ref_array<unsigned char> ra(szInt, 10);

    ra.assign(100);
    ra[0] = 25;
    assert(ra.front() == 25);

    xor(ra.c_array(), ra.size());
}
```

在 7.9.2 小节 (304 页) 有一个 ref_array 的具体使用例子。

7.2 dynamic_bitset

C++98 标准为处理二进制数值提供了两个工具：vector<bool>和 bitset。

vector<bool>是对元素类型为 bool 的 vector 特化，它内部并不真正存储 bool 值，而是以 bit 来压缩保存、使用代理技术来操作 bit，造成的后果就是它很像容器，大多数情况下的行为与标准容器一致，但它不是容器，不满足容器的定义。

bitset 与 vector<bool>类似，同样存储二进制位，但它的大小固定，而且比 vector<bool>支持更多的位运算。

vector<bool>和 bitset 各有优缺点：vector<bool>可以动态增长，但不能方便地进行位运算；bitset 则正好相反，可以方便地对容纳的二进制位做位运算，但不能动态增长。

boost.dynamic_bitset 的出现恰好填补了这两者之间的空白，它类似标准库的 bitset，提供丰富的位运算，同时长度又是动态可变的。

dynamic_bitset 位于名字空间 boost，为了使用 dynamic_bitset 组件，需要包含头文件<boost/dynamic_bitset.hpp>，即：

```
#include <boost/dynamic_bitset.hpp>
using namespace boost;
```

7.2.1 类摘要

dynamic_bitset 的类摘要如下：

```

template <typename Block, typename Allocator>
class dynamic_bitset
{
public:
    explicit dynamic_bitset();
    dynamic_bitset(const dynamic_bitset& b);

    void swap(dynamic_bitset& b);
    void resize(size_type num_bits, bool value = false);
    void clear();
    void push_back(bool bit);
    void append(Block block);

    dynamic_bitset& operator+=(const dynamic_bitset& b);
    dynamic_bitset& operator|=(const dynamic_bitset& b);
    ...

    dynamic_bitset& set();
    dynamic_bitset& reset();
    dynamic_bitset& flip();

    bool test(size_type n) const;
    bool any() const;
    bool none() const;
    dynamic_bitset operator~() const;
    size_type count() const;

    bool operator[](size_type pos) const;

    unsigned long to_ulong() const;

    size_type size() const;
    size_type num_blocks() const;
    size_type max_size() const;
    bool empty() const;

    bool is_subset_of(const dynamic_bitset& a) const;
    bool is_proper_subset_of(const dynamic_bitset& a) const;

    size_type find_first() const;
    size_type find_next(size_type pos) const;
};

```

就像 `array` 和 `ref_array` 的关系一样, `dynamic_bitset` 也几乎与 `std::bitset` 相同, 包括接口和行为, 唯一的区别是 `dynamic_bitset` 的大小是在构造函数中由参数指定的, 而且运行时是动态可变的。

注意：与 `vector<bool>` 和 `bitset` 一样，`dynamic_bitset` 不符合标准容器的定义，不是严格意义上的“容器”，为了避免误用，它特意不提供迭代器支持。不过为了叙述方便，接下来还会用容器来称呼，但读者一定要留意这一点。

7.2.2 创建与赋值

`dynamic_bitset` 的模板参数是：

```
template <typename Block, typename Allocator>
```

第一个类型参数 `Block` 指示 `dynamic_bitset` 以什么整数类型存储二进制位，必须是一个无符号整数，默认是 `unsigned long`。第二个类型参数 `Allocator` 是类内部使用的内存分配器，默认是 `std::allocator<Block>`。这两个类型参数一般情况下我们无需变动，默认类型会工作得很好。^①

有很多种方式可以创建 `dynamic_bitset` 对象，例如：

- 不带参数的构造函数创建一个大小为 0 的 `dynamic_bitset` 对象（可以在之后增长）；
- 传入参数指定 `dynamic_bitset` 的大小并赋初值，像标准的容器一样；
- 从另一个 `dynamic_bitset` 拷贝构造；
- 从 01 字符串构造（很遗憾，`dynamic_bitset` 与 `std::bitset` 有相同的缺陷，要求字符串必须是 `std::string`，而不能是 C 字符串）。

示范 `dynamic_bitset` 创建与赋值的各种形式的代码如下：

```
dynamic_bitset<> db1; //空的 dynamic_bitset
dynamic_bitset<> db2(10); //大小为 10 的 dynamic_bitset
dynamic_bitset<> db3(0x16, BOOST_BINARY(10101));
dynamic_bitset<> db4(string("0100")); //字符串构造
dynamic_bitset<> db5(db3); //拷贝构造

dynamic_bitset<> db6;
db6 = db4; //赋值操作符

cout << hex << db5.to_ulong() << endl;
cout << db4[0] << db4[1] << db4[2] << endl;
```

① 如果程序中经常使用不超过 32 位的二进制数，那么 `unsigned long` 可能会造成一定的空间浪费，这时就可以考虑变动 `dynamic_bitset` 的 `Block` 类型，使用 `unsigned char` 或者 `unsigned short`。

这段代码中值得注意的是 `db3` 对象, 它使用了 `BOOST_BINARY` 宏 (参见 4.12.1 小节, 157 页) 构造了一个编译期的二进制数, 没有运行时开销, 较 `db4` 使用 `string` 临时变量的构造效率要高。

`dynamic_bitset` 内部按照由高到低的顺序存储二进制位, 也就是说, 第 0 个元素存储最低位 (二进制字面值最右边的值)。因此, 代码最后一行的输出是:

```
001
```

7.2.3 容器操作

`dynamic_bitset` 可以使用 `resize()` 成员函数在运行时调整容器的大小, 扩展或者收缩都是允许的, 并且可以用 `true/false` 或者 `1/0` 指定扩展后新元素的默认值。如果是扩展, 那么原有的二进制位保持不变, 新增加的二进制位被置为指定的值; 如果是收缩, 那么收缩后容量的二进制位保持不变, 多余的二进制位被抛弃, 此时设置新元素的默认值是无意义的。例如:

```
dynamic_bitset<> db;                //空的 dynamic_bitset
db.resize(10, true);                //扩展为 10 个二进制位, 值全为 1
cout << db << endl;                //输出 1111111111
db.resize(5);                        //缩小容量为 5
cout << db << endl;                //输出 11111
```

清空 `dynamic_bitset` 可以使用成员函数 `resize(0)`, 它将把容器内所有的二进制位删除, 但我们更应该调用 `clear()` 函数, 它更快速。`dynamic_bitset` 也提供与标准容器相同的 `size()` 和 `empty()` 函数, 可以返回 `dynamic_bitset` 当前容纳的二进制位的个数和判断容器是否为空, `empty()` 等价于 `size()==0`, 但更快速:

```
dynamic_bitset<> db(5, BOOST_BINARY(01110));
cout << db << endl;                //输出 01110
assert(db.size() == 5);              //目前有 5 个二进制位
db.clear();                           //清空 dynamic_bitset
assert(db.empty() && db.size()==0);  //判断 dynamic_bitset 是否为空
```

由于 `dynamic_bitset` 使用 `Block` 来存储二进制位, 因此 `size()` 函数不能反映 `dynamic_bitset` 所占用的内存大小, `dynamic_bitset` 提供 `num_blocks()` 返回所有二进制位占用的 `Block` 数量, 即 `size()/sizeof(Block)*8+1`。如果使用默认模板参数 `unsigned long`, 那么一个 `Block` 就可以存储 32 个二进制位, `num_blocks()` 的计算公式是 `size()/32+1`, 故下面的断言成立:

```
assert(dynamic_bitset<>(32).num_blocks()==1);
assert(dynamic_bitset<>(33).num_blocks()==2);
```

`dynamic_bitset` 可以像 `vector` 那样使用 `push_back()` 向容器末尾 (二进制数的最高位) 追加一个值:

- `test()` 函数检验第 `n` 位是否是 1;
- 如果容器中存在二进制位 1, 那么 `any()` 返回 `true`;
- 如果容器中不存在二进制位 1, 那么 `none()` 返回 `true`;
- `count()` 函数统计容器中所有值为 1 的元素的数量。

例如:

```
dynamic_bitset<> db(4, BOOST_BINARY(0101));
assert(db.test(0) && !db.test(1));
assert(db.any() && !db.none());
assert(db.count() == 2);
```

有三个翻转二进制位的函数:

- `set()` 函数可以置全部或者特定位置的值为 1 或 0;
- `reset()` 可以置全部或者特定位置的值为 0;
- `flip()` 可以反转全部或者特定位置的值。

这三个函数都有两种重载形式, 无参的版本操作 `dynamic_bitset` 的所有二进制位, 带参数的版本操作指定的二进制位。例如:

```
dynamic_bitset<> db(4, BOOST_BINARY(0101));
db.flip(); //翻转所有二进制位
assert(db.to_ulong() == BOOST_BINARY(1010))

db.set(); //置所有位为 1
assert(!db.none()); //此时无 0 二进制位

db.reset(); //置所有位为 0
assert(!db.any()); //此时无 1 二进制位

db.set(1, 1); //设置 db[1] 为值 1
assert(db.count() == 1);
```

`dynamic_bitset` 还为访问元素提供了查找操作:

- `find_first()` 从第 0 位置开始查找, 返回第一个值为 1 的位置;
- `find_next(pos)` 则从第 `pos` 位置开始查找, 返回第一个值为 1 的位置; 如果找不到这样的值, 则返回 `npos`。

由于 `dynamic_bitset` 不是容器, 不提供迭代器, 故这两个查找函数返回的不是迭代器对

象，而是容器内二进制位的索引值，其类型是内部类型定义 `size_type`，即 `size_t`。它们的使用如下：

```
dynamic_bitset<> db(5, BOOST_BINARY(00101));
BOOST_AUTO(pos, db.find_first());    //从最低位找二进制位 1
assert(pos == 0);
pos = db.find_next(pos);              //从 pos 位置开始找下一个二进制位 1
assert(pos == 2);
```

7.2.6 类型转换

我们之前已经多次使用了 `dynamic_bitset` 的成员函数 `to_ulong()`，它可以直接把内部存储的二进制转换成一个类型为 `unsigned long` 的整数。

```
dynamic_bitset<> db(10, BOOST_BINARY(1010101));
cout << db.to_ulong() << endl;      //85
```

但 `to_ulong()` 的前提是 `dynamic_bitset` 内部存储的二进制位表示不能超过 `unsigned long` 的最大值（即 `std::numeric_limits<unsigned long>::max()`），否则会转换失败，抛出 `std::overflow_error` 异常。

例如，下面的代码先用 `append()` 为 `dynamic_bitset` 增加了一个整数，虽然整数值很小，但它为 `dynamic_bitset` 增加了 32 位的二进制位（前面全是 0），当再追加一个 1 位的时候就会超过 `unsigned long` 的上限：

```
db.append(10);                      //追加一个整数
cout << db.to_ulong() << endl;      //10325
db.push_back(1);                    //再追加一个二进制位, 超过 ulong 的上限
cout << db.to_ulong() << endl;      //抛出异常
```

当 `dynamic_bitset` 无法转换成一个 `unsigned long` 表示的时候，可以使用自由函数 `void to_string()` 转换成一个标准字符串，它不会抛出异常（除非无法分配内存）。`to_string()` 把转换后的字符串通过参数传出，而不是以返回值的形式：

```
string str;
to_string(db, str);
cout << str << endl;
```

7.2.7 集合操作

`dynamic_bitset` 容纳的是 01 值，这使得它适合用于表示集合。`dynamic_bitset` 支持集合的基本运算，如交集、并集和差集，但并没有提供专门的成员函数，因为这些集合运算都可以由位运算实现，如 `operator&` 和 `operator|` 对应于交集和并集。

`dynamic_bitset` 为子集运算实现了两个成员函数：`is_subset_of` 和 `is_proper_subset_of`，分别检测一个对象是否是另一个对象的子集或真子集，要求这两个检查的对象里的元素数量必须相等，否则会引发运行时断言异常。

示范 `dynamic_bitset` 集合操作作用法的代码如下：

```
dynamic_bitset<> db1(5, BOOST_BINARY(10101));
dynamic_bitset<> db2(5, BOOST_BINARY(10010));

cout << (db1 | db2) << endl;           //并集 10111
cout << (db1 & db2) << endl;           //交集 10000
cout << (db1 - db2) << endl;           //差集 00101

dynamic_bitset<> db3(5, BOOST_BINARY(101));
assert(db3.is_proper_subset_of(db1));    //是真子集

dynamic_bitset<> db4(db2);                //拷贝构造
assert(db4.is_subset_of(db2));           //是子集
assert(!db4.is_proper_subset_of(db2));   //不是真子集
```

7.2.8 综合运用

本小节给出一个综合运用 `dynamic_bitset` 的例子：使用筛法求质数。

我们先来快速复习一下概念。质数，又称素数，是指只能被 1 和它自身整除的整数，如 2、3、5、11，而 6、15 这样不符合要求的数被称为合数。0 和 1 这两个数比较特殊，它们既不是质数也不是合数^①。

筛法求素数的原理很简单，首先确定一个整数集合，然后逐个筛选出不符合要求的数，最后剩下的就是质数。在这里我们使用 `dynamic_bitset` 来实现从 2 开始的整数区间筛法求质数。

首先我们需要含入 `dynamic_bitset` 头文件，并打开 `boost` 名字空间：

```
#include <boost/dynamic_bitset.hpp>
using namespace boost;
```

然后我们从用户输入得到要筛选的整数区间，并创建 `dynamic_bitset` 对象。这体现了 `dynamic_bitset` 动态的优越性，`std::bitset` 无法在运行时指定大小：

```
int main()
```

① 题外话，0 和 1 是数学的基石，它们与另外三个数： e 、 i 和 π 并称为数学中最重要的五个数。把这五个数联系起来的是一个完美的等式——欧拉公式： $e^{i\pi} + 1 = 0$ 。

```
{
    int n ;
    cin >> n;
    dynamic_bitset<> db(n);
```

随后我们使用 `set()` 置集合中的所有位标志，进行筛选前的准备：

```
db.set();
```

筛选时我们跳过了 `dynamic_bitset` 里的 0 和 1，因为它们既不是质数也不是合数，使用 `find_next()` 来查找集合中的元素，运用筛法删除它的倍数：

```
for (dynamic_bitset<>::size_type i = db.find_next(1);
     i != dynamic_bitset<>::npos ;
     i = db.find_next(i) )
{
    for (dynamic_bitset<>::size_type j = db.find_next(i);
         j != dynamic_bitset<>::npos ;
         j = db.find_next(j) )
    {
        if ( j % i == 0 )
        {
            db[j] = 0;                //被整除，非质数
        }
    }
}
```

当计算完毕后，`dynamic_bitset` 中所有为 1 的二进制位就是质数：

```
for (dynamic_bitset<>::size_type i = db.find_next(2);
     i != dynamic_bitset<>::npos ;
     i = db.find_next(i) )
{
    cout << i << ", ";              //输出质数
}
//end of main
```

输入 10，程序的运行结果是：3, 5, 7,

输入 50，程序的运行结果是：3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,

7.3 unordered

散列容器 (hash container) 是一种非常重要的容器类型，它通常比二叉树的存储方式可以提供更高的访问效率。C++98 标准中并未规定散列容器，因为它的提案来得太晚了，但这并不

妨碍编译器厂商实现自己的散列容器。SGISTL、STLport、Dinknumware STL 等实现都向用户提供了各自的散列容器。

由于散列容器没有标准规范，各家的实现也就各不相同，性能、接口也有差异，不过在容器的名字上倒是难得的一致，基本都是 `hash_map`、`hash_set`。

等到 C++ 标准委员会有了时间，开始整理散列容器的标准时，这才发现，数年间非标准实现已经广泛流传，接近“事实上的标准”，标准的名字 `hash_xxx` 已经被“鸠占鹊巢”。因此，C++ 标准委员会不得不选用了另一个 `unordered_xxx` 以避免与现存代码冲突，不过这个名字也更好地表现了散列容器的本质——它是无序的。

因为散列容器是无序的，因此不需要容器的元素类型提供 `operator<`，而是使用散列函数和键值比较的 `operator==`，比标准容器的要求要略微放宽一些。

`boost.unordered` 库提供了一个完全符合 C++ 新标准草案 (TR1) 的散列容器实现，包括无序集合 (`set`) 和无序映射 (`map`)。它们位于名字空间 `boost`，为使用 `unordered` 库，需要包含头文件 `<boost/unordered_set.hpp>` 或 `<boost/unordered_map.hpp>`，即：

```
#include <boost/unordered_set.hpp>
#include <boost/unordered_map.hpp>
using namespace boost;
```

由于 STLport 中也提供了散列容器的实现，故这里将同 `unordered` 一起对比介绍。

7.3.1 散列集合简介

`unordered` 库提供两个散列集合类 `unordered_set` 和 `unordered_multiset`，STLport 也提供 `hash_set` 和 `hash_multiset`。它们的接口、用法与 STL 里的标准关联容器 `set/multiset` 相同，只是内部使用散列表代替了二叉树实现，因此查找复杂度由对数降为常数。如果读者熟悉 STL 容器，那么就可以立刻开始使用散列容器。

`unordered_set/unordered_multiset` 的简要声明如下：

```
template <
    class Key,
    class Hash = boost::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<Key> >
class unordered_set;

template<
    class Key,
    class Hash = boost::hash<Key>,
```

```
class Pred = std::equal_to<Key>,
class Alloc = std::allocator<Key> >
class unordered_multiset;
```

与 `std::set` 相比, `unordered_set` 的模板增加了一个计算散列值的模板类型参数, 通常是 `boost::hash`, 最好不要去改变它。另外比较谓词参数是使用 `std::equal_to<>`, 而不是 `set` 中的 `less<>`, 这是因为散列容器不需要保持有序。

7.3.2 散列集合的用法

`hash_set/unordered_set` 具有与 `std::set` 相同的功能, 可以用 `size()` 获得容器大小, 用 `empty()` 判断空, 支持比较操作符, 可以用 `count()`、`find()`, 大多数应用 `std::set` 的场景都能用 `hash_set/unordered_set` 替换。

唯一需要注意的是散列容器的无序性, 我们不能在散列容器上使用 `binary_search`、`lower_bound` 和 `upper_bound` 这样用于已序区间的算法, 散列容器自身也不提供这样的成员函数。

下面的代码示范了 `hash_set/unordered_set` 的一些用法, `unordered_multiset` 用法与之类似, 只不过它可以容纳重复的元素。

程序定义了一个模板函数 `hash_func()`, 用以操作 `hash_set/unordered_set`, 两者的表现是完全一样的。首先使用 `boost::assign` 初始化散列集合, 以迭代器遍历输出, 然后用 `size()` 显示容器大小, 用 `clear()` 清空集合, 再用 `insert()` 插入两个元素, 用 `find()` 查找元素, 最后用 `erase()` 删除一个元素, 这些都是标准容器的标准操作。

```
#include <hash_set> //stlport 的散列集合
#include <boost/unordered_set.hpp> //boost 的散列集合
using namespace boost;
template<typename T> //模板参数要求为散列集合类型
void hash_func()
{
    using namespace boost::assign;

    T s = (list_of(1),2,3,4,5); //初始化数据
    for (T::iterator p = s.begin(); p != s.end(); ++p) //可以使用迭代器遍历集合
    {
        cout << *p << " ";
    }
    cout << endl;
    cout << s.size() << endl;

    s.clear(); //清空集合
    cout << s.empty() << endl; //判断集合是否空
```

```

s.insert(8); //同标准容器一样使用 insert() 函数
s.insert(45);
cout << s.size() << endl;
cout << *s.find(8) << endl;

s.erase(45);
}

```

然后我们在 `main()` 函数中使用 `hash_set<int>` 和 `unordered_set<int>` 调用这个模板函数:

```

int main()
{
    hash_func<hash_set<int> >();
    hash_func<unordered_set<int> >();
}

```

7.3.3 散列映射简介

`unordered` 库提供两个散列映射类 `unordered_map` 和 `unordered_multimap`, `STLport` 也提供 `hash_map`/`hash_multimap`。它们的接口、用法与 STL 里的标准关联容器 `map`/`multimap` 相同, 只是内部使用散列表代替了二叉树实现, 模板参数多了散列计算函数, 比较谓词使用 `equal_to<>`。

`unordered_map` 和 `unordered_multimap` 的简要声明如下:

```

template <
    class Key, class Mapped,
    class Hash = boost::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<Key> >
    class unordered_map;

template<
    class Key, class Mapped,
    class Hash = boost::hash<Key>,
    class Pred = std::equal_to<Key>,
    class Alloc = std::allocator<Key> >
    class unordered_multimap;

```

7.3.4 散列映射的用法

`unordered_map`/`hash_map` 属于关联式容器, 采用 `std::pair` 保存 key-value 形式的数 据, 可以理解成一个关联数组, 提供 `operator[]` 重载, 用法与标准容器 `map` 完全相同。

示范 `unordered_map`/`hash_map` 一些用法的代码如下, `unordered_multimap` 用法与

之类似，但因为它允许有重复的 key-value 映射，因此不提供 operator[]。

```
#include <hash_map>
#include <boost/unordered_map.hpp>
using namespace boost;

int main()
{
    using namespace boost::assign;

    unordered_map<int, string> um =                //使用 assign 初始化
        map_list_of(1,"one")(2, "two")(3, "three");

    um.insert(make_pair(10,"ten"));                //可以使用 insert() 函数
    cout << um[10] << endl;                        //使用 operator[] 访问元素
    um[11] = "eleven";                             //关联数组用法
    um[15] = "fifteen";

    BOOST_AUTO(p, um.begin());                    //使用 BOOST_AUTO 获得迭代器
    for (p; p != um.end(); ++p)
    {   cout << p->first << "-" << p->second << ", ";   }
    cout << endl;

    um.erase(11);                                 //删除键值为 11 的元素
    cout << um.size() << endl;                      //输出 5

    hash_map<int, string> hm =                    //stlport 提供的散列映射类
        map_list_of(4,"four")(5, "five")(6, "six");

    BOOST_AUTO(q, hm.begin());                    //获得迭代器
    for (q; q != hm.end(); ++q)                  //迭代器遍历
    {   cout << q->first << "-" << q->second << ", ";   }
    cout << endl;
}
```

代码运行结果如下：

```
ten
11-eleven,1-one,2-two,3-three,15-fifteen,10-ten,
5
4-four,5-five,6-six,
```

7.3.5 性能比较

一般情况下散列容器的性能表现要比标准容器更好，下面的代码简单测试了 std::multiset、stlport::multiset 和 boost::unordered_multiset 三者的性能。

程序使用 boost 随机数库 random(参见 9.4 小节, 357 页)向容器插入 10000 个 1 到 100 之间的整数, 然后执行 count 和 find 操作, 使用 2.7.10 小节实现的高精度 ptimer 来计时。

```
#include <typeinfo>
#include <set>
#include <hash_set>
#include <boost/unordered_set.hpp>
#include <boost/random.hpp>
using namespace boost;

template<typename T>
void fill_set(T &c) //向容器插入数据
{
    variate_generator<mt19937, uniform_int<> >
        gen(mt19937(), uniform_int<>(0, 100));

    for (int i = 0 ; i < 10000; ++i) //插入一万个整数
        c.insert(gen());
}

template<typename T>
void test_perform() //性能测试函数
{
    T c;
    cout << typeid(c).name() << endl;
    {
        ptimer t;
        fill_set(c); //测试插入数据的性能
    }
    {
        ptimer t;
        c.count(10); //测试 count() 的性能
    }
    {
        ptimer t;
        c.find(20); //测试 find() 的性能
    }
}

int main()
{
    test_perform<multiset<int> >(); //测试标准集合
    test_perform<hash_multiset<int> >(); //测试 stlport 散列集合
    test_perform<unordered_multiset<int> >(); //测试 unordered 集合
}
```

表 7-1 展示了一次测试的运行结果, 单位是秒。

表 7-1 一次测试的运行结果

	Multiset	hash_multiset	unordered_multiset
insert	0.125	0.126	0.062
Count	0.000	0.000	0.000
Find	0.000	0.000	0.000

当然，这个测试很简单也很不全面，但应该能够给出一个散列容器性能比较直观的印象。

7.3.6 高级议题

本小节讨论关于 unordered 库的一些高级议题。

内部数据结构

unordered 库使用“桶”(bucket)来存储元素，散列值相同的元素被放入同一个桶中。当前散列容器的桶的数量可以用成员函数 bucket_count() 来获得， bucket_size() 返回桶中的元素数量。例如：

```
unordered_set<int> us = {list_of(1),2,3,4);
cout << us.bucket_count() << endl;
for (int i = 0; i < us.bucket_count(); ++i)           //访问每个桶
{ cout << us.bucket_size(i) << ", "; }
```

当散列容器中有大量的数据时，桶中的元素数量也会增多，会造成访问冲突。为了提高散列容器的性能，unordered 库会在插入元素时自动增加桶的数量。用户不能直接指定桶的数量（由散列容器自己管理会更好），但可以在构造函数或者 rehash() 函数指定最小的桶的数量。例如：

```
unordered_set<int> us(100);                               //要求使用 100 个桶存储数据
us.rehash(200);                                           //改为使用 200 个桶
```

C++0x TR1 草案还规定有一个函数 max_load_factor()，它可以获取或设定散列容器的最大负载因子，即桶中元素的最大平均数量。通常最大负载因子都是 1，用户不应当去改变它，过大或过小都没有意义。

支持自定义类型

unoredered 库支持 C++ 内建类型 and 大多数标准库容器，但不支持用户自定义的类型，因为它无法计算自定义类型的散列值。

如果要使 unordered 支持自定义类型，需要定制类模板的第二个和第三个参数，也就是提供散列函数和相等比较谓词。

相等比较谓词很容易实现, `unordered` 库默认使用 `std::equal_to`, 这是一个标准库中的函数对象, 它使用 `operator==`, 只要自定义类实现了这个操作符就可以了, 不必再特意编写一个函数对象。如果需要用个特别的相等判断规则, 那么可以额外写函数对象, 传递给 `unordered` 容器。

散列函数则是必须要实现的, 这也是为什么它被放在模板参数列表前面的原因。我们需要使用 `boost.hash` 库来计算自定义类型的散列值。最简单的使用方法是编写一个 `hash_value()` 函数, 创建一个 `hash` 函数对象, 然后使用它的 `operator()` 返回散列值。

下面的代码定义了一个类 `demo_class`, 它实现了 `operator==` 和散列函数, 可以被 `unordered` 所容纳:

```
#include <boost/unordered_set.hpp>
#include <boost/functional/hash.hpp>
struct demo_class
{
    int a;
    friend bool operator==(const demo_class& l, const demo_class& r)
    { return l.a == r.a; }
};
size_t hash_value(demo_class & s)
{ return boost::hash<int>()(s.a); }
int main()
{
    unordered_set<demo_class> us;
}
```

与 TR1 的差异

`boost.unordered` 基本上是依据 C++0x 标准草案来实现的, 可以说是完全符合标准的规定。但它有个小的“扩充”, 增加了比较操作符 `operator==` 和 `operator!=`, 可以比较两个容器包含的元素是否相等。

```
using namespace boost::assign;
unordered_set<int> us1 = list_of(1)(2)(3);
unordered_set<int> us2 = list_of(3)(2)(1);
assert(us1 == us2);

unordered_map<int, string> um1 = map_list_of(1, "11")(2, "22");
unordered_map<int, string> um2 = map_list_of(1, "one")(2, "two");
assert(um1 != um2);
```

这两个操作符在实际工作中很有用, 但要注意: 它不属于 C++0x 标准, 如果将来程序要切换到新标准可能会遇到不可移植的问题。

7.4 bimap

C++标准提供了映射型容器map和multi_map,它们就像是一个关联数组,把一个元素(key)映射到另一个元素(value),但这种映射关系是单向的,只能是key到value,而不能反过来。

boost.bimap 扩展了标准库的映射型容器,提供双向映射的能力,功能强大,其接口被特意设计为符合 STL 规范,以减少学习的负担。

bimap 位于名字空间 boost::bimaps, 但被 using 语句引入了名字空间 boost, 为了使用 bimap 组件, 需要包含头文件<boost/bimap.hpp>, 即:

```
#include <boost/bimap.hpp>
using namespace boost;
```

7.4.1 类摘要

bimap 是一个具有复杂内部结构的类,它基于 boost.multi_index 库实现,类摘要如下:

```
template<class LeftCollectionType, class RightCollectionType>
class bimap
{
public:

    //左右视图
    left_map left;
    right_map right;

    // 构造与赋值
    bimap();
    template< class InputIterator >
    bimap(InputIterator first,InputIterator last);
    bimap(const bimap &);
    bimap& operator=(const bimap& b);

    //标签操作
    template< class Tag >
    struct map_by;
    template< class Tag >
    map_by<Tag>::type by();
    template< class Tag >
    const map_by<Tag>::type & by() const;
    template< class Tag, class IteratorType >
    map_by<Tag>::iterator project(IteratorType iter);

    //迭代器投射操作
```

```

template< class IteratorType >
left_iterator project_left(IteratorType iter);
template< class IteratorType >
right_iterator project_right(IteratorType iter);
template< class IteratorType >
iterator project_up(IteratorType iter);
};

```

7.4.2 基本用法

bimap 可以容纳两个类型的元素，是这两种元素关系的集合，这是 bimap 的基本视图。此外，这个关系还可以有两个视图：左视图和右视图，分别用成员变量 left 和 right 访问，相当于两个不同方向的 std::map，其用法也同 std::map 一样。

对于 bimap<X,Y>，bimap.left 相当于 map<X,Y>，bimap.right 相当于 map<Y,X>。例如，下面的代码定义了一个 int 和 string 的 bimap，分别用左视图和右视图插入一些数据，最后用左视图来查看：

```

#include <boost/bimap.hpp>
using namespace boost;
int main()
{
    bimap<int, string> bm; //定义一个双向视图对象

    //使用左视图 map<int, string>
    bm.left.insert(make_pair(1, "111")); //插入数据
    bm.left.insert(make_pair(2, "222"));

    //使用右视图 map<string, int >
    bm.right.insert(make_pair("string", 10)); //插入数据
    bm.right.insert(make_pair("bimap", 20));

    //对左视图使用迭代器迭代
    for (BOOST_AUTO(pos, bm.left.begin());
         pos != bm.left.end(); ++pos)
    {
        cout << "left[" << pos->first << "]= "
              << pos->second << endl;
    }
}

```

程序的运行结果如下：

```

left[1]=111
left[2]=222
left[10]=string
left[20]=bimap

```

bimap 的左右视图的基本用法与 `std::map` 相同，但也有一些区别。首先，因为它的双向性，key/value 值对必须都是唯一的，插入任何一个重复的值都将是无效操作，例如：

```
bm.left.insert(make_pair(2, "222"));           //左视图插入数据
bm.left.insert(make_pair(2, "333"));           //无效插入操作
bm.right.insert(make_pair("string", 2));        //无效插入操作
```

其次，bimap 的两个视图的迭代器返回的值都是常量对象，相当于 `pair<const, const>`，不能如 `std::map` 那样修改 value 的值。代码

```
bm.right.begin()->second = 234;
```

会导致编译错误，提醒你不能给一个常量赋值。

最后一点区别是 bimap 不能使用 `operator[]` 和 `at()`，不能像关联数组那样使用。^①

bimap 有两个内部类型 `value_type` 和 `relation`，它是 bimap 关系集合元素的类型，可以在不使用左右视图的情况下像操作集合一样插入 bimap 元素：

```
bimap<int, string> bm;
typedef bimap<int, string>::value_type vt;
bm.insert(vt(3, "333"));
```

7.4.3 值的集合类型

bimap 以自然的方式扩展了映射的语义，将 `std::map` 的 key/value 值的映射关系扩展为左组和右组两个集合类型的映射关系。`std::map` 的声明是：

```
map<X, Y>
```

而 bimap 的声明是：

```
bimap<collection_type_of<X>, collection_type_of<Y> >.
```

在这个概念下，`std::map` 和 `std::multi_map` 的左组是一个有序的集合，而右组则无任何约束，它们对应的 bimap 是：

```
bimap<set_of<X>, unconstrained_set_of<Y> >和
bimap<multiset_of<X>, unconstrained_set_of<Y> >
```

bimap 定义的集合类型包括如下：

① 但这个区别不是决定性的，通过配置模板参数可以使 bimap 支持 `operator[]` 和 `at()`，之后我们会看到这样的用法。

- `set_of`: 可以用作键值 (索引), 有序且唯一, 视图相当于 `map`;
- `multiset_of`: 可以用作键值 (索引), 有序, 视图相当于 `multimap`;
- `unordered_set_of`: 可以用作键值 (索引), 无序且唯一, 视图相当于 `unordered_map`;
- `unordered_multiset_of`: 可以用作键值 (索引), 无序, 视图相当于 `unordered_multimap`;
- `list_of`: 不能用作键值 (索引), 序列集合, 无对应的 STL 容器;
- `vector_of`: 不能用作键值 (索引), 随机访问集合, 无对应的 STL 容器;
- `unconstrained_set_of`: 不能用作键值 (索引), 无任何约束关系, 无对应的 STL 容器。

在 `bimap` 的模板参数列表中使用这些集合类型, 就可以任意定义 `bimap` 的映射关系; 如果不指定集合类型, `bimap` 将默认使用 `set_of` 类型。示范一些 `bimap` 定义的代码如下:

```
//左组是有序集合, 右组是无序集合
bimap<int, unordered_set_of< string> > bm1;

//左组和右组都是有序多值集合
bimap< multiset_of<int>, multiset_of< string> > bm2;

//左组是无序集合, 右组是一个序列, 只能有左视图
bimap< unordered_set_of<int>, list_of< string> > bm3;

//左组是随机访问集合, 右组无约束
bimap< vector_of<int>, unconstrained_set_of< string> > bm4;
```

这些集合类型位于名字空间 `boost::bimaps`, 但不在 `<boost/bimap.hpp>` 中。为了使用这些集合类型, 需要包含各自的同名头文件, 如 `list_of` 集合类型需要包含 `<boost/bimap/list_of.hpp>`。

7.4.4 集合类型的用法

`bimap` 的集合类型引入了很多的变化, 使 `bimap` 的类型产生大量的组合, 变得十分复杂。下面仅介绍基本的用法。

如果要使用双向映射, 左组和右组必须都是有序的。如果其中的一组是无序的, 那么就没有相应方向的映射视图; 如果两组都是无序的, 那么只能用 `bimap` 的集合视图来访问元素。

我们先实现一个辅助函数 `print_map()`, 它遍历一个 `map`, 输出其中的所有元素, 将被用于简化接下来的示范代码:


```
template<typename T>
void print_map(T &m)
{
    for (BOOST_AUTO(pos, m.begin());
         pos != m.end(); ++pos)
    {
        cout << pos->first << "<-->" << pos->second << endl;
    }
}
```

接下来我们声明一个无序多值的 bimap，它的左组和右组可以插入任意可重复的值：

```
bimap<unordered_multiset_of<int>, unordered_multiset_of< string> > bm;
bm.left.insert(make_pair(1, "111"));
bm.left.insert(make_pair(2, "222"));
bm.left.insert(make_pair(2, "555"));

bm.right.insert(make_pair("string", 10));
bm.right.insert(make_pair("bimap", 20));
bm.right.insert(make_pair("bimap", 2));

print_map(bm.left);
```

输出将会是：

```
1<-->111
2<-->bimap
2<-->555
2<-->222
10<-->string
20<-->bimap
```

如果声明一个右值的集合类型是序列或者随机访问类型的 bimap：

```
bimap<set_of<int>, vector_of< string> > bm;
```

那么它只能使用左视图，企图使用右视图会引发编译错误。但它有一个优点，可以在左视图使用 operator[] 和 at()，像操作一个关联数组。如果左组类型是 multiset_of/unordered_multiset_of 则无法使用 operator[]：

```
bm.left.insert(make_pair(1, "111"));
bm.left[2] = "222";
bm.left[300] = "bimap";

//bm.right.insert(make_pair("string", 10));           //编译错误

print_map(bm.left);
```

输出是：

```
1<-->111
```

```
2<-->222
300<-->bimap
```

如果左右两组都不是 `set` 类型，那么左右视图都将无法使用，`bimap` 将变得很难操纵，最好不要这样使用 `bimap`。

7.4.5 使用标签类型

`bimap` 的左视图和右视图分别用 `bimap.left` 和 `bimap.right` 来访问，通常这很方便，也很容易理解。但它太平常而缺乏具体的含义，不能够准确地表达左右视图的用途，如果程序中存在很多 `bimap` 实例，程序员很容易在不同的 `bimap` 之间造成混乱。

最简单的解决办法是使用注释，说明 `bimap` 的用途以及它的左右视图的含义，高级的 C++ 集成开发环境在编写代码时会自动地给出提示。但这不是根本的解决办法，不可能给所有 `bimap` 都提供良好的注释，而且总会有人有意或无意地忽略注释。

`bimap` 可以使用 `bimaps::tagged` 类给左组和右组数据在语法层面上贴“标签”，从而更清晰地说明左右视图的含义。

标签实际上是对 `bimap` 集合数据类型的一个包装，声明如下：

```
template< class Type, class Tag >
struct tagged
{
    typedef Type value_type;
    typedef Tag tag;
};
```

其用法与 `singleton_pool` (3.11 小节, 91 页) 和 `exception` (4.9 小节, 136 页) 类似，例如：

```
tagged<int, struct id>
```

包装了 `int` 类型，标签名字是 `id`。

加了标签的数据类型可以直接作为 `bimap` 的 `key/value`，也可以配合集合类型使用，例如：

```
bimap<tagged<int, struct id>, vector_of< string> > > bm1;
bimap<multiset_of<tagged<int, struct id> >,
    unorderedset_of<tagged< string, name> > > > bm2;
```

加入标签后的 `bimap` 依然可以使用 `left` 和 `right` 访问左视图和右视图，但同时还可以用模板成员函数 `by<tagged>()` 通过标签来访问。例如：

```
#include <boost/bimap.hpp>
```

```
using namespace boost::bimaps;
int main()
{
    bimap<tagged<int, struct id>, tagged< string, struct name> > bm;

    bm.by<id>().insert(make_pair(1, "samus"));           //使用标签"id"
    bm.by<id>().insert(make_pair(2, "adam"));           //相当于左视图

    bm.by<name>().insert(make_pair("link", 10));         //使用标签"name"
    bm.by<name>().insert(make_pair("zelda", 11));        //相当于右视图

    print_map(bm.by<name>());
}
```

程序的运行结果如下：

```
adam<-->2
link<-->10
samus<-->1
zelda<-->11
```

bimap 的许多操作都支持使用标签，用法与普通的左右视图差不多，只是要加上模板标签，例如左右视图的类型变成 `map_by<left>/map_by<right>`，迭代器访问变成 `pos->get<left>()/ pos->get<right>()`，通常这要比原来不带标签的写法更清楚，不容易出错。

7.4.6 使用 assign 库

bimap 的接口兼容 STL，因此可以很容易地使用 assign 库（4.4 小节，106 页）为它赋初值，使用时需要注意 bimap 左右视图的集合类型，选择恰当的 assign 库工具。

如果我们有一个 bimap 类型定义 `bm_t`：

```
typedef bimap<multiset_of<int>, vector_of< string> > bm_t;
```

那么可以直接使用 `assign::list_of` 为它初始化：

```
bm_t bm = assign::list_of<bm_t::relation>(1, "111")(2, "222");
```

注意：我们必须使用 `assign` 名字空间来使用 `list_of`，即 `assign::list_of`，因为在 `boost::bimaps` 名字空间内存在同名的集合类型 `list_of`，如果不使用名字空间限定会引发编译错误。

`bm` 的左视图是个 `multiset`，可以使用 `assign` 库的 `insert()` 函数赋值，这里不需要使用 `assign` 名字空间限定：

```
insert(bm.left)(3, "333")(4, "444");
```

bm 的右视图是个 vector，可以使用 `push_back()` 函数赋值。如果右视图是 `list_of` 定义的，那么还可以使用 `push_front()` 函数：

```
push_back(bm.right) ("555", 5) ("666", 6);
```

7.4.7 查找与替换

当 bimap 的左右视图是 set 时，可以如 `std::map` 一样调用成员函数 `find()`，以键值为索引查找元素，例如：

```
#include <boost/bimap.hpp>
#include <boost/assign.hpp>
int main()
{
    using namespace boost::bimaps;
    typedef bimap<int, string > bm_t;

    using namespace boost::assign;
    bm_t bm = assign::list_of<bm_t::relation>(1, "mario")(2, "peach");
                                                    //使用 relation 作为元素类型初始化

    BOOST_AUTO(pos, bm.left.find(1));           //左视图查找键值 1
    cout << "[" << pos->first
        << "]" = " << pos->second << endl;
    BOOST_AUTO(pos2, bm.right.find("peach"));   //右左视图查找键值"peach"
    cout << "[" << pos2->first
        << "]" = " << pos2->second << endl;
}
```

程序的运行结果如下：

```
[1]=mario
[peach]=2
```

bimap 还通过视图提供 `std::map` 所没有的替换功能，可以直接修改 key 或者 value，这在很多时候都是非常方便的功能。视图的成员函数 `replace_key()` 可以替换键，`replace_data()` 可以替换 value，它们接受一个指示位置的迭代器和值作为参数，返回 bool 表示替换是否成功。例如：

```
BOOST_AUTO(pos, bm.left.find(1));
bm.left.replace_key(pos, 111);                 //替换键 1->111
bm.left.replace_data(pos, "luigi");            //替换值 peach->luigi
```

将把元素 `[1]=mario` 替换成 `[111]=luigi`。

除了使用 `replace_key()` 和 `replace_data()` 函数，bimap 还使用了 `boost.lambda` 库

提供更快速的修改函数 `modify_key()` 和 `modify_data()`。它们的行为与 `replace` 略有不同：如果因为集合类型的约束导致修改失败将会删除修改的元素，而不是抛出异常通知用户，所以在使用 `modify` 时要小心。

`modify` 功能需要包含额外的头文件 `<boost/bimap/support/lambda.hpp>`，它使用了 `lambda` 表达式，在名字空间 `bimaps` 定义了占位符 `_key` 和 `_data`，用法与 `replace` 基本一致。

使用 `modify`，刚才的替换代码可以写成：

```
#include <boost/bimap/support/lambda.hpp>
...
BOOST_AUTO(pos, bm.left.find(1));
using namespace boost::bimaps;
bm.left.modify_key(pos, _key = 111);
bm.left.modify_data(pos, _data = "luigi");
```

因为这里 `bimap` 的左右组类型都是 `set`，不允许有重复的值，如果 `modify` 修改 `key` 时违反这个约束，将导致数据丢失：

```
bm.left.modify_key(pos, _key = 2);           //修改 key=2, 与集合中其他数据冲突
assert(bm.left.size() == 1);                 //修改的元素被删除
assert(bm.right.find("mario") == bm.right.end());
```

修改 `value` 也会导致同样的结果：

```
bm.left.modify_data(pos, _data = "peach");    //冲突，修改的元素将被删除
```

但如果我们变动 `bimap` 的定义，把集合类型改为允许重复，那么 `modify` 将不会发生这样的错误：

```
typedef bimap<multiset_of<int>, string > bm_t;
...
bm.left.modify_key(pos, _key = 2);
assert(bm.left.size() == 2);
assert(bm.right.find("mario") != bm.right.end());
```

7.4.8 投射

`bimap` 提供三个成员函数 `project_left()`、`project_right()` 和 `project_up()`，可以把 `bimap` 的迭代器分别投射到左视图、右视图和关系视图上，这允许用户以一种视图查找，然后很容易地转换到其他视图再进行其他操作。

沿用 7.4.6 小节初始化的 `bimap` 对象，投射的使用例子如下：

```
BOOST_AUTO(left_pos, bm.left.find(3));
```

```
BOOST_AUTO(right_pos, bm.project_right(left_pos));
cout << "right:[" << right_pos->first
    << "]" << right_pos->second;
```

输出是:

```
right:[333]=3
```

投射也可以使用 bimap 的标签用法, 例如:

```
typedef bimap<set_of<tagged<int,struct id> >,
    multiset_of< tagged<string,struct name> > > bm_t;

using namespace boost::assign;
bm_t bm = assign::list_of<bm_t::relation>(1, "mario")(2, "peach");
insert(bm.by<id>())(3, "wario")(4, "luigi");
insert(bm.by<name>())("yoshi", 5)("olima", 6);

BOOST_AUTO(right_pos, bm.by<name>().find("yoshi"));
BOOST_AUTO(left_pos, bm.project<id>(right_pos));
++left_pos;
cout << "left:[" << left_pos->get<id>()
    << "]" << left_pos->get<name>();
```

这段代码中的 bimap 类型与之前的稍有不同, 集合类型分别是 set_of 和 multiset_of, 因此 assign 库不能使用 push_back() 或者 push_front(), 只能使用 insert()。除了投射, 这段代码还演示了带标签 bimap 的其他用法。

7.4.9 高级议题

bimap 是一个复杂的组件, 本小节讨论关于它的一些高级议题, 但没有覆盖 bimap 的全部内容。

视图的值类型

使用视图插入元素时可以使用 make_pair, 也可以使用视图的内置类型 bimap<>::left_value_type 或者 bimap<>::right_value_type, 它们通常比 make_pair() 速度更快, 因为可以避免对象的拷贝构造函数的调用。例如:

```
typedef bimap<int, string> bm_t;
bm_t bm;
bm.left.insert(bm_t::left_value_type(1, "one"));
bm.right.insert(bm_t::right_value_type("two", 222));
```

如果使用了标签, 那么值类型就是 biamp<>::map_by<left>::value_type 和 biamp<>::map_by<right>::value_type。

值的集合类型扩展

bimap 的值集合类型除了接受数据类型作为模板参数外，也可以接受其他的模板参数，就如同是一个标准的容器类型。例如 set_of 可以接受一个比较谓词，unorderedset_of 可以如 unordered 库（参见 7.3 小节，253 页）那样接受散列函数对象。例如，下面的代码定义了一个使用 std::greater<> 作为排序准则的 bimap：

```
bimap<set_of<int, std::greater<int> >, unorderedset_of<string> >
```

使用这个功能，可以让 bimap 能够有更多的灵活性，能够容纳更多的自定义类型，扩大它的使用范围。但这也会使 bimap 的声明和用法变得更加复杂，更加难以理解，应该在确实有必要的时候才使用，而不是仅仅为了炫耀技术。

关系的集合类型

除了左组和右组两个模板类型参数，bimap 还可以接受更多的模板参数。第三个模板参数可以指定 bimap 的集合视图保存关系的形式，默认是 left_based。也可以修改成 list_of_relation、set_of_relation 或 unordered_set_of_relation 等。恰当地配置关系集合类型与类型集合类型可以获得更快的索引速度。

但与值的集合类型扩展一样，bimap 为用户提供了大量的可配置选项，它们不同的搭配产生的组合是一个庞大的数字，会使 bimap 的复杂度成倍地提高。

7.5 circular_buffer

circular_buffer 实现了循环缓冲区的数据结构，支持标准的容器操作（如 push_back）但大小是固定的，当到达容器末尾时将自动循环利用容器另一端的空間。

circular_buffer 位于名字空间 boost，为了使用 circular_buffer 组件，需要包含头文件 <boost/circular_buffer.hpp>，即：

```
#include <boost/circular_buffer.hpp>
using namespace boost;
```

7.5.1 类摘要

circular_buffer 是一个符合 STL 规范的容器，因此具有一个内存分配器的模板类型参数，很多构造、赋值操作也都有一个分配器参数，但通常我们不应该去改变它，故类摘要中将这“噪声”都省略了。

`circular_buffer` 的接口很多,但大部分都与 STL 标准容器类似,下面的类摘要仅列出一些最重要的操作:

```
template <class T, class Alloc>
class circular_buffer
{
public:
    explicit circular_buffer();
    explicit circular_buffer(capacity_type capacity);
    circular_buffer(size_type n, const_reference item);
    template <class InputIterator>
    circular_buffer(InputIterator first, InputIterator last);
    ~circular_buffer();

    iterator begin();
    iterator end();
    reference operator[](size_type index);
    reference at(size_type index);
    reference front();
    reference back();

    void rotate(const_iterator new_begin);
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    bool full() const;
    pointer linearize();

    void assign(size_type n, const_reference item);
    void push_back(const_reference item = value_type());
    void push_front(const_reference item = value_type());
    void pop_back();
    void pop_front();
    iterator insert(iterator pos, const_reference item);
    iterator erase(iterator pos);
}
```

7.5.2 用法

`circular_buffer` 被设计成一种可以与标准容器无缝结合使用的容器,实现了一个大小固定的循环队列,就像是一个 `deque` 和 `stack` 的混合体,可以像普通双端队列一样执行 `push_back()`、`push_front()`、`insert()` 等操作来增加元素,也可以像栈一样用 `pop_back()`、`pop_front()` 来弹出元素。它也具有标准容器的共通功能,如判断大小,获取迭代器,删除元素、`operator[]` 等。例如:

```
circular_buffer<int> cb(5);           //声明一个大小为 5 的循环缓冲区
```



```

assert(cb.empty()); //缓冲区目前无数据

cb.push_back(1); //向后端添加元素 1
cb.push_front(2); //向前端添加元素 1
assert(cb.front() == 2);
cb.insert(cb.begin(), 3); // 向前端添加元素 3

//可以使用迭代器遍历容器
for (BOOST_AUTO(pos, cb.begin()); pos != cb.end(); ++pos)
{ cout << *pos << ", "; }
cout << endl; //输出 3,2,1

cb.pop_front(); //弹出首元素 3
assert(cb.size() == 2);
cb.push_back(); //弹出末元素 1
assert(cb[0] == 2);

```

circular_buffer 还重载了比较操作符，可以对两个同样大小的 circular_buffer 对象进行字典序比较：

```

using namespace boost::assign;
circular_buffer<int> cb1( (list_of(1),2,3));
circular_buffer<int> cb2( (list_of(3),4,5));
circular_buffer<int> cb3 = cb1; //从 cb1 拷贝构造

assert(cb1 < cb2);
assert(cb1 == cb3);

```

7.5.3 环型缓冲区

circular_buffer 的特殊之处在于它内部存储数据的方式，内存空间不是动态增长的，而是循环使用的，可以把 circular_buffer 内部想像成一个首尾相连的环，当元素数量达到容器的容量上限时将自动重用最初的空间。

为了简化接下来的代码，我们抽取之前遍历 circular_buffer 的代码，定义一个 print() 函数来输出其内部的所有元素：

```

template<typename T>
void print(T& cb)
{
    for (BOOST_AUTO(pos, cb.begin()); pos != cb.end(); ++pos)
    { cout << *pos << ", "; }
    cout << endl;
}

```

下面的代码声明了一个大小为 3 的 circular_buffer，使用 print() 来输出循环缓冲区内元素的变化情况：

```
using namespace boost::assign;

//circular_buffer 可以使用 assign 库初始化
circular_buffer<int> cb( (list_of(1),2,3));
print(cb); //1,2,3, 此时缓冲区已满

cb.push_back(4); //4 将覆盖最开始的 1,
print(cb); //2,3,4, begin() 从 2 开始

cb.push_back(5); //5 将覆盖最开始的 2,
print(cb); // 3,4,5, begin() 从 3 开始

cb.pop_front(); //弹出最开始的 3
print(cb); //4,5, 现在 circular_buffer 只有两个元素
```

circular_buffer 为这种环形结构提供以下几个特殊的操作函数:

- full() 判断缓冲区是否已满;
- linearize() 可以把缓冲区线性化成一个连续的普通数组, is_linearized() 可以检测缓冲区可否线性化;
- rotate() 从指定的迭代器位置旋转整个缓冲区。

例如:

```
using namespace boost::assign;
circular_buffer<int> cb( (list_of(1),2,3,4,5));
assert(cb.full());
print(cb); //1,2,3,4,5,

int *p = cb.linearize(); //获取线性数组
assert(p[0]== 1 && p[3] == 4);

cb.rotate(cb.begin()+ 2); //从第三个位置开始旋转
print(cb); //3,4,5,1,2,
```

在使用 circular_buffer 需要注意一点: 虽然 circular_buffer 内部存储是循环的, 但它的迭代器不是循环的, 迭代器仅在 begin() 和 end() 区间内有效, 超过这个区间迭代器就会失效。

例如, 下面的代码声明了一个容量为 10 的 circular_buffer, 当迭代器超过 end() 时会引发断言异常:

```
using namespace boost::assign;
circular_buffer<int> cb = (list_of(0),1,2,3,4,5,6,7,8,9);
BOOST_AUTO(pos, cb.begin());
```

```
assert( *(pos + 3) == 3);           //正常
cout << *(pos + 10) << endl;       //迭代器失效，引发异常
```

7.5.4 空间优化型缓冲区

`circular_buffer` 在创建时一次性分配所需的内存，这是标准容器的通常做法，但对于循环缓冲区数据结构可能不一定最合适。因此，`circular_buffer` 库还提供了一个 `circular_buffer_space_optimized` 类，它是 `circular_buffer` 的适配器，只有在确实需要时才分配内存空间，而且当容器内元素减少时也会自动释放内存。

`circular_buffer_space_optimized` 的用法几乎与 `circular_buffer` 完全相同：

```
using namespace boost::assign;

circular_buffer_space_optimized<int> cb( 10);
push_back(cb)(1),2,3,4;
assert(cb.size() == 4);
assert(cb.capacity() == 10);

cb.resize(100, 10);
assert(cb.size() == cb.capacity());
```

当程序需要容纳大量的元素，但多数情况下仅保存较少数量元素的时候就可以使用 `circular_buffer_space_optimized`，它可以有效地减少内存的占用，提高程序的运行效率。

7.6 tuple

`tuple`（元组）定义了一个有固定数目元素的容器，其中的每个元素类型都可以不相同，这与其他容器有着本质的区别。例如，标准容器 `std::vector` 或 `boost::array` 虽然可以容纳很多元素，但所有的元素都必须是同一类型。`tuple` 很有用，它是 `std::pair` 的泛化，可以从函数返回任意数量的值，也可以代替 `struct` 组合数据。

很多其他的编程语言（如 Python）都有内建的 `tuple` 类型，但 C++98 标准没有。`boost.tuple` 使用库的方式为 C++ 增加了这种很有用的数据结构，它已经被收入 C++ 0x TR1 标准草案。

`tuple` 位于名字空间 `boost::tuples`，但大部分功能被 `using` 语句引入了名字空间 `boost`，为了使用 `tuple` 组件，需要包含头文件 `<boost/tuple/tuple.hpp>`，即：

```
#include <boost/tuple/tuple.hpp>
using namespace boost;
```

7.6.1 最简单的 tuple: pair

标准库中提供的模板类 `std::pair` 是 `tuple` 的特例: 2-tuple——仅能持有两个成员的元组, 了解 `pair` 有助于我们接下来 `tuple` 的学习。

`pair` 的声明如下:

```
template <typename T, typename U>
struct pair
{
    T first;
    U second;
};
```

像这样定义一个 `pair` 对象:

```
pair<int, string> a_pair;
```

`pair` 的两个成员可以分别用 `first` 和 `second` 来访问, 它们可以是任意的类型。`pair` 重载了所有比较操作符, 因此只要元素支持比较, 那么同型的对象之间可以进行比较操作。工厂函数 `make_pair()` 还可以根据参数自动推导类型, 创建 `pair` 对象。

`pair` 是标准库中很多组件的基础, 例如关联容器 `map` 和 `multi_map` 就使用 `pair` 来保存不同元素之间的映射关系。`boost` 库的很多组件也使用了 `pair`, 如 5.5.3 小节 (199 页) 介绍的 `sub_match`, 因为这种持有两个不同类型对象的功能确实非常方便有用。

2-tuple 就已经如此有用, 那么可以持有任意数量成员的 `tuple` 又会是怎样的呢?

7.6.2 类摘要

`tuple` 是 `pair` 的泛化, 声明很类似 `pair`, 具有容纳不同类型元素的功能, 类摘要如下:

```
template <class T0, ... , class T9>
class tuple
{
public:
    tuple() ;
    tuple(T0 t0, ...);
    tuple(const tuple<...>&);
    tuple& operator=(const tuple&);
    tuple& operator=(const std::pair& );

    template <int N> T get();
};
```

tuple 的类声明非常简单，但它的用法却并不简单，我们接着往下看。

7.6.3 创建与赋值

tuple 默认最多支持 10 个模板类型参数，也就是说它最多能容纳 10 个不同类型的元素 (10-tuple)，对于绝大多数应用来说这已经足够了。^①

tuple 对元素的类型没有特殊的要求，但如果类型不支持缺省构造或者赋值操作，那么 tuple 也会相应地缺失功能。特别的，元素也可以是 tuple，也就是说 tuple 支持嵌套定义，例如：

```
typedef tuple<int , string> my_tuple1;           //容纳 int 和 string 的元组
typedef tuple<int, my_tuple1> my_tuple2;        //容纳 int 和元组的元组
```

单从类声明来看，tuple 与 pair 很相似，只是模板参数列表由两个变成了 10 个，把多个不同类型的值“打包”为一个整体。tuple 的构造函数与 pair 也差不多，可以直接输入 N 个值构造，或者从另一个 tuple 拷贝构造。例如：

```
typedef tuple<int, string, double> my_tuple;      //3-tuple
my_tuple t1;                                     //全部元素使用缺省构造
my_tuple t2(1, "123");                          //第三个元素使用缺省构造
my_tuple t3(t1);                                //拷贝构造
t2 = t3;                                         //赋值操作
```

如果 tuple 的元素类型是引用，那么在初始化时必须给予初值：

```
int x = 10;
tuple<int&> t4(x);                               //1-tuple
```

如果没有初值会引起编译错误，因为无法创建一个空的引用：

```
tuple<int&> t4;                                   //error
```

也存在不能实例化的 tuple 类型，例如当元素类型是 void 或者函数类型（不是函数指针）的时候，因为这两种类型没有实例（没有一个类型为 void 的变量），因而也就不存在相应的 tuple 对象：

```
typedef tuple<void> no_instance_t1;              //没有 tuple 实例
typedef tuple<double(int)> no_instance_t2;       //没有 tuple 实例
```

但指向 void 或者函数类型的指针是可以的：

^① 当然，也存在 1-tuple 和 0-tuple，但它们在软件实际开发中用处不大。

```
tuple<void*> t1; //void*指针 1-tuple 对象
tuple<double(*) (int)> t2; //函数指针 1-tuple 对象
```

make_tuple

为了方便创建 tuple 对象, tuple 库提供与 make_pair() 类似的 make_tuple() 函数, 它同样可以根据参数类型推导出要创建的 tuple 类型, 默认的类型是非引用类型。make_tuple() 的声明是:

```
template<typename T1,...>
tuple<T1,...> make_tuple(const T1& t1, ...);
```

make_tuple() 的使用就如同 make_pair() 一样便捷:

```
make_tuple(2, 3.0); //tuple<int, double>
make_tuple(string(), vector<int>()); //tuple< string, vector<int> >
```

如果能让 make_tuple 生产的 tuple 数据类型是引用, 那么要用到 ref 库 (11.2 小节, 422 页) 的 ref() 和 cref() 函数, 它们可以包装变量, 使 tuple 的类型为 T& 或者从 const T&, 这在用于不可拷贝的数据类型时是很有用的:

```
int i;
string s;
tuple<int, string&> t1 = make_tuple(i, ref(s));
tuple<const int&, string&> t2 = make_tuple cref(i), ref(s));
```

7.6.4 访问元素

tuple 提供成员函数 get<>() 访问内部的值。

get<>() 是一个模板函数, 它以整数为索引, 返回 tuple 中的第 N 个元素。索引遵循 C++ 的惯例从 0 开始, 如果索引超过了 tuple 内的元素个数则会导致编译错误。^①例如:

```
BOOST_AUTO(t, make_tuple(1, "char[]", 100.0)); //3-tuple
assert(t.get<0>() == 1); //取第一个元素,int 类型
assert(t.get<2>() == 100.0); //第三个元素,double 类型
cout << t.get<1>(); //第二个元素,const char 类型
cout << ++t.get<0>();
```

代码中要特别注意模板函数 get<>() 的用法, 尖括号内是一个表示元素位置的整数, 语法很像是访问数组元素的 operator[], 这使得它比较容易理解。小心: 不要只写尖括号而忘记了圆

① 一点小遗憾, tuple 没有提供类似 bimap 那样的标签用法, 不能写出 get<id>() 形式的代码。与简单的整数索引相比, 这种形式具有非常好的可读性。

括号，否则会导致编译错误——要意识到我们是在调用一个模板函数。

在了解了 `get<>()` 的用法后，初学者往往会犯一个错误，企图像遍历数组那样用 `for` 循环打印出 `tuple` 的所有元素，写出如下的代码：

```
for (int i = 0; i < 3; ++i)
{
    cout << t.get<i>();           //错误
}
```

这段代码是无法通过编译的。

只要明白了 `get<>()` 函数的工作原理，就不难知道无法编译的原因了。`get<>()` 是一个模板函数，编译进行模板实例化时要求模板参数 `<int N>` 必须是编译期可确定的，而 `for` 循环中的变量 `i` 只能在运行时确定，因而无法编译模板代码（在 7.6.10 小节，282 页提供了使用递归变量 `tuple` 的方法）。

除了成员函数 `get<>()`，`tuple` 库还提供一个自由函数 `boost::get<>()`，它的用法与同名成员函数完全相同，只是需要接受一个 `tuple` 变量作为操作的对象^①，例如：

```
BOOST_AUTO(t, make_tuple(1, "char[]", 100.0));
get<0>(t);
get<1>(t);
```

7.6.5 比较操作

`tuple` 全面支持比较操作，包括相等和不等的各种测试，它将比较操作符转发到内部的各个元素进行比较，因此要求 `tuple` 的元素必须能够执行比较操作，否则会引发编译错误。作为比较对象的两个 `tuple` 也必须有相同的元素个数，否则也会引发编译错误。

`tuple` 的大小比较是基于“字典序”的，而且是“短路”操作，从第一个元素开始，一旦得出比较结果就停止比较操作。

要使用 `tuple` 的比较功能，必须要包含头文件 `<boost/tuple/tuple_comparison.hpp>`，例如：

```
#include <boost/tuple/tuple_comparison.hpp>
using namespace boost;
int main()
```

① 实际上，自由函数 `get<>()` 才是访问 `tuple` 元素的真正实现，成员函数 `get<>()` 在内部调用了自由函数 `get<>()`。

```
{
    typedef tuple<int ,double ,string> my_tuple;    //3-tuple

    my_tuple t1 = make_tuple(1, 100.0, string("abc"));
    my_tuple t2 = make_tuple(1, 200.0, string("def"));
    assert(t1 < t2);

    my_tuple t3(t2);
    assert(t2 == t3);
}
```

下面的代码定义了一个空类 A，它没有定义比较操作，因此无法通过编译：

```
struct A{};
tuple<A> ta1, ta2;                                //1-tuple
ta1 == ta2;                                        //编译错误
```

7.6.6 输入输出

tuple 支持 C++ 标准库的流输入输出操作，与比较操作相同，对 tuple 内的所有元素逐个调用 operator<<或 operator>>，因此要求每个元素都支持流输入输出，否则也会引发编译错误。

要使用 tuple 的流输入输出功能，必须要包含 <boost/tuple/tuple_io.hpp>，例如：

```
#include <boost/tuple/tuple_io.hpp>
using namespace boost;
int main()
{
    typedef tuple<int ,double ,string> my_tuple;

    my_tuple t1(1, 2.0, "string");
    cout << t1 << endl;
    cout << "please input tuple:";
    cin >> t1;
    cout << t1 << endl;
}
```

tuple 的输入输出格式默认是：整个 tuple 用圆括号包围，元素值间用空格分隔。因此，这段代码的显示是：

```
(1 2 string)
please input tuple:(3 5 asdfg )
(3 5 asdfg)
```

输入时也要注意，整个 tuple 必须用圆括号包围起来。

tuple 库还在名字空间 boost::tuples 里定义了三个流格式操作符函数，用来改变输入输

出的格式，它们是：

- `set_open(char)` : 设置 tuple 开始时的字符；
- `set_close(char)` : 设置 tuple 结束时的字符；
- `set_delimiter(char)` : 设置元素之间的分隔符。

下面的代码使用这些操作符函数，把 tuple 的输出由默认的圆括号改成了方括号，元素分隔符改成了逗号：

```
my_tuple t1(1, 2.0, "string");
cout << tuples::set_open '[' << tuples::set_close ']' << endl;
cout << tuples::set_delimiter ',' << endl;
cout << t1 << endl; //输出[1,2,string]
```

7.6.7 连结变量

tuple 库提供一个类似 `make_tuple()` 的函数 `tie()`，正如它的名字，它可以把变量连结到 tuple 上，生成一个元素类型全是引用的 tuple，相当于 `make_tuple(ref(a), ref(b), ...)`，可以被用于左值，如果有：

```
int i; double d;
```

那么 `tie(i, d)` 将生成一个临时的 tuple 变量，其类型为 `tuple<int&, double&>`。

`tie()` 可以方便地利用现有普通变量创建一个可赋值的 tuple 对象，因此可以对 tuple 执行“解包”操作，这在接收返回 tuple 的函数返回值时特别有用：

```
#include <boost/tuple/tuple.hpp>
using namespace boost;
typedef tuple<int ,double ,string> my_tuple; //3-tuple 定义
my_tuple func() //返回 3 个值的函数
{ return make_tuple(1, 2.0, "string"); }
int main()
{
    int i; double d; string s; //3 个普通变量
    tie(i, d, s) = func(); //使用 tie() 接受函数返回的 tuple
    cout << i;
}
```

`tie()` 不仅能够应用于 tuple，它也可以应用于 `std::pair`：

```
int i; string s;
tie(i, s) = make_pair(100, "abc");
```

关于 `tie()` 还有一个特殊的对象 `tuples::ignore`，它相当于一个占位符，可以在赋值时“忽略”某些对象，在只关心 `tuple` 中少数元素值的时候就可以使用，从而不必声明一堆必须声明又不用的变量。例如，下面的代码使用 `ignore` 忽略了函数返回的 `int` 和 `string` 结果，只接收 `double` 值：

```
double d;
tie(tuples::ignore, d, tuples::ignore) = func();
```

7.6.8 应用于 assign 库

`assign` 库（参见 4.4 小节，106 页）提供了一个初始化工具 `tuple_list_of`，它可以初始化元素类型为 `tuple` 的容器，用法类似 `map_list_of` 和 `pair_list_of`。

示范 `tuple_list_of` 用法的代码如下：

```
typedef tuple<int ,double ,string> my_tuple;
using namespace boost::assign;

vector<my_tuple> v = tuple_list_of(1, 1.0, "123")(2, 2.0, "456");
assert(v.size() == 2);
```

其他的 `assign` 库操作函数需要使用 `make_tuple()` 或者 `tie()` 函数来创建可插入容器的 `tuple` 对象，例如：

```
v += make_tuple(3, 3.0, "789"),make_tuple(4, 4.0, "abc");
assert(v.size() == 4);
```

7.6.9 应用于 exception 库

`tuple` 可以把多个不同类型的数据“打包”，`exception` 库就利用了 `tuple` 的这个功能，可以把多个错误信息“打包”成一个 `tuple`，然后一次性完成所有错误信息的创建，传递给 `boost::exception`，详见 4.9.8 小节（145 页）。

7.6.10 内部结构

`tuple` 的内部结构是一个部件（cons）的链表，cons 位于名字空间 `boost::tuples`，它的声明如下：

```
template<typename Head, typename Tail>
struct cons
{
    typedef Head head_type;
    typedef Tail tail_type;
    head_type head;
```

```
tail_type tail;

head_ref get_head() { return head; }
tail_ref get_tail() { return tail; }
};
```

通过 cons 的模板类型 Head 和 Tail, tuple 使用了类似 operators 库的基类链技术 (4.8.3 小节), 通过链式继承实现类型的容纳, 的例如:

```
tuple<int, double, string>
```

概念上相当于:

```
cons<int, cons<double, <string, null_type> > >
```

基类链的末尾是一个 null_type, 它是一个空类, 起到哨兵的作用, 等价于 tuple<> (0-tuple)。

tuple 利用 cons 提供了两个很有用的函数: get_head() 和 get_tail(), 它们分别返回 tuple 首元素和尾部链表的引用, 使用它们就可以递归地访问 tuple 内的元素。

之前我们在 7.6.4 小节说过, 无法使用循环来处理 tuple 元素, 现在我们可以改用递归来处理, 用一个 get_head() 和 get_tail() 函数实现遍历输出 tuple 内的元素:

```
template <typename Tuple>
void print_tuple(const Tuple& t)
{
    cout << t.get_head() << ', ';           //输出 tuple 首元素
    print_tuple(t.get_tail());               //递归输出剩余的 tuple 元素
}
```

print_tuple() 是一个模板函数, 因此可以接受任意的 tuple 对象, 它先用 get_head() 获得 tuple 的第一个元素, 然后递归地处理尾部链表 get_tail()。为了使递归能够停止, 我们必须有一个处理 null_type 的重载函数, 它什么都不做, 只是用来终止递归过程:

```
void print_tuple(const tuples::null_type&) {}
```

那么, 下面的代码:

```
typedef tuple<int, double, string> t_type;
t_type t(1, 2.1, "string tuple");
print_tuple(t);
```

将输出:

```
1,2.1,string tuple
```

7.6.11 使用访问者模式

使用 tuple 的部件链表形式，我们可以很容易地用递归的方式操作 tuple，处理的方式都很相似，反复调用 get_head() 和 get_tail()，因此值得把这种手法规范化。

我们可以使用访问者模式来封装对 tuple 的递归操作，它可以分离数据的表示与操作。由于 C++ 中的模板类不能自动推导参数，而可自动推导参数的模板函数不支持偏特化，因此这个访问者并不是很完美。

访问者函数 visit_tuple() 泛化了 tuple 的递归操作，增加了一个模板参数 Visitor，它是一个函数对象，具有 operator()，可以被调用。函数的返回值使用 boost::result_of 组件（参见 7.6.11 小节，284 页）来实现推导：

```
template<typename Visitor, typename Tuple>
typename result_of<Visitor(Tuple&)>::type           //自动推导返回值类型
visit_tuple(Visitor v, const Tuple &t)
{
    v(t.get_head());                               //访问者对象访问 tuple 的首元素
    return visit_tuple(v, t.get_tail());           //递归访问剩余元素
}
```

为了操作 tuple，我们还需要编写一个函数对象，它同 7.6.10 节(282 页)的 print_tuple() 类似，向标准输出打印元素值：

```
struct print_visitor                               //一个访问者函数对象
{
    typedef void result_type;                       //返回值类型定义
    template<typename T>
    result_type operator()(const T &t)              //调用操作符重载
    {
        cout << t << ", ";                         //简单地输出元素值
    }
};
```

为了终止递归，我们还必须写出 visit_tuple() 针对 null_type 的重载形式：

```
template<>
void visit_tuple< print_visitor, tuples::null_type>
(print_visitor , const tuples::null_type &)
{}
```

访问者模式可以这样使用：

```
tuple<int , double ,string> t(1, 2.1, "string visitor");
visit_tuple(print_visitor(), t);
```

代码将输出:

```
1,2.1,string visitor
```

使用 `visit_tuple()`, 我们还可以实现对 `tuple` 更多的操作, 更加方便地操作 `tuple` 对象, 比如查找 `tuple` 中最大的元素, 选择出特定类型的元素等等。

例如, 下面的代码定义了一个查找最大元素的 `tuple` 访问者函数对象和 `visit_tuple()` 的特化形式:

```
template<typename T>
struct max_visitor
{
    T *tmp; //最大值
    max_visitor(T *t):tmp(t){}
    typedef void result_type;
    result_type operator()(const T &t)
    {
        *tmp = *tmp < t? t : *tmp;
    }
};
template<>
void visit_tuple< max_visitor<double>, tuples::null_type>
(max_visitor<double> , const tuples::null_type &)
{}
```

`max_visitor` 函数对象可以在元素是数字的 `tuple` 中查找最大值^①, 例如:

```
tuple<int, long, float, double> t(100, 5, 3.14, 314.15);
double *max_value = new double(t.get_head()); //设置初始值
max_visitor<double> mv(max_value); //创建访问者对象
visit_tuple(mv, t); //访问 4-tuple
cout << *max_value << endl; //输出最大值 314.15
```

`boost.fusion` 库进一步深化了访问者模式, 它为 `tuple` 提供了非常全面的操作, 读者可以参考 Boost 库文档以了解更多的用法。

7.6.12 高级议题

本小节讨论关于 `tuple` 的一些高级议题。

代替 struct

`tuple` 的实现很紧凑很小, 构造与访问函数都是内联的, 因而很容易被编译器优化, 一个

① 这里为了简单而直接使用原始指针传递数据, 存在内存泄漏的问题, 实际的代码应该使用 `shared_ptr` 来管理指针。

tuple 和一个持有同样元素类型的 struct 几乎拥有同样的运行时性能, 因此我们完全可以用 tuple 来代替 struct 来打包数据。

例如:

```
struct demo //一个拥有 3 个成员变量的结构
{
    int x;
    double y;
    vector<string> str;
};
typedef tuple<int, double, vector<string> > t_type; //同样拥有 3 个元素的 3-tuple
assert(sizeof(demo) == sizeof(t_type)); //大小通常是相同的
```

很明显, 拥有同样功能的情况下, tuple 的代码要比 struct 更加简洁, 而且提供了许多立即可用的操作元素的方法, 使用起来更方便。而 struct 则必须手工定义所需的成员访问和操作函数, 不仅麻烦, 而且容易增加出错的几率和调试的时间。

但由于 tuple 采用了模板技术, 即使很少量的 tuple 代码也会带来大量的模板实例化推演, 导致编译时间增加 (泛型编程的普遍现象)。因此, 当使用 tuple 时, 需要考虑编译的时间成本。但一般来说, 在大型程序中编译 tuple 的时间只是很小的一部分, 完全可以被 tuple 带来的好处抵消。

tuple 的辅助类

tuple 库在 `boost::tuples` 名字空间里提供了两个辅助类 `element` 和 `length`, 它们可以获得 tuple 的一些相关信息。

`element<N, T>::type` 可以给出 T 中第 N 个元素的类型, `length<T>::value` 可以给出 T 的元素数量, 模板参数列表中的类型 T 是一个 tuple 类型。

例如:

```
typedef tuple<int, string> my_tuple1;
typedef tuple<int, my_tuple1> my_tuple2;
assert(typeid(int) == typeid(tuples::element<0, my_tuple1>::type));
assert(typeid(string) == typeid(tuples::element<1, my_tuple1>::type));
cout << tuples::length<my_tuple2>::value << endl; //输出 2
```

正确地运用 tie

熟悉 Python 的读者也许会记得 Python 中使用 tuple 交换变量值的经典用法 (虽然效率不高):

```
(a, b) = (b, a) #swap
```

很自然，一旦学会了 `tie()` 的使用方法就会想到如下的代码：

```
int a = 10, b = 20;
tie(a, b) = tie(b, a);
```

然而实际运行结果却是错误的，变量值并没有交换。因为这行代码等价于：

```
tuple<int&,int&> x(b,a),y(a,b);
y = x;
```

把 `tuple` 解包，实际的操作就是：

```
a = b;
b = a;
```

最后两个变量都被赋了相同的值。

使用 `tie()` 交换变量的正确写法应该是令右值的 `tuple` 为非引用类型：

```
tie(a, b) = make_tuple(b, a);
```

当然，`tuple` 的 `swap` 解法并不很优雅，也不够高效，但提供了一种使用 Boost 的新思路。

7.7 any

`any` 是一种很特殊的容器，它只能容纳一个元素，但这个元素可以是任意的类型——`int`、`double`、`string`、STL 容器或者任何的自定义类型。程序可以用 `any` 保存任意的数据，在任何需要的时候将它取出。这种功能与 `shared_ptr<void>` 有些类似，但 `any` 是类型安全的。

`any` 位于名字空间 `boost`，为了使用 `any` 组件，需要包含头文件 `<boost/any.hpp>`，即：

```
#include <boost/any.hpp>
using namespace boost;
```

7.7.1 类摘要

`any` 不是一个模板类，它的类摘要如下：

```
class any
{
public:
    any();
    any(const any &);
    template<typename ValueType> any(const ValueType &);
    any & operator=(const any &);
    template<typename ValueType> any & operator=(const ValueType &);
```

```

~any();

any & swap(any &);

bool empty() const;
const std::type_info & type() const;
};

```

any 能够容纳任意类型的原因在于它的构造函数 `any(const ValueType &)` 和赋值函数 `operator=(const ValueType &)`，它们是模板函数，可以接受任意的类型^①，将值存入内部的一个模板类 holder。从这个意义上讲，any 实际上是一个包装类。

any 的析构函数删除内部的 holder 对象。如果类型是指针，any 并不会对指针执行 delete 操作，因此，如果用 any 保存原始指针会造成内存泄漏，替代方法是使用智能指针来存入 any 容器，如 `shared_ptr` (3.4 小节，69 页)。例如：

```

int *p = new int(10);           //应该用 shared_ptr<int> p(new int(10))
any a = p;                      //危险，会造成内存泄漏

```

空的 any 构造函数创建一个空的 any 对象，不持有任何值。成员函数 `empty()` 可以判断 any 对象是否是空的。如果 any 持有一个对象，那么函数 `type()` 返回对象的类型信息，是一个标准的 `type_info` 类的引用。

7.7.2 访问元素

any 类本身不提供任何对内部元素的访问函数，而是使用了一个友元函数 `any_cast()`，这个函数的命名模仿了标准库的转型操作符 `xxx_cast`^②。

`any_cast()` 的用法也很类似转型操作符，可以取出 any 内部持有的对象，声明如下：

```

template<typename T> T any_cast(any & operand);
template<typename T> T any_cast(const any & operand);

```

这两个重载形式返回 any 存储对象的拷贝，如果转型的目标类型 T 是一个引用，则返回持有值的引用。例如，下面的代码使用 any 存储了 int，并使用 `any_cast()` 转换了值和值的引用两种类型：

```

any a(10);
int n = any_cast<int>(a);           //获得一个值的拷贝
assert(n == 10);

```

① any 对类型 T 有一个小小的要求，它必须是可拷贝构造的，并且析构不能抛出异常。

② 这种用法我们已经在 Boost 库中多次看到了，比如 `lexical_cast` (5.1 小节，163 页)。


```
any_cast<int&>(a) = 20;           //获得值的引用, 被用于左值
assert(any_cast<int>(a) == 20);;
```

如果要转换的类型不是 any 内部对象的类型, 或者 any 不持有任何对象 (empty ()==true), 则这两个 any_cast() 函数会抛出一个 bad_any_cast 异常, 它是 std::bad_cast 的子类, 但经过了 exception 库 (4.9 小节, 136 页) 的包装, 因而 catch 捕获后可以使用 boost::exception 的全部功能。例如:

```
try
{
    any a;           //一个空 any 对象, 不持有任何值
    any_cast<int>(a); //转换将导致异常
}
catch(boost::exception&) //捕获异常, 输出诊断信息
{
    cout << current_exception_diagnostic_information();
}
```

any_cast() 还有另外两个接受指针参数的重载形式, 它们把 any 转换成持有值的指针:

```
template<typename ValueType> ValueType* any_cast(any *operand);
template<typename ValueType> const ValueType * any_cast(const any * operand);
```

当 any 以指针的方式传入 any_cast() 的时候, 返回的指针类型与传入的 any 对象具有相同的常量性。如果 any 不持有任何对象, 这两个 any_cast() 不会抛出异常, 而是返回一个空指针:

```
any a1, a2(2.0);
assert(any_cast<int*>(&a1) == 0); //转换空 any 对象
assert(any_cast<string*>(&a2) == 0); //转换错误的类型
```

7.7.3 用法

any 的类接口很小很简单, 因此很容易使用, 它的出现让 C++ 的强类型语法检查失去了作用, C++ 仿佛变成了一种弱类型的动态语言。

创建或赋值一个 any 对象就像是使用一个具有魔力的类型, 它可以被初始化或者赋值任意类型的数据:

```
any a(100);           //创建一个 any 对象, 初始化为一个整数
a = string("char*"); //any 存储一个 string 字符串
a = vector<vector<int>> >(); //any 存储一个二维 vector
```

在 any 中存储字符串的时候我们必须使用 std::string, 如果直接使用 C 字符串会引发编译错误:

```
a = "c str"; //企图直接向 any 存入 C 字符串, 无法通过编译
这是因为 C 字符串是一个普通数组, 它没有拷贝构造函数, 不符合 any 对类型的要求。
```

在给 any 赋值后, 可以使用 any_cast() 取回存入的值, 就像用一个转型操作, 但我们必须知道 any 内部值的确切类型才能调用成功:

```
a = string("avatar"); //any 存储 string 字符串
cout << any_cast<string>(a); //用 any_cast 取得存储的值
any_cast<int>(a); //类型错误, 抛出异常
```

在 any 中保存指针时要特别小心, 因为空指针对于 any 也是有效的值, 在用 any_cast() 取出来之后必须要进行测试, 例如:

```
string *ps = new string("abc"); //不好的使用方式
a = ps;
if (!a.empty() && any_cast<string*>(a))
{ cout << *any_cast<string*>(a) << endl; }
```

这样的代码很麻烦, 而且还有内存泄漏的危险, 所以除非必要, 应该使用智能指针包装原始指针再放入 any, 进一步的讨论见后面的阐述。

7.7.4 简化的操作函数

any 类很有用, 但它提供的操作函数仅有一个 any_cast(), 使用不太方便, 因此有必要我们自己编写一些辅助类和函数来增加它的使用价值, 当然, 这些辅助类和函数必然要基于 any_cast()。

首先我们需要判断 any 持有对象的类型, 这个很容易做到, 因为 any 提供了运行时的类型检查函数 type:

```
template<typename T>
bool can_cast(any &a)
{ return typeid(T) == a.type(); }
```

接下来是 get() 函数, 它可以返回 any 内部值的引用, 因此可以当做左值, 直接修改 any:

```
template<typename T>
T& get(any &a)
{
    BOOST_ASSERT(can_cast<T>(a));
    return *any_cast<T>(&a);
}
```

注意 get() 的参数传递和用法, 我们使用引用的方式接受 any 对象, 在调用 any_cast() 时获取 any 对象内部的数据指针, 避免了 any 可能存储大对象时的拷贝代价。

最后是 `get_pointer()` 函数，它直接返回 `any` 内部值的指针：

```
template<typename T>
T* get_pointer(any &a)
{
    BOOST_ASSERT(can_cast<T>(a));
    return any_cast<T>(&a);
}
```

这些辅助函数可以这样使用：

```
any a;
int x = 1;
a = x;
assert(can_cast<int>(a));
get<int>(a) = 10;
*get_pointer<int>(a) = 20;
```

这些辅助函数比直接使用 `any_cast()` 要好，因为它们的名字更易读易懂，比起“万能型”的 `any_cast()`，它们能够显著地改善代码的可维护性。

7.7.5 保存指针

`any` 可以持有原始指针，但这样的用法很不安全，会导致内存泄漏。应该使用智能指针包装原始指针，这样在 `any` 析构时智能指针会自动地调用 `delete`，从而安全地释放资源。

不是所有的智能指针都可以作为 `any` 存储的对象。`auto_ptr` 是一个错误的选择，它特有的拷贝转移语义使得它不能被用作容器的元素；`scoped_ptr` (3.2 小节，63 页) 也不行，它不能被拷贝，不符合 `any` 的类型要求。我们应该使用 `shared_ptr` (3.4 小节，69 页)，它是最智能的智能指针，几乎完全可以替代原始指针，而只需要很小的代价。

在之前的示范代码中我们已经看到了 `shared_ptr` 用于 `any` 的用法，但这种写法不够安全，用户会偶尔忘记使用 `shared_ptr` 包装原始指针再放入 `any`，应该提供一种方式，使用户可以更方便地存储指针而不关心存储的方式。

我们可以自行实现一个工厂函数 `make_ptr_any()` 来满足这个要求，它封装了 `any` 使用 `shared_ptr` 的用法，简化了存储指针的调用接口，因而更加安全：

```
template<typename T>
any make_ptr_any(T *p = 0)
{ return any(shared_ptr<T>(p)); }
```

简单的封装带来的是调用接口的简化，代码如下：

```
any a = make_ptr_any<string>(new string("long"));
```

```
a = make_ptr_any<vector<int> >(new vector<int>);
```

在获取 `make_ptr_any()` 生产的 `any` 对象时我们必须在 `any_cast` 的模板参数中指明 `shared_ptr`, 否则无法取出保存的指针:

```
any a = make_ptr_any<string>(new string("avatar"));
cout << *any_cast<shared_ptr<string> >(a);    //使用 any_cast
cout << *get<shared_ptr<string> >(a);          //使用自定义的 get() 函数
```

为了简化访问存储 `shared_ptr` 的 `any` 对象的代码, 也可以再编写一个便捷函数, 它指定 `shared_ptr` 指向的类型, 直接返回一个 `shared_ptr` 的引用:

```
template<typename T>
shared_ptr<T>& get_shared(any &a)
{
    BOOST_ASSERT(can_cast<shared_ptr<T> >(a));
    return *any_cast<shared_ptr<T> >(&a);
}
```

`get_shared()` 可以这样使用:

```
cout << *get_shared<string>(a);                //返回 shared_ptr<string>
```

7.7.6 输出

`any` 不支持内部值流输出或者转化为字符串, 这是它所缺乏又很常用的功能, 特别是在调试的时候。

我们希望有一个泛型的输出功能, 它可以打印出 `any` 内部保存的值, 不仅能够支持原始类型, 也能够支持智能指针。这个需求用普通的模板函数难以实现, 因为无论是编译期还是运行时都很难区分类型参数 `T` 是指针还是智能指针, 因此我们使用“智能函数”——函数对象来实现这个功能。

模板类 `any_print` 的模板类型参数 `T` 通常情况下是 `any` 内部的数据类型, 但允许被偏特化。`any_print` 的声明是这样:

```
template<typename T > struct any_print;
```

内部的 `operator()` 使用 `any_cast` 来访问 `any` 的元素, 使用 `function-try` 块来捕获异常:

```
void operator()(any &a)
try
{
    cout << *any_cast<T>(&a) << endl;
}
catch(bad_any_cast &)
{
}
```

```
    cout << "print error" << endl;
}
```

普通的 `any_print` 只能输出原始类型，如果 `any` 持有的是 `shared_ptr`，则需要对 `any_print` 类进行模板偏特化：

```
template<typename T>
struct any_print<shared_ptr<T>>
{
    void operator()(any &a)
    try
    {
        cout << **any_cast<shared_ptr<T>>>(&a) << endl;
    }
    catch(bad_any_cast &)
    {
        cout << "print error" << endl;
    }
};
```

偏特化的 `any_print< shared_ptr<T>>` 在 `any_cast` 访问后使用了连续的两个解引用操作符`*`，以获得 `shared_ptr` 所指向的真正内容。

`any_print` 类的调用方式稍微有点古怪，需要使用两对括号：第一对括号创建一个 `any_print` 的临时对象，第二对括号调用操作符 `operator()`，执行真正的输出工作，例如^①：

```
any a;
char* str = "string";
a = str;
any_print<char*>()(a);
a = 10;
any_print<int>()(a);
shared_ptr<string> ps = make_shared<string>("metroid");
a = ps;
any_print< shared_ptr<string>>()(a);
```

`any_print` 函数对象包装了 `any_cast`，并且使用了 `try/catch` 块处理了可能发生的异常，较原始的 `any_cast` 用法更简单也更安全。

如果读者不习惯 `any_print` 的临时对象用法，那么可以使用一个 `inline` 函数来包装它：

```
template<typename T>
void inline any_print_func(any &a)
{ any_print<T>()(a); }
```

① 这种调用形式类似 4.10 小节（146 页）`uuid` 的生成器用法。

它的使用要稍微简单一些:

```
any_print_func<shared_ptr<string> >(a);
```

7.7.7 应用于容器

`any` 类可以用在那些要保存的值的类型是未知的应用程序, 当容器的元素类型是 `any` 时, 容器的表现就像是一个可持有多个不同类型对象的动态 tuple (7.6 小节, 275 页):

```
vector<any> v;
v.push_back(10); //int
v.push_back(1.414); //double
v.push_back(shared_ptr<int>(new int(100) )); //shared_ptr
```

也可以对 `any` 容器使用 `assign` 库 (4.4 小节, 106 页) 来初始化或赋值, 例如:

```
using namespace boost::assign;
vector<any> v2 = list_of<any>(10) (0.618) (string("char"));
cout << v2.size();
```

如果希望一种数据结构具有 tuple 那样容纳任意类型的能力, 又可以在运行时动态变化大小, 那么我们就可以用 `any` 作为元素类型搭配容器。

`any` 应用于容器的典型例子就是 7.10 小节(314 页)的 `property_tree` 和 10.4 小节(400 页)的 `program_options`, 它们使用 `any` 来存储类型不确定的属性值。在 7.9.3 小节 (306 页) 有另一个实际的例子。

7.8 variant

`variant` 与 `any` 有些类似, 是一种可变类型, 是对 C/C++ 中 `union` 概念的增强和扩展。普通的 `union` 只能持有 POD (普通数据类型), 而不能持有如 `string`、`vector` 等复杂类型, `variant` 则没有这个限制。

`variant` 位于名字空间 `boost`, 为了使用 `variant` 组件, 需要包含头文件 `<boost/variant.hpp>`, 即:

```
#include <boost/variant.hpp>
using namespace boost;
```

7.8.1 类摘要

`variant` 的接口与 `any` 也很类似, 但它是一个模板类, 类摘要如下:

```

template<typename T1, typename T2, ..., typename TN >
class variant
{
public:
    variant();
    variant(const variant &);
    template<typename T> variant(T &);
    template<typename U1, typename U2, ..., typename UN>
        variant(variant<U1, U2, ..., UN> &);
    ~variant();

    void swap(variant &);
    variant & operator=(const variant &);

    bool empty() const;
    const std::type_info & type() const;
    int which() const;

    bool operator==(const variant &) const;
    bool operator<(const variant &) const;
};

```

variant 与 any 不同，它是有界类型（而 any 是无界类型），允许保存的数据类型必须在模板参数列表中声明，其形式很像 tuple（7.6 小节，275 页）。对类型参数 TN 的最低要求是可拷贝构造且析构不抛出异常。

无参的构造函数将把 variant 的值用第一个模板参数（T1）的缺省构造函数初始化，如果 T1 没有缺省构造函数，那么 variant 也不能缺省构造。带参数的构造函数参数可以是模板类型参数中任意类型的值，variant 将初始化为这种类型。

variant 支持拷贝构造和赋值，赋值操作要求模板类型参数也都支持赋值操作。

成员函数 empty() 和 type() 的用法与 any 相同，用来检测当前 variant 持有的对象。但 empty() 永远返回 false，因为 variant 不能是空的，它总有值。which() 函数可以返回 variant 当前值的类型在模板参数列表的索引号（从 0 开始计数）。

variant 提供了比较操作符，这使得它可以被用作关联容器的元素或者用于排序，但要求 variant 的所有模板参数也是可以比较的，这个要求很容易理解。

variant 也支持流输出，但同样要求每一个模板参数是可流输出的。

7.8.2 访问元素

variant 的操作要比 any 方便，能够直接访问元素的值，例如：

```
variant<int, float, string> v;           //可容纳 int,float 和 string
v = "123";                             //v 持有一个 string 对象
```

但为了实现泛型编程, `variant` 也提供一个外界自由函数来访问内部元素。`variant` 没有提供 `variant_cast` 的函数, 而是使用 `boost` 库中的一个泛型函数 `get()`。这是因为 `variant` 的设计出发点与 `any` 不同, 它的目的是存储多个数据的联合, 而不是任意类型的容器。

用于访问 `variant` 元素的 `get()` 函数声明如下:

```
template<typename U,...> U * get(variant * operand);
template<typename U,...> const U * get(const variant * operand);
template<typename U,...> U & get(variant & operand);
template<typename U,...> const U & get(const variant & operand);
```

它的声明与 `any_cast()` 类似, 用法也基本相同。

如果 `variant` 当前的值不是 `get()` 想取的类型, 那么会抛出 `boost::bad_get` 异常, 它是 `std::exception` 的子类, 但没有使用 `boost.exception` 库 (参见 4.9 小节, 136 页) 进行包装。如果是返回指针的 `get()` 形式, 那么会返回一个空指针。

7.8.3 用法

使用 `variant`, 我们必须要在模板参数中指定它所能容纳的类型。类型的数量在 `boost1.42` 中默认最多是 20 个 (一个相当惊人的数字)。例如, 声明一个能容纳 `int`、`double` 和 `string` 的 `variant` 类型, 其写法与 `tuple` 很像:

```
typedef variant<int, double, string> var_t;
```

有很多的构造函数和赋值函数可以创建、赋值 `variant` 对象。无参的构造函数使用第一个模板类型参数缺省构造 `variant`, 如果传入模板参数列表中的类型值, 那么 `variant` 就存储这个值, 例如:

```
var_t v(1);           //v->int
v = 2.13;             //v->double
var_t v2("string type"); //v2->string
v2 = v;               //v2->double
```

成员函数 `empty()` 是无意义的, 它总返回 `false`, 它的存在只是为了兼容 `any`, 让泛型代码可以把 `variant` 当做 `any` 一样进行操作。成员函数 `type()` 可以检测 `variant` 当前值的类型, 功能与 `any` 的同名成员函数相同如:

```
assert(v.type() == typeid(double));
```

一旦 `variant` 对象被创建, 它就总是可用的, 不需要使用 `union` 的点号操作符来访问内部

的成员，它用起来就像个普通的数据类型，非常方便。与 any 不同的是，它自身支持更多的操作，比如比较、流输出，因此用起来更方便。

我们也可以使用自由函数 get() 来获取 variant 的值：

```
cout << get<string>(v);
cout << get<int>(var(108));
```

但 get() 函数通常不是最方便最有效的访问方法，它与 any_cast 同样存在着类型不安全的隐患，操作时必须查询 variant 当前值的类型。

进一步示范 variant 用法的代码如下：

```
#include <boost/variant.hpp>
using namespace boost;
int main()
{
    typedef variant<int, double, string> var_t;
    var_t v; //缺省构造, v==0
    assert(v.type() == typeid(int)); //v->int
    assert(v.which() == 0); //v 现在持有第一个类型的元素

    v = "variant demo"; //v->string
    cout << *get<string>(&v) << endl; //使用 get() 函数取值

    try
    {
        cout << get<double>(v) << endl; //抛出异常
    }
    catch (bad_get &)
    {
        cout << "bad_get" << endl;
    }
}
```

7.8.4 访问器

variant 可以如 any 那样使用，在运行时通过 type() 检测 variant 的类型，再施以必要的操作，例如：

```
typedef variant<int, double> var_t;
void var_print(var_t &v)
{
    if (v.type() == typeid(int)) //if-else 分支语句检测类型
    {
        get<int>(v) *=2;
        cout << v << endl;
    }
}
```

```

    }
    else if (v.type() == typeid(double))
    {
        get<double>(v) *=2;
        cout << v << endl;
    }
    else
    {
        cout << "don't konw type" << endl;
    }
}

```

函数 `var_print()` 将输入的 `variant` 对象内容翻倍并输出，它使用了 RTTI（运行时类型信息）技术，效率低，并且一旦 `variant` 的模板参数发生变化，`var_print()` 也必须改动，其 `if-else` 的处理结构也很不优雅。

`variant` 基于访问者模式（推荐书目 [1]）提供了模板类 `static_visitor`，它解耦了 `variant` 的数据存储和访问操作，把访问操作集中在访问器类，易于增加新的访问操作，使这两者可以彼此独立地变化。

`static_visitor` 是一个静态访问 `variant` 的基类，它应该被实际访问器函数对象所继承来使用，其类声明如下：

```

template<typename ResultType>
class static_visitor {
public:
    typedef ResultType result_type;
protected:                                     // for use as base class only
    static_visitor() { }
    ~static_visitor() { }
};

```

`static_visitor` 把构造函数和析构函数都声明为保护的，表明它是一个抽象类，只能被继承使用。其中的模板参数 `ResultType` 指明了访问者子类 `operator()` 的返回值类型，默认是 `void`。

使用 `static_visitor` 首先要从 `static_visitor` 继承，然后重载 `operator()`，用来访问 `variant` 的内部值。访问器必须能够处理 `variant` 所可能拥有的所有类型，不能仅处理其中的一部分，否则会引发编译错误：

```

struct var_print : public static_visitor<>    //返回值类型为 void
{
    template<typename T>                     //使用模板函数来处理所有元素类型
    void operator()(T &i) const              //通常是 const 函数
    {

```

```

        i *= 2;
        cout << i << endl;           //输出以验证
    }
};

```

函数对象 `var_print` 的 `operator()` 是一个模板函数，可以对普通类型执行加倍操作，然后再输出。

假设我们把 `variant` 增加一个 `vector<int>` 的类型：

```
typedef variant<int, double, vector<int> > var_t;
```

那么访问器的 `operator()` 需要增加一个针对 `vector<int>` 类型的重载，而原有的处理代码不必做任何变动：

```

template<>
void operator()<vector<int> >(vector<int> &v) const
{
    copy(v.begin(),v.end(),back_inserter(v)) ;
    for (int i = 0;i< v.size();++i)
    {
        cout << v[i] << ", ";       //输出以验证
    }
    cout << endl;
}

```

请读者注意针对 `vector<int>` 类型的 `operator()` 的特化形式，特化的模板参数在第一个括号之后，因为模板参数必须在函数名称之后，而括号操作符函数的全名是 `operator()`。

将访问器对象应用于 `variant`，需要使用函数 `apply_visitor()`。`apply_visitor()` 有多种重载形式，但最常用的形式是接受一个访问器对象和一个 `variant` 对象，对 `variant` 内的值调用访问器对象的 `operator()`。例如：

```

var_t v(1);                               //一个 variant 对象

apply_visitor(vp, v);                      //应用访问器，翻倍并输出 v 的值
v = 3.414;                                 //v->double
apply_visitor(vp, v);                      //应用访问器，翻倍并输出 v 的值

using namespace boost::assign;
v = vector<int>(list_of(1)(2));            //v-> vector<int>
apply_visitor(vp, v);                      //应用访问器，翻倍并输出 v 的值

```

函数 `apply_visitor()` 还可以只接受访问器对象，这时它会返回一个新的函数对象 `apply_visitor_delayed_t`，用于延后对 `variant` 的访问。这个新的函数对象包装了访问器，同样提供 `operator()`，可以直接操作 `variant` 对象而无需再使用 `apply_visitor()`，例如：

```
var_t v(1); //一个 variant 对象
var_print vp; //一个访问器对象
BOOST_AUTO(vp, apply_visitor(var_print())); //获得新的函数对象
vp(v); //直接访问 variant, 不必在调用
//apply_visitor()
```

`apply_visitor()` 的单参用法可以用于存储了 `variant` 的容器, 生成的函数对象可以被标准库算法调用, 例如:

```
//对容器内的所有 variant 元素实施访问器操作
for_each(vec.begin(), vec.end(), apply_visitor(var_print()) );
```

7.8.5 与 any 的区别

`variant` 很像 `any`, 它们都能容纳一个可变类型的元素。但 `variant` 是有界类型, 元素类型范围由用户指定, 而 `any` 则是无界类型, 可以容纳任意的类型。

可能有人会认为 `variant` 没有存在的必要, 因为 `any` 的功能比它更强。但这是一个不正确的认识。

`any` 任意类型的自由往往也意味着程序员更多的责任, 就像 `void*` 指针, 灵活但要付出更多的安全检查代价。`variant` 的有界类型将取值类型限定在一个较小的范围内, 因而更容易处理、更容易理解。因为所有允许的类型都在模板参数列表中指定, `variant` 可以在编译期进行类型检查, 充分利用 C++ 作为静态强类型语言的好处。

`variant` 还提供泛型的 `visitor` 方式访问内部元素, 这是一个强大的设计模式, 从而导致了 `variant` 与 `any` 根本不同的使用方法。

`variant` 和 `any` 是两个互有关联但不能互相替代的容器, 虽然很多方面都很像。

7.8.6 高级议题

本小节讨论关于 `variant` 的一些高级议题。

二元访问器

之前的访问器都只能操作一个 `variant` 对象, 但操作两个 `variant` 对象的访问器也是允许的, 这在某些情况下是有用的, 比如对两个 `variant` 进行比较。

二元访问器同样要继承模板类 `static_visitor`, 例如一个比较访问器可以是这样:

```
struct is_var_equal: public static_visitor<bool>
{
    template <typename T, typename U>
```

```
bool operator()() { const T &t, const U &u } const
{
    if (typeid(T) != typeid(U))
        return false;
    return t == u;
}
};
```

函数 `apply_visitor()` 也支持使用二元访问器:

```
apply_visitor(is_var_equal(), v1, v2);
```

`apply_visitor()` 也能够生成二元访问器的延后访问函数对象:

```
apply_visitor(is_var_equal())(v1, v2);
```

BOOST_VARIANT_ENUM_PARAMS

在编写操作 `variant` 的泛型代码时, `variant` 的模板参数列表是一个很麻烦的问题, 需要手工写出 N 个含有从 1 到 N 个类型的 `variant` 类型, 比如:

```
template<typename T1, typename T2, ...>
void op_var(variant<T1, T2, ...> &v) {...}
```

为了简化这种代码的编写, `variant` 库提供了宏 `BOOST_VARIANT_ENUM_PARAMS`, 它使用了 `boost.preprocessor` 库的预处理技术, 可以简化模板函数或者模板类的特化声明。

宏 `BOOST_VARIANT_ENUM_PARAMS` 将展开为 `BOOST_VARIANT_LIMIT_TYPES` (默认为 20) 个的参数序列, 并为每个参数增加一个从 0 开始的后缀。因此:

```
template<BOOST_VARIANT_ENUM_PARAMS(typename T)>
void op_var(variant<BOOST_VARIANT_ENUM_PARAMS(T)> &v)
{ cout << v << endl;}
```

相当于:

```
template<typename T1, typename T2, ..., typename T19>
void op_var(variant<T1, T2, ..., T19> &v)
{ cout << v << endl;}
```

这样, 我们就可以很简单地写出操作任意 `variant` 对象的泛型代码。

使用 mpl

`variant` 可以使用模板元库 `mpl` 的类型容器 `mpl::vector` 来代替模板参数列表, 这时使用的类型数量可以不受默认 20 的限制。

工厂类 `make_variant_over` 的内部类型定义 `make_variant_over<mpl>::type` 定义

了 variant 的类型。例如：

```
#include <boost/mpl/vector.hpp>
typedef boost::mpl::vector<int, double, std::vector<string> > var_types;
                                                    //定义一个容纳 int、double 和
                                                    //vector<string>的类型容器
make_variant_over<var_types>::type v;
                                                    //产生 variant 类型
v = 2.13;
```

7.9 multi_array

C++98 标准库提供了 string 和 vector，它们是一维的数组，另有个组件 valarray 可以实现多维的数值数组，但它不是容器，而且设计存在一些问题。多维数组虽然实际应用中没有一个维数组那么普遍，但也是很有用的，在 C++98 中除了原始数组，只能用 vector<vector<T>> 来代替，虽然可以用，但不够方便。

multi_array 库解决了这个问题，它是一个多维容器，高效地实现了 STL 风格的多维数组，比使用原始多维数组或者 vector of vector 更好。

multi_array 位于名字空间 boost，为了使用 multi_array 组件，需要包含头文件 <boost/multi_array.hpp>，即：

```
#include <boost/multi_array.hpp>
using namespace boost;
```

7.9.1 类摘要

multi_array 很像标准容器，具有标准容器的大部分接口，它的类摘要如下：

```
template <typename ValueType, std::size_t NumDims>
class multi_array
{
public:
    //template typedefs
    template <std::size_t Dims> struct          subarray;
    template <std::size_t Dims> struct          const_subarray;
    template <std::size_t Dims> struct          array_view;
    template <std::size_t Dims> struct const_array_view;

    //constructors
    multi_array();
    template <typename ExtentList>
    explicit multi_array(const ExtentList& sizes, const storage_order_type& store
        = c_storage_order());
```

```

explicit multi_array(const extents_tuple& ranges, const storage_order_type&
    store = c_storage_order());

//iterators:
iterator      begin();
iterator      end();

//capacity:
size_type     size() const;
size_type     num_elements() const;
size_type     num_dimensions() const;

//element access:
template <typename IndexList>
    element&    operator()(const IndexList& indices);
reference      operator[](index i);
array_view<Dims>::type operator[](const indices_tuple& r);
const_array_view<Dims>::type operator[](const indices_tuple& r) const;

//queries
element*      data();
element*      origin();
const element* origin() const;
const size_type* shape() const;
const index*  strides() const;
const index*  index_bases() const;
const storage_order_type& storage_order() const;

//comparators
bool operator==(const multi_array& rhs);

//modifiers:
template <typename InputIterator>
    void    assign(InputIterator begin, InputIterator end);
template <typename SizeList>
    void    reshape(const SizeList& sizes)
template <typename BaseList> void reindex(const BaseList& values);
    void    reindex(index value);
template <typename ExtentList>
    multi_array&    resize(const ExtentList& extents);
multi_array&    resize(extents_tuple& extents);
};

```

上面的类摘要很长，但仍然省略了大量的成员函数，如拷贝构造函数和赋值函数、比较操作符重载、返回常量的迭代器等。

multi_array 是递归定义的，它的每个维度都是一个 multi_array，最底层的则是个一维的 multi_array，因此 multi_array 是组合模式的一个具体应用。

7.9.2 用法

`multi_array` 的模板参数很像标准容器，除了容纳的元素类型外，它多了一个维数的模板参数，用来指定多维数组的维数。例如，下面的代码

```
multi_array<int, 3> ma;
```

定义了一个三维数组 `ma`，相当于 `int ma[X][Y][Z]`。

但仅仅指定了维数还不够，多维数组还需要指定每个维度的具体值。

`multi_array` 为完成这个工作特意提供了 `extent_gen` 类和预定义的一个实例 `boost::extents`，它重载了 `operator[]`，用起来就像是一个原始多维数组，例如，下面的代码声明了一个维度为 2/3/4 的三维数组：

```
multi_array<int, 3> ma(extents[2][3][4]);
```

`multi_array` 的总维数可以用成员函数 `num_dimensions()` 获得，它的返回值就是模板参数中的 `NumDims`。函数 `shape()` 返回一个常量指针（数组），里面有 `NumDims` 个元素，表明了具体各个维度的值。例如：

```
BOOST_AUTO(shape, ma.shape());           //获得维度数组指针
for (size_t i = 0; i < ma.num_dimensions(); ++i)
{
    cout << shape[i] << ", ";             //逐个输出维度信息
}
cout << endl << ma.num_elements();
```

`multi_array` 重载了 `operator[]`，可以像使用普通数组那样访问内部的元素。`operator[]` 在 debug 模式下执行范围检查，如果访问索引超过了多维数组的范围会引发 `BOOST_ASSERT`（6.1 小节，213 页）断言失败从而退出程序，但在 release 模式下的行为则未定义。^①

示范 `multi_array` 基本用法的代码如下：

```
#include <boost/multi_array.hpp>
using namespace boost;
int main()
{
    //声明一个三维数组，维度为 2/3/4
```

① 很遗憾，`multi_array` 与其他标准容器一样，没有使用异常机制来处理错误，因此，保证数组范围不越界是库用户自己的责任。


```

multi_array<int, 3> ma(extents[2][3][4]);

//可以像普通多维数组那样遍历
for (int i = 0, v = 0; i < 2; ++i)
    for (int j = 0; j < 3; ++j)
        for (int k = 0; k < 4; ++k)
        {
            ma[i][j][k] = v++;           //给数组元素赋值
        }

//输出多维数组内所有元素
for (int i = 0; i < 2; ++i)
{
    for (int j = 0; j < 3; ++j)
    {
        for (int k = 0; k < 4; ++k)
        {
            cout << ma[i][j][k] << ",";
        }
        cout << endl;
    }
    cout << endl;
}

cout << ma[2][3][4];           //将引发一个越界错误
}

```

访问多维数组的另一种形式是使用一个位置索引序列, 传递给 `operator()`, `multi_array` 将从序列中提取 `NumDims` 个值定位直接访问元素, 某些时候这种访问方式比 `operator[]` 的效率更高。

例如, 下面的代码使用 `boost::array` 指定访问 `[0][1][2]` 位置的元素:

```

array<size_t, 3> idx = {0,1,2};
ma(idx) = 10;           //使用 operator()
cout << ma(idx);

```

`multi_array` 另外提供一个 `data()` 成员函数, 以指针的形式返回内部的数组, 它包含了所有的元素, 成员函数 `num_elements()` 可以获得元素的总数。

下面的代码使用了 7.1.6 小节, 243 页实现的 `ref_array`, 它代理了 `multi_array` 的元素指针, 因此就可以像一个标准容器那样使用, 也能用 `assign` 库初始化。例如:

```

multi_array<int, 2> ma(extents[3][4]) ;           //2 维数组

```

```
//使用 7.1.6 小节的 ref_array 适配数组
ref_array<int> ra(ma.data(), ma.num_elements());
ra.assign(list_of(1)(2)(3).repeat(9, 255));
```

7.9.3 多维数组生成器

`multi_array` 使用 `extents` 的创建方式很方便,但也有个小缺点,它硬编码了对象的创建,使客户代码缺乏灵活性。我们可以使用生成器模式 (Builder) 来把创建对象的过程封装起来。

我们把这个生成器命名为 `multi_builder`,它专门用于生产 `multi_array` 对象,因此具有与 `multi_array` 同样的模板参数。为了高效地返回多维数组对象避免拷贝,可以使用 `shared_ptr` (3.4 小节, 69 页) 来包装它:

```
template<typename T, std::size_t N>
class multi_builder:boost::noncopyable           //不可拷贝
{
public:
    typedef boost::multi_array<T, N> array_type;
    typedef boost::shared_ptr<array_type> type;
```

出于简化代码的目的,在 `multi_builder` 内部使用了两个 `typedef`,它们也可以被 `multi_builder` 的客户代码使用。

创建 `multi_array` 需要使用 `extents` 对象,它实际上是在名字空间 `boost::detail::multi_array` 里的一个模板类 `extent_gen<0>` 的实例, `operator[]` 接受维度递归地生成 `extent_gen<n>` 对象,因此我们只能使用 `any` (7.7 小节, 287 页) 来保存它^①:

```
private:
    boost::any ext;                               //any 对象
public:
    multi_builder():ext(boost::extents){}
    ~multi_builder(){}

```

接下来我们定义生成器的构建步骤 `dim()`,它是一个模板函数,可以逐个地传入维度,用 `any` 对象保存递归的 `extent_gen<n>` 对象:

```
template<std::size_t n>
void dim(std::size_t x)
{
    BOOST_STATIC_ASSERT(n >= 0 && n < N);        //静态断言

```

① 出于演示目的, `multi_builder` 使用了 `any` 类,其实也可以不使用 `any`,内部有一个成员变量 `ranges_`,它可以保存递归的 `extent_gen<n>`。

```

    ext = any_cast<boost::detail::multi_array::extent_gen<n> >(ext)[x];
}

```

当获得全部维度后, multi_builder 使用 create() 方法以 shared_ptr 的形式返回创建好的多维数组对象:

```

type create()
{
    return type(new
        array_type(any_cast<boost::detail::multi_array::extent_gen<N> >(ext)));
}
};

```

使用 multi_builder, 我们可以一步一步地构造出 multi_array 对象, 使多维数组的创建过程更加直观。示范它用法的代码如下:

```

#include "multi_builder.hpp"
...
multi_builder<int, 2> builder;           //生成器对象
builder.dim<0>(2);                       //第 1 个维度
builder.dim<1>(2);                       //第 2 个维度
BOOST_AUTO( mp , builder.create() );    //创建一个 2*2 的数组
cout << mp->size();                      //使用智能指针的箭头操作符访问成员
for (int i = 0, v = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
        (*mp)[i][j] = v++;              //也可以用解引用操作符*来使用[]
    }

```

现在看来 multi_builder 似乎没有什么用处, 不如直接的 extents 方式简单, 但之后 (7.9.7 小节) 我们会看到它给程序带来的好处: multi_builder 封装了 multi_array 的构造过程, 隔离了构造与客户代码。

7.9.4 改变形状和大小

multi_array 可以在运行时使用成员函数 reshape() 改变多维数组的形状, 即变动各个维度的大小, 但总维数和元素数量保持不变, 变动前的维度乘积与变动后的维度乘积必须相同。

reshape() 接受一个维度序列作为参数, std::vector 或者 boost::array 都可以, 通常 array 会更快。

例如, 把一个三维数组由 [2][3][4] 变为 [4][2][3], 可以这样:

```

multi_array<int, 3> ma(extents[2][3][4]);
assert(ma.shape()[0] == 2);

```

```
array<std::size_t, 3> arr = {4,3,2};
ma.reshape(arr);
assert(ma.shape()[0] == 4);
```

`multi_array` 也可以在运行时使用成员函数 `resize()` 改变多维数组的大小，动态增长或缩减元素的数量，但总维数不能变（它已经在模板参数中固定了）。`resize()` 函数既可以接受维度序列，也可以接受 `extents` 表达式。

下面的代码把刚才的三维数组 `ma` 由 `[4][2][3]` 变为 `[2][9][9]`：

```
ma.resize(extents[2][9][9]);
assert(ma.num_elements() == 2*9*9);
assert(ma.shape()[1] == 9);
```

在变动大小时 `multi_array` 将重新分配内存，原有的元素将被复制到新内存，然后被析构，新增的元素使用缺省构造函数构造。如果 `multi_array` 的元素总数变少，那么原有的多余元素将被删除。

7.9.5 创建子视图

多维数组的操作是比较复杂的，`multi_array` 库允许用户为多维数组创建一个只查看其中一部分数据的子视图（view），子视图既可以与原数组拥有相同的维数，也可以少于原来的维数。后一种情况被称为多维数组的“切片”（slice），它可以降低数组的维数，使之更容易处理。

我们用最容易理解的二维数组作为例子，假设我们有一个 `3*4` 的矩阵，：

```
typedef multi_array<int, 2> ma_type;
multi_array<int, 2> ma(extents[3][4]) ;
```

它里面的数据是：

```
(0,1, 2, 3)
(4,5, 6, 7)
(8,9,10,11)
```

我们需要使用 `index_gen` 类的预定义实例 `indices` 来定义子视图的索引范围。

`indices` 用法类似 `extents` 对象，重载了 `operator[]` 来限定范围，但它的参数不是单个的整数，而是一个 `multi_array<T, N>::index_range` 对象，它用来指定从多维数组中抽取的维度范围。例如：

```
typedef ma_type::index_range range; //简化类型定义
indices[range(0,2)][range(0,2)]; //创建一个临时 indices 对象
```

`multi_array` 的 `operator[]` 接受 `indices` 对象返回子视图，子视图的类型是

`multi_array<T, N>::array_view<N>::type`, 也可以直接用 `BOOST_AUTO`, 从而无需写出这个烦琐的类型声明。因此, 获取一个 2*2 的子视图可以是这样:

```
BOOST_AUTO(view, ma[indices[range(0,2)][range(0,2)]]);
```

相当于:

```
ma_type::array_view<2>::type view =  
    ma[indices[range(0,2)][range(0,2)]];
```

它的数据是 `ma` 的一部分:

```
(0,1)  
(4,5)
```

`multi_array` 的子视图也是个 `multi_array`, 拥有相同的接口, 可以用 `operator[]`、`num_dimensions()` 等成员函数操作, 但不能改变子视图的形状和大小。例如:

```
cout << view.num_elements() << endl;  
for (int i = 0; i < 2; ++i)  
{  
    for (int j = 0; j < 2; ++j)  
    {  
        cout << view[i][j] << ",";  
    }  
    cout << endl;  
}  
cout << *view.shape() << endl;
```

使用 `indices` 的时候也可以改变子视图的步长, 只需要在 `index_range` 的 `operator()` 里增加第三个步长的参数即可, 比如:

```
indices[range(0, 2)][range(0,4,2)];
```

将跳过第 2 维的 1、3 列数据, 子视图将会是:

```
(0,2)  
(4,6)
```

`view` 的成员函数 `strides()` 可以获得每个维度的步长, 类似 `shape()`, 它返回的是一个数组指针。

创建多维数组的切片与子视图类似, 只需要向 `indices` 的 `operator[]` 传入要被切片维度的整数值即可, 例如:

```
BOOST_AUTO(view, ma[indices[1][range(0,4)]]);
```

将把 `view` 降低到一维数组, 它切取了 `ma` 的第 2 行, 即:

(4,5,6,7)

7.9.6 适配普通数组

`multi_array` 是多维数组最常用的类，它自己管理内存，可以动态增长，但有的时候我们需要将一个一维数组适配成多维数组进行处理。

`multi_array` 库提供了另外两个类 `multi_array_ref` 和 `const_multi_array_ref` 来满足这一需求。它们可以把一段连续的内存（原始数组）适配成多维数组，用法类似 `smart_ptr` 库中的 `scope_array`（3.3 小节，67 页）。

`multi_array_ref` 和 `const_multi_array_ref` 都通过构造函数接受一块外界的内存，把它适配成多维数组的接口，适配后的用法除了不能动态增长外与 `multi_array` 完全相同，包括支持 `operator[]` 访问和 `reshape()` 改变维度。`const_multi_array_ref` 的功能要更少一些，它是只读的，适配后不能修改数组元素的值。^①

示范这两个类用法的代码如下：

```
int arr[12];
for (int i = 0; i < 12; ++i)                //数组赋初值
{
    arr[i] = i;
}

//multi_array_ref 适配成 3*4 的二维数组
multi_array_ref<int, 2> mar(arr, extents[3][4]);
for (size_t i = 0; i < 3; ++i)
{
    cout << "(";
    for(size_t j = 0; j < 4; ++j)
    {
        cout << mar[i][j]++;                //可以修改内部的值
        cout << (j!=3?', ':' ');
    }
    cout << ")" << endl;
}

//const_multi_array_ref 适配成 2*6 的二维数组
const_multi_array_ref<int, 2> cmar(arr, extents[2][6]);
for (size_t i = 0; i < 2; ++i)
{
    cout << "(";
    for(size_t j = 0; j < 6; ++j)
```

① 实际上，`multi_array_ref` 和 `multi_array` 都是 `const_multi_array_ref` 的子类，它们通过继承的方式逐步增加了多维数组的功能。

```

{
    cout << cmar[i][j];
    cout << (j!=5?' ':' ');
}
cout << ")" << endl;
}

```

程序的运行结果如下：

```

(0,1,2,3 )
(4,5,6,7 )
(8,9,10,11 )
(1,2,3,4,5,6 )
(7,8,9,10,11,12 )

```

概念上，7.1.6 小节（243 页）实现的 `ref_array<T>` 类似 `multi_array_ref<T, 1>`，它们均代理了一维数组，但两者的用途是不同的。`ref_array<T>` 是为了给普通数组或指针一个 STL 风格的接口，使得它们可以更容易地配合标准库工作，而 `multi_array_ref<T, 1>` 则是一个退化了的多维数组，在它上面依然可以执行多维数组的操作，但很多是无意义的。

7.9.7 高级议题

本小节讨论关于 `multi_array` 的一些高级议题。

用序列指定维数

除了使用 `extents` 对象，`multi_array` 还支持另一种创建对象的方法，直接向构造函数传入一个维度序列。例如声明一个维度为 2/3/4 的三维数组：

```

array<std::size_t, 3> arr = {2, 3, 4};
multi_array<int, 3> ma(arr);

```

这里必须使用 `array` 或者 `vector`，通常 `array` 会更快。

注意：我们不能直接使用 `assign` 库的 `list_of`，因为 `list_of` 生成的匿名数组不满足 `multi_array` 的序列概念要求，但可以把 `array` 或者 `vector` 和 `list_of` 组合起来使用，这样就可以在一句话中完成 `multi_array` 的构造，例如：

```

multi_array<int, 3> ma(vector<int>(assign::list_of(2)(2)));

```

使用序列的构造方式通常要比 `extents` 对象要快一些，也节省一点内存，而且有利于编写维度无关的代码。

在之前的 7.9.3 小节（306 页）我们实现了一个生成器 `multi_builder`，它封装了 `multi_array` 的创建过程，这里我们就把 `multi_builder` 的内部创建方式改为使用序列。

multi_builder 的对外接口和基本结构不变, 但我们不必再使用 any 和 extent_gen<n> 进行递归推导, 只需要增加一个 array 的成员变量就可以了, 实现明显比之前的要简洁很多, 也容易更理解:

```
//multi_builder.hpp
#include <boost/noncopyable.hpp>
#include <boost/array.hpp>
#include <boost/static_assert.hpp>
#include <boost/smart_ptr.hpp>
#include <boost/multi_array.hpp>

template<typename T, std::size_t N>
class multi_builder:boost::noncopyable
{
private:
    boost::array<std::size_t, N> d;
public:
    typedef boost::multi_array<T, N> array_type;
    typedef boost::shared_ptr<array_type> type;

    multi_builder(){}
    ~multi_builder(){}

    template<std::size_t n>
    void dim(std::size_t x)
    {
        BOOST_STATIC_ASSERT(n >= 0 && n < N);
        d[n] = x;
    }
    type create()
    {
        return type(new array_type(d));
    }
};
```

新的实现用 array 保存了维度信息, 也可以使用 extent_gen 来一次性完成维度计算, create() 可以是这样:

```
type create()
{
    typedef boost::detail::multi_array::extent_gen<N> gen_type;
    gen_type r;

    for (int i=0; i != N; ++i)
    {
        typedef typename gen_type::range range_type;
        r.ranges_[i] = range_type(0,d[i]);
    }
}
```



```
    return type(new_array_type(r));
}
```

使用生成器 `multi_builder`，我们就可以在 `multi_array` 的两种创建方式间随意切换，而不会影响到使用 `multi_array` 的客户代码。

元素的存储顺序

`multi_array` 默认以 C 数组的顺序存储元素，所以可以直接用 `data()` 获得 C 风格的数组指针，但为了与 Fortran 等老接口提供兼容，它也可以使用其他的存储顺序。

`multi_array` 的构造函数可以多接受一个存储顺序的标志，取值是 `c_storage_order`、`fortran_storage_order`、和 `general_storage_order`，默认是 `c_storage_order`。

成员函数 `storage_order()` 可以获得 `multi_array` 当前的存储顺序。

设置索引基数

`multi_array` 的数组索引遵循 C 的惯例，从 0 开始计数，但有的时候多维数组都从 0 计数会不太方便，有的维度使用其他的计数方式可能会更好。`multi_array` 提供了多种方式来变更索引基数来满足这个特殊的要求。

我们可以在创建多维数组时使用 `extent_range` 对象的构造函数指定索引范围，用来代替简单的整数维度定义，`extent_range` 的用法类似 `index_gen`。

例如，下面的代码创建了一个 4*4*4 的三维数组，第一维的索引从 1 到 4，第二维的索引从 0 到 3，第三维的索引从 -2 到 -1：

```
typedef multi_array<int, 3> ma_type;
typedef ma_type::extent_range range;
ma_type ma(extents [range(1,5)][4][range(-2,2)]);
ma[1][0][-2] = 10; //这是多维数组的第一个元素
```

成员函数 `reindex()` 可以在多维数组创建后随时更改索引基数，它有两种重载形式：如果传入一个整数 N，那么它把所有维度的基数都改为这个整数 N；如果传入一个整数序列，那么就按序列中的值逐个变更维度的基数；例如：

```
ma.reindex(1);
assert(ma[1][1][1] == 10); //多维数组的第一个元素
ma.reindex(array<size_t,3>(assign::list_of(1)(0)(-4)));
assert(ma[1][0][-4] == 10); //多维数组的第一个元素
```

成员函数 `index_bases()` 返回各维度索引的起始值数组指针，它的行为类似 `shape()`，如获取第一维的索引起始值：

```
cout << *ma.index_bases() << endl;
```

禁用全局对象

在创建多维数组时我们会用到 `extents` 和 `indices` 这两个“全局”对象，它们并不是真正的全局对象，而是被定义在一个匿名的名字空间里，相当于两个静态全局变量，只能被本源文件使用。

如果工程中有多处使用了 `multi_array` 组件，那么可能会有很多个 `extents` 和 `indices` 对象，它们的构造将会是一个不小的开销，可以在包含头文件 `<boost/multi_array.hpp>` 前定义宏 `BOOST_MULTI_ARRAY_NO_GENERATORS`，以禁止它们的预定义。这时多维数组的创建可以改用维度序列的方式。

使用 7.9.3 小节（306 页）实现的生成器 `multi_builder` 可以做到隔离这个变化。

更灵活的生成子视图

7.9.5 小节（308 页）在生成子视图时我们使用了 `index_range` 类，它接受三个整数，定义了一个左闭右开的索引范围，步长默认为 1。但当时我们只展示了基本的用法，它还有许多其他的功能，善加利用可以使子视图和切片的工作更加简单。

不带参数的 `index_range` 构造函数表示该维度的所有元素，这可以在不知道具体维度的情况下获取该维度的元素。

`index_range` 还重载了比较操作符，可以使用比较操作符指定索引的范围，配合不带参数的 `index_range` 构造函数可以灵活地指定子视图的范围。

下面的代码定义了一个维度为 [3][4][7] 的子视图，它使用 `index_range` 分别取了第一维的前 3 列，第二维的 2-5 列和第三维的所有元素：

```
typedef multi_array<int, 3> ma_type;
typedef ma_type::index_range range;
ma_type ma(extents [9][8][7]);
BOOST_AUTO(view , ma[indices[range()< 3][2<=range()<= 5][range()]] );
cout << *view.shape() << endl;
```

7.10 property_tree

`property_tree` 是一个保存了多个属性值的树形数据结构，可以用类似路径的简单方式访问任意节点的属性，而且每个节点都可以用类似 STL 的风格遍历子节点。`property_tree` 特别适合于应用程序的配置数据处理，可以解析 xml、ini、json 和 info 四种格式的文本数据，使

用它能够减轻自己开发配置管理的工作。

property_tree 支持多种数据格式，接下来将以 XML 作为重点介绍对象，因为 XML 格式是现在被广泛应用的数据交换格式，很多人都很熟悉它，并且有很多处理 XML 的应用软件。

property_tree 的属性树结构与 XML 的结构非常相似，处理方式也很相近，两者对比可以获得最佳的学习效果。实际上，property_tree 内部使用的就是一个小巧快速的开源 xml 解析器——rapidxml。

property_tree 位于名字空间 boost::property_tree，要使用 property_tree 和它的 XML 解析组件，需要包含头文件 <boost/property_tree/ptree.hpp> 和 <boost/property_tree/xml_parser.hpp>，即：

```
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/xml_parser.hpp>
using namespace boost::property_tree;
```

7.10.1 类摘要

property_tree 库的核心类是 basic_ptree，它的类摘要如下：

```
template<typename Key, typename Data, typename KeyCompare>
class basic_ptree {
public:
    typedef basic_ptree< Key, Data, KeyCompare > self_type;
    typedef std::pair< const Key, self_type >    value_type;

    //构造与赋值
    basic_ptree();
    basic_ptree& operator=(const self_type &);

    //基本容器操作
    void swap(self_type &) ;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    iterator begin() ;
    iterator end() ;
    value_type & front() ;
    value_type & back() ;
    iterator insert(iterator, const value_type &) ;
    iterator erase(iterator) ;
    iterator push_front(const value_type &) ;
    iterator push_back(const value_type &) ;
    void pop_front() ;
    void pop_back() ;
```

```

void reverse() ;
void sort() ;

//数据访问
data_type & data() ;
void clear() ;
template<typename Type> Type get(const path_type &) const;
self_type & get_child(const path_type &) ;
self_type & put_child(const path_type &, const self_type &) ;
self_type & add_child(const path_type &, const self_type &) ;
};

```

`basic_ptree` 的接口很像标准容器 `std::list`，可以执行很多基本的元素操作，如使用 `begin()` 和 `end()` 遍历当前属性树的所有子节点。此外它还增加了操作属性树的 `get()`、`get_child()`、`get_value()`、`data()` 等额外的操作，因而很容易学习。

`basic_ptree` 有两个重要的内部类型定义 `self_type` 和 `value_type`。`self_type` 是 `basic_ptree` 模板实例化后自身的类型，它也是子节点的类型。`value_type` 是节点的数据结构，它是一个 `std::pair`，含有节点的属性名 (`first`) 和节点自身 (`second`)。这两个类型定义意味着 `basic_ptree` 使用了组合模式，它是由多个 `basic_ptree` 复合而成的集合体，就像是一个 `std::list<key, basic_ptree>`。

同标准字符串类 `basic_string` 一样，通常我们不直接使用 `basic_ptree`，而是使用预定义好的 `typedef`: `ptree`、`wptree`、`iptree`、`wiptree`，这些名字中的前缀 `i` 表示忽略大小写，前缀 `w` 表示支持宽字符。

接下来我们主要使用 `ptree`，它的定义是：

```
typedef basic_ptree<std::string, std::string> ptree;
```

7.10.2 读取配置信息

我们使用一个简单的 xml 配置文件 `conf.xml` 作为例子，它保存了一些基本的配置信息：

```

<conf>
  <gui>l</gui>
  <theme>matrix</theme>
  <urls>
    <url>http://www.url1.com</url>
    <url>http://www.url2.com</url>
    <url>http://www.url3.com</url>
  </urls>
  <clock_style>24</clock_style>
</conf>

```

要解析这个配置文件，我们首先要用 `read_xml()` 函数解析 xml 并初始化 `ptree`，它有两种

重载形式，可以接受文件名或 IO 流，声明分别是：

```
void read_xml(const string &, Ptree &pt);
void read_xml(basic_istream &, Ptree &pt);
```

调用 `read_xml()` 函数就可以将配置信息读入到 `ptree` 中：

```
ptree pt; //声明一个 ptree 对象
read_xml("conf.xml", pt); //读取 xml 配置信息
```

在初始化 `ptree` 后，我们可以用模板成员函数 `get()` 通过路径访问属性树内的节点，用模板参数指明获取属性的类型。

`property_tree` 的路径语法很自然很容易理解，与 `xpath` 的路径语法类似，但它使用点号 (.) 作为路径分隔符，例如：

```
pt.get<string>("conf.theme") //获取<theme>节点的字符串值
pt.get<int>("conf.clock_style") //获取 clock_style 的 int 值
```

`ptree` 的 `get()` 函数支持缺省值的用法，在指定路径的同时可以指定缺省值。如果属性树中不存在该属性，则使用缺省值。由于缺省值已经有类型信息，因此 `get()` 的返回值类型可以被自动推导出来，不必再加模板参数。例如：

```
pt.get("conf.no_prop", 100); //取 no_prop 节点的值，如果不存在返回整数 100
```

对于有多个子节点的节点，比如要获得 `conf.urls` 子节点的值，我们就需要使用 `get_child()` 先获得子节点对象，再像使用标准容器一样用 `begin()` 和 `end()` 遍历子节点。迭代器指向 `ptree` 的 `value_type`，它的 `second` 成员是子节点自身，值可以直接用 `data()` 或者 `get_value<>()` 访问：

```
BOOST_AUTO(child, pt.get_child("conf.urls"));
for (BOOST_AUTO(pos, child.begin()); pos != child.end(); ++pos)
{ cout << pos->second.get_value<string>() << ", "; }
```

完整的示范代码如下：

```
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/xml_parser.hpp>
int main()
{
    using namespace boost::property_tree;

    ptree pt; //声明 property_tree 对象
    read_xml("conf.xml", pt); //解析 xml 配置文件

    //示范 get<>() 函数的用法
```

```

cout << pt.get<string>("conf.theme") << endl;
cout << pt.get<int>("conf.clock_style") << endl;
cout << pt.get<long>("conf.gui")<< endl;
cout << pt.get("conf.no_prop", 100)<< endl;           //缺省值用法

BOOST_AUTO(child, pt.get_child("conf.urls"));
for (BOOST_AUTO(pos, child.begin());
     pos != child.end(); ++pos)
{
    cout << pos->second.data() << ", ";           //改用 data() 函数
}
cout << endl;
}

```

程序的运行结果如下:

```

matrix
24
1
100
http://www.url1.com,http://www.url2.com,http://www.url3.com,

```

7.10.3 写入配置信息

`property_tree` 不仅能够读取配置信息,也可以写入配置信息。它的操作具有对称性,使用模板成员函数 `put()` 可以修改属性树的节点值,如果子节点不存在就相当于新增节点,如果子节点存在则就地修改。不过 `put()` 通常不需指定模板参数,因为属性值的类型可以自动推导。例如:

```

pt.put("conf.theme", "Matrix Reloaded");
pt.put("conf.clock_style", 12);

```

对于深层次的节点,如 `conf.urls.url`,也可以同样操作:

```

pt.put("conf.urls.url", "http://www.url4.org");

```

当修改完成后,使用 `write_xml()` 函数保存配置信息,它的声明与 `read_xml()` 类似:

```

void write_xml(const string &, Ptree &pt);
void write_xml(basic_ostream &, Ptree &pt);

```

完整的示范代码如下:

```

#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/xml_parser.hpp>
int main()
{
    using namespace boost::property_tree;

    ptree pt;

```

```

read_xml("conf.xml", pt);

pt.put("conf.theme", "Matrix Reloaded");
pt.put("conf.clock_style", 12);
pt.put("conf.gui", 0);
pt.put("conf.urls.url", "http://www.url4.org");

write_xml(cout, pt);
}

```

这里我们使用了 `write_xml()` 的第二种形式，向标准输出流 `cout` 写入数据。程序的运行结果是：

```

<?xml version="1.0" encoding="utf-8"?>
<conf><gui>0</gui><theme>Matrix Reloaded</theme>
<urls><url>http://www.url4.org</url><url>http://www.url2.com</url><url>http://www.url3.com</url></urls>
<clock_style>12</clock_style></conf>

```

输出结果中的 `gui`、`theme` 属性都如预期，但 `urls` 的子节点却有点奇怪，原来的 `url1` 不见了，被替换成了 `url4`。这是由于 `put()` 的操作性质决定的，它只是新增或者修改已有属性值，不能单纯添加新节点。

如果要向属性树加入新节点，我们需要使用 `add()` 函数，例如：

```
pt.add("conf.urls.url", "http://www.url4.org");
```

7.10.4 更多用法

`basic_tree` 有一部分接口完全符合 STL 容器规范，使用起来就像是一个 `std::list<string, ptree>`，可以在它上面执行各种基本操作。

假设我们有如下的 `xml` 数据：

```

<!-- a.xml -->
<a>string</a>
<b>100</b>
<c>false</c>

```

那么可以用 `find()` 成员函数查找 `<a>` 节点的值：

```

ptree pt;
read_xml("a.xml", pt);

ptree::assoc_iterator pos = pt.find("a");
cout << pos->second.data() << endl;

```

但这样的操作明显没有 `get()` 函数方便，而且 `find()` 也不能实现深层次查找。

`get()` 函数有三种形式, 除了前面我们见过的基本形式和缺省值形式外, 还有一种 `get_optional()`, 它也是缺省值形式的用法, 但使用了 `optional` 库 (4.3 小节, 113 页) 来包装返回值, 如果 `get` 失败则 `optional` 对象是无效的, 例如:

```
assert(pt.get_optional<string>("a"));
assert(!pt.get_optional<string>("not_exists"));
```

同样地, 直接获取树节点值的 `get_value()` 也有 `get_value_optional()` 的形式, 用法类似。

7.10.5 XML 数据格式

`property_tree` 本身没有实现 xml 解析器, 而是使用非 Boost 的开源项目 `rapidxml`, 因此, `property_tree` 的 xml 解析功能受限于 `rapidxml`。

`rapidxml` 是一个基于 DOM 模型的小而快速的 xml 解析器, 比市面上能够找到的大部分 xml 解析器都要快。但它不完全支持 xml 标准, 不支持 DTD、名字空间, 也不支持编码转换, 默认 xml 是 UTF-8 的格式。

`property_tree` 使用 `rapidxml` 将所有的 xml 节点转换为属性树对应的节点, 节点的标签名是属性名, 节点的内容是属性值。节点的属性保存在该节点的 `<xmlattr>` 下级节点, 注释保存在 `<xmlcomment>` 中, 节点的文本内容 (如 CDATA) 保存在 `<xmltext>` 中。

`property_tree` 解析 xml 的方式有点像 `xpath/xquery`, 但它的功能有限, 不能像 `xpath` 那样执行模糊查找、条件查找, 也不支持如 `find("xxx.yyy")` 的深度查找。

`property_tree` 对 xml 的转换不是可逆的, 因为它不是用来专门处理 xml 数据, 因此, 读取 xml 数据再以 xml 格式写回可能会丢失一些格式信息和空白字符, 与原文件不完全等同, 但 xml 的配置数据不变。

我们将 `conf.xml` 做少许修改, 增加一些属性和注释, 看一下 xml 特殊值的访问方法:

```
<conf>
  <!-- conf comment -->
  <gui>1</gui>
  <theme id="001">matrix</theme>
  <urls>
    <!-- urls comment -->
    <url>http://www.url1.com</url>
  </urls>
  <clock_style name="local">24</clock_style>
</conf>
```


那么，下面的语句

```
ptree pt;
read_xml("conf.xml", pt);
cout << pt.get<string>("conf.<xmlcomment>") << endl;
cout << pt.get<string>("conf.clock_style.<xmlattr>.name") << endl;
cout << pt.get<long>("conf.theme.<xmlattr>.id") << endl;
cout << pt.get<string>("conf.urls.<xmlcomment>") << endl;
```

的输出是：

```
conf comment
local
1
urls comment
```

7.10.6 其他数据格式

property_tree 除了支持工业标准 XML 外还支持 json、ini 和自有的 info 三种格式，下面就对这三种格式做简单的介绍。

json

json 是 JavaScript Object Notation 的缩写，是来自 JavaScript 语言的一种数据交换格式。它结构简单紧凑，类似 xml 但没有那么多的繁文缛节，详细的 json 语法不在这里介绍，读者可自行查找相关的资料。

下面定义了一个简单的 json 格式数据，与之前的 xml 数据类似，需要注意：属性名要用引号括起来后面接冒号；属性值如果是字符串也要使用引号；每个属性后要用逗号分隔，除非它是复合语句的最后一句。

```
{
  "conf":
  {
    "gui": 1,
    "theme": "matrix",
    "urls":
    {
      "url": "http://www.url1.com"
    },
    "clock_style": 24
  }
}
```

要解析 json 格式的数据，需要包含头文件 `<boost/property_tree/json_parser.hpp>`，使用 `read_json()` 和 `write_json()` 读写配置，其他的用法与 xml 完全相同：

```

#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>           //json 头文件
int main()
{
    using namespace boost::property_tree;

    ptree pt;                                           //声明 property_tree 对象
    read_json("conf.json", pt);                       //解析 json 配置文件

    cout << pt.get<string>("conf.theme") << endl;
    cout << pt.get<int>("conf.clock_style") << endl;
    cout << pt.get<long>("conf.gui")<< endl;
    cout << pt.get("conf.no_prop", 100)<< endl;

    BOOST_AUTO(child, pt.get_child("conf.urls"));
    for (BOOST_AUTO(pos, child.begin());
         pos != child.end(); ++pos)
    {
        cout << pos->second.data() << ", ";
    }
}

```

ini

ini 格式是 Windows 操作系统下非常流行的一种配置数据格式，曾经被广泛应用，但它较 xml 和 json 有些不足：不支持多级层次，只有简单的 key/value 形式，不能表达复杂的配置，因此其他格式的配置数据可能无法转换到 ini 格式。

ini 中的每个 section 被创建为一个属性树节点，section 里的属性成为它的子节点，全局属性（不属于任何 section）加入属性树的根节点，空的 section 则不加入属性树。

解析 ini 格式的数据需要包含头文件<boost/property_tree/ini_parser.hpp>，使用 read_ini() 和 write_ini() 读写配置，其他的用法与 xml 完全相同。

info

info 格式是 property_tree 的专用格式，它较 xml 简洁，风格类似 json，但带有许多 C++ 的特征，例如#include 语句、斜杠字符转义等等，特别适合 C++ 程序员使用。例如：

```

conf           ;comment
{
    gui 1
    theme "matrix"
    urls
    {
        url "http://www.url1.com"
        url "http://www.url2.com"
    }
    clock_style 24
}

```

```
}
```

要解析 info 格式的数据，需要包含头文件<boost/property_tree/info_parser.hpp>，使用 read_info() 和 write_info() 读写配置，其他的用法与 xml 完全相同。

info 格式简单高效，很适合在程序中初始化大量的数据。

7.10.7 高级议题

本小节讨论关于 property_tree 的一些高级议题。

使用其他路径分隔符

property_tree 默认使用点号分隔路径，虽然不太符合操作系统和 xml 的习惯，但用来表示有层次的属性还是很自然的。但有的时候属性名中会出现点号（这种情况应该很少），我们也可以使用其他分隔符，例如斜杠。

get() 等函数的路径参数通常我们传的都是字符串，但实际上 property_tree 的路径是一个 basic_ptree::path_type 类型，它是一个内部的 typedef，真正的类型是 string_path，类摘要如下：

```
template <typename String, typename Translator>
class string_path
{
    explicit string_path(char_type separator = char_type('.'));
    string_path(const String &value, char_type separator = char_type('.'));
    string_path& operator /=(const string_path &o);
}
```

string_path 的构造函数默认使用点号作为路径分隔符，但也可以传入其他字符修改，它还重载了 operator/= 操作符，可以组合路径。

使用 string_path，我们可以这样写 property_tree 的访问路径：

```
ptree::path_type p('/'); //改为使用斜杠分隔属性名
p /= "conf";
p /= "theme.name1"; //这时属性名可以含有点号
cout << pt.get<string>(p) << endl;
```

或者可以更简单，直接在 get() 函数参数里构造临时变量：

```
pt.get<string>( ptree::path_type("conf/theme.name1", '/'));
```

读取 xml 的选项

read_xml() 函数在读取 xml 配置文件时可以传入标志位控制读取行为，标志常量位于名字

空间 `boost::property_tree::xml_parser`, 它们是:

- `no_concat_text` : 文本节点不连接, 而是分别在各子节点存储;
- `no_comments` : 不读入 xml 注释信息;
- `trim_whitespace`: 压缩文本中的白空格。

例如, 下面的代码将这三个标志联合使用, 读取的 xml 将压缩空格没有注释:

```
read_xml("a.xml", pt, xml_parser::no_concat_text |  
        xml_parser::no_comments | xml_parser::trim_whitespace);
```

异常

如果配置文件有误(读取时不存在, 写入时只读、格式错误)、属性树路径错误或者数据不存在, `property_tree` 就会通过 `throw_exception()` 函数(参见 4.9.6 小节, 143 页)抛出异常报错。

配置文件的读写错误会抛出 `xxx_parser_error` 的异常, 如 `xml_parser_error`、`json_parser_error`。

`property_tree` 本身的异常有三种, 它们都派生自 `std::runtime_error`。 `ptree_error` 是 `property_tree` 库的异常基类, 子类 `ptree_bad_data` 和 `ptree_bad_path` 是具体的异常类型, 内部用 `any` (参见 7.7 小节, 287 页)保存了错误信息。例如:

```
try  
{  
    cout << pt.get<string>("conf.patherror") << endl;  
}  
catch (std::exception &e)  
{  
    cout << e.what() << endl;  
}
```

输出是:

```
No such node (conf.patherror)
```

7.11 总结

本章讨论了 10 个 Boost 容器和数据结构, 它们从不同的方面扩展了标准容器。

我们首先讨论了五个容器: `array`、`dynamic_bitset`、`unordered`、`bimap` 和 `circular_buffer`, 它们都是对标准容器在不同方向上的扩展。具体说明如下:

`array` 是对 C++ 内建数组的一个简单包装, 为数组增加了迭代器支持和一些有用的成员函数, 使数组的行为更像标准容器 (但不是, 永远也不可能是), 而性能上没有任何损失。如果读者还在使用普通数组, 建议尽快转向 `array`, 不需要花费太多的学习时间就可以使代码获得改善。

`dynamic_bitset` 类似 `std::bitset`, 可以容纳任意数量的二进制位, 用于位运算或者集合运算, 接口几乎与 `std::bitset` 相同, 很容易使用。`dynamic_bitset` 的最佳搭档是 `BOOST_BINARY` 宏, 可以获得最高的运行效率。

`unordered` 实现了与 `std::set`、`std::map` 类似的散列容器, 它使用散列函数、桶而不是二叉树来组织数据, 接口也基本与标准容器相同, 大多数情况下可以提供比标准容器更好的性能。`STLport`、`Dinkumware STL` 等很多标准库实现也同时提供了功能相同的散列容器, 其名称是 `hash_xxx`, 但性能都不能和 `boost` 库的实现相比。

`bimap` 是一个双向 `map`, 它扩展了 `std::map` 的内涵, 引入了左组、右组、关系、视图、投射等许多概念, 使 `bimap` 超越了简单的双向 `map`, 成为了一个威力强大的映射关系容器。`bimap` 乍看上去很简单, 但一旦深入就会发现拥有十分复杂的特性, 可以用模板参数进行各方面的配置, 组合的数量将以几何级数快速增长。使用 `bimap` 要谨记: 够用的、简单的就是好用的, 千万不要被 `bimap` 强大而复杂的能力而诱惑。

`circular_buffer` 是一个循环队列, 它的操作很接近 `deque`, 接口也符合标准容器, 因而很容易学习和使用, 完全可以把它当做标准容器用在各个地方。只需要注意一点, 它内部的内存是循环利用的, 理解环形的数组需要费点力气。很多应用程序都使用了生产者/消费者模式, 其中有用两者工作的有界缓冲区, `circular_buffer` 恰好可以满足这个需求, 可以很容易地基于 `circular_buffer` 实现这样的缓冲区类。

其次, 我们讨论了 `tuple`、`any` 和 `variant`, 它们都是能够容纳不同类型的特殊容器, 值得放在一起研究, 通过比较它们之间的相似和不同, 可以更深入地理解它们:

`tuple` 是对 `std::pair` 的泛化。它表现为一种数据类型, 类似一个 `struct`, 但也可以是一种能容纳任意类型的静态容器, 可以对不同种类的数据“打包”。`tuple` 最常用的一个方式是把它用做函数的返回值, 这样函数就不必使用输出参数就可以返回任意多个数据。有了 `tuple`, 作为组合数据用的 `struct` 就失去了存在的意义, `tuple` 不仅高效, 而且可以使用很多泛型的方法去操作它。`tuple` 还是一个威力强大的类型容器, 是模板元编程的基础。

`any` 和 `variant` 都是只能容纳一个可变类型的容器, 它们自身提供的操作也很有限, 需要有外部的自由函数的支持才能工作。`any` 可以持有任意类型的数据, `variant` 的类型则需要在编译期指定范围, 因此 `any` 的类型检查只能在运行时, 而 `variant` 则把一部分类型检查工作放在了编译期, 具有更多的类型安全性。`any` 和 `variant` 的另一个区别在于对内部值的访问方式上。

`any` 只能使用 `any_cast()` 函数, 类似于转型操作, 灵活但需要很多的检查工作。`variant` 可以使用具有同样功能的 `get()` 函数, 但提供了使用访问者模式的 `static_visitor`, 可以编写出更安全更易维护的代码。

`tuple` 与 `variant` 在类型的声明上有相似之处, 都有一个类型参数列表, 但 `tuple` 是每个类型一个元素, 而 `variant` 则是所有类型同时只能有一个元素。

最后, 我们研究了 `multi_array` 和 `property_tree` 这两个组件, 它们都使用了组合模式实现了复杂的数据结构。

`multi_array` 是一个泛型的多维容器, 比原始多维数组或者 `vector of vector` 更高效, 操作更灵活。它提供了丰富的操作接口, 可以在运行时动态地改变维度、大小和索引基数, 也可以使用子视图或切片访问其中的部分数据。多维数组的用法很复杂, 但 `multi_array` 把它大大地简化了。

`property_tree` 库是自 1.41 版新出现的容器, 但已经相当成熟。它可以解析 `xml`、`json`、`ini` 和 `info` 四种格式的配置文件, 在内部构造出一个属性树, 可以用类似 `xpath` 的简便语法访问任意节点的数据。很多应用程序都或多或少地需要配置文件, 我们已经重复开发了太多这样的类, 可以保证, `property_tree` 完善的功能能够满足所有对配置文件的要求, 节省这方面的大量开发成本。

本章还实现了一个很有用的模板类: `ref_array`, 它补充了 `array` 的不足, 可以代理一个普通数组以提供类似 STL 容器的行为, 在兼容 C 风格 API 时会非常有用。

第 8 章

算法

C++标准库中已经提供了大量的算法（超过 100 个），可以对容器执行统计、查找、赋值、排序等许多操作，差不多囊括了实际软件开发工作中所有可能用到算法。如果开发人员能够对这些算法捻熟于心，那么它们将发挥与 STL 容器同样甚至更强大的威力。

C++标准库珠玉在前，Boost 算法库几乎“无事可做”。但它还是贡献了几个非常重要的库：graph——基于图数据结构的算法、gil——图像处理算法和 string_algo——字符串处理算法（已经在 5.3 节，175 页有详细介绍）。

本章并不讨论 graph 和 gil，它们已经超出了本书的范围，而是讨论 Boost 库中的另两个算法组件 foreach 和 minmax，这两个组件不是很大但很实用，相信读者会很快阅读完并掌握它们。

8.1 foreach

循环是程序设计中经常用到的基本语言结构，最常见的就是 for 循环了。我们都很熟悉如 `for(int i = 0; i < 10; ++i)` 或者 `for(p = c.begin(); p != c.end(); ++p)` 这样的经典形式，然而随着历史车轮的前进，这种写法越来越显出了其烦琐和冗余的缺点。

较新的编程语言如 Java、C#、Python 等都为遍历容器的循环提供了特别的语法。比如，在 Java 中，循环遍历一个 List 容器，可以这样写：

```
///java 代码，目前 C++98 不能使用
for(int x : aList)                                //aList 是一个容器，如 ArrayList<int>
{
    //do something...
}
```

在 Python 中则更加简洁自然:

```
for x in v:                                     #v 可以是一个 list、set 或 tuple
    print x
```

这种代码形式比老式的 for 循环更加简洁, 更清晰地表明了代码编写者的意图, 也更不容易犯遍历不全或者越界的错误。

C++标准委员会已经意识到了这一点, 新的 C++0x 标准将以类似的形式提供语言级别的容器遍历支持, 但在 C++0x 正式推出之前, 我们暂时不能享受这一便利。

Boost 的 foreach 库使用宏提供类似的序列遍历, 简便好用, 不需要使用麻烦的迭代器, 也不需要定义新的函数对象 (比如 lambda)。

使用 foreach 库, 需要包含头文件 <boost/foreach.hpp>, 即:

```
#include <boost/foreach.hpp>
```

8.1.1 用法

foreach 库提供两个宏 BOOST_FOREACH 和 BOOST_REVERSE_FOREACH, 分别来实现对序列的正向遍历和反向遍历, 使用起来非常简单方便, 就如同其他编程语言中的 foreach 循环:

```
#include <boost/foreach.hpp>
#include <boost/assign.hpp>
int main()
{
    using namespace boost::assign;           //用 assign 初始化序列值
    vector<int> v = (list_of(1),2,3,4,5);    //boost::assign

    BOOST_FOREACH(int x, v)                  //foreach 遍历 vector 容器
    {
        cout << x << ", ";
    }
    cout << endl;

    string str("boost foreach");
    BOOST_FOREACH(char& c, str)              //也可以遍历 string 容器
    {
        cout << c << "-";
    }
}
```

运行结果如下:

```
1,2,3,4,5,
b-o-o-s-t- -f-o-r-e-a-c-h-
```


8.1.2 详细解说

虽然 `BOOST_FOREACH` 表现为一个宏，但内部却包含了精巧的设计，它不使用动态内存分配、虚拟函数、函数指针调用等会降低效率的手法，其循环的执行效率几乎与手写循环同样高效。

使用 `BOOST_FOREACH` 要注意一点，它遍历序列有一个前提：假设序列是稳定的，也就是说在 `BOOST_FOREACH` 循环中不能改变序列的长度，也不能增减序列的元素，否则会导致遍历使用的迭代器失效，发生未定义错误。此外，`BOOST_FOREACH` 宏的使用方法几乎与普通的 `for` 完全相同，可以在循环体里做任何事情（除了刚才提到的变动序列）：`break`，`continue`，`return`，函数调用，甚至是再嵌入一个 `BOOST_FOREACH`。它与 `for` 唯一不同的是循环开始时的声明，只接受两个参数，其定义如下：

```
#define BOOST_FOREACH(VAR, COL)
```

第一个参数 `VAR` 是循环变量声明，第二个 `COL` 是要遍历的序列。`COL` 参数很容易使用，只需要把序列变量的名字放在那里就可以了，也允许是一个函数的返回值（右值），而第一个参数 `VAR` 则有较多的变化。

通常情况下 `VAR` 可以直接声明循环变量，但在 `BOOST_FOREACH` 循环中获得的将是序列内元素的拷贝，可以使用但无法修改。如果想要高效地使用序列内元素或者修改序列，则需要声明时使用引用的形式，如：

```
BOOST_FOREACH(int &x, v)           //循环变量引用容器的元素
    ++x;
```

与普通的 `for` 循环一样，变量类型声明的位置也是自由的，可以把它拿到 `BOOST_FOREACH` 前面定义：

```
set<int> s = list_of(10)(20)(30);    //标准集合容器
int y;                               //外置循环变量声明
BOOST_FOREACH (y, s)                //foreach 循环
{
    if (++y % 7 == 0)
    {
        cout << y << endl;
        break;
    }
}
```

8.1.3 使用 `typeof`

因为 `BOOST_FOREACH` 要求的仅仅是个变量的声明，没有表达式，所以这里无法直接使用

BOOST_TYPEOF 或 BOOST_AUTO 来简化。但可以采用一个略微取巧的办法来解决——通过序列的迭代器得到元素的类型，只需要付出很小的代价：

```
set<int> s = list_of(10)(20)(30);
BOOST_TYPEOF(*s.begin()) y;           //也可以用 BOOST_AUTO(y, *s.begin());
BOOST_FOREACH(y, s)
{ ... }
```

甚至可以把 BOOST_FOREACH 与 BOOST_TYPEOF 嵌套起来使用（但不能用 BOOST_AUTO），不会有任何问题，这两个宏可以正常地配合工作，像这样：

```
BOOST_FOREACH(BOOST_TYPEOF(*s.begin()) y, s)
{ ... }
```

需要注意的是，这个技巧不是一个通用的解决方案，有时候 typeid 库获得的类型不与 BOOST_FOREACH 循环所要求的匹配，这时我们只能再回归到原始的类型声明方式上。

8.1.4 更优雅的名字

如果读者觉得大写的宏定义看起来过于“刺眼”，破坏了代码的“优雅”，那么可以再进行宏定义，把 BOOST_FOREACH 改成自己喜欢的名字，用起来更方便。例如使用小写的 foreach 和 reverse_foreach：

```
#define foreach BOOST_FOREACH
#define reverse_foreach BOOST_REVERSE_FOREACH
```

注意，不要定义 foreach (x,y) 的宏形式，当 foreach 的参数也是宏的话可能引发问题。还有，选择简化名称时小心不要与 std::for_each 算法或者其他名字发生冲突。

下面的代码使用 foreach 名字演示了 BOOST_FOREACH 更多的用法：

```
#include <boost/foreach.hpp>
#include <boost/assign.hpp>
#define foreach BOOST_FOREACH
#define reverse_foreach BOOST_REVERSE_FOREACH
int main()
{
    using namespace boost::assign;

    //遍历普通数组
    int ar[] = {1,2,3,4,5};
    foreach(int& x, ar)
        cout << x << " ";
    cout << endl;

    //遍历 map
```

```

map<int, string> m = map_list_of(1, "111")(2, "222")(3, "333");

pair<int, string> x;                                //map 容器只能使用外置变量声明
foreach(x, m)
    cout << x.first << x.second << endl;

//遍历用 vector 实现的二维数组
vector< vector<int> > v =
    list_of( list_of(1)(2))(list_of(3)(4));
foreach(vector<int> & row, v)
{
    reverse_foreach(int &z, row)
        cout << z << ", ";
    cout << endl;
}
}

```

8.1.5 支持的序列类型

BOOST_FOREACH 建立在 boost.range 的概念上,任何符合 range 概念的容器序列就自动地被支持。具体来说,BOOST_FOREACH 支持所有 STL 容器 (vector、deque、list、set、map、string)、原始数组和 C 字符串 (即以 NULL 结尾的字符数组) 以及元素是迭代器的 std::pair。此外,BOOST_FOREACH 还支持 Boost 库内的大部分容器类型。

示范 BOOST_FOREACH 应用于 Boost 容器的代码如下:

```

using namespace boost::assign;

//boost::array
array<int, 5> ar = (list_of(1), 2, 3, 4, 5);
foreach(int x, ar)
    cout << x << " ";
cout << endl;

//迭代器的 pair
pair<array<int, 5>::iterator, array<int, 5>::iterator>
    rng(ar.begin(), ar.end() - 2);
foreach(int x, rng)
    cout << x << " ";
cout << endl;

//循环缓冲区 boost::circular_buffer
circular_buffer<int> cb = list_of(1)(2)(3);
foreach(int x, cb)
    cout << x << " ";
cout << endl;

```

```
//无序容器 boost:: unordered_set
unordered_set<double> us = list_of(3.14) (2.717) (0.618);
foreach(double x, us)
    cout << x << " ";
cout << endl;
```

如果需要 BOOST_FOREACH 支持自定义的容器序列, 则要做一些其他工作。如果自定义类型类似 STL 容器, 提供了内部的 iterator 和 const_iterator 类型以及 begin() 和 end() 成员函数, 那么 BOOST_FOREACH 就可以自动支持。在本书 7.1.6 小节(243 页)实现的 ref_array 完整地模仿了 STL 容器, 故它也可用于 BOOST_FOREACH。

```
int arr[] = {1, 2, 3, 4, 5};
ref_array<int> ra(arr, 5);
foreach(int x, ra)
    cout << x << " ";
cout << endl;
```

另一种方法是让自定义类型满足 range 的概念。本书未介绍 boost.range, 故在此不对这种方法详述, 读者可参考 foreach 文档了解更多信息。

8.1.6 一个小问题

BOOST_FOREACH 是宏, 这的确有时会带来问题。当变量声明是一个“含有逗号的”模板类型时它就会失效, 所以在遍历 map 类型时无法写出如下的语句:

```
foreach(pair<int, string> x, m) //错误
```

解决的办法是用 typedef 消除模板形式, 或者在循环体外声明变量, 就像之前示范代码做的那样。但很遗憾的是, 到目前为止还没有其他更好的解决办法。

8.2 minmax

minmax 库是对 C++98 标准库中的算法 std::min/max 和 std::min_element/max_element 的增强, 可在一次处理中同时获得最大最小值, 执行效率上有很大提高。minmax 库由两个组件构成, 下面先介绍其中的 minmax() 函数组件。

minmax() 函数改进了 std::min/max 函数, 可同时返回两个参数的最大最小值。它位于名字空间 boost, 为了使用 minmax 组件需要包含头文件 <boost/algorithm/minmax.hpp>, 即:

```
#include <boost/algorithm/minmax.hpp>
using namespace boost;
```

minmax() 函数的简要声明如下:

```
template <typename T>
tuple< T const&, T const& >
minmax(T const& a, T const& b) ;

template <typename T, class BinaryPredicate>
tuple< T const&, T const& >
minmax(T const& a, T const& b, BinaryPredicate comp) ;
```

8.2.1 用法

我们先来复习一下 `std::min/max` 的用法，很简单：

```
cout << std::max(200, 12) << endl;
cout << std::min(200, 12) << endl;
```

这段代码将分别输出两者之间的最大最小值。

`minmax()` 函数的使用方法与之类似，但因为要同时返回两个值，故它返回的是一个包含两个元素的 `tuple` 类型：`tuple< T const&, T const& >`（注意不是 `std::pair`），但如果想省些力气的话，可以用 `BOOST_AUTO` 来自动获得返回值的类型定义。

返回 `tuple` 中的第一个元素是小值，第二个元素是大值，必须使用 `tuple.get<>()` 来访问：

```
BOOST_AUTO(x, minmax(1, 2)) ; //使用 BOOST_AUTO
cout << x.get<1>() << " " << x.get<0>();
```

同其他 STL 算法一样，`minmax()` 也可以接受一个比较谓词函数对象（小于含义）作为第三个参数：

```
struct Comp //定义一个函数对象，有“BOSS”的字符串为大值
{
    bool operator()(const string &a, const string &b)
    {
        return (a < b) || (b.find("BOSS") != string::npos);
    }
};
string s1("5000"), s2("123BOSS");
BOOST_AUTO(x, minmax(s1, s2)) ;
cout << x.get<1>() << " " << x.get<0>() << endl;
BOOST_AUTO(y, minmax(s1, s2, Comp())) ;
cout << y.get<1>() << " " << y.get<0>();
```

运行结果如下：

```
5000 123BOSS
123BOSS 5000
```

8.2.2 使用 `tuples::tie`

如果读者已经熟悉了 `boost.tuple` (7.4 小节, 261 页) 的用法, 那么会很自然地想到使用便捷函数 `tie()`:

```
int a, b;
tie(a, b) = minmax(5, 3);
cout << a << b;
```

这要比直接使用 `tuple` 类型更简洁方便。但同样地, 不能企图用 `minmax` 来自动交换两个变量的值, 例如这样:

```
tie(a,b)=minmax(a,b) //错误, 不能实现预想的功能
```

其原因已经在 `tuple` 中做过解释。如果一定要实现大小值交换功能, 可以用下面的模板函数 `minmax_swap<>`, 它使用标准库的 `swap()` 函数:

```
template<typename T>
void minmax_swap(T &a, T &b)
{
    if(a > b) swap(a,b); //大小值交换
};
int main()
{
    int a = 100, b = 50;
    minmax_swap(a,b);
    cout << a << b;
}
```

8.3 `minmax_element`

`minmax_element` 组件包含一组算法, 它们改进了 `std::min_element()` 和 `std::max_element()` 函数, 可同时返回容器中的最大最小值。

`minmax_element` 位于名字空间 `boost`, 为了使用 `minmax_element` 组件需要包含头文件 `<boost/algorithm/minmax_element.hpp>`, 即:

```
#include <boost/algorithm/minmax_element.hpp>
using namespace boost;
```

8.3.1 用法

`minmax_element` 并不是一个算法, 而是一个算法族, 包括 `first_min_element()`、

`last_min_element()`、`first_min_first_max_element()`、`first_min_last_max_element()`等一系列类似的算法。`minmax_element()`是最基本也是最常用的，以下的介绍以 `minmax_element()` 为例，其他算法可以望文生义触类旁通。

`minmax_element()` 的简要声明如下：

```
template <typename ForwardIter>
std::pair<ForwardIter, ForwardIter>
minmax_element(ForwardIter first, ForwardIter last)
```

`minmax_element()` 函数接受一个序列的区间作为参数，查找其中第一次出现的最大最小值，并以 `std::pair` 的形式返回其迭代器（注意不是值）。

示范 `minmax_element()` 用法的代码如下：

```
#include <boost/algorithm/minmax_element.hpp>

using namespace boost;
int main()
{
    using namespace boost::assign;

    vector<int> v = (list_of(633), 90, 67, 83, 2, 100);

    BOOST_AUTO(x, minmax_element(v.begin(), v.end()));
    cout << "min: " << *x.first << endl;
    cout << "max : " << *x.second << endl;
}
```

`minmax_element()` 的使用风格与标准库的算法是一致的，如果你熟悉标准库，那么几乎没有学习难度。唯一需要注意的就是它的返回值，一个迭代器的 `pair`，因此必须用 `first` 和 `second` 来访问指向最小最大值的迭代器，并用解引用操作符 `*` 来获取值。

程序的运行结果如下：

```
min: 2
max : 633
```

8.3.2 其他函数的用法

`minmax_element()` 函数的其他同族函数用于序列中存在相同元素的情况，它们可以找到第一个或者最后一个最大最小元素，形式为 `xxx_min/max_yyy_max/min_element()`，其中的 `xxx` 和 `yyy` 可以是 `first` 或者 `last`。

下面的代码示范了其中几个函数的用法：

```
#include <boost/typeof/typeof.hpp>
#include <boost/assign.hpp>
#include <boost/algorithm/minmax_element.hpp>
using namespace boost;
int main()
{
    using namespace boost::assign;
    vector<int> v = (list_of(3),5,2,2,10,9,10,8); //初始化测试数据
    vector<int>::iterator pos; //保存查找到的迭代器位置
    pos = first_min_element(v.begin(),v.end()); //找第一个最小值
    assert(pos - v.begin() == 2); //v[2]
    pos = last_min_element(v.begin(),v.end()); //最后一个最小值
    assert(pos - v.begin() == 3); //v[3]
    BOOST_AUTO(x, first_min_last_max_element(v.begin(),v.end()));
    //第一个最小值和最后一个最大值
    assert(x.first - v.begin() == 2 && //v[2]
           x.second - v.begin() == 6); //v[6]
}
```

8.4 总结

本章介绍了两个 Boost 算法库：foreach 和 minmax。

有了 foreach 库，我们就可以告别 STL 那些烦琐的迭代器，随心所欲地遍历泛型容器，再也不需要关心 begin()、end()、rbegin()、rend() 这些迭代器函数的用法和细微差异，BOOST_FOREACH 为我们做了所有的工作。

foreach 库还有些高级议题，如优化速度、应用于不可拷贝的序列，如果读者想深入研究请参考 Boost 说明文档。

minmax 库提供了对 C++98 标准库中的最大最小值算法的增强，可以在一次运算中同时获得两个结果，它的复杂度要比 C++98 的算法好很多。如果我们在实际工作中有这样的需求就应该使用它。

这两个算法库可以搭配 typeof 库和 assign 库，以获得更好的使用效果。

本章还实现了一个有用的模板函数 minmax_swap<>，用来交换两个值的大小，但它并没有使用我们讲述的任何算法，这说明，有时候不拘泥于使用算法也能够解决实际的问题。

第 9 章

数学与数字

科学计算一直是 C++ 和其他计算机语言的一个重要应用领域。现实生活中数学随处可见，而计算机本质上就是对大量数字的运算。从纯数学的角度看，程序也不过是一个非常大的整数而已。

C++98 涵盖了 C89 所定义的数学运算函数，例如 `sqrt()`、`abs()`、`log()` 等等，标准库还提供了复数 `complex` 和数值序列 `valarray`，但对于近代数学来说这些工具还很不够用。

Boost 程序库针对标准库的不足提供了有益的补充，增加了 C99 标准中引入的大量特殊数学函数和复数函数，还有四元数、八元数、拉格朗日多项式、贝塞尔曲线等许多扩充，几乎覆盖了近现代数学的大部分领域，包括高等代数、线性代数和统计学，大大增强了 C++ 语言用于科学计算和理论研究的实用性和能力。再配合自身简洁而富于表现力的语法，C++ 完全可以在专业计算领域与 Fortran 一较高下。

本章介绍 Boost 数学领域的四个库：`integer`、`rational`、`crc` 和 `random`。

9.1 integer

`integer` 库提供了一组有关整数处理的头文件和类，具有良好的可移植性，让 C++ 能够更方便、更准确、更容易地处理整数类型，其中的功能包括编译期计算两个整数的最大值或最小值、编译期 `log2` 计算等。本章将介绍其中的三个组件。

9.1.1 integer_traits

C++98 标准在头文件 `<limits>` 里定义了模板类 `std::numeric_limits<>`，它使用模板特化技术给出了 `int`、`double` 等数据类型的相关特性，如最大最小值、是否有符号等等。这些特性

大部分是编译期的常量，但最大最小值函数却不是常量，只能在运行时使用，这在泛型编程时会带来一些不方便。

`integer_traits` 类派生自 `std::numeric_limits`，弥补了这一缺点。`integer_traits` 位于名字空间 `boost`，为了使用 `integer_traits`，需要包含头文件 `<boost/integer_traits.hpp>`，即：

```
#include <boost/integer_traits.hpp>
using namespace boost;
```

类摘要

`integer_traits` 的类摘要如下：

```
template<class T>
class integer_traits : public std::numeric_limits<T>
{
public:
    BOOST_STATIC_CONSTANT(bool, is_integral = false);
};
```

`integer_traits` 的名字表明了它的功能：它是一个整数特征类。由于继承自 `std::numeric_limits<>`，因此拥有 `std::numeric_limits<>` 的全部能力。此外，它运用了模板特化技术和宏 `BOOST_STATIC_CONSTANT`（参见 4.11.2 小节，155 页），为各种整数类型提供编译期的常量最大最小值。

例如，`integer_traits` 针对 `short` 的特化版本如下（为示范而做了简化）：

```
template<>
class integer_traits<short>
    : public std::numeric_limits<short>,
{
public:
    BOOST_STATIC_CONSTANT(bool, is_integral = true);
    BOOST_STATIC_CONSTANT(T, const_min = -32768);
    BOOST_STATIC_CONSTANT(T, const_max = 32767);
};
```

`integer_traits<short>` 为 `std::numeric_limits<short>` 增加了两个静态常量成员 `const_min` 和 `const_max`，其他方面均与 `std::numeric_limits< short >` 相同。

用法

`integer_traits` 是 `std::numeric_limits` 的子类，因此任何可以使用 `std::numeric_limits` 的地方也都可以使用 `integer_traits`，用来获取整数类型的相关信息。函数 `min()` 和 `max()` 也同样能够使用，但新增加的 `const_min` 和 `const_max` 这两个静态成员变

量可以完全取代它们，并且能够用于编译期的泛型编程。

示范 `integer_traits` 用法的代码如下：

```
#include <boost/integer_traits.hpp>
using namespace boost;
int main()
{
    cout << integer_traits<int>::const_max << endl;
    cout << integer_traits<bool>::const_min << endl;
    cout << integer_traits<long>::is_signed << endl;
}
```

9.1.2 标准整数类型

由于 C++98 标准制定于 1998 年，因此它只能涵盖 C89 的内容，未能赶上 1999C 标准，不是新的 C 标准的超集。为弥补这个缺陷，头文件 `<boost/cstdint.hpp>` 基于 1999 年 C 标准中规定的 `<stdint.h>`，为标准的整型数提供了精确的定义，而且如果编译器提供了 `<stdint.h>`，那么就会自动包含它以保证兼容性。

`<boost/cstdint.hpp>` 声明的所有整数类型都位于名字空间 `boost`，要使用 `<boost/cstdint.hpp>`，只需要包含它即可：

```
#include <boost/cstdint.hpp>
using namespace boost;
```

解说

`<boost/cstdint.hpp>` 以形如 `xxx_yyy#_t` 的 typedef 提供了各种宽度（二进制位）的整数精确定义，其中的宽度可以是 8、16、32 和 64，例如 `int8_t`、`uint_least16_t`。

整数类型的名字规则如下：

xxx：表明该类型是否有正负符号。有符号数为 `int`，无符号数是 `uint`。

yyy：表明该类型的特性。`least` 表示该类型至少具有 # 位的宽度（可能会比 # 宽）；`fast` 表示该类型不仅具有 `least` 的性质，而且是 CPU 处理速度最快的类型；如果没有 `yyy` 标识，则该类型是一个精确宽度的类型，其宽度恰好是 # 位。

#：表明该类型的宽度，以位为单位。

另外 `<boost/cstdint.hpp>` 还有两个最大宽度整数类型：`intmax_t` 和 `uintmax_t`，用

来表示编译器支持的最大整数，可以兼容其他任意整数类型。^①

用法

<boost/cstdint.hpp>提供了很多精确的整数类型定义，读者需要做的就是依据程序的要求选择一个恰当的类型、声明变量，并如同一个普通整数那样使用。

示范<boost/cstdint.hpp>用法的代码如下：

```
#include <boost/cstdint.hpp>
#include <limits>
using namespace boost;
int main()
{
    uint8_t u8;           //一个简单的8位无符号整数,相当于unsigned char
    int_fast16_t i16;      //最快的有符号16位整数
    int_least32_t i32;     //至少有32位的有符号整数
    uintmax_t um;         //编译器支持的最大无符号整数类型

    u8 = 255;
    i16 = 32000;
    i32 = i16;
    um = u8 + i16 + i32;

    //输出各整数类型的名称、大小和值
    cout << typeid(u8).name() << ":" << sizeof(u8)
         << " v = " << (short)u8 << endl;
    cout << typeid(i16).name() << ":" << sizeof(i16)
         << " v = " << i16 << endl;
    cout << typeid(i32).name() << ":" << sizeof(i32)
         << " v = " << i32 << endl;
    cout << typeid(um).name() << ":" << sizeof(um)
         << " v = " << um << endl;

    //输出各整数类型的极值
    cout << (short)numeric_limits<int8_t>::max() << endl;
    cout << numeric_limits<uint_least16_t>::max() << endl;
    cout << numeric_limits<int_fast32_t>::max() << endl;
    cout << numeric_limits<intmax_t>::min() << endl;
}
```

注意：在流输出 int8_t 类型时需转换为 short 型，否则流输出会认为是一个字符类型，输出其 ASCII 码，而不是数字。程序在 Windows 环境的运行结果如下：

^① 有的读者可能对于这种 _t 后缀的整数类型不太适应，其实这只是遵循了 C++ 标准对类型的定义惯例而已（后缀 t 表示 type），如标准库中的 size_t、wchar_t、ptrdiff_t 等等，本书中的有些代码也使用了这种命名方式。

```

unsigned char:1 v = 255
short:2 v = 32000
long:4 v = 32000
unsigned __int64:8 v = 64255
127
65535
2147483647
-9223372036854775808

```

9.1.3 整数类型模板类

头文件<boost/integer.hpp>与<boost/cstdint.hpp>功能类似，也提供一系列的整数类型定义，但它不使用 typedef 的形式，而是用模板类，可以为程序员自动地选择最合适的整数类型。

头文件<boost/integer.hpp>内的类型位于名字空间 boost，要使用<boost/integer.hpp>，只需要包含它即可：

```

#include <boost/integer.hpp>
using namespace boost;

```

类摘要

<boost/integer.hpp>包含两组模板类。最容易使用的模板类是 int_fast_t<>，它的声明如下：

```

template< typename LeastInt >
struct int_fast_t
{
    typedef implementation_supplied fast;
};

```

int_fast_t<>的内置类型 fast 可以自动给出模板参数类型 LeastInt 相应的处理速度最快的整数类型。如果 LeastInt 已经是最快的整数，则返回 LeastInt，例如：

```

#include <boost/integer.hpp>
using namespace boost;
int main()
{
    int_fast_t<char>::fast a;           //char 类型的最快类型
    cout << typeid(a).name() << endl;

    int_fast_t<int>::fast b;            //int 类型的最快类型
    cout << typeid(b).name() << endl;

    int_fast_t<uint16_t>::fast c;       //uint16 类型的最快类型
}

```

```
cout << typeid(c).name() << endl;
}
```

程序的运行结果如下:

```
char
int
unsigned short
```

另一组模板类使用指定的整数位数或数值作为模板参数, 内置类型 `least` 返回支持的最小整数类型, `fast` 返回最快的整数类型。这组类包括 `int_t`、`uint_t`、`int_max_value_t`、`int_min_value_t` 和 `uint_value_t`, 它们的类摘要如下:

```
template< int Bits >
struct int_t
{
    typedef implementation_supplied least;
    typedef int_fast_t<least>::fast fast;
};
template< int Bits >
struct uint_t
{
    typedef implementation_supplied least;
    typedef int_fast_t<least>::fast fast;
};
```

`int_t` 和 `uint_t` 的模板参数指明需要的整数宽度 (以 bit 为单位), 不一定是 8 的整数倍, 但必须是正整数, 内部类型 `least` 和 `fast` 返回能容纳 Bits 位的最小和最快整数类型。

```
template< long MaxValue >
struct int_max_value_t
{
    typedef implementation_supplied least;
    typedef int_fast_t<least>::fast fast;
};
```

`int_max_value_t` 的模板参数表明要容纳的最大整数数值, 参数必须是一个正整数, 内部类型 `least` 和 `fast` 返回能容纳 MaxValue 的最小和最快有符号整数类型。

```
template< long MinValue >
struct int_min_value_t
{
    typedef implementation_supplied least;
    typedef int_fast_t<least>::fast fast;
};
```

`int_min_value_t` 与 `int_max_value_t` 类似, 但它的模板参数表明要容纳的最小整数数值, 参数必须是一个负整数, 内部类型 `least` 和 `fast` 返回能容纳 MinValue 的最小和最快有符

号整数类型。

```
template< unsigned long Value >
struct uint_value_t
{
    typedef implementation_supplied least;
    typedef int_fast_t<least>::fast fast;
};
```

`uint_value_t` 与 `int_max_value_t` 类似，它的模板参数表明要容纳的最大整数数值，参数必须是一个正整数，内部类型 `least` 和 `fast` 返回能容纳 `Value` 的最小和最快无符号整数类型。

用法

整数类型模板类与普通的整数类型一样容易使用，只需要多写一点点模板类代码，它会根据所需要的整数宽度或者处理的整数值自动为你选择最合适的整数类型，这无疑大大方便了程序员的工作。

由于整数类型模板类内仅有 `typedef`，没有其他数据成员或成员函数，因此这些类不会产生任何运行时的开销，与使用内置整数类型或者 Boost 整数类型同样高效。

示范这些整数类型模板类用法的代码如下：

```
#include <boost/integer.hpp>
#include <boost/format.hpp>
#include <typeinfo>
int main()
{
    format fmt("type:%s,size=%dbit\n");    //一个 format 对象

    uint_t<15>::fast a;                    //可容纳 15 位的无符号最快整数
    cout << fmt % typeid(a).name() % (sizeof(a) * 8) ;

    int_max_value_t<32700>::fast b;        //可处理 32700 的最快整数
    cout << fmt % typeid(b).name() % (sizeof(b) * 8);

    int_min_value_t<-33000>::fast c;       //可处理-33000 的最快整数
    cout << fmt % typeid(c).name() % (sizeof(c) * 8);

    uint_value_t<33000>::fast d;          //可处理 33000 的最快无符号整数
    cout << fmt % typeid(d).name() % (sizeof(d) * 8);
}
```

代码中使用 `format` 库（参见 5.2 小节，167 页）对输出进行格式化，运行结果如下：

```
type:unsigned short,size=16bit
type:short,size=16bit
```

```
type:int,size=32bit
type:unsigned short,size=16bit
```

9.2 rational

科学计算的基础是数字运算，C++语言内建了 `int` 和 `float/double`，分别用于支持有限精度的整数和实数（小数），C++98 标准库提供了 `complex`，支持复数的概念。但是，数学中另一种非常重要的数被遗漏了，那就是有理数（分数）。

`boost.rational` 库实现了有理数，补充了 C++ 的数字概念。它基于 C++ 内建整数类型，数字运算时没有精度损失，某种程度上相当于 Java 语言中的 `BigDecimal`，可以应用于金融财务等需要准确值的领域。

`rational` 位于名字空间 `boost`，为了使用 `rational`，需要包含头文件 `<boost/rational.hpp>`，即：

```
#include <boost/rational.hpp>
using namespace boost;
```

9.2.1 类摘要

`rational` 类的简要声明如下：

```
template<typename I> class rational
{
public:

    //构造函数
    rational();                //缺省构造函数，值为零
    rational(I n);            //整数，等于 n/1
    rational(I n, I d);        //分数 (n/d)

    //赋值
    rational& operator=(I n);
    rational& assign(I n, I d);

    //获取内部的分子分母
    I numerator() const;
    I denominator() const;

    //算术操作符
    rational& operator+= (const rational& r);
    ... //其他操作符重载
```



```

//与整数的算术运算
rational& operator+= (I i);
... //其他操作符重载

//递增与递减
const rational& operator++();
const rational& operator--();

//逻辑非操作符
bool operator!() const;

//布尔类型转换
operator bool_type() const;

//比较操作符
bool operator< (const rational& r) const;
bool operator== (const rational& r) const;

//与整数进行比较
bool operator< (I i) const;
... //其他比较操作符重载
};

```

`rational` 类内部把有理数用分子/分母的形式保存, 并运算时使用最大公约数、最小公倍数等操作加以恰当的规范化。它还使用 `operators` 库 (4.8 小节, 125 页), 实现了完整的算术操作符重载, 因此其基类链上总共有 16 个类。

类似标准库的复数类, `rational` 需要使用模板定义基本整数类型, 如:

```
rational<int> pi(22,7); //圆周率的“约率”
```

9.2.2 创建与赋值

`rational` 有三种形式的构造函数, 缺省构造函数 (无参) 创建一个值为零的有理数, 单参的构造函数创建一个整数, 双参数的构造函数创建一个经规范化的分数:

```

rational<int> a; //a=0
rational<int> b(20); //b=20
rational<int> c(31415, 10000); //c=3.1415

```

`rational` 重载了 `operator=`, 可以直接从另一个整数赋值, 也可以使用成员函数 `assign()` 接受分子/分母形式的两个整数参数赋值:

```

rational<int> r; //r=0
r = 0x31; //r = 0x31 (16 进制) 或 r = 49 (10 进制)
r.assign(7, 8); //r = 7/8

```

但 `rational` 不能从浮点数 (`float/double`) 构造或者赋值, 下面的代码是错误的:

```
rational<int> r(1.0);           //错误
r = 3.14;                       //错误
r.assign(7.23, 100);           //错误
```

9.2.3 算术运算与比较运算

`rational` 使用 `operators` 库 (参见 4.8 小节, 125 页) 大大简化了操作符重载的代码量, 实现了完整的算术操作, 包括基本的加减乘除 (`+*/`)、递增递减 (`++--`) 和各种算术比较运算 (`==`、`<`、`<=`、`>`、`>=` 等), 其语法与 `int`、`double` 等类型没有任何区别, 操作起来非常符合自然语言的习惯。

示范 `rational` 各种运算的代码如下:

```
#include <boost/rational.hpp>
using namespace boost;
int main()
{
    rational<int> a(3), b(65534), c(22, 7);    //三个有理数
    b += a;                                     //加法运算
    c -= a;                                     //减法运算
    if (c >= 0)                                 //比较运算
    {
        c = c * b;                             //乘法运算
        ++a;                                    //递增运算
    }
    assert(a == 4);                             //比较运算
}
```

9.2.4 类型转换

与整数的 `bool` 涵义一样, `rational` 也可以在 `bool` 语境里直接隐式转换为 `bool` 类型, 当它值是 0 时为 `false`, 否则为 `true`。

`rational` 也重载了逻辑非操作符 `operator!`, 以相反的规则转换为 `bool` 类型。

示范 `rational` 与 `bool` 类型转换的代码如下:

```
rational<int> r(10);           //值为 10 的有理数
if (r)                         //r 不为 0, 被隐式转换为 bool 值 true
{
    r -= 10;                   //r 现在值为 0
}
assert(!r);                   //r 可隐式转换为 false, 逻辑非操作为 true
```

除了 bool 以外, rational 不能隐式转换到其他任何类型 (即使是分母为 1 的有理数转换到整数), 而必须使用 rational_cast<> 函数。这个函数模仿了标准库的转型操作符, 可以显式地转换到其他数字类型:

```
rational<int> r(2718, 1000);           //自然对数的底
cout << rational_cast<int>(r) << endl; //2
cout << rational_cast<double>(r) << endl; //2.718
```

9.2.5 输入输出

rational 重载了流输入输出操作符, 可以直接与 IO 流配合工作, 格式为用斜杠 (/) 分隔的两个整数, 在流输入时要求斜杠与整数之间不能有空格等空白字符, 例如:

```
rational<int> r;
cin >> r;           //输入: 2/3
cout << ++r;        //r=5/3
```

9.2.6 分子与分母

rational 的成员函数 numerator() 和 denominator() 可以直接访问有理数对象的分子和分母, 但返回的是个右值, 只能读而不可写。用户可以使用这两个成员函数编写出操作有理数的任意代码。例如:

```
rational<int> r(22,7);
cout << r.numerator() << ":" << r.denominator()
    << "=" << rational_cast<double>(r); //输出 22:7=3.14286
```

9.2.7 与数学函数配合工作

rational 没有实现到 float/double 的隐式转换, 故不能直接使用 sqrt()、pow() 等标准 C 函数, 它们要求参数是 int、double 等类型。但取绝对值的 abs() 函数是一个例外, 因为 rational 实现了对它的重载。

如果要对 rational 对象实施这些数学函数运算, 就必须使用 rational_cast<> 转型成 float 或者 double 值才可以, 这一点的确很不方便。

示范数学函数应用于 rational 对象的代码如下:

```
rational<int> a(1414,1000), pi(314, 100); //2 的平方根和  $\pi$ 

cout << abs(a) << endl; //可以直接取绝对值, 707/500
cout << pow(rational_cast<double>(a), 2) << endl; //1.9994
cout << cos(rational_cast<double>(pi)) << endl; // -0.999999
```

9.2.8 异常

有理数是分数，是以除法来表示的数，因此 `rational` 的分母不能是 0，当构造、赋值或者其他操作导致发生除以 0 的情形时，`rational` 会抛出 `bad_rational` 异常。但这种情况一般很少发生，只要稍微留意就可以避免。

示范 `bad_rational` 异常的代码如下：

```
rational<int> a(22,7), b; //a=22/7,b=0
try
{
    a / b; //被 0 除，发生异常
}
catch (bad_rational& e)
{
    cout << typeid(e).name() << e.what() << endl;
}
```

9.2.9 rational 的精度

`rational` 虽然能够精确地处理有理数，但并不能提供无限精度，它的精度完全取决于模板参数 `I`。`rational` 的最大精度是 `numeric_limits<I>::max()`，最小精度是 `1.0/numeric_limits<I>::max()`。

对于 32 位的 `int`，`rational<int>` 的最小精度是 $4.65661e-10$ ，比同样为 32 位的 `float` 精度要高 ($1.19209e-07$)，比 64 位的 `double` 低 ($2.22045e-16$)，但 `rational<int64_t>` 的最小精度是 $1.0842e-19$ 。一般情况下，`rational<int>` 的精度足以满足日常工作。

另外，`rational` 也没有针对 `numeric_limits<>` 提供特化版本，因此无法对它求数值极限，所有 `numeric_limits<rational<I>>` 都是无意义的，因为：

```
assert(numeric_limits<rational<I>>::is_specialized == true)
```

总成立。但这并不妨碍你自己为它写一个 `numeric_limits<>` 的特化。

9.2.10 实现无限精度的整数类型

`rational` 被设计为可以与具有无限精度的整数类型一起使用，如果将来某个库实现了无限精度的整数类型，那么 `rational` 也就具有了无限精度。

`rational` 对模板参数 `I` 的要求是“类似于整数”，具有一个整数类型应该具有的操作，包括加减乘除四则运算、模运算、非运算，可以进行比较操作和拷贝构造、赋值。

在这里，本书给出一个无限精度的整数类型的部分实现——`iint (infinity int)`，不以实用和效率为目的，仅仅用于演示，也许会给读者一点启发。^①

iint 的内部实现

首先遇到的问题是无限精度整数的内部表示。我们不能使用语言内建的整数类型，即使是 `int64_t`，它也是有限的，无限精度整数的表示应该只能受计算机内存的限制。`iint` 采用 `std::string<char>` 来存储整数，每个字节存储一个十进制位，因此可以表现无限大的整数。当然这有很大的空间浪费，实际上一个十进制位只需要半个字节。

接下来是符号的设计。整数可以分为正数和负数，还有一个特殊的数：0，它即不是正数也不是负数，理论上存在+0 和-0。这符合三态布尔的概念，可以用 `tribool`（参见 4.7 小节，120 页）来实现。

`iint` 的内部结构如下：

```
class iint
{
private:
    tribool is_positive;      //正负号表示
    std::string intstr;      //数位存储
```

构造与赋值

解决了 `iint` 的内部表示问题，接下来就是 `iint` 对象的创建。我们决定先从最简单开始：接受一个 `int` 整数构造，暂不考虑从字符串等其他形式。

为了今后运算的方便，`string<char>` 内不保存十进制位的 ASCII 码，而是直接保存数字。可以采用对整数反复除 10 取余的办法，逐个获得该位上的值存入 `string`。但我们可以有更方便的办法，利用 `lexical_cast`（参见 5.1 小节，163 页）或者 `format`（参见 5.2 小节，167 页）直接将数字转换为字符串，然后再减去 `0x30` 即可。因为还要处理符号，因此选用具有更多功能的 `format` 库。

使用 `tribool` 的话，在处理符号时必须要小心，对它要有特别的操作，比普通的二值布尔要复杂一点。

`iint` 的构造函数实现如下：

```
iint(int n = 0):
    is_positive(n > 0 ? true: (n < 0 ? false: indeterminate))
```

① Boost 社区已经启动了无限精度整数的工作，暂定名是 `BigInt` 或 `xint`，但何时完成还是个未知数。

```

{
    static format fmt("%d");
    intstr = (fmt % (is_positive || indeterminate(is_positive)?n:-n)).str();

    std::reverse(intstr.begin(), intstr.end());
    std::transform(intstr.begin(), intstr.end(), intstr.begin(),
        std::bind2nd(std::minus<char>(), 0x30));
}

```

这段代码要注意 `tribool` 的使用，必须对 `true/false/ indeterminate` 三态做完整的判断处理。`format` 将整数转化为字符串后使用标准库算法 `reverse` 将之转置，使字符串的最低位（[0]）存储整数的最低位，然后用算法 `transform`，搭配函数对象 `minus<>` 减去 `0x30`，把 ASCII 码转换成真正的数字。

另外，构造函数特意没有使用 `explicit` 修饰，带来的好处是可以支持 `int` 到 `iint` 的隐式转换。

`iint` 的赋值操作很简单：

```

iint& operator=(const iint & r)
{
    is_positive = r.is_positive;
    intstr = r.intstr;
    return *this;
}

```

转换为字符串输出

`iint` 需要一个输出功能，以方便调试和显示。我们先开发一个 `to_string()` 函数，随后使用这个函数可以很容易地实现流输出。`to_string()` 完全是构造函数的反操作，将保存整数的字符串重新变成 ASCII 码，但 `tribool` 操作要简单一些。

```

std::string to_string() const
{
    std::string tmp(intstr);
    if (!is_positive) //增加负号显示
    {
        tmp += "-";
    }

    std::reverse(tmp.begin(), tmp.end()); //重新转置
    std::transform(tmp.begin() + (is_positive?0:1),
        tmp.end(), tmp.begin() + (is_positive?0:1),
        std::bind2nd(std::plus<char>(), 0x30));
    return tmp;
}

```

有了 `to_string()` 函数，流输出可以实现如下：

```

friend std::ostream& operator<<(std::ostream& os, const iint& x)

```

```

{
    os << x.to_string();
    return os;
}

```

实现比较操作

`iint` 应该支持基本的比较运算，我们需要使用 `operators` 库（参见 4.8 小节，125 页）的 `totally_ordered` 复合运算概念，`iint` 的继承关系是：

```
class iint: totally_ordered<iint>
```

相同操作 `operator==` 仍然需要对 `tribool` 特殊处理：

```

friend bool operator==(const iint& l, const iint& r)
{
    return
        (indeterminate(l.is_positive) && indeterminate(r.is_positive) ) ||
        //处理 0 的符号
        (l.is_positive == r.is_positive &&
         l.intstr == r.intstr);
}

```

小于操作符的处理要更复杂一些，但只要明白 `indeterminate` 状态是专门处理 0 的就容易理解了：

```

friend bool operator<(const iint& l, const iint& r)
{
    if (indeterminate(l.is_positive) && indeterminate(r.is_positive) ||
        !l.is_positive && indeterminate(r.is_positive))
    { return false; }

    return (indeterminate(l.is_positive) && r.is_positive) ||
        bool(l.is_positive) < bool(r.is_positive) ||
        l.intstr.size() < r.intstr.size() ||
        l.intstr < r.intstr;
}

```

实现一元操作符

整数的一元操作符有 `+`、`-` 和 `!`，用于改变符号位和逻辑非运算，它们很简单：

```

iint& operator-()
{
    is_positive = !is_positive;
    return *this;
}
iint& operator+()
{ return *this; }

```

```
bool operator!()
{ return !indeterminate(is_positive); }
```

这里 tribool 的操作比较简单，特别是!的处理。

实现加法操作

加法运算是一切运算的基础，需要用到 operators 库的 addable 概念，它要求类具有 operator+=操作符。我们要模仿手算加法的步骤，从最低位开始相加，超过 10 的数要上进位，这时 string<char>的优点就体现出来了，它可以在一个位置上容纳大于 10 的数字，不必开额外的空间保存进位信息。

加法操作的实现如下：

```
iint& operator+=(const iint & r)
{
    BOOST_AUTO(x, boost::minmax(intstr.size(), r.intstr.size()));

    intstr.append(x.get<1>() + 1 - intstr.size(), 0)

    for (size_t i = 0; i < x.get<1>(); ++i)
    {
        if (i < r.intstr.size())
        {   intstr[i] += r.intstr[i];   }
        if (intstr[i] > 9)
        {
            intstr[i] -= 10;
            ++intstr[i + 1];
        }
    }
    return *this;
}
```

这个加法实现不是最好的，它没有考虑符号的问题，但作为演示足够了。它首先用 minmax 算法计算两个加数中最多位的一个（用 std::max 也可以），将结果增加一位以防有进位。随后从最低位开始逐位相加，如果有进位则向它的后一个元素加 1。

有了这个加法运算，operator++也可以轻松实现：

```
iint& operator++()
{
    *this += 1;
    return *this;
}
```

实现其他运算

要使 iint 能够被 rational 使用，我们还需要实现减法、乘法、除法和模运算。

减法运算类似加法运算，但要考虑借位问题，并且要对加法运算做修改，支持正负数的混合运算。乘法是加法的 n 次重复操作，可以用连续调用 n 次加法实现，但效率会很低，可以模拟手算乘法以提高速度。除法和取模的实现要困难一些，可以用连减除数的方法。

这些运算如果全部实现代码量会较多，在此就不列出了，读者可以把它们作为一个练习，在 `iint` 已有代码的基础上补充完整。

9.2.11 最大公约数和最小公倍数

`rational` 库还提供了两个工具函数：最大公约数 `gcd()` 和最小公倍数 `lcm()`。这两个函数实际是另一个库 `math` 的组成部分，位于名字空间 `boost::math`，被 `rational` 库引入了名字空间 `boost`，因而可以更方便地使用。

`gcd()` 和 `lcm()` 的函数声明如下：

```
template <typename IntType>
IntType gcd(IntType n, IntType m);

template <typename IntType>
IntType lcm(IntType n, IntType m);
```

示范这两个函数用法的代码如下：

```
#include <boost/rational.hpp>
using namespace boost;
int main()
{
    int a = 37, b = 62;
    format fmt("gcd(%1%, %2%) = %3%.lcm(%1%, %2%) = %4%\n");
    cout << fmt % a % b % gcd(a,b) % lcm(a,b);
}
```

程序的运行结果是：

```
gcd(37, 62) = 1.lcm(37, 62) = 2294
```

9.3 crc

CRC（循环冗余校验码）是一种被广泛应用的错误验证机制，它使用一定的规则处理数据，计算出一个校验和，并附在数据末尾发送给接收方。接收方使用同样的规则计算 CRC，如果两个计算结果一致则说明传输正确，否则表明传输过程发生了错误。

`boost::crc` 库实现了计算 CRC 的功能，使用很方便。它位于名字空间 `boost`，为了使用 `crc`

库，需要包含头文件<boost/crc.hpp>，即：

```
#include <boost/crc.hpp>
using namespace boost;
```

crc 库提供了两个类用于计算循环冗余校验码：一个是 `crc_basic`，以 bit 为单位进行计算，速度慢，仅供理论研究；另一个是 `crc_optimal`，是优化过的处理机，以 byte（字节）为单位进行计算，速度很快，适合于实际应用。crc 库的所有实现均基于 `crc_optimal`，故接下来仅针对 `crc_optimal` 进行介绍。

9.3.1 类摘要

`crc_optimal` 的类声明如下：

```
template < std::size_t Bits, impl_def TruncPoly = 0u,
           impl_def InitRem = 0u,
           impl_def FinalXor = 0u, bool ReflectIn = false,
           bool ReflectRem = false >
class crc_optimal
{
public:

    //构造函数
    explicit crc_optimal( value_type init_rem = InitRem );

    void reset( value_type new_rem = InitRem );

    //计算 CRC 值
    void process_byte( unsigned char byte );
    void process_block( void const *bytes_begin, void const *bytes_end );
    void process_bytes( void const *buffer, std::size_t byte_count );

    value_type checksum() const;

    //调用操作符重载
    void operator ()( unsigned char byte );
    value_type operator ()() const;
};
```

`crc_optimal` 有六个模板参数，但我们无需关心这些细节（除非读者想自己实现新的 CRC 计算方法），因为这涉及 CRC 计算的许多数学理论知识，只有第一个模板参数 `Bits` 是比较重要的，它定义了 CRC 模板类的位数，一般取值为 16 或 32。

9.3.2 预定义的实现类

`crc_optimal` 类的模板参数过多，实际使用时很不方便，因此 `crc` 库基于 `crc_optimal`

预定义了四个实现类：`crc_16_type`、`crc_ccitt_type`、`crc_xmodem_type` 和 `crc_32_type`。这四个类代表了历史上被广泛使用的 CRC 计算方法，前三个是 16 位的 CRC 码（2 字节），第四个是 32 位的 CRC 码（4 字节）。

较常用的是 `crc_32_type`，它使用的算法被用于 PKZip 计算 CRC 值，定义如下：

```
typedef crc_optimal<32, 0x04C11DB7, 0xFFFFFFFF, 0xFFFFFFFF, true, true>
    crc_32_type;
```

9.3.3 计算 CRC

为优化速度，`crc_optimal` 只能以 `byte` 为单位处理数据（`crc_basic` 可以处理 `bit`）。

成员函数 `process_byte()` 一次只能处理一个字节。`process_bytes()` 和 `process_block()` 函数则可以处理任意长度的数据块，两者的输入参数不同，`process_bytes()` 用数据块 + 数据块长度的形式，而 `process_block()` 接受数据块的开始和结束指针。

在任何时候 `crc_optimal` 都可以调用 `checksum()` 函数获得当前已计算出的 CRC 值，或者调用 `reset()` 重置计算结果。

使用 `crc_32_type` 演示 CRC 计算的代码如下：

```
#include <boost/crc.hpp>
using namespace boost;
int main()
{
    crc_32_type crc32;                                // 一个 crc 对象

    cout << hex;                                       // 置输出流为 hex 格式
    cout << crc32.checksum() << endl;                 // 无输入数据时也可取 crc 值
    crc32.process_byte('a');                          // 计算一个字节
    cout << crc32.checksum() << endl;
    crc32.process_bytes("1234567890", 10);           // 计算 10 个字节
    cout << crc32.checksum() << endl;

    char szCh[] = "1234567890";                      // 一个字符数组
    crc32.reset();                                     // 复位 crc 处理机
    crc32.process_block(szCh, szCh + 10);            // 使用区间的形式输入数据
    cout << crc32.checksum() << endl;
}
```

程序运行结果如下：

```
0
e8b7be43
```

```
3f4a3e87
261daee5
```

crc_optimal 还重载了括号操作符 operator(), 它有两种用法。不带参数的形式直接返回 CRC 值, 相当于调用 checksum(), 可以很方便地获取返回值; 带参数的形式接受一个字节, 相当于调用 process_byte(), 因此可以把 crc_optimal 对象当作函数对象传递给 STL 算法。例如:

```
#include <boost/crc.hpp>
using namespace boost;
int main()
{
    crc_32_type crc32;                //一个 crc 对象

    cout << hex;
    crc32.process_bytes("1234567890", 10); //处理 10 个字节
    cout << crc32() << endl;             //使用 operator() 获得 crc 值

    string str = "1234567890";
    crc32.reset();                    //复位 crc 处理机

    //把 crc 对象作为函数对象传递给 for_each 算法处理字符串
    cout << std::for_each(str.begin(), str.end(), crc32)() << endl;
    cout << crc32() << endl;
}
```

对这个例子的最后两条语句需要解释如下:

std::for_each 算法接受一个单参的函数对象, 把它拷贝一个副本在算法内部使用, 不会影响原函数对象 (即 crc32)。算法在数据块区间内逐个元素调用函数对象的 operator(), 即 process_byte() 计算 CRC 值, 循环完成后返回存储的函数对象副本——另一个 crc_32_type 对象, 因此可以再调用无参版 operator() 来获取 CRC 值。(想了解更多 for_each 算法的信息请见推荐书目[2])

9.3.4 CRC 函数

crc_optimal 模板类及其预定义实现类可以非常容易地计算 CRC 值, 但 crc 库还提供了两个自由函数: crc() 和 augmented_crc(), 声明如下:

```
template<...>
typename boost::uint_t<Bits>::fast
boost::crc( void const *buffer, std::size_t byte_count );

template < ... >
typename boost::uint_t<Bits>::fast
```

```
boost::augmented_crc( void const *buffer, std::size_t byte_count );
```

`crc()` 函数使用模板参数在内部构造 `crc_optimal` 对象，直接计算 CRC 值并立即返回，例如：

```
crc<16, 0x8005, 0, 0, true, true>("1234567890", 10);
```

等同于

```
crc_16_type().process_bytes("1234567890", 10);
```

`augmented_crc()` 的用法与 `crc()` 类似，只需要传递较少的参数，但它的计算规则与 `crc()` 和 `crc_optimal` 模板类有所不同，主要是为了方便计算附加 CRC 码的数据，详细用法可参考 `crc` 的说明文档。

9.3.5 自定义 CRC 函数

`crc` 库提供的 `crc()` 和 `augmented_crc()` 函数需要较多的模板参数，使用起来不够方便，有悖其本意。我们可以自己实现一个模板函数，来进一步简化计算 CRC 值。

```
template<typename T>
typename T::value_type crc( void const * buffer, std::size_t byte_count)
{
    T crc_obj;
    crc_obj.process_bytes(buffer, byte_count);
    return crc_obj();
}
```

模板函数 `crc<T>` 的原理很简单：使用预定义的 `crc` 实现类（或者任意 `crc_optimal<>` 实现类）作为参数，直接计算 CRC 值并返回。它的用法如下：

```
int v = crc<crc_xmodem_type>("1234567890", 10);

crc_xmodem_type crc16;
crc16.process_bytes("1234567890", 10);
assert(crc16() == v);
```

9.4 random

“随机数”是计算机科学中一个历史悠久的议题，关于随机数有着许多相当深刻的讨论，比如计算机能否产生“真随机数”，什么是好的“伪随机数”等等。`random` 库专注于伪随机数的实现，有多种算法可以产生高质量的伪随机数，并提供随机数发生器、分布等很多有用的数学、统计学相关概念。它已被收入 C++0x TR1 标准草案。

random 库位于名字空间 boost，为了使用 random 库，需要包含头文件<boost/random.hpp>，即：

```
#include <boost/random.hpp>
using namespace boost;
```

9.4.1 伪随机数发生器

random 库提供了 26 个伪随机数发生器，使用的算法包括线性同余算法、逆同余算法、Mersenne Twister 算法、fibonacci 算法、ranlux 算法及它们的混合。这些随机数发生器的随机数质量、内存需求、产生速度都各不相同，程序员可以仔细评估以选择最合适自己应用的一个。

在这里作者推荐三个伪随机数类：rand48、mt19937 和 lagged_fibonacci19937，它们产生随机数的速度由高到低，产生的随机数质量和所需内存则由低到高。

虽然这些随机数发生器使用的算法不同，实现也有较大的差异，但基本接口是类似的，像这样：

```
class random_demo                                //不是真正的 boost 代码!
{
    random_demo(int x0)                          //使用种子值的构造函数
    int seed(int x0);                            //设置种子值
    int operator()();                            //调用操作符,产生随机数
};
```

这段代码说明了伪随机数发生器的两个基本接口。构造函数和成员函数 seed() 为发生器赋一个种子值，操作符 operator() 产生一个随机数。

例如，下面的代码使用时间作为种子值，随机数发生器 mt19937 产生 100 个伪随机数：

```
#include <ctime>
#include <boost/random.hpp>
using namespace boost;
int main()
{
    mt19937 rng(time(0));                        //以时间为种子创建一个随机数发生器
    for (int i = 0; i < 100; ++i)
    {
        cout << rng() << ", ";                //产生随机数
    }
}
```

9.4.2 伪随机数发生器的构造

伪随机数发生器在程序中应尽量少发生构造操作，最好是实现为单件。原因有两个，一是伪随机数发生器的构造成本是相当昂贵的，越是高质量的伪随机数发生器越需要初始化大量的内部状态，需要大量的时间和空间开销；二是由于构造发生器提供的种子一般是使用系统时间，如果两个发生器构造的时间间隔很短，那很可能种子值是相同的，从而得到两个行为相同的伪随机数发生器，令随机数失去意义。

我们可以利用 4.6 小节,116 页介绍的 singleton 组件来保证伪随机数发生器的唯一创建。但由于伪随机数发生器的缺省构造函数使用的是固定种子，会导致重复的随机数序列，故不能使用它的缺省构造函数，需要使用一个 wrapper 来包装它：

```
template<typename T>                                //模板类型是随机数发生器
class rng_wrapper
{
private:
    T rng;                                           //随机数发生器成员变量
public:
    rng_wrapper():rng((typename T::result_type)time(0)){}
    typename T::result_type operator()()           //重载调用操作符
    { return rng(); }
};
```

rng_wrapper 是内部保存伪随机数发生器的一个实例，构造时使用当前时间作为随机数种子初始化发生器。

有了这个 wrapper，伪随机数发生器的单件用法就可以实现如下：

```
#include <boost/random.hpp>
#include <ctime>
#include <boost/pool/detail/singleton.hpp>
using boost::details::pool::singleton_default;
class rng_wrapper {...}
int main()
{
    typedef singleton_default< rng_wrapper<mt19937> > rng_t;

    rng_t::object_type &rng = rng_t::instance();
    for (int i = 0; i < 100; ++i)
    { cout << rng() << ", "; }
}
```

9.4.3 伪随机数发生器的拷贝

伪随机数发生器都是可以拷贝、赋值的，在有些情况下这种能力是很有用的：拷贝后的副本保存了伪随机数发生器的状态，可以产生相同的随机数序列，用于再现测试。例如：

```
mt19937 rng(time(0));           //以时间为种子创建一个随机数发生器
cout << rng() << endl;
mt19937 rng2(rng);              //拷贝构造随机数发生器
for (int i = 0; i < 10; ++i)
{ assert(rng() == rng2()); } }
```

这段代码首先创建了一个 mt19937 伪随机数发生器对象 rng，然后 rng2 从 rng 拷贝构造，因而这两个伪随机数发生器具有了相同的状态，对它们的调用将产生完全一致的随机数序列，与伪随机数发生器构造时的种子值无关。

9.4.4 随机数分布器

伪随机数发生器能够产生高质量的随机数，但产生的随机数都是整数类型，均匀分布在整数域。但很多时候我们需要的是某些具有特定要求的随机数，如在指定区间内的任意整数、 $[0, 1)$ 区间的实数、正态分布等等，前面提到的伪随机数发生器显然无法满足要求。

常用的解法是使用模、除以整数最大值等运算处理随机数，使之转变为所要求的数，例如下面的代码将 rand48 产生的随机数转换成 $[0, 1]$ 区间内的小数：

```
rand48 rng;
for (int i = 0; i < 10; ++i)
{
    cout << rng() * 1.0 / numeric_limits<rand48::result_type>::max();
}
```

这种做法虽然可用，但却不是最有效的方法。它不具有可复用性，会导致在开发过程中出现大量重复代码，应该尽量避免。

random 库为此提出了随机数分布器的概念，把伪随机数发生器产生的整数域随机数映射到另一种分布，提高了代码的复用性。

random 库提供的映射类是相当全面的，覆盖了应用的大部分领域，下面列出了其中的所有 15 个分布：

- uniform_smallint: 在小整数域内的均匀分布；
- uniform_int : 在整数域上的均匀分布；

- `uniform_01` : 在区间 $[0, 1]$ 上的实数连续均匀分布;
- `uniform_real` : 在区间 $[\min, \max]$ 上的实数连续均匀分布;
- `bernoulli_distribution` : 伯努利分布;
- `binomial_distribution` : 二项分布;
- `cauchy_distribution` : 柯西(洛伦兹)分布;
- `gamma_distribution` : 伽马分布;
- `poisson_distribution` : 泊松分布;
- `geometric_distribution` : 几何分布;
- `triangle_distribution` : 三角分布;
- `exponential_distribution` : 指数分布;
- `normal_distribution` : 正态分布;
- `lognormal_distribution` : 对数正态分布;
- `uniform_on_sphere` : 球面均匀分布。

9.4.5 随机数分布器类摘要

`random` 库中的随机数分布器有的相当专业, 如对数正态分布、球面均匀分布、三角分布, 涉及的数学理论也较深, 通常的应用软件开发工作能用到的机会较少, 故本书在此仅介绍几个最有用最常用的分布。

`uniform_smallint`

`uniform_smallint` 的类摘要如下:

```
template<class IntType = int>
class uniform_smallint
{
public:
    uniform_smallint(IntType min = 0, IntType max = 9);
    result_type min() const;
    result_type max() const;
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
```

```
};
```

随机数分布器 `uniform_smallint` 实现了在小整数区间的均匀分布。它的模板参数是分布的整数类型，默认为 `int`，构造函数指定分布区间的最大最小值。其他接口与伪随机数发生器类似，同样可以用 `operator()` 来获得随机数，另外还提供 `min()` / `max()` 函数返回分布映射的区间范围。

uniform_int

`uniform_int` 的类摘要如下：

```
template<class IntType = int>
class uniform_int
{
public:
    explicit uniform_int(IntType min = 0, IntType max = 9);
    result_type min() const;
    result_type max() const;
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
    template<class UniformRandomNumberGenerator>
};
```

随机数分布器 `uniform_int` 实现了在任意整数区间的均匀分布，它的接口与 `uniform_smallint` 几乎完全相同。两者的区别在于产生随机数分布的质量。`uniform_smallint` 基于分布区间要远小于随机数源的整数域的假设，不考虑量子化的问题，而 `uniform_int` 则无此限制，分布区间可以是任意大。

uniform_01

`uniform_01` 的类摘要如下：

```
template<class UniformRandomNumberGenerator, class RealType = double>
class uniform_01
{
public:
    explicit uniform_01(base_type rng);
    result_type operator()();
    result_type min() const;
    result_type max() const;
};
```

随机数分布器 `uniform_01` 实现了在 $[0, 1]$ 区间的小数均匀分布。它的模板参数是伪随机数发生器类型和产生的浮点数类型，默认是 `double`。构造函数传入伪随机数发生器对象，其他接口同 `uniform_smallint`。需要注意：模板参数 `UniformRandomNumberGenerator` 不仅可以是普通类型，也可以是引用类型，即 `T&`，这样可以避免对象的拷贝代价。

uniform_real

uniform_real 的类摘要如下:

```
template<class RealType = double>
class uniform_real
{
public:
    uniform_real(RealType min = RealType(0), RealType max = RealType(1));
    result_type min() const;
    result_type max() const;
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
};
```

随机数分布器 uniform_real 是 uniform_01 在任意实数区间的扩展, 实现了在任意区间的实数均匀分布。它的模板参数是产生的浮点数类型, 默认是 double。构造函数传入实数区间的两个端点, 默认位于 $[0, 1)$, 其他接口同 uniform_smallint。

normal_distribution

normal_distribution 的类摘要如下:

```
template<class RealType = double>
class normal_distribution
{
public:
    explicit normal_distribution(const result_type& mean = 0,
                                const result_type& sigma = 1);
    RealType mean() const;
    RealType sigma() const;
    void reset();
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
};
```

随机数分布器 normal_distribution 实现了正态分布。它的模板参数是产生的浮点数类型, 默认是 double。构造函数传入均值和方差, 默认是 0 和 1。成员函数 mean() 和 sigma() 分别返回分布的参数均值和方差。操作符重载 operator() 接受一个 $[0, 1]$ 上均匀分布的随机数发生器 (通常是 uniform_01)。

9.4.6 随机数分布器用法

随机数分布器必须搭配随机数发生器才能工作, 因此必须先定义一个随机数发生器作为随机数源, 然后再选择恰当的模板参数定义一个分布器 (如 uniform_smallint<>), 把随机数映射

到某个区间的分布，最后调用分布器的 `operator()` 获得所需的随机数。

示范随机数分布器用法的代码如下。它使用 `uniform_int<>`、`uniform_01` 和 `normal_distribution` 各产生十个随机数：

```
#include <boost/random.hpp>
using namespace boost;
int main()
{
    mt19937 rng(time(0));

    uniform_int<> ui(0, 255);
    for (int i = 0; i < 10; ++i)
    {   cout << ui(rng) << ", "; }
    cout << endl;
    uniform_01<mt19937&> u01(rng);           //注意：引用类型
    for (int i = 0; i < 10; ++i)
    {   cout << u01() << ", ";   }
    cout << endl;

    //定义一个均值为 1，方差是 2 的正态分布
    normal_distribution<> nd(1, 2);
    int count = 0;
    for (int i = 0; i < 10000; ++i)
    {

        //计算正态分布的统计学概率
        if (abs(nd(u01) - 1) <= 2.0)
        {   ++count;   }
    }
    cout << 1.0 * count / 10000 << endl;
}
```

9.4.7 变量发生器

有了随机数发生器和分布器，已经基本可以解决大部分与随机数相关的问题了。但为了方便使用，`random` 库还提供一个模板类 `variate_generator<>`（变量发生器），组合了随机数发生器和分布器，使产生随机数更加容易便捷。

`variate_generator` 的类摘要如下：

```
template<class Engine, class Distribution>
class variate_generator
{
public:
    variate_generator(Engine e, Distribution d);
```

```

    result_type operator()();
    template<class T>
    result_type operator()(T value);

    engine_value_type& engine();
    const engine_value_type& engine() const;

    distribution_type& distribution();
    const distribution_type& distribution();

    result_type min() const;
    result_type max() const;
};

```

`variate_generator` 的模板参数是随机数发生器 `Engine` 和分布器 `Distribution`, `Engine` 的类型可以是 `T` (原始类型)、`T*` (指针类型) 或者 `T&` (引用类型), `Distribution` 则必须是 `T` (原始类型), 不能是指针或引用。

`variate_generator` 在构造函数中指定随机数发生器和分布器的实例, 可以用 `engine()`、`distribution()` 获得随机数发生器和分布器的引用, `operator()` 产生随机数。

`variate_generator` 也是可以拷贝和赋值的, 这意味着它同伪随机数发生器一样可以保存状态以备后用, 但如果模板参数的 `Engine` 是引用类型则无法拷贝、赋值。

`variate_generator` 必须组合随机数发生器和分布器, 可以这样用:

```

mt19937 rng((int32_t)std::time(0));
uniform_smallint<> us(1,100);

variate_generator<mt19937&, uniform_smallint<>> gen(rng, us);
for (int i = 0; i < 10 ; ++i)
{
    cout << gen() << endl;
}

```

也可以把这些构造工作都放在一条语句中:

```

variate_generator<mt19937, uniform_smallint<>>
gen(mt19937((int32_t)std::time(0)), uniform_smallint<>(1,100));

```

变量发生器也能够封装为单件的形式, 避免初始化的成本, 手法与 `rng_wrapper` 的类似, 读者可自行实现。

9.4.8 产生随机数据块

随机数发生器和分布器一次只能产生一个随机数, 但在密码学和其他应用领域经常需要产生一定长度的随机数据块。利用 `random` 库可以实现模板函数 `rand_bytes()` 如下:

```

template<typename Rng >
void rand_bytes(unsigned char *buf, int buf_len)
{
    typedef variate_generator<Rng, uniform_smallint<>> var_gen;
    static var_gen
        gen(Rng((typename Rng::result_type)std::time(0)),
            uniform_smallint<>(1,255));           //随机字节值在 1-255 之间

    for (int i = 0; i < buf_len; ++i)           //循环填入随机数
        buf[i] = gen();
}

```

`rand_bytes()` 要求传入一个随机数发生器模板参数，在内部定义了一个静态变量 `gen`，保证随机数发生器在函数调用时仅会初始化一次，避免了多次构造的代价，随后使用 `for` 循环向缓冲区填入 1 至 255 的随机数。

这里不使用标准算法 `std::generate_n` 的原因是标准算法会在内部保留一个函数对象的拷贝，而变量发生器被拷贝后就保存了状态，如果时间点很接近的两个调用很可能会输出相同的伪随机数序列。

如果要使用 `std::generate_n`，就需要对变量的引用进行包装，使得它可以被拷贝。

我们可以在函数内部定义一个包装类，实现引用包装：

```

void rand_bytes(...)
{
    ...                                     //同之前的代码
    class var_wrapper                       //引用包装类，定义了 operator() 供算法调用
    {
    public:
        var_wrapper(var_gen&t_):t(&t_){}    //构造函数
        typename var_gen::result_type operator() ()
        { return (*t)(); }
    private:
        var_gen *t;                        //内部保存变量发生器的指针
    };
    generate_n(buf, buf_len, var_wrapper(gen));
}

```

这种实现比较简陋，不具有通用性。C++0x 的 TR1 库提供了引用包装器 `reference_wrapper<T>`，可以很容易地实现这个用法：

```
generate_n(buf, buf_len, std::tr1::ref(gen));
```

Boost 库虽然也实现了 `tr1::ref` 库（参见 11.2 小节，422 页），非常遗憾的是，它不支持标准草案中规定的函数调用，没有 `operator()`。但本书提供了一个弥补的措施，可以如 TR1 库同样使用，例如：

```
generate_n(buf, buf_len, boost::ref(gen));
```

不管模板函数 `rand_bytes()` 内部是如何实现的，它对外的接口是稳定的，用法如下：

```
unsigned char buf[10]; // 一个 10 字节的缓冲区

rand_bytes<mt19937>(buf, 10); // 向缓冲区填入随机数
for (int i = 0; i < 10 ; ++i) // 输出随机数
{   cout << (short)buf[i] << ", "; }
cout << endl;
rand_bytes<rand48>(buf, 10); // 改用 rand48 随机数发生器
for (int i = 0; i < 10 ; ++i) // 输出随机数
{   cout << (short)buf[i] << ", "; }
```

9.4.9 真随机数发生器

之前介绍的随机数发生器产生的都是伪随机数，它们并不是真正的随机数，而是有一个非常大的循环周期（例如，`mt19937` 的循环周期是 $2^{19937}-1$ ），真随机数无法用纯软件产生（因为计算机本身是个确定的有限状态自动机）。

`random` 库并没有实现也无法实现真随机数发生器，因为那需要操作系统或者硬件特性支持，无法做到平台无关性。

但 `random` 库在头文件 `<boost/nondet_random.hpp>` 声明了一个 `random_device` 类，它使用桥接模式和 `pimpl` 惯用法定义了一个真随机数模型，用户可以依据操作系统的特性自行实现真随机数发生器。

`random_device` 类的简要声明如下：

```
class random_device : noncopyable
{
public:
    result_type min() const;
    result_type max() const;
    explicit random_device(const std::string& token = default_token);
    ~random_device();
    double entropy() const;
    unsigned int operator()();
private:
    class impl; // 桥接模式，用户自定义实现
    impl * pimpl;
};
```

`random_device` 定义了硬件随机数设备所必需的接口，真正的实现应该是内部类 `random_device::impl`。很遗憾，`pimpl` 指针没有使用 `smart_ptr` 库，我们必须自己在 `random_device` 的析构函数中调用 `delete` 来删除它。

9.4.10 实现真随机数发生器

实现真随机数发生器需要操作系统支持，在这里作为演示，我们暂使用伪随机数发生器来实现 `random_device`。^①

`random_device::impl` 实现了真随机数的产生，其实现代码因具体操作系统而不同，但至少要提供一个接口用于产生随机数。在这里我们使用 `rand48` 来模仿硬件随机数，使用 `operator()` 来产生随机数：

```
class boost::random_device::impl
{
private:
    rand48 rng;
public:
    impl():rng(time(0))           //用时间种子初始化随机数发生器
    {      cout << "random_device::impl ctor\n";      }
    ~impl() { cout << "random_device::impl dtor\n"; }

    unsigned int operator()()
    {      return rng();      }
};
```

接下来要实现 `random_device` 类声明而未定义的构造、析构函数、取熵值和 `operator()`。

```
const char * const boost::random_device::default_token = "some_device";
boost::random_device::random_device(const std::string& token)
: pimpl(new impl)
{}

boost::random_device::~~random_device()
{ delete pimpl; }

double boost::random_device::entropy() const
{ return 10; }

unsigned int boost::random_device::operator()()
{ return (*pimpl)(); }
```

这样，这个真随机数发生器就完成了，可以把它像 `rand48`、`mt19937` 一样使用，也可以配合分布器、变量发生器：

```
#include <boost/random.hpp>
```

① Boost1.43 版为 `random_device` 提供了 Windows 和 Linux 下的实现 `cpp` 文件，分别使用了微软 CSP 和 `/dev/urandom`，可以直接编译后使用，也可以作为实现的范例。


```
#include <boost/nondet_random.hpp>
using namespace boost;
class boost::random_device::impl
{...}

int main()
{
    //通常用法
    random_device rng;
    for (int i = 0 ;i < 10 ; ++i)
    {
        cout << rng() << ",";
    }
    cout << endl;

    //配合分布器
    uniform_real<> ur(1.0, 2.0);
    for (int i = 0 ;i < 10 ; ++i)
    {
        cout << ur(rng) << ",";
    }
    cout << endl;

    //配合变量发生器
    variate_generator<random_device&, uniform_smallint<>>
        gen(rng, uniform_smallint<>(0,255));
    for (int i = 0 ;i < 10 ; ++i)
    {
        cout << gen() << ",";
    }
    cout << endl;
}
```

9.5 总结

本章介绍了 Boost 库在数学领域的几个成就。

首先我们讨论了 `integer` 库，它为处理整数这种基本且重要的数据类型提供了很多小工具，例如精确宽度的整数类型和自动选择可以快速处理的整数类型。这个库虽然很小，用法也比较平淡无奇，但在泛型编程和跨平台开发时却很有用，有利于保证代码的通用性和可移植性。也许某个时候，你的程序会运行在 64 位 CPU 和操作系统，或者向下移植到 8 位或 16 位的嵌入式系统里，那时候你就会发现精确宽度整数的重要性。

`rational` 库实现了有理数的 C++ 表述，完善了 C++ 的数字域。`rational` 重载了各种算术操作符，可以像普通数据类型一样使用，而且比 `float/double` 好的是没有精度损失，可以用于货币处理等日常工作。但 `rational` 不是无限精度的类型，没有无穷大的概念，存在上溢的可能，在与 C 标准数学函数配合使用时，必须使用不方便的 `rational_cast<>` 转型为 `float/double`，因此不适合更高级的科学计算。

循环冗余码是一种很有用的错误验证机制，使用 `crc` 库可以非常容易地计算、验证各种 CRC 值。它利用了 `integer` 库提供的快速整数，可以快速高效地处理大批量的数据。

本章最后讨论了 `random` 库。随机数在密码学、理论研究、图像处理、仿真、视频游戏等很多领域都有广泛的应用，`random` 可以生成高质量的随机数，提供了大量且全面的伪随机数发生器和分布器，可以满足各种应用需求。

本章实现的工具函数和工具类如下：

- `iint`：一个具有无限精度的整数类型，但没有完全实现，仅支持比较运算和加法运算，可以作为进一步开发的基础；
- `crc<>()`：比 `Boost` 库提供的函数更容易地计算 CRC 值；
- `rng_wrapper<>`：一个对随机数发生器的简单包装类，采用系统时间作为随机数种子，可以用于构造单件；
- `rand_bytes<>()`：简单方便地产生随机数数据块。

第 10 章

操作系统相关

跨平台、可移植一直是很多 C++ 程序员所追求的目标。但由于 C++ 是一种“中级”语言，很多时候都要与操作系统打交道，因而功能代码常常与平台结合的很紧密，难以做到真正的跨平台，在向其他平台移植代码的时候修改操作系统相关的代码通常是必不可少的工作。

Boost 库提供了数个操作系统相关的库，部分地屏蔽了操作系统的底层细节，能够提高程序的可移植性，其中的两个库（system 和 filesystem）即将成为 C++ 新标准的一部分。

本章首先介绍 io_state_savers 库，它可以保存输入输出流的状态并自动恢复；随后是 system 库，它封装了操作系统底层的错误代码，为上层提供了一个可移植的统一的处理接口；第三个是 filesystem 库，提供跨平台的文件系统处理能力，能够以一致的方式处理不同操作系统中的文件和目录；最后我们将看到 program_options 库，它为程序员提供了强大的命令行参数解析和程序运行选项配置能力。

本章的库除 io_state_savers 外都需要编译，这是由其平台相关特性所决定的。

10.1 io_state_savers

C++ 标准库为输入输出提供了流操作功能，它代替了 C 的文件 handle 和相关的函数，使用起来灵活方便，而且支持国际化，是一个功能强大且富有弹性的处理框架。

流操作一般都很简单明了，但它的状态管理却是个麻烦的事情。如果执行了某些改变状态的操作，如流重定向、修改了输出格式标志（如 `std::hex`），则流的状态就会一直改变下去，而这通常不是程序的本意。我们必须手工保存流的状态，并在使用完后及时恢复，以避免流发生错误——这是个麻烦却又不得不做的工作。

`io_state_savers` 库可以简化恢复流状态的工作，它能够保存流的当前状态，自动恢复流的状态或者由程序员控制恢复的时机。

`io_state_savers` 库位于名字空间 `boost::io`，为了使用 `io_state_savers` 组件，需要包含头文件 `<boost/io/ios_state.hpp>`，即：

```
#include <boost/io/ios_state.hpp>
using namespace boost::io;
```

10.1.1 类摘要

`io_state_savers` 库包含很多 `saver` 类，它们被细分用于保存流的不同状态，如数字精度、输出宽度、流缓冲、填充字符等等，但基本的形式都差不多，大概是这样：

```
class ios_saver                                     //伪代码
{
    typedef std::ios_base          state_type;
    typedef implementation_defined aspect_type;

    explicit saver_class( state_type &s );
    saver_class( state_type &s, aspect_type const &new_value );
    ~saver_class();

    void restore();
};
```

`ios_saver` 有两个内部类型定义：`state_type` 和 `aspect_type`，它们标记了 `ios_saver` 的基本属性：

- `state_type` 是 `ios_saver` 要保存的流类型，如 `ios_base`、`basic_ios<>`；
- `aspect_type` 是 `ios_saver` 要保存的具体流状态，如 `fmtflags`、`iostate`。

`ios_saver` 的构造函数接受一个流对象的引用，同时保存该流的 `aspect_type` 值。另一种形式的构造函数则在保存的同时变更流的状态为 `new_value`。

`ios_saver` 在析构时会自动把保存的状态恢复到流对象，程序员也可以在任何时刻使用 `restore()` 函数手工恢复流状态。

`ios_saver` 把赋值操作符 `operator=` 声明为私有的，因此都是不可赋值的，但允许拷贝构造，可以在程序中拷贝一个 `ios_saver` 以备后用。

10.1.2 用法

`io_state_savers` 库里有很多用于保存并恢复流状态的类，它们被分成四组，分别是：

- 基本的标准属性保存器，如 `ios_flags_saver`、`ios_width_saver`;
- 增强的标准属性保存器，如 `ios_iostate_saver`、`ios_rdbuf_saver`;
- 自定义的属性保存器，如 `ios_istate_saver`、`ios_pword_saver`;
- 组合的的属性保存器，如 `ios_all_saver`。

最简单也是最常用的类是 `ios_all_saver` 和 `wios_all_saver`，它们可以保存流的所有状态，让我们不必费心去决定保存流的哪个状态。`wios_all_saver` 是 `ios_all_saver` 的宽字符形式，用来配合宽字符流 `wcout` 使用。

假设我们有如下的日志函数 `logging()`，它用来向标准输出打印日志信息：

```
void logging(const char* msg)
{ cout << msg << endl;}
```

使用流的重定向功能，我们可以把日志的输出转到一个文件流中，这样可以把日志保存起来用于日后审查：

```
ofstream fs("logfile.log");           //输出文件流
cout.rdbuf(fs.rdbuf());              //cout 流重定向
logging("fatal msg");                //向文件流输出日志
```

那么下面的代码将会发生崩溃的错误：

```
int main()
{
    string filename("c:/test.txt");
    cout << "log start" << endl;           //开始
    if(!filename.empty())
    {
        ofstream fs(filename.c_str());    //打开一个文件流
        cout.rdbuf(fs.rdbuf());          //标准输出重定向到文件流
        logging("fatal msg");             //重定向输出日志
    }                                     //文件流被自动析构!!
    cout << "log finish" << endl;          //cout 无法输出，运行错误
}
```

出错的原因在于流的重定向。当文件流 `fs` 被设置为 `cout` 的缓冲区后，`cout` 将总向它输出数据。但 `fs` 是一个局部变量，当离开 `if` 语句的作用域后它被自动销毁，导致缓冲区失效。但 `cout` 对此并不知情，仍然向一个无效缓冲区写入数据，从而发生了严重的运行错误。

要修复这个 bug 非常容易，只需要在重定向或任何可能改变流状态的操作前用 `ios_all_saver` 保存 `cout` 的状态，那么在离开作用域时它就会自动把 `cout` 恢复到最初的状态，从而避免了灾难的发生：

```

{
    ...
    ios_all_saver ifs(cout);           //保存流的所有状态
    cout.rdbuf(fs.rdbuf());
    ...
}                                     //离开作用域，导致保存器析构，自动恢复流的状态

```

10.1.3 简化 new_progress_timer

在 2.3.3 小节我们实现了一个 progress_timer 的扩展类——new_progress_timer，它可以度量很高精度的时间，但析构函数的流状态的保存与恢复代码十分繁琐，现在到了使用 io_state_savers 库来改进它代码的时候了。

我们直接使用 ios_all_saver 来保存标准输出流 cout 的状态，将 new_progress_timer 的析构函数修改如下：

```

#include <boost/io/ios_state.hpp>
template<int N = 2 >
class new_progress_timer:public boost::timer
{
    ...
    ~new_progress_timer(void)
    {
        try
        {
            //保存流的状态
            boost::io::ios_all_saver ios(m_os);

            //设置输出格式
            m_os.setf( std::istream::fixed,
                       std::istream::floatfield );
            m_os.precision( N );

            //输出时间
            m_os << elapsed() << " s\n" // "s" 表示秒
                  << std::endl;
        }
        //自动恢复流状态
    }
}

```

10.2 system

C++中处理错误的最佳方式是使用异常，但操作系统和许多底层 API 不具有这个能力，它们一般使用更通用也更难以操作的错误代码来表示出错的原因，不同的操作系统的错误代码通常不

是兼容的，这给编写跨平台的程序带来了很大的麻烦。

system 库使用轻量级的对象封装了操作系统底层的错误代码和错误信息，使调用操作系统功能的程序可以被很容易地移植到其他操作系统。它作为基础部件已经被 filesystem (10.3 小节, 380 页) 和 asio (12.2 小节, 493 页) 等库调用，并且被接受为 C++0x TR2。

10.2.1 编译 system 库

system 库需要编译才能使用，bjam 命令如下：

```
bjam -toolset=msvc -with-system -build-type=complete stdlib=stlport stage
```

如果要使用工程中嵌入源码的方式，可以直接在 cpp 文件中包含 system 库的实现代码，非常简单，只有短短的两行代码：

```
//sysprebuild.cpp
#define BOOST_SYSTEM_NO_LIB
#include <libs/system/src/error_code.cpp>
```

system 库位于名字空间 boost::system，在工程中使用 system 库时，需要包含头文件 <boost/system/error_code.hpp>。

如果使用嵌入源码编译的方式，还要在头文件之前定义宏 BOOST_SYSTEM_NO_LIB 或者 BOOST_ALL_NO_LIB，即：

```
#define BOOST_SYSTEM_NO_LIB
#include <boost/system/error_code.hpp>
using namespace boost::system;
```

10.2.2 错误值枚举

system 库在名字空间 boost::system::errc 下定义了一个很大的枚举类 errc_t，它定义了通用的可移植的错误代码^①：

```
namespace errc                                     //boost::system::errc
{
    enum errc_t
    {
        success = 0,
        address_family_not_supported,             //EAFNOSUPPORT
        address_in_use,                           //EADDRINUSE
```

① 名字空间 errc 和枚举 errc_t 在 system 库早期曾经被命名为 posix 和 posix_errno，虽然现在还可以使用，但已经不被推荐，将来可能会取消。

```

    address_not_available,          //EADDRNOTAVAIL
    already_connected,             //EISCONN
    ....                           //其他定义
}
}

```

system 库在头文件 <boost/system/windows_error.hpp> 和 <boost/system/linux_error.hpp> 里分别定义了 Windows 和 Linux 操作系统特定的错误值枚举，分别位于名字空间 boost::system::windows_error 和 boost::system::linux_error。如果针对这两个操作系统做特定编程，则可以包含这两个头文件。

10.2.3 错误类别

system 库的核心类是 error_category、error_code 和 error_condition。

error_category 是一个抽象基类，被用于标识错误代码的类别，它的类摘要如下：

```

class error_category : public noncopyable
{
public:

    virtual const char *    name() const = 0;
    virtual string message( error_code::value_type ev ) const = 0;
    virtual error_condition default_error_condition( int ev ) const;
    virtual bool equivalent( int code, const error_condition & condition ) const;
    virtual bool equivalent( const error_code & code, int condition ) const;
};

```

error_category 的核心函数共有四个：

- 成员函数 name() 可以获得类别的名称；
- message() 可以获得错误代码 ev 对应的描述信息；
- default_error_condition() 是一个工厂方法，它由错误代码 ev 产生一个 error_condition 对象（将在下一小节介绍）；
- equivalent() 用于比较两个错误代码是否相等。

error_category 不能直接使用，必须使用继承的方式生成它的子类。system 库预定义了两个类别实例 system_category 和 generic_category，用于表示系统错误和通用的可移植错误。它们位于名字空间 boost::system，也可以用自由函数 get_system_category() 和 get_generic_category() 直接访问。

如果我们需要建立一个新的错误码类别，则可以从 error_category 派生一个新类。新类

中只有 `name()` 和 `message()` 是必须实现的，因为它们是纯虚函数，其余的函数可以使用 `error_category` 的缺省实现。

例如，定义一个新的错误码类别 `my_category` 的代码如下：

```
class my_category : public error_category
{
public:
    virtual const char *name() const
    { return "myapp_category"; }
    virtual string message(int ev) const
    {
        string msg;
        switch(ev)                //使用 switch-case 语句产生错误消息
        {
            case 0:
                msg = "ok";break;
            default:
                msg = "some error";break;
        }
        return msg;
    }
};
```

10.2.4 错误代码

`error_code` 和 `error_condition` 都用于表示错误代码，但 `error_code` 更接近操作系统和底层 API，而 `error_condition` 则更偏重于可移植。两者的声明很类似，但 `error_code` 多了一个 `default_error_condition()` 的方法，可以把 `error_code` 转换成 `error_condition`。

`error_code` 的类摘要如下：

```
class error_code
{
public:
    error_code();
    error_code(int val, const error_category & cat );

    void assign( int val, const error_category & cat );
    void clear();

    int value() const;
    const error_category & category() const;
    error_condition default_error_condition() const;
    string message() const;
    operator unspecified-bool-type() const;
};
```

`error_code` 和 `error_condition` 都可以从一个整数错误值或者 `errc_t` 枚举构造，并同时指定所属的错误类别。如果无参构造，那么错误值将会是 0（无错误）。

`error_code` 对象内部的值可以用 `value()` 获得，获得错误描述要使用 `message()`，它将调用错误码类别对象的 `message()` 返回描述信息。函数 `category()` 可以返回错误代码所属的类别，因此 `message()` 相当于：

```
ec.category().message(ec.value())
```

不同的错误类别决定了 `error_code` 的含义，相同的错误代码如果属于不同的错误类别，那么将具有不同的含义。

```
my_category my_cat;                                //一个自定义错误类别的实例
error_code ec(10, my_cat);                          //错误码 10, 自定义类别
cout << ec.value() << ec.message() << endl;
ec = error_code(10, system_category);               //系统错误类别
cout << ec.value() << ec.message() << endl;
```

注意：在使用自定义错误类别的时候我们不能向 `error_code` 传递临时对象构造：

```
error_code ec(10, my_category());                   //错误
```

因为 `error_code` 的构造函数接受的是 `error_category` 的引用，不会拷贝副本。上面的代码虽然可以正确编译，但运行时因为临时对象在语句结束后析构，再调用 `message()` 成员函数时会发生未定义行为。

这段代码在 Windows 环境下运行结果是：

```
10some error
10 环境不正确。
```

`error_code` 和 `error_condition` 都提供隐式 `bool` 转换的能力，可以在一个 `bool` 上下文中转换成 `bool` 值，含义与我们通常处理的错误代码含义相同，当错误值非 0 时返回 `true`。

进一步示范这两个错误代码类用法的代码如下：

```
#define BOOST_SYSTEM_NO_LIB
#include <boost/system/error_code.hpp>
using namespace boost::system;
int main()
{
    cout << system_category.name() << endl;           //输出类别名称

    error_code ec;                                    //缺省构造一个错误代码对象
    assert(ec.value() == errc::success);               //默认无错误
    assert(!ec);                                       //默认无错误, 可隐式 bool 转换
```

```

assert(ec.category() == system_category);

ec.assign(3L, system_category);           //错误值为 3
error_condition econd = ec.default_error_condition();
                                           //产生一个可移植的错误代码 econd

//也可以直接用 system_category 产生可移植的错误代码
assert(econd == system_category.default_error_condition(3L));

cout << ec.message() << endl;             //输出错误描述信息
cout << econd.message() << endl;
cout << econd.value() << endl;            //输出可移植的错误代码
}

```

程序在 Windows 环境的运行结果如下：

```

system
系统找不到指定的路径。
No such file or directory
2

```

最后一行的输出表明相同错误的 `error_code` 和 `error_condition` 可能错误码并不相同，因此在编写跨平台的程序时应尽量使用 `error_condition`，并且使用不依赖于具体的错误值的 `errc_t` 枚举。

10.2.5 错误异常

`system` 库还提供一个异常类 `system_error`，它是 `std::runtime_error` 的子类，是对 `error_code` 的一个适配，可以把 `error_code` 应用到 C++ 的异常处理机制之中。

`system_error` 位于名字空间 `boost::system`，使用时需要另外包含头文件 `<boost/system/system_error.hpp>`，即：

```

#include <boost/system/system_error.hpp>
using namespace boost::system;

```

它的类摘要如下：

```

class system_error : public std::runtime_error
{
public:
    system_error( error_code ec );
    system_error( error_code ec, const char * what_arg );

    const error_code & code() const throw();
    const char *      what() const throw();
};

```

`system_error` 的用法与标准异常没有太多的差别, 构造时需要传入 `error_code` 对象, 新增的成员函数 `code()` 可以返回这个 `error_code` 对象的拷贝。

示范 `system_error` 用法的代码如下:

```
#define BOOST_SYSTEM_NO_LIB
#include <boost/system/system_error.hpp>
using namespace boost::system;
int main()
{
    try
    {
        //抛出 system_error 异常
        throw system_error(error_code(5, system_category()));
    }
    catch (system_error& e)
    {
        cout << e.what();
    }
}
```

10.3 filesystem

目录、文件处理是脚本语言如 Shell、Python、Perl 所擅长的领域, 但 C++ 语言缺乏对操作系统中文件的查询和操作能力, 因此 C++ 程序员经常需要再掌握另外一门脚本语言以方便自己的工作, 这增加了学习的成本。

`filesystem` 库是一个可移植的文件系统操作库, 它在底层做了大量的工作, 使用 POSIX 标准表示文件系统的路径, 使 C++ 具有了类似脚本语言的功能, 可以跨平台操作目录、文件, 写出通用的脚本程序。

`filesystem` 库同样被接受为 C++0x TR2, 即将出现在 C++ 新标准中。^①

10.3.1 编译 filesystem 库

`filesystem` 库需要 `system` 库支持, 因此必须先编译 `system` 库, 请参考 10.2.1 小节。

`filesystem` 库需要编译才能使用, `bjam` 命令如下:

```
bjam -toolset=msvc -with-filesystem -build-type=complete stdlib=stlport stage
```

① 在 Boost1.42 版中的 `filesystem` 库是 V2, 目前库作者正在规划 V3, 现有的部分接口可能将来会发生变化。

如果要使用工程中嵌入源码的方式，可以直接在 cpp 文件中包含 filesystem 库的实现代码，如下：

```
//fsprebuild.cpp
#define BOOST_SYSTEM_NO_LIB //使用 system 的预编译形式
#define BOOST_FILESYSTEM_NO_LIB
#include <boost/filesystem.hpp>

#include <libs/filesystem/src/utf8_codecvt_facet.hpp>
#include <libs/filesystem/src/utf8_codecvt_facet.cpp>
#include <libs/filesystem/src/path.cpp>
#include <libs/filesystem/src/operations.cpp>
#include <libs/filesystem/src/portability.cpp>
```

filesystem 位于名字空间 `boost::filesystem`，为了使用 filesystem 组件，需要包含头文件 `<boost/filesystem.hpp>`。

如果使用嵌入源码的方式，需要在包含头文件之前定义宏 `BOOST_FILESYSTEM_NO_LIB`、`BOOST_SYSTEM_NO_LIB` 或者直接使用 `BOOST_ALL_NO_LIB`。即：

```
#define BOOST_SYSTEM_NO_LIB
#define BOOST_FILESYSTEM_NO_LIB
#include <boost/filesystem.hpp>
using namespace boost::filesystem;
```

10.3.2 类摘要

filesystem 库的核心类是 `basic_path`，它屏蔽了不同文件系统的差异，使用可移植的 POSIX 语法提供了通用的目录、路径表示，并且支持 POSIX 的符号链接概念。

`basic_path` 的类摘要如下：

```
template <class String, class Traits>
class basic_path
{
public:

    //构造函数
    basic_path();
    basic_path(const string_type& s);
    template <class InputIterator>
        basic_path(InputIterator first, InputIterator last);

    //赋值操作
    basic_path& operator=(const string_type& s);
    template <class InputIterator>
        basic_path& assign(InputIterator first, InputIterator last);
```

```

basic_path& operator/=(const string_type& s);
template <class InputIterator>
    basic_path& append(InputIterator first, InputIterator last);

void clear();
void swap( basic_path & rhs );
basic_path& remove_filename();
basic_path& replace_extension(const string_type & new_extension = "");

const string_type string() const;
const string_type file_string() const;
const string_type directory_string() const;

const external_string_type external_file_string() const;
const external_string_type external_directory_string() const;

string_type root_name() const;
string_type root_directory() const;
basic_path root_path() const;
basic_path relative_path() const;

basic_path parent_path() const;
string_type filename() const;

string_type stem() const;
string_type extension() const;

bool empty() const;
bool is_complete() const;
bool has_root_name() const;
bool has_root_directory() const;
bool has_root_path() const;
bool has_relative_path() const;
bool has_filename() const;
bool has_parent_path() const;

//iterators
iterator begin() const;
iterator end() const;
};

```

`basic_path` 提供了丰富的成员函数，基本涵盖了文件系统的大部分操作。但通常我们不直接使用 `basic_path`，而是使用预定义的 typedef: `path` 和 `wpath`^①，它们的定义如下：

① `property_tree` 库中也有一个同名的 `path` 类，用来表示属性树的查找路径，它位于名字空间 `boost::property_tree`，读者务必小心，不要弄混了。

```
typedef basic_path< std::string, path_traits > path;
typedef basic_path< std::wstring, wpath_traits > wpath;
```

`basic_path` 还支持流输出操作，可以以字符串的形式打印出内部保存的路径表示，为我们提供了一个方便的调试手段。

接下来的讨论，我们将主要使用 `path` 类。

10.3.3 路径表示

`path` 的构造函数可以接受 C 字符串和 `string`，也可以是一个指定首末迭代器字符串序列区间。`path` 使用标准的 POSIX 语法提供可移植的路径表示，它也是 Unix、Linux 的原生路径。POSIX 语法使用斜杠 (/) 来分隔文件名和目录名，点号 (.) 表示当前目录，双点号 (..) 表示上一级目录。例如：

```
path p1("./a_dir");
path p2("/usr/local/lib");
```

`path` 也支持操作系统的原生路径表示，比如在 Windows 下使用盘符，分隔符使用反斜杠 (\)：

```
path p3("c:\\tmp\\test.text");
path p4("d:/boost/boost/filesystem/");
```

因为反斜杠被 C++ 定义为转义符，在字符串中使用反斜杠表示路径的时候必须连续写 \\ 才能被识别成一个 \，因此这种方式通常没有 Unix 风格的斜杠方式方便。

空的构造函数创建一个空路径对象，不表示任何路径，成员函数 `empty()` 可以判断路径是否为空：

```
path p5; // 缺省构造为空路径
assert(p.empty());
```

`path` 的构造函数没有被声明为 `explicit`，因此字符串可以被隐式转换为 `path` 对象，这在编写操作文件系统的代码时非常方便，可以不用创建一个临时的 `path` 对象。

`path` 重载了 `operator/=`，可以像使用普通路径一样用 / 来追加路径，成员函数 `append()` 也可以追加一个字符串序列。例如：

```
char str[] = "the path is (/root)."; // 一个普通字符串数组
path p(str + 13, str + 14); // 区间方式，取字符串中的一个斜杠
// 字符
assert(!p.empty()); // 路径非空
```

```

p /= "etc"; //使用 operator/=追加路径
string filename = "xinetd.conf";
p.append(filename.begin(), filename.end()); //追加字符序列
cout << p << endl; //p = /etc/xinetd.conf

```

自由函数 `system_complete()` 可以返回路径在当前文件系统中的完整路径（也就是通常所说的绝对路径），例如

```
cout << system_complete(p) << endl;
```

在 Windows 下的输出可能是（依据可执行程序所在盘而不同）：

```
d:/etc/xinetd.conf
```

而在 Linux 下则是：

```
/etc/xinetd.conf
```

需要注意：`path` 仅仅是用于表示路径，而并不关心路径中的文件或目录是否存在，路径也可能是个在当前文件系统中无效的名字，例如在 Windows 下不允许文件名或目录名使用冒号、尖括号、星号、问号等等，但 `path` 并不禁止这种表示：

```
path p("/::/*/?/<>"); //完全合法
```

10.3.4 可移植的文件名

为了提高程序在不同文件系统上的可移植性，`filesystem` 库提供了一系列的文件名（或目录名）检查函数，可以根据系统的命名规则判断一个文件名字符串的有效性，从而尽可能地为程序员编写可移植的程序创造便利。

自由函数 `portable_posix_name()` 和 `windows_name()` 分别检测文件名字符串是否符合 POSIX 规范和 Windows 规范，保证名字可以移植到符合 POSIX 的类 Unix 操作系统和 Windows 操作系统上。

POSIX 规范只有一个很小的字符集用于文件名，包括大小写字母、点号、下画线和连字符，而 Windows 则范围要广一些，仅不允许 `<>?:|/\` 等少量字符。例如：

```

string fname("w+abc.xxx");
assert(!portable_posix_name(fname)); //posix 非法文件名
assert(windows_name(fname)); //Windows 合法文件名

```

函数 `portable_name()` 判断名字是否是一个可移植的文件名，相当于 `portable_posix_name()` && `windows_name()`，但名字不能以点号或者连字符开头，并允许表示当前目录的“.”和父目录的“..”。它保证文件名可以移植到所有现代操作系统和一些旧有的操作系统上。

`portable_directory_name()` 的判断规则进一步严格, 它包含了 `portable_name()`, 并且要求名字中不能出现点号, 目录名可以移植到 OpenVMS 操作系统上。

`portable_file_name()` 类似 `portable_directory_name()`, 提供更可移植的文件名, 它要求文件名中最多有一个点号, 且后缀名不能超过 3 个字符。

最后, `filesystem` 库提供一个 `native()` 函数, 它判断文件名是否符合本地文件系统命名规则, 在 Windows 下它等同于 `windows_name()`, 而在其他操作系统下则只是简单地判断文件名不是空格且不包含斜杠。

这些文件名检查函数的用法如下:

```
assert(!portable_name("w+() abc.txt")
    && !portable_name("./abc"));
assert(!portable_directory_name("a.txt") &&
    portable_directory_name("abc"));
assert(portable_file_name("a.bc")
    && !portable_file_name("y.conf"));
```

10.3.5 路径处理

`path` 类提供了丰富的函数用于处理路径, 可以获取文件名、目录名、判断文件属性等等, 本节不能全部介绍, 只能叙述其中最常用的一部分。^①

`path` 的成员函数 `string()` 返回标准格式的路径表示, `directory_string()` 返回文件系统格式的路径表示, `parent_path()`、`stem()`、`filename()` 和 `extension()` 分别返回路径中的父路径、不含扩展名的全路径名、文件名和扩展名:

```
path p("/usr/local/include/xxx.hpp");

cout << p.string() << endl;
cout << p.directory_string() << endl;
cout << p.parent_path() << endl;
cout << p.stem() << endl;
cout << p.filename() << endl;
cout << p.extension() << endl;
```

在 Windows 下运行结果是:

```
/usr/local/include/xxx.hpp
```

① `filesystem` 库早期版本曾经把文件系统比拟成一棵目录树, 有 `leaf()`、`branch()` 等比喻函数来获得文件名和路径名, 但现在这些名字都已经被废弃, 不推荐使用。

```

\usr\local\include\xxx.hpp
/usr/local/include/
/usr/local/include/xxx
xxx.hpp
.hpp

```

成员函数 `is_complete()` 用于检测 `path` 是否是一个完整（绝对）路径，这需要依据具体的文件系统的表示，例如下面代码的第一个断言在 Windows 下是成立的，而在 Linux 下则不成立，因为 Windows 系统的完整路径需要包括盘符：

```

assert(!p.is_complete());
assert(system_complete(p).is_complete());           //总是完整路径

```

`root_name()`、`root_directory()`、`root_path()` 这三个成员函数用于处理根目录，如果 `path` 中含有根，那么它们分别返回根的名字、根目录和根路径：

```

cout << p.root_name() << endl;
cout << p.root_directory() << endl;
cout << p.root_path() << endl;

```

这段代码在 Linux 下将输出一个空字符串和两个斜杠 (/)，因为 Linux 下的根没有名字。而在 Windows 下，如果 `path` 是如 “c:/xxx/yyy” 的形式，输出将会是：“c:”、“/” 和 “c:/”。

成员函数 `relative_path()` 返回 `path` 的相对路径，相当于去掉了 `root_path()`。

根路径和相对路径的这四个函数都有对应的 `has_xxx()` 的形式，用来判断是否存在对应的路径，同样，`has_filename()` 和 `has_parent_path()` 用于判断路径是否有文件名或者父路径。例如：

```

assert(!p.has_root_name());
assert(p.has_root_path());
assert(p.has_parent_path());

```

之前讨论的成员函数都不会改变 `path` 的值，接下来的两个函数能够原地修改 `path`：

- `remove_filename()` 函数可以删除路径中最后的文件名，把 `path` 变为纯路径表示；
- `replace_extension()` 可以变更文件的扩展名。

例如：

```

cout << p.replace_extension() << endl;
cout << p.replace_extension("hxx") << endl;
cout << p.remove_filename() << endl;

```

它的运行结果是：

```
/usr/local/include/xxx
/usr/local/include/xxx.hxx
/usr/local/include/
```

也可以对两个 `path` 对象执行比较操作，它基于字典序并且大小写敏感比较路径字符串，提供 `operator<` 操作符，并基于 `<` 提供其他的比较操作，因此它的 `operator==` 是等价语义的：

```
path p1("c:/test/1.cpp");
path p2("c:/TEST/1.cpp");
path p3("c:/abc/1.cpp");

assert(p1 != p2);
assert(p2 < p3);
```

`path` 类还提供迭代器 `begin()` 和 `end()`，可以迭代路径中的字符串，例如：

```
path p = "/boost/tools/libs";

BOOST_AUTO(pos, p.begin());
while(pos != p.end())
{
    cout << "[" << *pos << "];"
    ++pos;
}
```

输出将会是：

```
[/][boost][tools][libs]
```

如果 `path` 类提供的这些操作还不足以满足要求，那么我们可以使用 Boost 中的字符串处理库，如 `string_algo` (5.3 小节, 175 页)、`xpressive` (5.5 小节, 196 页)，从 `path` 的字符串表达中提取字符串做处理，然后再赋值给 `path` 对象。

10.3.6 异常

`filesystem` 库使用异常来处理文件操作时发生的错误，异常类是一个模板类：`basic_filesystem_error`，它是 `system` 库中 `system_error` (10.2.5 小节, 379 页) 的子类。

`basic_filesystem_error` 的类摘要如下：

```
template <class Path>
class basic_filesystem_error : public system_error
{
public:
    typedef Path path_type;

    explicit basic_filesystem_error(const std::string& what_arg, error_code ec);
```

```
basic_filesystem_error(const std::string& what_arg,
    const path_type& p1, error_code ec);
basic_filesystem_error(const std::string& what_arg,
    const path_type& p1, const path_type& p2, error_code ec);

const path_type& path1() const;
const path_type& path2() const;

const char * what() const;
};
```

`basic_filesystem_error` 的接口很简单, 由于继承自 `system_error`, 因此可以使用 `code()` 获得 `error_code` 值, 它还可以使用 `path1()` 和 `path2()` 来获取发生异常时的路径对象。

`basic_filesystem_error` 也有两个 typedef: `filesystem_error` 和 `wfilesystem_error`, 我们通常应该使用这两个预定义的异常类。它们的定义如下:

```
typedef basic_filesystem_error<path> filesystem_error;
typedef basic_filesystem_error<wpath> wfilesystem_error;
```

例如, 下面的代码检查文件的大小, 但文件不存在, 将抛出异常:

```
path p("c:/test.txt");

try
{
    file_size(p);
}
catch(filesystem_error& e)
{
    cout << e.path1() << endl;
    cout << e.what() << endl;
}
```

由于文件系统位于程序之外, 是不可控且全局共享的, 因此访问目录或文件随时都有可能发生异常, 例如文件已经删除、文件已经存在等。

读者需要注意, 本书的示范代码都没有使用 try-catch, 只是出于使代码清晰易读的目的。为了程序的健壮性, 应总使用 try-catch 块来保护文件访问代码。

10.3.7 文件状态

`filesystem` 库提供一个文件状态类 `file_status` 及一组相关函数, 用于检查文件的各种属性, 如是否存在、是否是目录、是否是符号链接, 等等。

`file_status` 的类摘要如下:

```

class file_status
{
public:
    explicit file_status( file_type v = status_unknown );

    file_type type() const;
    void type( file_type v );
};

```

`file_status` 的成员函数 `type()` 可以获得文件的状态，它是一个枚举值。但通常我们不直接使用 `file_status` 类（就像是 `std::typeinfo`），而是用相关函数返回的 `file_status` 对象。

函数 `status(const Path& p)` 和 `symlink_status(const Path& p)` 测试路径 `p` 的状态，结果可以用 `file_status` 的 `type()` 获得，如果路径 `p` 不能被解析则会抛出异常 `filesystem_error`。

文件可能的状态 `file_type` 是个枚举类型，值如下：

- `file_not_found` : 文件不存在；
- `status_unknown` : 文件存在但状态未知；
- `regular_file` : 是一个普通文件；
- `directory_file` : 是一个目录；
- `symlink_file` : 是一个链接文件；
- `block_file` : 是一个块设备文件；
- `character_file` : 是一个字符设备文件；
- `fifo_file` : 是一个管道设备文件；
- `socket_file` : 是一个 `socket` 设备文件；
- `type_unknown` : 文件的类型未知。

下面的断言在 Windows 环境下均成立（使用了 `path` 的隐式转换）：

```

assert(status("d:/boost").type() == directory_file);
assert(status("d:/boost/nofile").type() == file_not_found);
assert(status("d:/boost/README.txt").type() != symlink_file);
assert(status("d:/boost/README.txt").type() == regular_file);

```

下面的断言在 Linux 环境下均成立（使用了 path 的隐式转换）：

```
assert(status("/dev/null").type() == character_file);
assert(status("/root").type()    == directory_file);
assert(status("/bin/sh").type()  == regular_file);
```

filesystem 库还提供了一些便利的谓词函数 `is_xxx()`，可以简化对文件状态的判断，例如：

```
assert(is_directory("d:/boost");
assert(!exists("d:/boost/nofile"));
assert(!is_symlink("d:/boost/README.txt"));
assert(!is_other("d:/boost/README.txt"));
assert(is_regular_file("d:/boost/README.txt"));
assert(!is_empty("d:/boost/README.txt"));
```

大部分谓词函数都可以望文知意，比较特别的是 `is_other()` 和 `is_empty()`。当文件存在且不是普通文件、目录或链接文件时 `is_other()` 返回 true。如果 path 对象是目录，那么当目录里无文件时 `is_empty()` 返回 true，如果 path 对象是文件，那么文件长度为 0，`is_empty()` 返回 true。

函数 `equivalent(const Path1& p1, const Path2& p2)` 可以比较两个目录实体是否是同一个，但如果 `(!exists(s1) && !exists(s2)) || (is_other(s1) && is_other(s2))` 则无法比较，会抛出异常。

10.3.8 文件属性

受可移植的限制，很多文件属性不是各平台共通的，例如 Windows 下的只读、归档属性、Linux 下的不同用户的读写执行权限等等，因此 filesystem 库仅提供了少量的文件属性操作：

- 函数 `initial_path()` 返回程序启动时（进入 `main()` 函数）的当前路径；
- 函数 `current_path()` 返回当前路径。它和 `initial_path()` 返回的都是一个完整路径（绝对路径）；
- 函数 `file_size()` 以字节为单位返回文件的大小；
- 函数 `last_write_time()` 返回文件的最后修改时间，是一个 `std::time_t`。

`last_write_time()` 还可以额外接受一个 `time_t` 参数，修改文件的最后修改时间，就像是使用 Linux 下的 `touch` 命令。

这些函数都要求操作的文件必须存在，否则会抛出异常，`file_size()` 还要求文件必须是个普通文件（`is_regular_file(name) == true`）。

示范这几个函数用法的代码如下：

```
cout << initial_path() << endl;
cout << current_path() << endl;

path p("d:/boost/README.txt");
cout << file_size(p) << endl;
time_t t = last_write_time(p);
last_write_time(p, time(0));           //更新 README.txt 的修改时间
```

此外，函数 `space()` 可以返回一个 `space_info` 结构，它表明了该路径下的磁盘空间分配情况，`space_info` 结构的定义如下：

```
struct space_info
{
    uintmax_t capacity;
    uintmax_t free;
    uintmax_t available;
};
```

`space()` 函数可以这样使用：

```
const int GBYTES = 1024*1024*1024;
space_info si = space("d:/");
cout << si.capacity / GBYTES << endl;
cout << si.available / GBYTES << endl;
cout << si.free / GBYTES << endl;
```

10.3.9 文件操作

`filesystem` 库基于 `path` 的路径表示提供了基本的文件操作函数，如创建目录（`create_directory`）、文件改名（`rename`）、文件删除（`remove`）、文件拷贝（`copy_file`）等等，这些函数的名字都很容易理解。

示范文件操作用法的代码如下：

```
#define BOOST_SYSTEM_NO_LIB
#define BOOST_FILESYSTEM_NO_LIB
#include <boost/filesystem.hpp>
using namespace boost::filesystem;
int main()
{
    namespace fs = boost::filesystem;

    path ptest = "c:/test";
    if (exists(ptest))
    {
        if (is_empty(ptest))
```

```

    {
        fs::remove(ptest) ; //remove 只能删除空目录或文件
    }
    else
    {
        remove_all(ptest); //remove_all 可以递归删除
    }
}

assert(!exists(ptest));
create_directory(ptest) ; //创建一个目录

copy_file("d:/boost/readme.txt", ptest / "a.txt"); //拷贝
assert(exists(ptest / "a.txt"));
rename(ptest / "a.txt", ptest / "b.txt"); //改名
assert(exists(ptest / "b.txt"));

//使用 create_directories 可以一次创建多级目录
create_directories(ptest / "sub_dir1" / "sub_dir1");
}

```

注意：在这段代码中我们为 `remove()` 函数加上了名字空间限定，因为在 VC 中有一个同名的宏会掩盖 `filesystem` 库的 `remove()` 函数。

10.3.10 迭代目录

`filesystem` 库使用 `basic_directory_iterator` 提供了迭代一个目录下的所有文件的功能，它预定义了两个 `typedef`: `directory_iterator` 和 `wdirectory_iterator`。

`directory_iterator` 用法有些类似 `string_algo` 库的 `find_iterator`、`split_iterator` (5.3.12 小节, 188 页) 或 `xpressive` 库的 `regex_token_iterator` (5.5.8 小节, 207 页)，空的构造函数生成一个逾尾 `end` 迭代器，传入 `path` 对象构造将开始一个迭代操作，反复调用 `operator++` 即可迭代目录下的所有文件。例如：

```

directory_iterator end;
for (directory_iterator pos("d:/boost/"); pos != end; ++pos)
{
    cout << *pos << endl;
}

```

需要注意：`basic_directory_iterator` 迭代器返回的对象并不是 `path`，而是一个 `basic_directory_entry` 对象，但 `basic_directory_entry` 类定义了一个到 `path` 的类型转换函数，因此可以在需要 `path` 的语境中隐式转换到 `path`。

`basic_directory_entry` 的类摘要如下：

```

template <class Path> class basic_directory_entry

```



```

{
public:
    typedef Path path_type;
    const Path& path() const;

    file_status status() const;
    file_status symlink_status() const;
};

```

`basic_directory_entry` 可以用 `path()` 方法返回路径, `status()` 返回路径的状态, 它也有两个 `typedef`, 与 `directory_iterator` 对应的是 `directory_entry`。

`directory_iterator` 只能迭代本层目录, 不支持深度遍历目录, 可以使用递归来实现这个功能, 并不是很难:

```

void recursive_dir(const path& dir)                //递归遍历目录
{
    directory_iterator end;                        //结束迭代器调用
    for (directory_iterator pos(dir); pos != end; ++pos)
    {
        if (is_directory(*pos))
        {
            recursive_dir(*pos);                  //是目录则递归遍历
        }
        else
        {
            cout << *pos << endl;                //不是目录则输出路径
        }
    }
}

```

由于递归遍历文件系统目录的功能非常有用也很有必要, 因此 `filesystem` 库专门提供了另外一个类 `basic_recursive_directory_iterator`, 它遍历目录的效率要比递归的 `directory_iterator` 高很多。

`basic_recursive_directory_iterator` 的基本用法与 `directory_iterator` 相同, 只是它可以递归遍历目录结构。 `basic_recursive_directory_iterator` 的类摘要如下:

```

template <class Path>
class basic_recursive_directory_iterator
{
    //构造、析构函数以及迭代器通用操作被略去
public:
    int level() const;
    void pop();
    void no_push();
}

```

```
private:
    int m_level;
};
```

通常我们使用 `basic_recursive_directory_iterator` 的两个 typedef: `recursive_directory_iterator` 和 `wrecursive_directory_iterator`。

因为它的名字实在是太长了，下面为了叙述方便将它简称为 `rd_iterator`，相当于有：

```
typedef recursive_directory_iterator rd_iterator;
```

`rd_iterator` 具有目录迭代器的基本功能，`operator++` 会令它返回目录中的下一个文件。它的特别之处是可以深度搜索目录，迭代当前目录及子目录下的所有文件。

成员函数 `level()` 返回当前的目录深度 `m_level`，当 `rd_iterator` 构造时（未开始遍历）`m_level == 0`，每深入一层子目录则 `m_level` 增加，退出时减少。成员函数 `pop()` 用于退出当前目录层次的遍历，同时 `--m_level`。当迭代到一个目录时，`no_push()` 可以让目录不参与遍历，使 `rd_iterator` 的行为等价于 `directory_iterator`。

使用 `rd_iterator`，深度遍历目录的操作会变得非常简单：

```
rd_iterator end;
for (rd_iterator pos("d:/test"); pos != end; ++pos)
{
    cout << "level" << pos.level() << ":" << *pos << endl;
}
```

下面的代码使用 `no_push()` 令 `rd_iterator` 的行为等价于 `directory_iterator`：

```
rd_iterator end;
for (rd_iterator pos("d:/test"); pos != end; ++pos)
{
    if (is_directory(*pos))
    {
        pos.no_push(); //使用 no_push(), 不深度遍历
    }
    cout << *pos << endl;
}
```

恰当地使用 `level()` 和 `no_push()`，还可以实现指定深度的目录遍历，读者可以自行试验一下。

10.3.11 实例 1：实现查找文件功能

了解了 `filesystem` 库迭代目录的功能后，作为示范，本小节来实现查找文件的功能，我们

将查找文件的函数命名为 `find_file()`。

因为查找文件很有可能找不到，因此我们选择使用 `optional<path>` 作为返回值类型。函数中首先判断目录的状态，如果不存在或者非目录则直接返回空的 `optional` 对象，表示查找失败。然后使用 `rd_iterator` 递归迭代目录内容，如果是文件则比较 `path` 对象的文件名，相等则查找成功。

```
#include <boost/optional.hpp>
optional<path>
find_file(const path& dir, const string& filename)
{
    typedef optional<path> result_type;           //返回值类型定义
    if (!exists(dir) || !is_directory(dir))
    {
        return result_type();
    }

    rd_iterator end;                             //递归迭代器
    for (rd_iterator pos(dir); pos != end; ++pos)
    {
        if(!is_directory(*pos) &&                //不是目录
            pos->path().filename() == filename)  //文件名相等
        {
            return result_type(pos->path());
        }
    }
    return result_type();
}
```

`find_file()` 可能并不是很高效，但作为通常的使用足够了，用法很简单：

```
int main()
{
    optional<path> r = find_file("d:/atest", "README.txt");
    if (r)
    {
        cout << *r << endl;
    }
    else
    {
        cout << "file not found." << endl;
    }
}
```

`find_file()` 还有个缺陷，不支持通配符模糊查找，如果想要进一步增强它的话就需要使用正则表达式。

10.3.12 实例 2：实现模糊查找文件功能

使用正则表达式处理字符串的强大功能，我们可以很容易地实现完全的模糊文件查找。

作为示范，我们只支持通配符`*`，大小写敏感并且不处理点号以外的正则表达式其他特殊符号（如括号和竖线），读者可以自己扩展增强它的功能（这很容易）。

下面是模糊查找的几个实现要点:

- 文件名中用于分隔主名与扩展名的点号必须转义, 因为点号在正则表达式中是一个特殊的字符;
- 通配符*应该转换为正则表达式的.*, 以表示任意多的字符;
- 在判断文件名是否查找到时我们应该使用正则表达式而不是简单的判断相等。
- 函数的返回值不能再使用 optional<path>, 因为模糊查找可能会返回多个结果, 因此应该使用 vector<path>。

find_files()的实现代码如下:

```
#include <boost/xpressive/xpressive_dynamic.hpp>
#include <boost/algorithm/string.hpp>
using namespace boost::xpressive;
void find_files(const path& dir, const string& filename, vector<path>& v)
{
    static xpressive::sregex_compiler rc;           //正则表达式工厂
    if (!rc[filename].regex_id())
    {
        string str = replace_all_copy(
            replace_all_copy(filename, ".", "\\."),
            "*", ".*");
        rc[filename] = rc.compile(str);             //处理文件名
                                                    //创建正则表达式
    }
    typedef vector<path> result_type;               //返回值类型定义
    if (!exists(dir) || !is_directory(dir))         //目录检查
    {
        return ;
    }

    rd_iterator end;                                //递归迭代器逾尾位置
    for (rd_iterator pos(dir); pos != end; ++pos)
    {
        if (!is_directory(*pos) &&
            regex_match(pos->path().filename(), rc[filename]))
        {
            v.push_back(pos->path());               //找到, 加入 vector
        }
    }
}
```

代码中在处理正则表达式时我们使用了工厂类 sregex_compiler, 并把它声明为函数内部的静态变量:

```
static xpressive::sregex_compiler rc;
```

因为 `sregex_compiler` 本身是一个 flyweight 工厂，它能够保证创建唯一的正则表达式对象，可以避免重入 `find_files()` 反复创建正则表达式对象的代价。如果我们不使用 `rd_iterator` 而是使用 `directory_iterator` 的递归形式就更能显现出它的好处：`sregex_compiler` 和正则表达式对象在反复的递归调用中都仅会被创建一次，节约大量的内存和时间。

为了仅使用一个正则表达式对象，我们需要使用 `sregex_compiler` 的 flyweight 用法，使用 `filename` 作为享元的索引键值，`sregex` 的 `regex_id()` 函数用来确定该享元是否已经被创建。如果已经被创建则直接使用，如果未被创建则使用 `string_algo` 库的 `replace_all_copy()` 把文件名中的 `.` 和 `*` 分别替换为 `\.` 和 `*`，然后编译为正则表达式对象，生成一个享元。即：

```
if (!rc[filename].regex_id())
{
    //cout << "should be once" << endl;
    string str = replace_all_copy(
        replace_all_copy(filename, ".", "\\."),
        "*", "\\*");
    rc[filename] = rc.compile(str);
}
```

`find_files()` 的查找目录过程与 `find_file()` 基本相同，文件名的比较改为使用 `regex_match()`，如果匹配成功则用 `push_back()` 加入到 `vector` 中。

`find_files()` 的用法同样很简单：

```
int main()
{
    vector<path> v;
    find_files("d:/atest", "*.txt", v);
    cout << v.size() << endl;

    BOOST_FOREACH(path &p, v)
    {
        cout << p << endl;
    }
}
```

顺便说一句，本小节查找文件的目标目录特意没有选用 `d:/boost`，因为 `boost` 目录里含有数千个目录和数万个文件，完全遍历需要相当长的时间。

10.3.13 实例 3：实现拷贝目录功能

`find_files()` 是一个功能强大的函数，利用它可以做很多事情，比如为 `filesystem` 库再增加一个 `copy_files()` 函数，它对应于 `remove_all()`。

`copy_files()` 函数利用 `find_files()` 的功能，首先查找源目录里的所有文件（使用通配

符*), 把它们加入到一个 `vector<path>`。然后使用 `BOOST_FOREACH` 遍历这个 `vector`, 把路径拆分成父路径和子路径两部分, 拼接并创建目标路径, 最后调用 `copy_file()` 函数拷贝文件。

`copy_files()` 还使用了 `progress_display` (2.4 小节, 23 页), 为用户提供了一个友好的进度显示。

完整的实现代码如下:

```
#include <boost/progress.hpp>
size_t copy_files(const path& from_dir, const path& to_dir ,
                  const string& filename = "")
{
    if (!is_directory(from_dir))                //源必须是个目录
    {
        cout << "args is not a dir." << endl;
        return 0;
    }
    cout << "prepare for copy, please wait..." << endl;
    vector<path> v;
    find_files(from_dir, filename, v);          //查找源的所有文件
    if (v.empty())                               //空目录则不拷贝
    {
        cout << "0 file copied." << endl;
        return 0;
    }

    cout << "now begin copy files ..." << endl;
    path tmp;
    progress_display pd(v.size());              //进度条控件
    BOOST_FOREACH(path &p, v)                   //foreach 算法
    {
        //拆分基本路径与目标路径
        tmp = to_dir / p.string().substr(from_dir.string().length());
        if (!exists(tmp.parent_path()))         //创建子目录
        {
            create_directories(tmp.parent_path());
        }
        copy_file(p, tmp);                      //拷贝文件
        ++pd;                                   //更新进度
    }

    cout << v.size() << " file copied." << endl;
    return v.size();                            //完成拷贝
}
```

`copy_files()` 的使用方法如下, 非常简单易用:

```
int main()
{
    copy_files("d:/python", "d:/atest");
}
```

`copy_files()` 不仅能够拷贝目录下的所有文件，也可以过滤只符合通配符的文件，这需要向 `copy_files()` 传递第三个参数 `filename`，例如要拷贝所有的文本文件可以使用 `*.txt`。

不过，这里实现的 `copy_files()` 也有小小的缺陷，它不能够拷贝空目录，这是因为 `find_files()` 在查找时不考虑目录匹配。但这个缺陷很容易弥补，读者可以把对 `copy_files()` 的改进作为一个练习，实践一下 `filesystem` 库的使用。

10.3.14 文件流操作

`filesystem` 库提供了大量的文件系统操作方法，可以很方便地操作文件或目录，但它使用的是 `path` 对象，而 C++98 标准库中的文件流类 `ifstream/ofstream/fstream` 只支持 `char*` 打开文件（非常遗憾，也许是标准制订时的一个疏忽），因此使用起来很不方便，我们必须调用 `path` 对象的 `string()` 方法，然后再把 `string` 转换为 C 字符串，像这样：

```
path p("d:/boost/README.txt");
std::ifstream ifs(p.string().c_str());
```

`filesystem` 库已经被提议进入 C++0x 标准的 TR2 草案，因此新标准中的标准文件流 `fstream` 将会增加对 `path` 的支持，到那时候 `filesystem` 库将会和标准库无缝对接，使 C++ 程序员获得完整的文件处理能力。

但在 C++0x 到来之前，为了让 Boost 库用户提前享受这个便利，`filesystem` 库在额外的头文件 `<boost/filesystem/fstream.hpp>` 中提供了在名字空间 `boost::filesystem` 下的同名文件流类，它们可以如标准文件流一样使用，而且支持 `path` 对象。

`boost::filesystem` 的文件流的使用方法与标准文件流一样，例如下面的代码使用 `path` 对象打开了一个文本文件，并把它打印到标准输出上：

```
#include <boost/filesystem/fstream.hpp>
namespace newfs = boost::filesystem;
int main()
{
    path p("d:/boost/README.txt");
    newfs::ifstream ifs(p.string().c_str());
    assert(ifs.is_open());
    cout << ifs.rdbuf();
}
```

这段代码中我们使用了名字空间别名 `newfs`，用来限定使用的是 `filesystem` 库的文件流类。

这种做法带来了一个好处，如果将来代码迁移到 C++ 新标准，那里的标准库支持 `path` 的使用，那么只需要把 `newfs` 指向 `std` 名字空间，原有的所有代码都不需要变动。

10.4 program_options

在以命令行为主要操作界面的操作系统上（如 UNIX、Linux）命令行参数解析程序配置选项是一个很重要的功能。通过命令行参数，用户可以定制程序的行为，比有提示的交互式操作更有效率，而且可用于编写脚本程序，执行更复杂的功能。对于 UNIX/Linux 环境下的 C++ 程序员来说，可能没有比 `gcc/g++` 更熟悉的命令行程序了，`-c`、`-o`、`-I`、`-l`、`-L` 等选项就像是咒语一样印在他们的脑海里。

在以图形界面为主的操作系统下（如 Windows）指定程序配置选项也很有用，很多图形界面的程序也支持命令行参数，可以实现一些特殊功能，或者给管理员等高级用户使用。如果有了命令行程序，也可以很容易地编写一个图形界面的 Shell，它可以把图形界面的选项转化为一系列的命令行参数，再让命令行程序执行。

简单的命令行参数解析可以手工实现，在 UNIX/Linux 下也有库函数可以解析命令行，但他们的功能都不全面，很难实现复杂的命令行参数处理。`program_options` 库为我们提供了如 `gcc` 那样功能强大的命令行参数处理功能，它不仅能够分析命令行，也能够从配置文件甚至环境变量中获取参数，实现了非常完善的程序配置选项处理功能。^①

10.4.1 编译 program_options 库

`program_options` 库需要编译才能使用，`bjam` 命令如下：

```
bjam -toolset=msvc -with-program_options -build-type=complete stdlib=stlport stage
```

如果要使用工程中嵌入源码的方式，可以直接在 `cpp` 文件中包含 `program_options` 库的实现代码，如下^②：

```
//poprebuild.cpp
#define BOOST_PROGRAM_OPTIONS_NO_LIB
#include <boost/program_options.hpp>
```

① 如果读者还不太理解 `program_options` 库的工作，请回忆一下 `test` 库的 6.3.9 小节（229 页），那里的命令行参数就示范了 `program_options` 的用法。

② VC8 下编译 `program_options` 库会产生一些整数类型转换的警告，但不影响编译。


```
#include <libs/program_options/src/cmdline.cpp>
#include <libs/program_options/src/config_file.cpp>
#include <libs/program_options/src/convert.cpp>
#include <libs/program_options/src/options_description.cpp>
#include <libs/program_options/src/parsers.cpp>
#include <libs/program_options/src/positional_options.cpp>
#include <libs/program_options/src/utf8_codecvt_facet.cpp>
#include <libs/program_options/src/value_semantic.cpp>
#include <libs/program_options/src/variables_map.cpp>
#include <libs/program_options/src/winmain.cpp>
```

program_options 库位于名字空间 `boost::program_options`，为了使用 program_options 组件，需要包含头文件 `<boost/program_options.hpp>`。

如果使用嵌入源码的方式，需要在包含头文件之前定义宏 `BOOST_PROGRAM_OPTIONS_NO_LIB` 或者直接使用宏 `BOOST_ALL_NO_LIB`。即：

```
#define BOOST_PROGRAM_OPTIONS_NO_LIB
#include <boost/program_options.hpp>
using namespace boost::program_options;
```

10.4.2 概述

program_options 库包含很多组件用来处理程序选项，本小节将简要介绍这些组件，并用使用一个简单的例子示范它们的基本用法。

处理程序选项功能的核心类是 `options_description`，它的成员函数 `add_options()` 重载了括号操作符，用来添加选项解析，用法很像 `assign` 库（4.4 小节，106 页）。

在定义了选项参数后，函数 `store()` 和 `parse_command_line()` 将解析命令行参数串。解析的结果保存在 `variables_map`，它是 `std::map` 的派生类，可以像关联数组一样使用。`variables_map` 的 `value_type` 是 `boost::any`，用来存储不确定类型的参数值，必须用 `as<type>()` 转型才能获取具体值。

在解析完成后，程序就可以随意使用 `variables_map` 获取选项值，执行不同的分支以满足用户的请求。

以上就是 program_options 库处理程序选项的基本流程，接下来我们将借用 10.4.1 节（400 页）的查找文件功能，把它改造为一个实用命令行工具（filesystem 库的实现代码暂时忽略）。

我们需要有 `help` 选项，用于显示帮助信息，这是任何程序都必须的功能。然后需要一个文件名选项，让用户指定想要查找的文件名，这个选项名称可以叫 `filename`。完整的程序代码如下：

```

#define BOOST_PROGRAM_OPTIONS_NO_LIB
#include <boost/program_options.hpp>
using namespace boost::program_options;
int main(int argc, char* argv[]) //需要命令行参数
{
    options_description opts("demo options");

    opts.add_options() //增加两个程序选项
        ("help", "just a help info")
        ("filename", value<string>(), "to find a file")
        ;
    variables_map vm; //选项存储 map 容器
    store(parse_command_line(argc, argv, opts), vm); //解析存储

    //解析完成,实现选项处理逻辑
    if (vm.count("help")) //处理帮助选项
    {
        cout << opts << endl; //输出帮助信息
        return 0; //然后退出程序
    }
    if (vm.count("filename")) //要查找的文件名
    {
        cout << "find " << vm["filename"].as<string>() << endl;
    }
    if (vm.size() == 0)
    {
        cout << "no options" << endl;
    }
}

```

首先我们必须声明一个 `options_description` 对象 `opts`, 它是分析选项参数的基础, 然后调用它的 `add_options()` 成员函数添加两个选项, 选项的基本信息是选项名和选项的描述信息。

第一个选项是 `help`, 它不需要参数, 直接给出描述信息。第二个选项是要查找的文件名, 在给出选项名和描述信息之外, 我们还指定了选项值的类型。 `value<string>()` 是 `program_options` 库的一个辅助模板函数, 它产生一个 `typed_value` 对象, 用于保存选项值。

随后 `parse_command_line()` 函数执行命令行参数的解析, `store()` 把解析的结果存储到 `variables_map` 的实例 `vm` 中, 这样就完成了程序选项的处理。

最后我们检查 `vm` 里的选项, 成员函数 `count()` 测试选项的数量, 通常应该为 1。如果有 `help` 选项, 那么我们输出帮助信息, `options_description` 对象支持流输出, 会自动打印出所有的选项信息。如果有文件名选项, 那么我们输出文件名, 然后进行查找 (具体操作省略)。如果 `vm` 中没有任何选项信息 (`vm.size() == 0`), 那么我们提示用户没有输入选项 (当然我们也可以默认输出帮助)。

假设可执行程序名是 `demopo.exe`, 下面示范了这个程序的命令行用法, 选项名以双连字符开始, 选项值接在选项名后, 用等号或者空格分隔。如果选项值中有空格、连字符或者其他特殊

字符，那么可以使用双引号把整个值括起来：

```
D:>demopo.exe --help
demo options:
  --help          just a help info
  --filename arg  to find a file
D:>demopo.exe --filename="read me.txt"
find read me.txt
```

如果输入了未定义的选项名，程序会抛出异常，因此，这里应该使用 `try-catch` 块来包含选项解析代码，后面我们会看到允许未定义选项的用法。

使用上面介绍的 `program_options` 库基本用法，我们就可以编写程序选项处理代码了，但如果想进一步了解它的更多用法，请接着向下看。

`program_options` 库的解析程序选项功能由三个基本组件构成，分别是选项描述器、分析器和存储器。选项描述器定义选项及选项的值，分析器依据选项描述器的定义解析命令行或者数据文件，存储器则把分析器的结果保存起来供程序员使用。

我们首先来研究选项描述器组件，它是选项分析的核心部分，包括三个基本类：`value_semantic`、`option_description` 和 `options_description`。

10.4.3 选项值

`value_semantic` 是选项描述器的核心类，它定义了选项值的语义信息，但我们通常不直接使用它，而是使用它的一个子类 `typed_value`，`typed_value` 的类摘要如下：

```
template<typename T>
class typed_value
{
public:
    typed_value(T *);

    typed_value * default_value(const T &);
    typed_value * implicit_value(const T &);
    typed_value * multitoken();
    typed_value * zero_tokens();
    typed_value * composing();
    typed_value* required();
};
```

`typed_value` 的模板参数 `T` 表明它可以容纳类型为 `T` 的选项值，通常它是 `int`/`double`/`string` 等类型，因为我们必须在命令行上以文本的形式表示它。`typed_value` 使用了 `lexical_cast` (5.1 小节, 163 页) 来转换 `T` 的文本表示，因此要求类型 `T` 支持 `operator<<`

可以流输出，这对于 `int` 等类型来说都完全满足。

`typed_value` 特别支持 `vector<T>` 类型。如果一个选项的值是 `vector`，那么所有命令行上的选项值都会被保存到那个 `vector` 中，相当于指定了 `composing()`。

`typed_value` 的构造函数可以传入一个 `T*` 指针，指明值的存储位置。如果指针不为空，那么选项值将保存在指针指向的变量，这使得 `typed_value` 可以操作外部的变量，相当于一个输出参数。指针也可以为空，这时 `typed_value` 会在内部保存选项值。

`default_value()` 与 `implicit_value()` 两者的行为有些类似。`default_value()` 函数指定选项的缺省值，当选项没有在命令行出现时将使用该值。`implicit_value()` 则指定选项的“隐含值”，当选项出现但没有给值时将使用该值。如果显式给出值，长名必须用 `--x=y` 的形式，短名则要用 `-xy` 的形式，即不使用空格或等号，选项名与选项值连在一起。

`multitoken()` 和 `zero_tokens()` 是两个设置属性的成员函数，它们表示 `typed_value` 可以接受多个或者零个记号 (token)。`required()` 函数要求选项值必须被提供。

`composing()` 函数表示选项可以多次出现并且最后被合并处理，这时类型 `T` 必须能够容纳这些值。

这些成员函数都返回 `typed_value` 的 `this` 指针，因此可以用箭头操作符连续调用它们，在一个表达式中完成对 `typed_value` 的设置。

模板函数 `value<T>()` 是一个便捷工厂函数，它可以生产出 `typed_value<T>` 对象，就像是 `make_shared<>()`。

下面的代码示范了选项值的一些用法，但不能构成完整的代码，因为它不能单独使用，还必须配合选项分析器的其他组件：

```
value<string>(); // 一个存储 string 的选项

value<int>()->default_value(10) // 存储 int 的选项
->implicit_value(1); // 缺省值是 10, 隐含值是 1

double x; // 选项值的外部存储
value<double>(&x)->zero_tokens() // 存储 double 值, 必须提供
->multitoken()->required(); // 可以有 0 个或者多个记号
```

10.4.4 选项描述器

选项描述器包括两个名字很相似的类：`option_description` 和 `options_description`，注意后者多了一个字母 `s`。

option_description

`option_description` 用于描述单个选项，它提供了选项的名称、语法和详细的描述信息，并使用 `value_semantic` 类来分析处理选项值。`options_description` 则是 `option_description` 的一个聚集，它使用 `vector<shared_ptr<option_description>>` 来存储多个选项，提供了一个方便定义多个选项值的接口。

`option_description` 类有很多成员函数，但最重要的是它的构造函数，如我们在之前所看到的，接受选项名、选项值和选项描述信息创建选项对象。它的类摘要如下：

```
class option_description
{
public:
    option_description();
    option_description(const char *, const value_semantic *);
    option_description(const char *, const value_semantic *, const char *);
};
```

在构造函数指定选项名时我们可以同时指定长名和短名，两者用逗号隔开。长名在命令行用双连字符开始，选项值用空格或者等号跟在后面。短名用单连字符开始，使用空格连接后面的选项值，或者选项值紧挨着选项名。

描述信息也可以被格式化，很长的描述信息可以用“\n”分段，每段的开头可以使用空格或者“\t”来增减缩进以使格式更加美观。很遗憾，`program_options` 库只允许每段最多出现一个“\t”，因此如果文本很长，我们必须小心地排版格式。

下面的代码创建了一个 `option_description` 对象，它指定了选项的长名字是 `help`，短名字是 `h`，没有选项值需要存储，并定制了描述信息的格式：

```
option_description("help,h", 0, "help message\n"
                    "\ta bit of long text");
```

options_description

`options_description` 是 `option_description` 的容器，它可以用成员函数 `add()` 直接增加一个 `option_description` 或者一组 `option_description` 对象(即另一个 `options_description`)，但更方便的是使用 `add_options()` 函数产生一个辅助对象，然后利用 `operator()` 向容器内连续插入选项信息对象。这种写法有利于编写格式规范的代码，易于阅读和维护。

10.4.5 选项描述器的用法

下面的代码综合了前两节的知识，示范了选项描述器的使用。这次我们仍然使用 10.4.2 小

节（401 页）的例子，但为它增加更多的选项信息：

```
int main(int argc, char* argv[])
{
    options_description opts("demo options");

    string filename;                                //外部的选项值存储
    opts.add_options()                             //产生辅助对象,调用 operator()

        //帮助选项,使用空格缩进格式
        ("help,h", "help message\n a bit of long text")

        //文件名选项,值可存储到外部,缺省值是 test
        ("filename,f", value<string>(&filename)->default_value("test"), "to find
        a file")

        //搜索路径选项,可以多次出现,可以接受多个记号
        ("dir,D", value<vector<string>> >()->multitoken(), "search dir")

        //搜索深度选项,隐含值是 5,短名与搜索路径不同
        ("depth,d", value<int>()->implicit_value(5), "search depth")
        ;

    variables_map vm;                                //选项存储 map 容器
    store(parse_command_line(argc, argv, opts), vm); //解析
    notify(vm);                                       //解析结果存储外部变量

    //解析完成,调用函数实现选项处理逻辑
    print_vm(opts, vm);
}
```

这段代码与 10.4.2 小节（401 页）略有不同，增加了一个外部变量，这样可以使程序的其他部分不需要知道命令行解析的细节，也不需要访问 `variables_map` 就可以获得选项信息。在解析完成后，我们必须调用自由函数 `notify()`，来通知 `variables_map` 更新所有关联的外部变量。

下面我们定义一个单独的选项处理函数，分离了选项的定义与处理代码：

```
void print_vm(options_description &opts, variables_map &vm)
{
    if (vm.size() == 0)                                //无参数处理
    {
        cout << opts << endl;                        //输出帮助信息
        return;
    }

    if (vm.count("help"))                                //处理帮助选项
    {
        cout << opts << endl;                        //输出帮助信息
    }
}
```

```

}

//输出查找文件名, 因为它有缺省值, 故总存在
cout << "find opt:" << vm["filename"].as<string>() << endl;

if (vm.count("dir")) //处理搜索路径
{
    cout << "dir opt:";
    BOOST_FOREACH(string str, //使用 foreach 循环
        vm["dir"].as<vector<string> >())
    {
        cout << str << ","; }
    cout << endl;
}
if (vm.count("depth")) //处理搜索深度
{
    cout << "depth opt:" << vm["depth"].as<int>() << endl;
}
}

```

下面列出了程序使用不同参数的运行结果:

```

D:\>demopo.exe -h
demo options:
-h [ --help ]             help message
                           a bit of long text
-f [ --filename ] arg (=test) to find a file
-D [ --dir ] arg          search dir
-d [ --depth ] [=arg(=5)] search depth

find opt:test
D:\>demopo.exe --filename abc
find opt:abc
D:\>demopo.exe -f readme.txt --dir c: d: -d
find opt:readme.txt
dir opt:c:,d:,
depth opt:5
D:\>demopo.exe -f a.cpp --dir /usr/bin -d2 -D /etc
find opt:a.cpp
dir opt:/usr/bin,/etc,
depth opt:2

```

10.4.6 分析器

program_options 库的分析器负责解析命令行参数, 行为好像字符串处理的分词动作, 将命令行拆分成选项名与选项值, 存储在 basic_parsed_options 类中。

最简单的分析器是 parse_command_line() 函数, 它可以解析标准的 main() 函数入口的命令行参数, 我们之前一直在使用它, 它的声明如下 (省略了缺省参数):。

```

parse_command_line(int argc, charT * argv, const options_description &);

```

`parse_config_file()` 函数可以解析配置文件，输入可以是文件名或者是 IO 流 (`istream`)，这意味着我们可以让 `parse_config_file()` 直接解析文件，或者我们预先从某处读入配置到流中然后再进行解析。

配置文件的基本格式是 `name=value`，很类似 Windows 环境中常用的 `ini` 格式，选项名必须使用长名，等号两边允许有空格。

`parse_config_file()` 的基本形式声明如下：

```
parse_config_file(src, const options_description& desc,  
    bool allow_unregistered = false);
```

它的最后一个参数 `allow_unregistered` 如果为 `true`，则允许配置文件中出现未定义的选项，使分析时不会抛出异常。

`parse_config_file()` 可以这样使用：

```
stringstream ss;                                     //使用字符串流作为输入  
ss << "filename=a.cpp\ndir=/usr/bin\ndepth=10";  
store(parse_config_file(ss, opts), vm);              //解析
```

如果有一个配置文件 `config.ini`，它的内容是：

```
filename = a.cpp  
depth=10  
dir=/usr/bin  
[section]  
#comment  
no_def=xxx      #未定义的选项
```

那么下面的两种读取文件方式将获得同样的解析结果：

```
ifstream ifs("config.txt");  
store(parse_config_file(ifs, opts, true), vm);       //解析  
char *str = "config.ini";  
store(parse_config_file<char>(str, opts, true), vm); //解析
```

请读者注意上面代码最后一行的 `parse_config_file<char>()` 用法，在直接使用文件名的方式读取配置文件时，我们必须用显式地指明模板参数 `char`，以告诉编译器要使用那个模板函数，否则无论是 VC8 还是 GCC3.4.6 都会报模板推导的编译错误——这可能是编译器的一个 bug。

分析器组件中还有两个工具：

- `command_line_parser` 类，它被 `parse_command_line()` 函数在内部调用，可以提供一些扩展功能：

- 以及一个用于分析环境变量的函数 `parse_environment()`。

我们将在稍后介绍这两个工具。

10.4.7 存储器

存储器是 `program_options` 库里最后一个组件，它接受分析器的结果，用 `store()` 函数存储到 `variables_map` 对象，再调用 `notify()` 把值更新到用户指定的变量。

存储器的核心类是 `variables_map`，它是 `std::map` 的子类，具有 `std::map` 的所有功能，例如 `operator[]`、`count()`、`find()` 等等。另外，它还提供了一个专门的成员函数 `notify()`，用于把选项值更新到 `value_semantic` 关联的外部变量。之前我们看到的自由函数 `notify()` 是成员函数 `notify()` 的一个等价操作。

`variables_map` 的键类型是 `string`，它存放了选项描述器 `option_description` 定义的选项名。在使用 `variables_map` 时我们只能使用选项的长名，使用短名索引会发生运行时异常。

`variables_map` 的值类型是 `any` 对象的一个包装类 `variable_value`，它的成员函数 `as<T>()` 包装了 `any_cast<T>()`，取出 `any` 内存放的选项值。因此 `as<T>()` 函数的模板类型必须与存储的类型相符，否则会抛出 `bad_any_cast` 异常。

`store()` 函数接受分析器的输出 `basic_parsed_options`，逐个遍历分析结果，将选项值存储到 `variables_map` 中。`store()` 函数允许多次调用，实现同时从不同的数据源获得选项信息，比如同时使用命令行和配置文件，已经有值的选项如果再进行 `store()` 会自动忽略，除非它可以接受多个记号。例如：

```
variables_map vm;                                //选项存储 map 容器
store(parse_command_line(argc, argv, opts), vm); //解析命令行
store(parse_config_file(ifs, opts, true), vm);    //解析配置文件
```

10.4.8 使用位置选项值

我们的查找文件实用程序现在已经基本可用了，但还有点小问题。要查找的文件名必须使用 `--filename` 或者 `-f` 来指定，这显得很麻烦，最好是不用选项名，直接在命令行中写出，像这样：

```
find_file a.cpp
```

`program_options` 库将这种没有选项名的选项称为位置选项，它的描述要使用 `positional_options_description` 类来辅助。

`positional_options_description` 类摘要如下：

```
class positional_options_description
{
public:
    positional_options_description();
    positional_options_description & add(const char *name, int count) ;
};
```

`positional_options_description` 类的主要功能集中在 `add()` 函数，它指定选项的名字 `name` 和出现的个数 `count`。`add()` 函数返回对象自身的引用，因此可用连续调用添加选项。当 `count== -1` 时表明位置选项的设置结束，不能再添加位置选项。

位置选项的解析较为复杂，我们不能再使用 `parse_command_line()` 函数，而要使用 `command_line_parser` 类。它的用法与 `parse_command_line()` 函数类似，接受命令行参数，然后调用 `options()` 传入 `options_description` 对象，最后用 `run()` 进行解析。

`command_line_parser` 的类摘要如下：

```
class command_line_parser
{
public:
    command_line_parser(const vector< string > &);
    command_line_parser(int, char *);

    basic_parsed_options< charT > run() ;

    command_line_parser & options(const options_description &) ;
    command_line_parser &
    positional(const positional_options_description &) ;
    command_line_parser & style(int) ;
    command_line_parser & extra_parser(ext_parser) ;
    command_line_parser & allow_unregistered() ;
    command_line_parser & extra_style_parser(style_parser) ;
};
```

`command_line_parser` 类中除了 `run()` 和构造函数，其他的成员函数都用于设置分析选项，并返回它自身的引用，以便串联设置。`command_line_parser` 类可以定制很多分析设置，但我们目前只用到它的位置选项功能 `positional()`，它接受一个 `positional_options_description` 对象以设置位置选项信息。

下面我们使用 `positional_options_description` 和 `command_line_parser` 来完成位置选项的功能。原有的选项定义无需更改，只需要增加位置的定义，并使用 `command_line_parser` 来解析：

```
...//之前的 options_description 定义不变
positional_options_description pod;                //位置选项
```

```

pod.add("filename", 1);           //查找文件名仅出现一个
BOOST_AUTO(pr,                   //使用 BOOST_AUTO 保存解析结果
  command_line_parser(argc, argv). //构造一个解析对象
  options(opts).                  //传入选项描述
  positional(pod).                //传入位置描述
  run());                         //设置完成,开始解析
store(pr, vm);                   //保存解析结果到 vm 中

```

下面是程序的运行结果:

```

D:\>demopo.exe abc.txt
find opt:abc.txt

```

我们也可以使用 `positional_options_description` 定义多个位置信息。例如,假设我们在文件名后允许出现两个查找路径和一个查找深度,那么代码可以这样写:

```

pod.add("filename", 1).add("dir", 2).add("depth", -1);

```

程序的运行结果是:

```

D:\>demopo.exe abc.txt ddd eee 7
find opt:abc.txt
dir opt:ddd,eee,
depth opt:7
D:\>demopo.exe abc.txt ddd 7
find opt:abc.txt
dir opt:ddd,7,

```

请读者注意程序的第二次运行结果, `program_options` 库忠实地执行了程序员的设定,取第二个和第三个参数作为搜索路径。位置选项不支持缺省或者默认的位置,一旦指定了位置的个数,我们就必须在命令行上顺序提供所有的参数。因此,如果在使用位置选项时请谨慎设置 `count` 参数,除非必要,都应该是 1。

10.4.9 分析环境变量

分析器组件中的 `parse_environment()` 函数可以从环境变量中提取信息,这个功能对于很多程序都是非常有用的。

首先我们要确定使用操作系统中的那些环境变量,确定它们的名字,然后在我们的程序中为它们定义选项描述。

假设我们要获取 Windows 操作系统里的 `TMP` 和 `USERNAME` 这两个环境变量,为了方便易记,我们采用 `tmp` 和 `uname` 作为选项名,这两个选项既可以从环境变量中获取也可以从命令行参数中获取。增加选项描述的代码如下:

```
("tmp", value<string>(), "tmp var")
("uname", value<string>(), "user's name")
```

为了把环境变量名转换到定义的选项名，我们需要自定义一个名字转换函数 `name_mapper()`，它把环境变量名映射成选项名，如果环境变量不对应选项，则应该返回空字符串，表示忽略这个环境变量。

为了方便转换，我们使用 `std::map` 和 `assign` 库来实现 `name_mapper()`：

```
#include <boost/assign.hpp>
string name_mapper(const string& env_name)
{
    using namespace boost::assign;
    static map<string, string> nm = //静态变量
        map_list_of("TMP", "tmp") ("USERNAME", "uname");

    return nm[env_name];
}
```

`parse_environment()` 函数有很多重载形式，但我们只需要使用它的一个最基本的形式：接受 `options_description` 对象和一个名字转换函数，声明如下：

```
parsed_options parse_environment(const options_description &, name_mapper);
```

因此，我们可以像分析命令行一样在调用 `parse_environment()` 函数后用 `store()` 函数把环境变量存储到 `variables_map` 对象中：

```
store(parse_environment(opts, name_mapper), vm);
cout << vm["tmp"].as<string> () << endl;
cout << vm["uname"].as<string> () << endl;
```

程序的运行结果如下（假设已经有了那些环境变量）：

```
D:\>demopo.exe
C:\xxx\Temp
xxx_username
```

10.4.10 分组选项信息

有很多理由支持把选项信息分组。例如选项非常多，把它们按照一定的类别组织起来方便阅读；或者将选项分成初级用户和高级用户两类，对初级用户隐藏高级选项；位置选项也应该单独分组，它在 `help` 的时候不应该作为选项显示出来。

选项信息分组主要使用 `options_description` 类。我们可以依据分组设计定义多个 `options_description` 对象，然后用它的 `add()` 方法再组合起来，这样就可以在显示帮助时

使用一个 options_description 对象，而在分析时使用另一个 options_description 对象。

作为示范，我们将查找文件程序的选项分为三组：第一组是 help，只能用于命令行选项；第二组是位置选项 filename，它可以用于命令行和配置文件，但不应该被显示在帮助中；第三组是 dir 和 depth，它们既可以用于命令行也可以用于配置文件。因此，我们需要定义三个 options_description 对象：

```
options_description opts1("group 1");           //第一组
opts1.add_options()                             //帮助选项
    ("help,h", "help message");
options_description opts2("group 2(hidden)");    //第二组,可以无名,因为不会
                                                //显示给用户看
opts2.add_options()                             //文件名选项
    ("filename,f", value<string>(), "to find a file");
options_description opts3("group 3");           //第三组
opts3.add_options()                             //搜索路径和深度
    ("dir,D", value<vector<string>>()->composing(), "search dir")
    ("depth,d", value<int>(), "search depth");
```

为了清晰起见，这里我们省略了一些选项之前的设置。还要注意在定义搜索路径选项时我们使用了 composing() 函数，它将指示分析器把来自不同数据源的选项值组合起来。

接下来我们把这三组选项组合起来，分别用于解析命令行、解析配置文件和显示：

```
options_description opts_all;                   //解析命令行，不需要名字
opts_all.add(opts1).add(opts2).add(opts3);
options_description opts_cfgfile;               //解析配置文件，不需要名字
opts_cfgfile.add(opts2).add(opts3);
options_description opts_showhelp("demo options"); //显示
opts_showhelp.add(opts1).add(opts3);
```

在解析选项时我们需要调用两次分析器和存储器，因为它要从命令行和配置文件两个源中获取数据：

```
store(parse_command_line(argc, argv, opts_all), vm); //解析
store(parse_config_file<char>("config.ini", opts_cfgfile), vm); //解析
```

当显示帮助信息时，我们使用专门用于显示的 options_description 对象：

```
if (vm.count("help"))                           //处理帮助选项
{
    cout << opts_showhelp << endl;             //输出帮助信息
}
```

这样，我们就完成了选项信息分组的全部工作，运行结果如下：

```
D:\>demopo.exe -h
```

```
demo options:
group 1:
  -h [ --help ]      help message
group 3:
  -D [ --dir ] arg    search dir
  -d [ --depth ] arg  search depth
D:\>demopo.exe -f abc.txt -d 5
find opt:abc.txt
depth opt:5
```

10.4.11 高级用法

分析器中的 `command_line_parser` 类是一个强大的选项解析器，有很多高级功能，之前我们已经介绍了位置选项的用法，下面再来看看它的其他用法。

命令行风格

`program_options` 库使用的程序选项默认是 UNIX 风格，使用双连字符和单连字符指定选项名，但如果用户喜欢，它也可以改为使用 Windows 风格的斜杠。

`command_line_parser` 的 `style()` 函数可以改变分析器的行为，它接受一个 `command_line_style::style_t` 的枚举值，缺省是 `unix_style`。如果增加 `allow_slash_for_short` 值，将可以在短名选项前使用斜杠。例如：

```
using namespace boost::program_options::command_line_style;
BOOST_AUTO(pr,
  command_line_parser(argc, argv).
  options(opts).
  style(unix_style | allow_slash_for_short).           //变更风格
  run());
store(pr, vm );
```

程序运行结果如下：

```
D:\ >demopo.exe /f abc.txt
find opt:abc.txt
```

未定义选项

`program_options` 库的默认行为是不允许未定义选项，如果命令行中出现了未在 `options_description` 中定义的选项名则会抛出异常。但这通常不是一个好的选择，它迫使我们必须用 `try-catch` 块来保护选项分析代码。

`command_line_parser` 的成员函数 `allow_unregistered()` 允许我们改变这个行为，类似在 `parse_config_file()` 函数中置 `allow_unregistered` 为 `true`。

当使用了 `allow_unregistered()` 函数后, `command_line_parser` 遇到未定义但又“像”是选项的命令行参数时, 会把它保存在分析结果 `parsed_options` 对象中, 而不是抛出异常。这些无法识别的参数可以使用 `collect_unrecognized()` 获得。例如:

```
BOOST_AUTO(pr,
    command_line_parser(argc, argv).
    options(opts).allow_unregistered().           //允许未定义选项
    run());
vector<string> ur_opts =                          //使用 vector<string>接收结果
    collect_unrecognized(pr.options, include_positional);
BOOST_FOREACH(string str, ur_opts)
{
    cout << str << endl;
}
```

程序运行结果如下:

```
D:\>demopo.exe -f abc.txt -o unknown -c a.
cpp
-o
unknown
-c
a.cpp
find opt:abc.txt
```

分析 Winmain 命令行

在 Windows 环境下的图形界面程序有时候也需要使用命令行, 用于在启动时指定某些行为。但 `winmain()` 函数没有如 `main()` 那样的 `argc`、`argv` 参数, 不能分解参数, 为了支持 Windows 的这种用法, `program_options` 库提供了一个函数 `spilt_winmain()`, 用于把未分解的字符串拆分成 `vector<string>`, 然后用 `command_line_parser` 的另一个构造函数接收再进行解析。

`spilt_winmain()` 的用法如下:

```
string wincmd = "--filename=a.cpp --dir=/usr/bin --depth=2";
vector<string> args = split_winmain(wincmd);
BOOST_AUTO(pr,
    command_line_parser(args).
    options(opts).run());
```

附加分析器

`command_line_parser` 可以使用成员函数 `extra_parser()` 接受一个附加的选项分析器 `ext_parser`, 用来处理特别格式的选项名。`ext_parser` 是一个函数, 它的输入是命令行参数, 输出是一个 `pair<string, string>`, 存储解析出的选项名和选项值。

假设我们为查找文件程序增加一个大小写无关选项 `-cy` 和 `-cn`, 在解析时把它转化成 `true` 和 `false`, 那么, 附加分析器可以是:

```

pair<string, string> extra_parser(const string& s)
{
    if (s.find("-c") == 0)
    {
        if (s.substr(2, 1) == "y")
            return make_pair("case", string("false"));
        else
            return make_pair("case", string("true"));
    }
    else
    {
        return pair<string, string>(); //返回空 pair 表示不处理
    }
}

```

extra_parser() 函数在遇到 -c 选项时, 把选项名转换成了 case, 选项值转换成了 true/false, 因此我们还需要增加 case 选项, 以支持这个附加分析器:

```
("case", value<string>(), "case switch")
```

最后, 我们使用 command_line_parser 来使用这个附加分析器:

```

BOOST_AUTO(pr,
    command_line_parser(argc, argv).
    options(opts).extra_parser(::extra_parser).
    run());
cout << vm["case"].as<string>(); //测试分析结果

```

程序的运行结果如下:

```

D:\>demopo.exe -cy
false

```

响应文件

有的操作系统支持响应文件 (Response File) 的概念, 它不同于配置文件, 里面的内容是与命令行相同的语法选项, 通常响应文件选项的标准格式是 @file。

program_options 库没有直接提供对响应文件的支持, 但可以手工实现。首先我们要定义一个附加分析器, 以解析出响应文件选项:

```

("response", value<string>(), "@file") //响应文件选项定义
...
pair<string, string> extra_parser(const string& s)
{
    if (s[0] == '@')
    { return make_pair("response", s.substr(1)); }
    else

```



```
{ return pair<string, string>(); }
}
```

然后我们使用 `command_line_parser` 和附加分析器像通常一样处理，在获得响应文件名后再进一步解析。用文件流 `ifstream` 读入文件内容，然后再用 `spilt_winmain()` 分解参数解析：

```
if (vm.count("response")) //处理响应文件选项
{
    //我们必须用 string 的 c_str() 方法，原因见 10.3.14 小节
    ifstream ifs(vm["response"].as<string>().c_str());
    stringstream ss;
    ss << ifs.rdbuf(); //读入文件内容
    vector<string> args = split_winmain(ss.str()); //解析
    BOOST_AUTO(pr,
        command_line_parser(args).
        options(opts).
        run());
    store(pr, vm ); //存储响应文件中的选项值
}
```

如果响应文件的内容是：

```
--filename=a.cpp
--dir=/usr/bin
--depth=2
```

那么程序的运行结果是：

```
D:\>demopo.exe @response.txt
find opt:a.cpp
dir opt:/usr/bin,
depth opt:2
```

10.5 总结

本章讨论了 Boost 库在操作系统方面提供的一些工具，它们有助于程序员编写跨平台的操作系统功能调用程序。

`io_state_savers` 是一个配合 IO 流的辅助类，它可以保存 IO 流的状态，在需要时候恢复，防止某些意外操作损坏 IO 流导致的未定义行为。它的用法非常简单，只需要在改变流前用保存器保存一下流的状态，之后就可以自动恢复。

`system` 库封装了操作系统的底层错误代码，并提供 `error_code` 和 `system_error` 两种

方式供用户使用。基于 `system` 库可以开发出充分利用操作系统的能力同时又具有高可移植性的软件，`filesystem` 库就是很好的例子。

`filesystem` 库是文件系统的抽象，它提供了兼容各种文件系统的路径表示，并提供了大量的文件、目录操作，对于主流的 Windows、UNIX 都支持得非常好。使用 `filesystem` 库，我们可以写出类似 Python、Perl 的 C++ 脚本程序，做到一切开发都不离开 C++ 环境。

由于 `system` 和 `filesystem` 库提供了 C++ 所急需的操作系统相关功能，它们已经被收入 C++0x TR2，不久的将来将成为新标准的一部分。

最后我们讨论了 `program_options` 库，它提供了从外界获取程序选项的能力。程序的选项可以来源自很多地方，最常见的就是命令行参数，此外 `program_options` 库还可以从配置文件、响应文件和环境变量中获得选项，因此可以适应将来维护期的变化。

`program_options` 库使用了非常灵活的机制来实现从多个数据源中获取数据，选项描述器、分析器和存储器三者都经过了精心的设计，而且被有机地组合起来，只需要编写少量的代码，我们的程序就可以获得非常漂亮且功能完善的配置选项解析功能。

本章实现了几个很有用的函数，如下：

- `find_file()` 和 `find_files()`：它们使用 `filesystem` 库的迭代目录功能提供了深度遍历文件系统查找文件的功能，是对 `filesystem` 库强大功能的一个很好的演示；
- `copy_files()`：它基于 `find_files()` 提供了深层次的目录拷贝，就像是“古老”的 DOS 命令 `xcopy`，它还使用 `progress_display` 提供了友好的界面显示。

第 11 章

函数与回调

本章的讨论主题围绕着函数与函数对象，总共有五个 Boost 库。它们是 Boost 中较为高级复杂的组件，应用了 C++ 中高级的语言特性和技巧，功能很强大，相应的用法也比较复杂、灵活，不太容易理解和掌握。

首先我们将学习两个小的工具类 `result_of` 和 `ref`，它们是本章其他库的基础。`result_of` 使用了复杂的技巧来自动推导函数的返回值类型，`ref` 可以包装对象的引用，在传递参数时消除对象拷贝的代价，或者将不可拷贝的对象变为可以拷贝。

`bind` 是 C++98 标准库中函数适配器的增强，可以适配任意的可调用对象，包括函数指针、函数引用和函数对象，把它们变成一个新的函数对象，它是迈向 C++ 函数式编程的第一步。`function` 库则是对 C/C++ 中函数指针类型的增强，它能够容纳任意的可调用对象，可以配合 `bind` 使用。

最后我们讨论 `signals2` 库，它实现了威力强大的观察者模式，如果读者曾经用过 C# 的 `event/delegate` 或者 Java 的 `Observable/Observer`，那么就会知道 `signals2` 对于 C++ 程序员的重要意义。

11.1 result_of

`result_of` 是一个很小但很有用的组件，可以帮助程序员确定一个调用表达式的返回类型，主要用于泛型编程和其他 Boost 库组件，它已被收入 TR1。

`result_of` 位于名字空间 `boost`，为了使用 `result_of` 组件，需要包含头文件 `<boost/utility/result_of.hpp>`，即：

```
#include <boost/utility/result_of.hpp>
using namespace boost;
```



11.1.1 原理

所谓“调用表达式”，是指一个含有 `operator()` 的表达式，函数调用或函数对象调用都可以称为调用表达式，而 `result_of` 可以确定这个表达式所返回的类型。

单从这一点来理解，`result_of` 的功能有些类似 `typeof` 库（4.2 小节，97 页）。`typeof` 可以确定一个表达式的类型，但它不具备推演调用表达式的能力。

类摘要

`result_of` 库含有一个 `result_of` 类型，它使用了模板元编程技术，声明的基本形式是：

```
template<typename ...>
struct result_of
{
    typedef ??? type;
};
```

给定一个调用表达式，可以通过内部类型定义 `result_of<...>::type` 获得返回值的类型。

假设我们有一个类型 `Func`，它可以是函数指针、函数引用或者成员函数指针，当然也可以是函数对象类型，它的一个实例是 `func`。`Func` 有一个 `operator()`，参数是 `(T1 t1, T2 t2)`，这里 `T1`、`T2` 是两个模板类型，那么

```
result_of<Func(T1,T2)>::type①
```

就是 `func(t1, t2)` 的返回值类型。

实例阐述

刚才的解释好像有些抽象难以理解，尤其是对泛型编程不熟悉的读者，我们用实例解说一下。

```
typedef double (*Func)(double d);
```

这行代码定义了一个函数指针类型 `Func`，它的调用式接受一个 `double` 类型，返回类型为 `double`。

```
Func func= sqrt;
```

这行代码声明了 `Func` 的一个实例（变量）`func`，一个具体的函数指针，并把它赋值为 `sqrt`

① 这种在模板中使用类似函数声明的形式还会在本书讲述 `function` 库时遇到，

——C 标准库中的开平方数学函数。那么

```
result_of<Func(double)>::type x = func(5.0);
```

这行代码必然可以正确通过编译，x 的类型将被推导为 double。

完整的程序如下：

```
#include <boost/utility/result_of.hpp>
using namespace boost;
int main()
{
    typedef double (*Func)(double d);
    Func func= sqrt;
    result_of<Func(double)>::type x = func(5.0);
    cout << typeid(x).name();
}
```

与 typeid 库的区别

上面的实例演示了 result_of 的基本用法，不是很具有吸引力，用 BOOST_AUTO 也可以完成同样的功能。但是，当处于一个泛型上下文之中，周围没有真实的类型，而且没有表达式的时候，BOOST_AUTO 就无能为力了，例如这样的一个简单的泛型函数：

```
template<typename T, typename T1>
??? call_func(T t, T1 t1)                //T 是个可调用的类型
{ return t(t1);}
```

无论如何，BOOST_AUTO 都派不上用场，这里不存在任何赋值表达式，只有函数调用式。

这正是 result_of 发挥威力的机会，它可以正确推导出返回类型，像这样：

```
template<typename T, typename T1>
typename result_of<T(T1)>::type call_func(T t, T1 t1)
{ return t(t1);}
```

这里必须在 result_of<>::type 前加上关键字 typename，否则编译器会认为 type 是 result_of 的成员变量，从而产生找不到声明的编译错误。

仍然使用刚才定义的函数指针，使用 result_of 的完整程序如下：

```
#include <boost/utility/result_of.hpp>
using namespace boost;
template<typename T, typename T1>
typename result_of<T(T1)>::type call_func(T t, T1 t1)
{ return t(t1);}
int main()
{
```

```
typedef double (*Func)(double d);
Func func= sqrt;
BOOST_AUTO( x, call_func(func, 5.0)); //赋值表达式, 可以用 typeid
cout << typeid(x).name();
}
```

11.1.2 用法

`result_of` 虽然小, 但它用到了很多 C++ 的高级特性, 如模板偏特化和 SFINAE, 并且部分依赖于编译器的能力。它不仅可以用于函数指针, 更重要的是用于函数对象进行泛型编程。

设类型 `Func` 可被调用 (具有 `operator()`), `func` 是 `Func` 的一个左值, 那么:

```
typeid(result_of<Func(T1,T2,...,TN)>::type)
```

必然等于

```
typeid(func(t1, t2, ..., tN)).
```

在 `result_of` 应用于函数对象时, 推导会更复杂一些, 例如:

- 如果 `Func` 有一个内部类型定义 (typedef) `result_type`, 那么 `typeid(result_of<Func(...)>::type) == typeid(Func::result_type);`
- 如果 `Func` 没有定义 `result_type` 类型, 那么如果是无参调用, `typeid(result_of<Func()>::type) == typeid(void)`。否则 `typeid(result_of<Func(...)>::type) == typeid(Func::result_of<Func(...)>::type);`
- 如果类型 `Func` 内部没有嵌套的 `result_of<Func(...)>::type` 定义, 那么 `result_of` 的类型推导将会失败, 报出编译错误。

如果读者对 `result_of` 的用法还不甚理解也不要紧, `result_of` 是底层编程技术的基本构件, 用于实现泛型库, 应用编程里用到的机会不多, 而且在本书仅用于 `ref` 库。

11.2.6 小节 (427 页) 示范了它用于函数对象的用法, 这里不再举例。

11.2 ref

STL 和 Boost 中的算法和函数大量使用了函数对象作为判断式或谓词参数, 而这些参数都是传值语义, 算法或函数在内部保留函数对象的拷贝并使用, 例如:

```
#include <boost/assign.hpp>
using namespace boost;
```

```
using namespace boost::assign;

int main()
{
    struct square                                //函数对象，计算整数的平方
    {
        typedef void result_type;                //返回结果的类型定义，很重要
        void operator()(int &x)
        {    x = x * x;    }
    };

    vector<int> v = (list_of(1),2,3,4,5);
    for_each(v.begin(), v.end(), square());

    //输出: '1', '4', '9', '16', '25'
}
```

一般情况下传值语义都是可行的，但也有很多特殊情况，作为参数的函数对象拷贝代价过高（具有复杂的内部状态），或者不希望拷贝对象（内部状态不应该被改变），甚至拷贝是不可行的（noncopyable、单件）。

boost.ref 应用代理模式，引入对象引用的包装器概念解决了这个问题^①。它位于名字空间 boost，为了使用 ref 组件，需要包含头文件<boost/ref.hpp>，即：

```
#include <boost/ref.hpp>
using namespace boost;
```

11.2.1 类摘要

ref 库定义了一个很小很简单的引用类型的包装器，名字叫 reference_wrapper，它的类摘要如下：

```
template<class T> class reference_wrapper
{
public:
    explicit reference_wrapper(T& t): t_(&t) {}
    operator T& () const { return *t_; }

    T& get() const { return *t_; }
    T* get_pointer() const { return t_; }
private:
    T* t_;
};
```

① 其实早在 9.4.8 小节（365 页）我们就已经运用类似的方法来解决这个问题了。

`reference_wrapper` 的构造函数接受类型 `T` 的引用类型, 内部使用指针存储指向 `t` 的引用, 构造出一个 `reference_wrapper` 对象, 包装了引用。 `get()` 和 `get_pointer()` 这两个函数分别返回存储的引用和指针, 相当于解开对 `t` 的包装。

请注意, `reference_wrapper` 的构造函数被声明为 `explicit`, 因此必须在创建 `reference_wrapper` 对象时就赋值初始化, 就像是使用一个引用类型的变量。

`reference_wrapper` 还支持隐式类型转换, 可以在需要的语境下返回存储的引用, 因此它很像引用类型, 能够在任何需要 `T` 出现的地方使用 `reference_wrapper`。

11.2.2 基本用法

`reference_wrapper` 的用法有些类似 C++ 中的引用类型 (`T&`), 就像是被包装对象的一个别名。但它只有在使用 `T` 的语境下才能够执行隐性转换, 其他的情况下则需要调用类型转换函数或者 `get()` 函数才能真正操作被包装对象。

此外, `reference_wrapper` 支持拷贝构造和赋值, 而引用类型是不可赋值的。

示范 `reference_wrapper` 简单用法的代码如下:

```
#include <boost/ref.hpp>
using namespace boost;
int main()
{
    int x = 10;
    reference_wrapper<int> rw(x);                //包装 int 类型的引用
    assert(x == rw);                             //隐式转换为 int 类型
    (int &)rw = 100;                             //显式转换为 int&类型, 用于左值
    assert(x == 100);

    reference_wrapper<int> rw2(rw);               //拷贝构造
    assert(rw2.get() == 100);

    string str;
    reference_wrapper<string> rws(str);           //包装字符串的引用
    *rws.get_pointer() = "test reference_wrapper"; //指针操作
    cout << rws.get().size() << endl;
}
```

请读者注意代码中最后一行的 `cout` 输出语句, 这里不存在一个要求 `string` 类型的隐式转换, 所以我们必须要使用 `get()` 获得被包装的真正对象, 然后调用它的方法。如果写出如下的语句:

```
cout << rws.size() << endl;
```


将引发一个编译错误: `reference_wrapper` 不存在 `size()` 成员函数。

因此, `reference_wrapper` 是一个很像引用的对象, 但它与真正的引用无论在用法还是用途上都存在差异。

11.2.3 工厂函数

`reference_wrapper` 的名字过长, 声明引用包装对象很不方便, 因而 `ref` 库提供了两个便捷的工厂函数 `ref()` 和 `cref()`, 可以通过参数类型推导很容易地构造 `reference_wrapper` 对象。

这两个函数的声明如下:

```
reference_wrapper<T> ref(T& t);  
reference_wrapper<T const> cref(T const& t);
```

`ref()` 和 `cref()` 会根据参数类型自动地推导生成正确的 `reference_wrapper<T>` 对象, `ref()` 产生的类型是 `T`, 而 `cref()` 产生的类型是 `T const`。例如:

```
double x = 2.71828;  
BOOST_AUTO(rw, cref(x)); //包装 double const  
cout << typeid(rw).name() << endl;  
  
string str;  
BOOST_AUTO(rws, ref(str)); //包装字符串  
cout << typeid(rws).name() << endl;
```

因为 `reference_wrapper` 支持拷贝, 因此 `ref()` 和 `cref()` 可以直接用在需要拷贝语义的函数参数中, 而不必专门使用一个 `reference_wrapper` 来暂存, 例如:

```
double x = 2.0;  
cout << std::sqrt(ref(x)) << endl; //计算平方根
```

11.2.4 操作包装

`ref` 库运用模板元编程技术提供两个特征类 `is_reference_wrapper` 和 `unwrap_reference`, 用于检测 `reference_wrapper` 对象:

- `is_reference_wrapper<T>` 的 `bool` 成员变量 `value` 可以判断 `T` 是否为一个 `reference_wrapper`;
- `unwrap_reference<T>` 的内部类型定义 `type` 表明了 `T` 的真实类型, 无论它是否经过 `reference_wrapper` 包装。

示范这两个特征类用法的代码如下：

```
vector<int> v(10, 2) ;
BOOST_AUTO(rw, cref(v));
assert(is_reference_wrapper<BOOST_TYPEOF(rw)>::value);
assert(!is_reference_wrapper<BOOST_TYPEOF(v)>::value;

string str;
BOOST_AUTO(rws, ref(str));
cout << typeid(unwrap_reference<BOOST_TYPEOF(rws)>::type).name() << endl;
cout << typeid(unwrap_reference<BOOST_TYPEOF(str)>::type).name() << endl;
```

自由函数 `unwrap_ref()` 为解开包装提供了简便的方法，它利用 `unwrap_reference<T>` 直接解开 `reference_wrapper` 的包装（如果有的话），返回被包装对象的引用。例如：

```
set<int> s ;
BOOST_AUTO(rw, ref(s));           //获得一个包装对象
unwrap_ref(rw).insert(12);        //直接解开包装

string str("test");
BOOST_AUTO(rws, cref(str));       //获得一个常对象的包装
cout << unwrap_ref(rws) << endl;  //解包装
```

直接对一个未包装的对象使用 `unwrap_ref()` 也是允许的，它将直接返回对象自身的引用：

```
cout << unwrap_ref(str) << endl;    //对未包装对象解包装
```

`unwrap_ref()` 的这个功能很有用，可以把 `unwrap_ref()` 安全地用在泛型代码中，从而不必关心对象的包装特性，总能够正确地操作对象。

11.2.5 综合应用

本小节我们给出 `ref` 库的一个综合运用的例子。

假设我们有一个很大的类 `big_class`，它具有复杂的内部状态，构造、拷贝都具有很高的代价：

```
class big_class           //一个简化的例子
{
private:
    int x;                //很复杂的内部状态,这里从简
public:
    big_class():x(0){}    //构造函数
    void print()          //一个操作函数,改变内部状态
    { cout << "big_class " << ++x << endl; }
};
```

模板函数 `print()` 接受任意类型的参数，调用它们的成员函数 `print()`。考虑到类型可能

会使用 `reference_wrapper` 包装，它使用 `unwrap_ref()` 函数：

```
template<typename T>
void print(T a)
{
    for (int i = 0; i < 2; ++i)
        unwrap_ref(a).print();           //解包装
}
```

最后，我们在 `main()` 调用模板函数 `print`：

```
int main()
{
    big_class c;
    BOOST_AUTO(rw, ref(c));
    c.print();                           //输出 1

    print(c);                            //拷贝传参，输出 2,3，内部状态不改变
    print(rw);                           //引用传参，输出 2,3，内部状态改变
    print(c);                            //拷贝传参，输出 4,5，内部状态不改变
    c.print();                           //输出 4
}
```

这段代码演示了拷贝传参和引用传参的不同。当调用 `print(c)` 时是拷贝传参，因此 `c` 在函数中被复制，它内部状态的变化不影响原对象，在函数调用完成后 `c` 的内部值仍然为 1。但调用 `print(rw)` 时由于使用了 `reference_wrapper` 包装，函数拷贝的是 `reference_wrapper` 对象，在函数内部被解包装为原对象的引用，因此改变了原对象的内部状态。

11.2.6 为 ref 增加函数调用功能

`ref` 将对象包装为引用语义，降低了复制的代价，使引用的行为更像对象（因为对象更有用更强大），可以让容器安全地持有被包装的引用对象，可以被称为是“智能引用”。因此它被收入了 C++0x 的 TR1 草案，成为了 C++ 新标准的一部分。

但很遗憾的是 `boost.ref` 库没有实现 TR1 的全部定义，尤其是不提供函数调用操作 `operator()`，这使得我们无法包装一个函数对象的引用并传递给标准库算法，而实际上这并不是一个太困难的事情。

9.4.8 小节（365 页）实现了一个类似的引用包装器，提供了 `operator()`。而在这里，我们将运用 `boost.result_of` 库为 `ref` 增加函数调用功能。`result_of` 库可以帮助确定一个调用表达式的类型，它也被收入了 C++ 新标准 TR1 库（参见 11.1 小节，419 页）。

在这之前，提醒读者先对 `<boost/ref.hpp>` 做一个备份，毕竟擅自修改 Boost 库的实现是

有一定风险的。

首先要在 `ref.hpp` 增加 `result_of` 的引用，在文件里添加包含语句：

```
#include <boost/utility/result_of.hpp>
```

随后我们在 `reference_wrapper` 类内部实现 `operator()` 函数：

```
//add operator() [2010 luojianfeng]
typename result_of<T()>::type operator() () const
{ return (*t_)(); }
```

这里我们用 `result_of<T()>::type` 确定了一个无参函数调用的返回类型，还需要在前面加上关键字 `typename`，让编译器知道 `type` 是一个类型而不是成员变量。另外 `operator()` 必须是 `const` 的，因为它不变动 `reference_wrapper` 类的状态。

使用成员模板函数重载，同样地我们可以实现带 `N` 个参数的函数调用：

```
template<typename T0>
typename result_of<T(T0)>::type operator()(T0 t0) const
{ return (*t_)(t0); }

template<typename T0, typename T1>
typename result_of<T(T0,T1)>::type operator()(T0 t0, T1 t1) const
{ return (*t_)(t0, t2); }

... //更多参数数量的重载形式
```

这样我们就完成了为 `ref` 增加函数调用的功能。但需要注意，函数调用依赖的是 `result_of` 的功能，因此 `reference_wrapper` 包装的对象类型可以是函数指针、函数引用、成员函数指针或函数对象。对函数对象有特别的要求，简单来说，其内部必须有 `typedef result_type`，用来定义返回值类型，否则无法推导。

下面的代码使用 `sqrt()` 函数和在 11.2 节（422 页）定义的 `square` 函数对象，示范了 `reference_wrapper` 的函数调用用法：

```
#include <boost/ref.hpp>
#include <boost/assign.hpp>
using namespace boost;
using namespace boost::assign;

struct square //函数对象，计算整数的平方
{
    typedef void result_type; //返回结果的类型定义，很重要
    void operator()(int& x)
    { x = x*x; }
};

int main()
```

```

{
    typedef double (*pfunc) (double);
    pfunc pf = sqrt;
    cout << ref(pf) (5.0);           //包装函数指针

    square sq;
    int x =5;
    ref(sq) (x);                     //包装函数对象
    cout << x;

    vector<int> v = (list_of(1),2,3,4,5);
    for_each(v.begin(), v.end(), ref(sq)); //传递给算法引用
}

```

11.3 bind

bind 是 C++98 标准库中函数适配器 bind1st/bind2nd 的泛化和增强, 可以适配任意的可调用对象, 包括函数指针、函数引用、成员函数指针和函数对象。bind 远远地超越了 STL 中的函数绑定器 (bind1st/bind2nd), 可以绑定最多 9 个函数参数, 而且对被绑定对象的要求很低, 可以在没有 result_type 内部类型定义的情况下完成对函数对象的绑定。bind 库很好地增强了标准库的功能, 它已经被收入 C++0x TR1 草案。

bind 位于名字空间 boost, 为了使用 bind 组件, 需要包含头文件 <boost/bind.hpp>, 即:

```

#include <boost/bind.hpp>
using namespace boost;

```

11.3.1 工作原理

bind 并不是一个单独的类或函数, 而是非常庞大的家族, 依据绑定的参数个数和要绑定的调用对象类型, 总共有数十个不同的形式, 但它们的名字都叫做 bind, 编译器会根据具体的绑定代码自动确定要使用的正确形式, bind 的基本形式如下:

```

template<class R, class F> bind(F f);
template<class R, class F, class A1> bind(F f, A1 a1);
namespace                               //匿名名字空间
{
    boost::arg<1> _1;
    boost::arg<2> _2;
    boost::arg<3> _3;
    ...                                 //其他 6 个占位符定义
}

```

bind 接受的第一个参数必须是一个可调用对象 f, 包括函数、函数指针、函数对象和成员函

数指针，之后 `bind` 接受最多九个参数，参数的数量必须与 `f` 的参数数量相等，这些参数将被传递给 `f` 作为入参。

绑定完成后，`bind` 会返回一个函数对象，它内部保存了 `f` 的拷贝，具有 `operator()`，返回值类型被自动推导为 `f` 的返回值类型。在发生调用时，这个函数对象将把之前存储的参数转发给 `f` 完成调用。

例如，如果有一个函数 `func`，它的形式是：

```
func(a1, a2)
```

那么，它将等价于一个具有无参 `operator()` 的 `bind` 函数对象调用：

```
bind(func, a1, a2)()
```

这是 `bind` 最简单的形式。`bind` 表达式存储了 `func` 和 `a1`、`a2` 的拷贝，产生了一个临时函数对象。因为 `func` 接受两个参数，而 `a1` 和 `a2` 都是实参，因此临时函数对象将具有一个无参的 `operator()`。当 `operator()` 调用发生时函数对象把 `a1`、`a2` 的拷贝传递给 `func`，完成真正的函数调用。

`bind` 的真正威力在于它的占位符，它们分别被定义为 `_1`、`_2`、`_3` 一直到 `_9`，位于一个匿名名字空间。占位符可以取代 `bind` 中参数的位置，在发生函数调用时才接受真正的参数。占位符的名字都以下划线开始，然后是一个数字，可能稍微有点奇怪，但它们是完全合法的 C++ 标识符，起这样的名字完全是为了方便书写、识别和记忆。

占位符的名字表示它在调用式中的顺序，而在绑定表达式中没有顺序的要求，`_1` 不一定必须第一个出现，也不一定只出现一次，例如：

```
bind(func, _2, _1)(a1, a2)
```

返回一个具有两个参数的函数对象，第一个参数将放在函数 `func` 的第二个位置，而第二个参数则放在第一个位置，调用时等价于

```
func(a2, a1)
```

接下来我们将通过数个例子来详细了解 `bind` 的用法。

11.3.2 绑定普通函数

`bind` 可以绑定普通函数，包括函数、函数指针，假设我们有如下的函数定义：

```
int f(int a, int b)                //二元函数
{ return a + b; }
int g(int a, int b, int c)         //三元函数
```

```
{ return a + b * c;}
typedef int (*f_type)(int, int);           //函数指针定义
typedef int (*g_type)(int, int, int);      //函数指针定义
```

那么 `bind(f, 1, 2)` 将返回一个无参调用的函数对象，等价于 `f(1, 2)`，`bind(g, 1, 2, 3)` 同样返回返回一个无参调用的函数对象，等价于 `g(1, 2, 3)`。这两个绑定表达式没有使用占位符，而是给出了全部具体参数，代码：

```
cout << bind(f, 1, 2)() << endl;
cout << bind(g, 1, 2, 3)() << endl;
```

相当于：

```
cout << f(1, 2) << endl;
cout << g(1, 2, 3) << endl;
```

输出结果是：

```
3
7
```

使用占位符 `bind` 可以有更多的变化，这才是它真正应该做的工作，下面列出了一些占位符的用法：

```
bind(f, _1, 9)(x) ;           //f(x, 9), 相当于 bind2nd(f, 9)
bind(f, _1, _2)(x, y) ;      //f(x, y)
bind(f, _2, _1)(x, y) ;      //f(y, x)
bind(f, _1, _1)(x, y) ;      //f(x, x), y 参数被忽略
bind(g, _1, 8, _2)(x, y) ;    //g(x, 8, y)
bind(g, _3, _2, _2)(x, y, z) ; //g(z, y, y), x 参数被忽略
```

注意：必须在绑定表达式中提供函数要求的所有参数，无论是真实参数还是占位符均可以。占位符可以出现也可以不出现，出现的顺序和数量也没有限定，但不能使用超过函数参数数量的占位符，比如在绑定 `f` 时不能使用 `_3`，在绑定 `g` 时不能使用 `_4`，也不能写 `bind(f, _1, _2, _2)` 这样的形式，否则会导致编译错误。

`bind` 完全可以代替标准库中的 `bind1st` 和 `bind2nd`，使用 `bind(f, N, _1)` 和 `bind(f, _1, N)`。要注意的是它们均使用了一个占位符，`bind1st` 把第一个参数用固定值代替，`bind2nd` 把第二个参数用固定值代替。

`bind` 也可以绑定函数指针，用法相同，例如：

```
f_type pf = f;
g_type pg = g;
int x = 1, y = 2, z = 3;
cout << bind(pf, _1, 9)(x) << endl;           //(*pf)(x, 9)
```

```
cout << bind(pg, _3, _2, _2)(x, y, z) << endl; //(*pg)(z,y,y)
```

11.3.3 绑定成员函数

bind 也可以绑定类的成员函数。

类的成员函数不同于普通函数，因为成员函数指针不能直接调用 `operator()`，它必须被绑定到一个对象或者指针，然后才能得到 `this` 指针进而调用成员函数。因此 bind 需要“牺牲”一个占位符的位置，要求用户提供一个类的实例、引用或者指针，通过对象作为第一个参数来调用成员函数，即：

```
bind( &X::func, x, _1, _2, ...)
```

这意味着使用成员函数时只能最多绑定八个参数。

例如，我们有一个类 demo：

```
struct demo //使用 struct 仅仅是为了方便，不必写出 public
{
    int f(int a, int b)
    {
        return a + b;
    }
};
```

那么，下面的 bind 表达式都是成立的：

```
demo a, &ra=a; //类的实例对象和引用
demo *p = &a; //指针

cout << bind(&demo::f, a, _1, 20)(10) << endl;
cout << bind(&demo::f, ra, _2, _1)(10, 20) << endl;
cout << bind(&demo::f, p, _1, _2)(10, 20) << endl;
```

注意：我们必须在成员函数前加上取地址操作符 `&`，表明这是一个成员函数指针，否则会无法通过编译，这是与绑定函数的一个小小的不同。

bind 能够绑定成员函数，这是个非常有用的功能，它可以替代标准库中令人迷惑的 `mem_fun` 和 `mem_fun_ref` 绑定器，用来配合标准算法操作容器中的对象。下面的代码使用 bind 搭配标准算法 `for_each` 用来调用容器中所有对象的 `print()` 函数：

```
#include <boost/bind.hpp>
using namespace boost;
struct point //一个二维点的类
{
    int x, y;
    point(int a = 0, int b = 0):x(a),y(b){}
    void print()
    {
        cout << "(" << x << ", " << y << ")\n";
    }
};
```



```
};
int main()
{
    vector<point> v(10);
    for_each(v.begin(), v.end(), bind(&point::print, _1));
}
```

bind 同样支持绑定虚拟成员函数，用法与非虚函数相同，虚函数的行为将由实际调用发生时的实例来决定。

11.3.4 绑定成员变量

bind 的另一个对类的操作是它可以绑定 public 成员变量，就像是一个选择器，用法与绑定成员函数类似，只需要把成员变量名像一个成员函数一样去使用。

仍然以 point 类为例子，假设我们已经在 vector 中存储了大量的 point 对象，而我们要得到它们的 x 坐标值，那么 bind 可以这样使用：

```
vector<point> v(10);
vector<int> v2(10);
transform(v.begin(), v.end(), v2.begin(), bind(&point::x, _1));

BOOST_FOREACH(int x, v2) //foreach 循环输出值
    cout << x << ", ";
```

代码中的 bind(&point::x, _1) 取出 point 对象的成员变量 x，transform 算法调用 bind 表达式操作容器 v，逐个把成员变量填入到 v2 中。

使用 bind，可以实现 SGISTL/STLport 中的非标准函数适配器 select1st 和 select2nd 的功能，直接选择出 pair 对象的 first 和 second 成员，例如：

```
typedef pair<int, string> pair_t;
pair_t p(123, "string");
cout << bind(&pair_t::first, p)() << endl;
cout << bind(&pair_t::second, p)() << endl;
```

但 bind 显然要比它们功能更强，可以选择类中任意名字的 public 成员变量。

11.3.5 绑定函数对象

bind 不仅能够绑定函数和函数指针，也能够绑定任意的函数对象，包括标准库中的所有预定义的函数对象。

如果函数对象有内部类型定义 result_type，那么 bind 可以自动推导出返回值类型，用法与绑定普通函数一样。但如果函数对象没有定义 result_type，则需要在绑定形式上做一点

改动，用模板参数指明返回类型，像这样：

```
bind<result_type>(Functor, ...);
```

标准库和 Boost 库中的大部分函数对象都具有 `result_type` 定义，因此不需要特别的形式就可以直接使用 `bind`，例如：

```
bind(std::greater<int>(), _1, 10);           //检查 x>10
bind(plus<int>(), _1, _2);                   //执行 x+y
bind(modulus<int>(), _1, 3);                 //执行 x%3
```

对于自定义函数对象，如果没有 `result_type` 类型定义，例如：

```
struct f
{
    int operator()(int a, int b)
    {
        return a + b;
    }
};
```

那么我们必须指明 `bind` 的返回值类型，像这样：

```
cout << bind<int>(f(), _1, _2)(10,20) << endl;
```

这种写法多少会有些不方便，因此，在编写自己的函数对象时，最好遵循规范为它们增加内部 `typedef result_type`，这将使函数对象与许多其他标准库和 Boost 库组件良好配合工作。

11.3.6 使用 ref 库

`bind` 采用拷贝的方式存储绑定对象和参数，这意味着绑定表达式中的每个变量都会有一份拷贝，如果函数对象或值参数很大、拷贝代价很高，或者无法拷贝，那么 `bind` 的使用就会受到限制。

因此 `bind` 库可以搭配 `ref` 库（11.2 小节，422 页）使用，`ref` 库包装了对象的引用，可以让 `bind` 存储对象引用的拷贝，从而降低了拷贝的代价。但这也带来了一个隐患，因为有时候 `bind` 的调用可能会延后很久，程序员必须保证 `bind` 被调用时引用是有效的。如果调用时引用的变量或者函数对象被销毁了，那么会发生未定义行为。

示范 `ref` 配合 `bind` 用法的代码如下：

```
int x = 10;
cout << bind(g, _1, cref(x), ref(x))(10) << endl;
f af;                                     //一个函数对象
cout << bind<int>(ref(af), _1, _2)(10, 20) << endl;
```

下面的代码则因为引用失效，引发了未定义行为：

```
int x = 10;
BOOST_AUTO(r, ref(x));                  //r 包装 x 的引用
```

```

{
    int *y = new int(20);
    r = ref(*y); //r 包装*y
    cout << r << endl;
    cout << bind(g, r, 1, 1)() << endl; //工作正常
    delete y; //引用的对象被销毁
}
cout << bind(g, r, 1, 1)() << endl; //未定义行为

```

11.3.7 高级议题

本小节讨论关于 bind 库的一些高级议题。

为占位符更名

bind 库默认使用 `_1`、`_2` 作为占位符的名称，通常它们很清晰地表明了被调用参数的含义，但也可能会有人对这些名字不够满意，想用其他更好懂易记的名字。bind 库允许这样做，方法有很多，最简单的是为原占位符使用引用创建别名，例如：

```

boost::arg<1> &_x = _1;
boost::arg<2> &_y = _2;
boost::arg<1> &arg1 = _1;
boost::arg<2> &arg2 = _2;

```

上面的代码前两行为占位符创建了类似数学函数变量的别名，而后两行则使用了稍长但更易理解的 `argN` 形式。

定义别名也可以使用 `BOOST_AUTO`（4.2 小节，97 页），这样就无需关心占位符的真实类型，把类型推导的工作交给编译器，有利于编写可移植的代码：

```

BOOST_AUTO(&_x, _1);
BOOST_AUTO(&_y, _2);

```

改名后的占位符用法与原占位符完全相同，例如：

```

assert(typeid(_x) == typeid(arg1));
cout << bind(f, _x, _y)(10, 20) << endl;
cout << bind(g, arg1, 2, arg2)(1, 2) << endl;

```

存储 bind 表达式

很多时候我们需要把写好的 bind 表达式存储起来，以便稍后再度使用，但 bind 表达式生成的函数对象类型声明非常复杂，通常无法写出正确的类型，因此可以使用 `typeof` 库（4.2 小节，97 页）的 `BOOST_AUTO` 宏来辅助我们，例如：

```

BOOST_AUTO(x, bind(greater<int>(), _1, _2));

```

```
cout << x(10, 20) << endl;
```

11.4 节 (437 页) 将要介绍的 function 库也可以达到同样的目的, 但它的功能更加强大。

嵌套绑定

bind 可以嵌套, 一个 bind 表达式生成的函数对象可以被另一个 bind 再绑定, 从而实现类似 $f(g(x))$ 的形式。

如果我们有 $f(x)$ 和 $g(x)$ 两个函数, 那么 $f(g(x))$ 的 bind 表达式就是:

```
bind(f, bind(g, _1))(x)
```

使用 bind 的嵌套用法必须小心, 它不太容易一次写正确, 也不容易理解, 超过两个以上的 bind 表达式通常只能被编译器读懂, 必须配合良好的注释才能够使用 bind 嵌套用法。

操作符重载

bind 重载了比较操作符和逻辑非操作符, 可以把多个 bind 绑定式组合起来, 形成一个复杂的逻辑表达式, 配合标准库算法可以实现语法简单但语义复杂的操作。

例如, 假设我们有一个存储有理数 rational 的 vector, 那么使用 bind 可以执行许多具有复杂逻辑的操作:

```
using namespace boost::assign;
typedef rational<int> ri; //有理数类
vector<ri> v = list_of(ri(1,2))(ri(3,4))(ri(5,6)); //初始化

//删除所有分子为 1 的有理数
remove_if(v.begin(), v.end(), bind(&ri::numerator, _1) == 1 );
assert(v[0].numerator() == 3); //有理数 1/2 被删除

//使用 find_if 算法查找分子是 1 的有理数, 不存在
assert(find_if(v.begin(), v.end(), bind(&ri::numerator, _1) == 1) == v.end());

//查找分子大于 3 且分母小于 8 的有理数
BOOST_AUTO(pos, find_if(v.begin(), v.end(),
    bind(&ri::numerator, _1) > 3 && bind(&ri::denominator, _1) < 8));

cout << *pos << endl; //输出 5/6
```

使用 bind 的操作符重载可以构建出复杂的逻辑表达式, 虽然好用, 但不熟悉 bind 用法的读者通常很难理解代码, 应当慎用少用。

很多情况下复杂的逻辑判断可以使用函数内部类来就地定义, 形式上要比 bind 组合清晰, 例如上面的 find_if 算法使用的 bind 表达式就等价于下面的函数对象:

```
struct pred
{
    bool operator()(ri &r)
    {
        return r.numerator() > 3 && r.denominator() < 8;
    }
};
pos = find_if(v.begin(), v.end(), pred());
```

比起 bind，简单明了的函数对象更容易理解。

绑定非标准函数

bind 库大大增强了 C++98 标准库中的函数绑定器，可以适配任何 C++ 中的函数。但标准形式 bind(f, ...) 不是 100% 适用于所有情况，有些非标准函数 bind 无法自动推导出返回值类型，典型的的就是 C 中的可变参数函数 printf()。必须用 bind<int>(printf, ...) (...) 的形式，例如：

```
bind<int>(printf, "%d+%d=%d\n", _1, 1, _2)(6, 7) ;
```

bind 的标准形式也不能支持使用了不同的调用方式(如 __stdcall、__fastcall、extern "C") 的函数，通常 bind 把它们看作函数对象，需要显式地指定 bind 的返回值类型才能绑定。

也可以在头文件 <boost/bind.hpp> 之前加上 BOOST_BIND_ENABLE_STDCALL、BOOST_BIND_ENABLE_FASTCALL 或 BOOST_BIND_ENABLE_PASCAL 等宏，明确地告诉 bind 支持这些调用方式。

11.4 function

function 是一个函数对象的“容器”，概念上像是 C/C++ 中函数指针类型的泛化，是一种“智能函数指针”。它以对象的形式封装了原始的函数指针或函数对象，能够容纳任意符合函数签名的可调用对象。因此，它可以被用于回调机制，暂时保管函数或函数对象，在之后需要的时机再调用，使回调机制拥有更多的弹性。

function 可以配合 bind 使用，存储 bind 表达式的结果，使 bind 可以被多次调用。

function 位于名字空间 boost，为了使用 function 组件，需要包含头文件 <boost/function.hpp>，即：

```
#include <boost/function.hpp>
using namespace boost;
```

11.4.1 类摘要

同 bind 一样，function 也不是一个单独的类，而是一个大的类家族。function 可以容

纳 0 到 10 个参数的函数，因此也就有多达 11 个类，命名分别是 `function0` 到 `function10`。但我们通常不直接使用它们，而是使用一个更通用的 `function` 类，它的类摘要如下：

```
template< typename Signature >
class function : public functionN <R, T1, T2, ..., TN>
{
public:

    //内部类型定义
    typedef R          result_type;
    typedef TN         argN_type;

    //参数个数常量
    static const int arity = N;

    //构造函数
    function();
    template<typename F> function(F);

    //基本操作
    void swap(const function&);
    void clear();
    bool empty() const;
    operator safe_bool() const;
    bool operator!() const;

    //访问内部元素
    template<typename Functor> Functor* target();
    template<typename Functor> const Functor* target() const;
    template<typename Functor> bool contains(const Functor&) const;
    const std::type_info& target_type() const;

    //调用操作符
    result_type operator()(arg1_type, ..., argN_type) const;
};
```

11.4.2 function 的声明

`function` 是一个模板类，但幸运的是我们不需要在模板参数列表中一一写出被容纳的函数原型的返回值和所有函数参数的类型。`function` 使用 `result_of` (11.1 小节, 419 页) 的方式，用类似函数声明的形式一次给出所有函数的类型信息，这被称为“首选语法”。例如：

```
function<int()> func;
```

将声明一个可以容纳返回值为 `int`、无参函数的 `function` 对象。这种声明形式很容易理解，尖括号中的类型声明很像是一个函数原型，只是没有函数名。它具有很好的可读性，可以读作

function of `int()`，表明这是一个容纳 `int()` 的 function 对象。

function 的函数类型声明也可以像真的函数声明那样带有参数名，例如下面的写法

```
function<int (int a, int b, int c)> func2;
```

与

```
function<int (int , int, int)> func2;
```

是完全等价的，读者可以任意选择喜欢的形式，但在本书之后的叙述中我们都将使用第二种形式，因为它更简洁清楚，同时也少打些字。

有的较老的编译器不支持这种便捷的形式，所以我们就不得不回到原始的模板类声明形式上，在模板参数列表中列出所有的模板参数，并且还要在 function 类名上指定参数的数量，这种用法被称为“兼容语法”。例如，刚才的两个 function 对象声明的兼容语法是：

```
function0<int> func;  
function3<int ,int , int, int> func2;
```

兼容语法可以为程序提供最大的兼容性，能够适应所有的编译器，但它的写法明显没有首选语法那样美观，也很难看出函数的原型，因此，除非真有必要，否则不推荐使用兼容语法。

11.4.3 操作函数

function 的构造函数可以接受任意符合模板中声明的函数类型的可调用对象，如函数指针和函数对象，也可以是另一个 function 对象的引用，之后在内部存储一份它的拷贝。

无参的构造函数或者传入空指针构造将创建一个空的 function 对象，不持有任何可调用物，调用空的 function 对象将抛出 `bad_function_call` 异常，因此在使用 function 前最好检测一下它的有效性。可以用 `empty()` 测试 function 是否为空，或者用重载操作符 `operator!` 来测试。function 对象也可以在一个 `bool` 上下文中直接测试它是否为空，它是类型安全的。

function 的其余成员函数功能如下：

- `clear()` 可以直接将 function 对象置空，它与使用 `operator=` 赋值 0 具有同样的效果；
- 模板成员函数 `target()` 可以返回 function 对象内部持有的可调用物 `Functor` 的指针，如果 function 为空则返回空指针 `NULL`；
- `contains()` 可以检测 function 是否持有一个 `Functor` 对象；
- 最后，function 提供了 `operator()`，它把传入的参数转交给内部保存的可调用物，完成真正的函数调用。

11.4.4 比较操作

`function` 重载了比较操作符 `operator==` 和 `operator!=`，可以与被包装的函数或函数对象进行比较。如果 `function` 存储的是函数指针，那么比较相当于

```
function.target<Functor>() == func_pointer
```

例如：

```
function<int(int,int)> func(f);
assert( func == f );
```

如果 `function` 存储的是函数对象，那么要求函数对象必须重载了 `operator==`，是可比较的。

两个 `function` 对象不能使用 `==` 和 `!=` 直接比较，这是特意的。因为 `function` 存在到 `bool` 的隐式转换，`function` 定义了两个 `function` 对象的 `operator==` 但没有实现，企图比较两个 `function` 对象会导致编译错误。

11.4.5 用法

`function` 就像是一个函数的容器，也可以把 `function` 想象成一个泛化的函数指针，只要符合它声明中的函数类型，任何普通函数、成员函数、函数对象都可以存储在 `function` 对象中，然后在任何需要的时候被调用。

`function` 这种能够容纳任意可调用对象的能力是非常重要的，在编写泛型代码的时候尤其有用，它使我们可以接受任意的函数或函数对象，增加程序的灵活性。

与原始的函数指针相比，`function` 对象的体积要稍微大一点（3 个指针的大小），速度要稍微慢一点（10%左右的性能差距），但这与它带给程序的巨大好处相比是无足轻重的。

示范 `function` 基本用法的代码如下：

```
#include <boost/function.hpp>
using namespace boost;
int f(int a, int b)                //声明一个二元函数
{ return a + b; }
int main()
{
    function<int(int,int)> func;    //无参构造一个 function 对象
    assert(!func);                //此时 function 不持有任何对象

    func = f;                      //func 存储了函数 f
    if (func)                     //function 可以转换为 bool 值
    {
```



```

    cout << func(10, 20);           //调用 function 的 operator()
}
func = 0;                          //function 清空, 相当于 clear()
assert(func.empty());
}

```

只要函数签名式一致, function 也可以存储成员函数和函数对象, 或者是 bind 表达式的结果。假设我们有如下的一个类 demo_class, 它既有普通成员函数, 又重载了 operator():

```

struct demo_class
{
    int add(int a, int b)           //加法操作
    {
        return a + b;    }
    int operator()(int x) const    //重载 operator()
    {
        return x*x;    }
};

```

存储成员函数时可以直接在 function 声明的函数签名式中指定类的类型, 然后用 bind 绑定成员函数:

```

function<int(demo_class&, int,int)> func1;
func1 = bind(&demo_class::add, _1, _2, _3);
demo_class sc;
cout << func1(sc, 10, 20);

```

也可以在函数类型中仅写出成员函数的签名, 在 bind 时直接绑定类的实例:

```

function<int(int,int)> func2;
func2 = bind(&demo_class::add,&sc, _1, _2);
cout << func2(10, 20);

```

11.4.6 使用 ref 库

function 使用拷贝语义保存参数, 当参数很大时拷贝的代价往往很高, 或者有时候不能拷贝参数。这时我们就可以向 ref 库 (11.2 小节, 422 页) 求助, 它允许以引用的方式传递参数, 能够降低 function 拷贝的代价。

function 并不要求 ref 库提供 operator(), 因为它能够自动识别包装类 reference_wrapper<T>, 并调用 get() 方法获得被包装的对象:

```

demo_class sc;                               //之前定义的函数对象
function<int(int)> func;
func = cref(sc);                             //使用 cref() 函数包装常对象的引用
cout << func(10);                           //调用被引用的对象

```

注意: 在这里我们使用的是 cref() 函数, 它是一个常引用包装, 因此只能调用 const 成员函数。

function 能够直接调用被 ref 库包装的函数对象，这个功能可以部分地弥补 boost.ref 没有 operator() 的遗憾。下面的代码定义了一个求总和的函数对象，它具有内部状态：

```
template<typename T>
struct summary                                     //函数对象，计算总和
{
    typedef void result_type;
    T sum;                                           //内部状态，总和
    summary(T v = T()):sum(v){}
    void operator()(T const &x)
    {    sum += x;    }
};
```

如果 ref 库不提供 operator()，那么它将无法用于标准库算法^①，因为标准库算法总使用拷贝语义，算法内部的改变不能影响原对象，function 可以提供一个略显麻烦但可用的解法：

```
int main()
{
    using namespace boost::assign;
    vector<int> v = (list_of(1),3,5,7,9);
    summary<int> s;                                //有状态的函数对象
    function<void(int const&)> func(ref(s));         //function 包装引用
    std::for_each(v.begin(), v.end(), func);       //使用标准库算法
    cout << s.sum << endl;                         //函数对象的状态被改变
}
```

11.4.7 用于回调

function 可以容纳任意符合函数签名式的可调用物，因此它非常适合代替函数指针，存储用于回调的函数，而且它的强大功能会使代码更灵活、富有弹性。

作为示范，我们定义一个 demo_class 类，它使用 function 代替函数指针作为内部类型保存回调函数，存储形式为 void(int) 的可调用物：

```
class demo_class
{
private:
    typedef function<void(int)> func_t;             //function 类型定义
    func_t func;                                   //function 对象
};
```

① 这句话不完全正确，有的算法（如 for_each）可以返回内部函数对象的拷贝，它持有被改变的内部状态，可以变通地获得内部状态，但大对象的拷贝仍然是个问题。

具体到本例子，直接使用函数对象和算法的代码可以是：

```
cout << (std::for_each(v.begin(), v.end(), s)).sum << endl;
```

```

int n;                                //内部成员变量
public:
    demo_class(int i):n(i){}

```

demo_class 使用模板函数 accept() 接受回调函数。之所以使用模板函数是因为这种形式更加灵活,用户可以在不知道也不关心内部存储形式的情况下传递任何可调用对象,包括函数指针和函数对象。例如:

```

template<typename Callback>
void accept(Callback f)                //存储回调函数
{
    func = f;
}

```

demo_class 的成员函数 run() 用来调用回调函数:

```

void run()
{
    func(n);
};
//demo_class 类定义结束

```

接下来我们定义一个用于回调的函数,它将输入翻倍:

```

void call_back_func(int i)
{
    cout << "call_back_func:";
    cout << i * 2 << endl;
}

```

demo_class 的回调可以这样使用:

```

int main()
{
    demo_class dc(10);
    dc.accept(call_back_func);        //接受回调函数
    dc.run();                          //调用回调函数,输出“call_back_func:20”
}

```

使用普通的 C 函数进行回调并不能体现 function 的好处,我们来编写一个带状态的函数对象,并使用 ref 库传递引用:

```

class call_back_obj
{
private:
    int x;                            //内部状态
public:
    call_back_obj(int i):x(i){}
    void operator()(int i)
    {
        cout << "call_back_obj:";
        cout << i * x++ << endl;      //先做乘法,然后递增
    }
}

```



```

    }
};
int main()
{
    demo_class dc(10);
    call_back_obj cbo(2);
    dc.accept(ref(cbo));           //使用 ref 库
    dc.run();                     //输出: call_back_obj:20
    dc.run();                     //输出: call_back_obj:30
}

```

demo_class 因为使用了 function 作为内部可调用物的存储, 因此不用做任何改变, 即可接受函数指针也可接受函数对象, 给用户以最大的方便。

function 还可以搭配 bind 库, 把 bind 表达式作为回调函数, 可以接受类成员函数, 或者把不符合函数签名式的函数 bind 为可接受的形式。下面我们定义一个回调函数工厂类, 它有两个回调函数:

```

class call_back_factory
{
public:
    void call_back_func1(int i)           //一个参数
    {
        cout << "call_back_factory1:";
        cout << i * 2 << endl;
    }
    void call_back_func2(int i, int j)    //两个参数
    {
        cout << "call_back_factory2:";
        cout << i * j * 2 << endl;
    }
};

```

function 搭配 bind 的用法如下:

```

int main()
{
    demo_class dc(10);
    call_back_factory cbf;
    dc.accept(bind(&call_back_factory::call_back_func1, cbf, _1));
    dc.run();                           //输出: call_back_factory1:20
    dc.accept(bind(&call_back_factory::call_back_func2, cbf, _1, 5));
    dc.run();                           //输出: call_back_factory2:100
}

```

通过以上的示例代码, 我们可以看到 function 用于回调的好处, 它无需改变回调的接口就可以解耦客户代码, 使客户代码不必绑死在一种回调形式上, 进而可以持续演化, 而 function

始终能够保证与客户代码正确沟通。

11.4.8 与 typeid 的区别

有的时候 typeid 库（4.2 小节，97 页）的 BOOST_AUTO 宏可以近似地取代 function，例如：

```
BOOST_AUTO(func, &f);           //存储一个普通函数指针
cout << func(10, 20) << endl;

demo_class sc;

//存储一个 bind 表达式
BOOST_AUTO(func2, bind(&demo_class::add, &sc, _1, _2));
cout << func2(10, 20) << endl;
```

但它们实现有很大的不同。function 类似一个容器，可以容纳任意有 operator() 的类型（函数指针、函数对象、lambda 表达式），它是运行时的，可以任意拷贝、赋值、存储其他可调用物。而 BOOST_AUTO 仅是在编译期推导出的一个静态类型变量，它很难再赋以其他值，也无法容纳其他的类型，不能用于泛型编程。

当需要存储一个可调用物用于回调的时候，最好使用 function，它具有更多的灵活性，特别是把回调作为类的一个成员的时候我们只能使用 function。

BOOST_AUTO 也有它的优点，它的类型是在编译器推导的，没有运行时的开销，效率上要比 function 略高一点。但它声明的变量不能存储其他类型的可调用物，不具有灵活性，只能用于有限范围的延后回调。

11.5 signals2

signals2 基于 Boost 的另一个库 signals，实现了线程安全的观察者模式。在 signals2 库中，观察者模式被称为信号/插槽（signals and slots），它是一种函数回调机制，一个信号关联了多个插槽，当信号发出时，所有关联它的插槽都会被调用。

许多成熟的软件系统都用到了这种信号/插槽机制（另一个常用的名称是事件处理机制：event/event handler），它可以很好地解耦一组互相协作的类，有的语言甚至直接内建了对它的支持（如 C#），signals2 以库的形式为 C++ 增加了这个重要的功能。

signals2 库位于名字空间 boost::signals2，为了使用 signals2 组件，需要包含头文件 <boost/signals2.hpp>，即：

```
#include <boost/signals2.hpp>
```

```
using namespace boost::signals2;
```

11.5.1 类摘要

signals2 库的核心是 signal 类, 相当于 C# 语言中的 event+delegate, 它的类摘要如下:

```
template<typename Signature,           //Function type R (T1,..., TN)
        typename Combiner = boost::signals2::optional_last_value<R>,
        typename Group = int,
        typename GroupCompare = std::less<Group> >
class signal : public boost::signals2::signal_base
{
public:

    signal(const combiner_type& = combiner_type(),
           const group_compare_type& = group_compare_type());
    ~signal();

    //插槽连接管理
    connection connect(const slot_type&, connect_position = at_back);
    connection connect(const group_type&, const slot_type&,
                       connect_position = at_back);
    void disconnect(const group_type&);
    template<typename S> void disconnect(const S&);
    void disconnect_all_slots();

    bool empty() const;
    std::size_t num_slots() const;

    //调用操作符
    result_type operator()( ...);

    //合并器
    combiner_type combiner() const;
    void set_combiner(const combiner_type&);
};
```

signal 的模板参数列表相当长, 总共有七个参数, 这里仅列出了最重要的前四个, 而且除了第一个是必须的外, 其他的都可以使用默认值:

- 第一个模板参数 Signature 的含义与 function 的一模一样, 也是一个函数类型签名, 表示可被 signal 调用的函数 (插槽、事件处理 handler)。例如:

```
signal<void(int, double)>
```

- 第二个模板参数 Combiner 是一个函数对象, 它被称为“合并器”, 用来组合所有插槽的调用结果, 默认是 optional_last_value<R>, 它使用 optional 库 (4.3 小节, 101 页) 返回最后一个被调用的插槽的返回值;

- 第三个模板参数 `Group` 是插槽编组的类型，缺省使用 `int` 来标记组号，也可以改为 `std::string` 等类型，但通常没有必要；
- 第四个模板参数 `GroupCompare` 与 `Group` 配合使用，用来确定编组的排序准则，默认是升序 (`std::less<Group>`)，因此要求 `Group` 必须定义了 `operator<`。

`signal` 继承自 `signal_base`，而 `signal_base` 又继承自 `noncopyable` (4.1 小节，95 页)，因此 `signal` 是不可拷贝的，如果把 `signal` 作为自定义类的成员变量，那么自定义类也将是不可拷贝的，除非使用 `shared_ptr` 来包装它。

11.5.2 操作函数

`signal` 最重要的操作函数是插槽管理 `connect()` 函数，它把插槽连接到信号上，相当于为信号（事件）增加了一个处理的 `handler`。

插槽可以是任意的可调用对象，包括函数指针、函数对象、以及它们的 `bind` 表达式和 `function` 对象，`signal` 内部使用 `function` 作为容器来保存这些可调用对象。连接时可以指定组号也可以不指定组号，当信号发生时将依据组号的排序准则依次调用插槽函数。

如果连接成功，`connect()` 将返回一个 `connection` 对象，表示了信号与插槽之间的连接关系，它是一个轻量级的对象，可以处理两者间的连接，如断开、重连接、或者测试连接状态。

成员函数 `disconnect()` 可以断开插槽与信号的连接，它有两种形式：传递组号将断开该组的所有插槽，传递一个插槽对象将仅断开该插槽。函数 `disconnect_all_slots()` 可以一次性断开信号的所有插槽连接。

当前信号所连接的插槽数量可以用 `num_slots()` 获得，成员函数 `empty()` 相当于 `num_slots() == 0`，但它的执行效率比 `num_slots()` 高。`disconnect_all_slots()` 的后果就是令 `empty()` 返回 `true`。

`signal` 提供 `operator()`，可以接受最多 9 个参数。当 `operator()` 被外界调用时意味着产生一个信号（事件），从而导致信号所关联的所有插槽被调用。插槽调用的结果使用合并器处理后返回，默认情况下是一个 `optional` 对象。

成员函数 `combiner()` 和 `set_combiner()` 分别用于获取和设置合并器对象，通过 `signal` 的构造函数也可以在创建的时候就传入一个合并器的实例。但通常我们可以直接使用缺省构造函数创建模板参数列表中指定的合并器对象，除非你想改用其他的合并方式。关于合并器我们将在 11.5.4 小节（449 页）详细介绍。

当 `signal` 析构时，将自动断开所有插槽连接，相当于调用 `disconnect_all_slots()`。

11.5.3 插槽的连接与调用

signal 就像是一个增强的 function 对象，它可以容纳（使用 connect() 连接）多个符合模板参数中函数签名类型的函数（插槽），形成一个插槽链表，然后在信号发生时一起调用。

例如，我们有如下两个无参的函数，它们可以被用作插槽：

```
void slots1()
{ cout << "slot1 called" << endl;}
void slots2()
{ cout << "slot2 called" << endl;}
```

除了类名字不同，signal 的声明语法与 function 几乎一模一样：

```
signal<void()> sig; //指定插槽类型 void(), 其他模板参数使用缺省值
```

然后我们就可以使用 connect() 来连接插槽，最后用 operator() 来产生信号：

```
int main()
{
    signal<void()> sig; //一个信号对象

    sig.connect(&slots1); //连接插槽 1
    sig.connect(&slots2); //连接插槽 2
    sig(); //调用 operator(), 产生信号（事件），触发插槽调用
}
```

在连接插槽时我们省略了 connect() 的第二个参数 connect_position，它的缺省值是 at_back，表示插槽将插入到信号插槽链表的尾部，因此 slots2 将在 slots1 之后被调用。程序的运行结果是：

```
slot1 called
slot2 called
```

如果在连接 slots2 的时候不使用缺省参数，而是明确地传入 at_front 位置标志，即：

```
sig.connect(&slots2, at_front);
```

那么 slots2 将在 slots1 之前被调用。

使用组号

connect() 函数的另一个重载形式可以在连接时指定插槽所在的组号，缺省情况下组号是 int 类型。组号不一定要从 0 开始连续编号，它可以是任意的数值，离散的、负值都允许。

如果在连接的时候指定组号，那么每个编组的插槽将是又一个插槽链表，形成一个略微有些

复杂的二维链表，它们的顺序规则如下：

- 各编组的调用顺序由组号从小到大决定(也可以在 signal 的第四个模板参数改变排序函数对象)；
- 每个编组的插槽链表内部的插入顺序用 at_back 和 at_front 指定；
- 未被编组的插槽如果位置标志是 at_front，将在所有的编组之前调用；
- 未被编组的插槽如果位置标志是 at_back，将在所有的编组之后调用。

我们使用一个新的函数对象 slots 来演示一下 signal 的编组，它是一个模板类：

```
template<int N>
struct slots                                //模板类,可以生成一系列的插槽
{
    void operator() ()
    {      cout << "slot"<< N <<" called" << endl;      }
};
```

signal 的连接代码如下：

```
sig.connect(slots<1>(),at_back);           //最后被调用
sig.connect(slots<100>(), at_front);       //第一个被调用

sig.connect(5,slots<51>(), at_back);       //组号 5,该组最后一个
sig.connect(5,slots<55>(), at_front);     //组号 5,该组第一个

sig.connect(3,slots<30>(), at_front);     //组号 3,该组第一个
sig.connect(3,slots<33>(), at_back);      //组号 3,该组最后一个

sig.connect(10,slots<10>());               //组号 10,该组仅有一个
```

当执行 sig() 后，插槽的调用结果如下：

```
slot100 called
slot30 called
slot33 called
slot55 called
slot51 called
slot10 called
slot1 called
```

11.5.4 信号的返回值

signal 如 function 一样，不仅可以把输入参数转发给所有插槽，也可以传回插槽的返回值。默认情况下 signal 使用合并器 optional_last_value<R>，它将使用 optional 对象

返回最后被调用的插槽的返回值。

我们修改一下之前定义的 slots 模板类，为它的 operator() 增加参数和返回值：

```
template<int N>
struct slots
{
    int operator()(int x)
    {
        cout << "slot"<< N <<" called" << endl;
        return x *N;
    }
};
```

signal 的声明对应也需要修改：

```
signal<int(int)> sig;
```

然后我们向信号连接三个插槽：

```
sig.connect(slots<10>());
sig.connect(slots<20>());
sig.connect(slots<50>());
```

signal 的 operator() 调用这时需要传入一个整数参数，这个参数会被 signal 存储一个拷贝，然后转发给各个插槽。最后 signal 将返回插槽链表末尾 slots<50>() 的计算结果，它是一个 optional 对象，必须用解引用操作符*来获得值，即：

```
cout << *sig(2); //输出 100
```

11.5.5 合并器

缺省的合并器 optional_last_value<R>并没有太多的意义，它通常用在我们不关心插槽返回值或者返回值是 void 的时候。但大多数时候，插槽的返回值都是有意义的，需要以某种方式处理多个插槽的返回值。

signal 允许用户自定义合并器来处理插槽的返回值，把多个插槽的返回值合并为一个结果返回给用户。合并器应该是一个函数对象（不是函数或者函数指针），具有类似如下的形式：

```
template<typename T>
class combiner //自定义合并器
{
public:
    typedef T result_type; //返回值类型定义
    template<typename InputIterator>
        result_type operator()(InputIterator, InputIterator) const;
```

```
};
```

combiner 类的调用操作符 operator() 的返回值类型可以是任意类型，完全由用户指定，不一定必须是 optional 或者是插槽的返回值类型。operator() 的模板参数 InputIterator 是插槽链表的返回值迭代器，可以使用它来遍历所有插槽的返回值，进行所需的处理。

作为示范，我们编写一个自定义的合并器，它使用 pair 返回所有插槽的返回值之和以及其中的最大值^①：

```
template<typename T>
class combiner
{
    T v; //计算总和的初始值
public:
    typedef std::pair<T, T> result_type;
    combiner(T t = T()):v(t) {} //构造函数
    template<typename InputIterator>
    result_type operator()(InputIterator begin, InputIterator end) const
    {
        if (begin == end) //如果返回值链表为空，则返回 0
        {
            return result_type();
        }
        vector<T> vec(begin, end); //使用容器保存插槽调用结果
        T sum = std::accumulate(vec.begin(), vec.end(), v);
        T max = *std::max_element(vec.begin(), vec.end());
        return result_type(sum, max);
    }
};
```

使用自定义合并器的时候我们需要改写 signal 的声明，在模板参数列表中增加第二个模板参数——合并器类型：

```
signal<int(int), combiner<int> > sig;
```

在这里我们没有向构造函数传递合并器的实例，因为 signal 的构造函数会缺省构造出一个实例，相当于：

```
signal<int(int), combiner<int> > sig(combiner<int>());
```

插槽的连接和调用如下：

```
sig.connect(slots<10>());
sig.connect(slots<20>());
sig.connect(slots<30>(), at_front); //最大值，第一个调用
```

① 注意：在这个合并器的代码里我们没有直接使用标准库的算法 accumulate 和 max_element 来操作迭代器，因为那样会遍历两次插槽列表，导致所有插槽被调用两次。详细解释见后（461 页）。

```
BOOST_AUTO(x, sig(2)); //用 BOOST_AUTO 获得信号的返回值
cout << x.first << ", " << x.second; //输出 120,60
```

当信号被调用时，`signal` 会自动把解引用操作转换为插槽调用，将调用给定的合并器的 `operator()` 逐个处理插槽的返回值，并最终返回合并器 `operator()` 的结果，因此，上面的代码将输出“120,60”。

如果我们不使用 `signal` 的缺省构造函数，而是在构造 `signal` 时传入一个合并器的实例，那么 `signal` 将使用这个合并器（的拷贝）处理返回值。例如，下面的代码使用了一个有初值的合并器对象，累加值从 100 开始：

```
signal<int(int), combiner<int> > sig(combiner<int>(100));
...
cout << x.first << ", " << x.second; //输出 220,60
```

11.5.6 管理信号的连接

信号与插槽的连接并不要求是永久的，当信号调用完插槽后，有可能需要把插槽从信号中断开，再连接到其他的信号上去。

`signal` 可以用成员函数 `disconnect()` 断开一个或者一组插槽，或者使用 `disconnect_all_slots()` 断开所有插槽连接，函数 `empty()` 和 `num_slots()` 用来检查信号当前插槽的连接状态。

要断开一个插槽，插槽必须能够进行等价比较，对于函数对象来说就是重载一个等价语义的 `operator==`。

我们可以为 `slots<N>` 增加一个等价比较，因为它没有状态，所以非常简单：

```
template<int N>
bool operator==(const slots<N>&, const slots<N>&)
{ return true;}
```

接下来的代码示范了插槽的连接与断开：

```
#include <boost/signals2.hpp>
using namespace boost::signals2;
int main()
{
    signal<int(int)> sig;
    assert(sig.empty()); //刚开始没有连接任何插槽

    sig.connect(0, slots<10>()); //连接两个组号为 0 的插槽
    sig.connect(0, slots<20>());
```

```

sig.connect(1,slots<30>());           //连接组号为 1 的插槽

assert(sig.num_slots() == 3);         //目前有 3 个插槽
sig.disconnect(0);                   //断开第 0 组插槽,共两个
assert(sig.num_slots() == 1);
sig.disconnect(slots<30>());          //断开一个插槽
assert(sig.empty());                 //信号不再连接任何插槽
}

```

使用 signal 自身管理插槽有一点不方便,因为它必须知道与它连接的所有插槽的信息,还要求插槽对象必须是可等价比较的,很多时候这些条件很难满足,比如有可能信号所连接的插槽是由另外一个库提供的,插槽并不支持比较操作。

11.5.7 更灵活的管理信号连接

signals2 库提供另外一种较为灵活的连接管理方式:使用 connection 对象。

每当 signal 使用 connect() 连接插槽时,它就会返回一个 connection 对象。connection 对象像是信号与插槽连接关系的一个句柄(handle),可以管理连接。它的类摘要如下:

```

class connection {
public:

    //构造函数与析构函数
    connection();
    connection(const connection&);
    connection& operator=(const connection&);

    //插槽连接管理
    void disconnect() const;
    bool connected() const;

    //阻塞操作
    bool blocked() const;

    //交换
    void swap(const connection&);

    //比较操作
    bool operator==(const connection&) const;
    bool operator!=(const connection&) const;
    bool operator<(const connection&) const;
};

```

connection 是可拷贝可赋值的,它也重载了比较操作符,因而可以被安全地放入标准序列

容器或者关联容器中，成员函数 `disconnect()` 和 `connected()` 分别用来与信号断开连接和检测连接状态。例如：

```
#include <boost/signals2.hpp>
using namespace boost::signals2;
int main()
{
    signal<int(int)> sig;

    connection c1 = sig.connect(0, slots<10>());
    connection c2 = sig.connect(0, slots<20>());
    connection c3 = sig.connect(1, slots<30>());

    c1.disconnect();                //断开第一个连接
    assert(sig.num_slots() == 2);    //sig 现在连接两个插槽
    assert(!c1.connected());        //c1 不再连接信号
    assert(c2.connected());         //c2 仍然连接
}
```

另外一种连接管理对象是 `scoped_connection`，它是 `connection` 的子类，提供类似 `scoped_ptr` (3.2 小节，63 页) 的 RAII 功能：插槽与信号的连接仅在作用域内生效，当离开作用域时连接就会自动断开。

当需要临时连接信号时 `scoped_connection` 会非常有用，例如：

```
#include <boost/signals2.hpp>
using namespace boost::signals2;
int main()
{
    signal<int(int)> sig;

    sig.connect(0, slots<10>());
    assert(sig.num_slots() == 1);
    {
        //进入局部作用域，建立临时连接
        scoped_connection sc = sig.connect(0, slots<20>());
        assert(sig.num_slots() == 2);
    }
    //离开局部作用域，临时连接自动断开
    assert(sig.num_slots() == 1);
}
```

插槽与信号的连接一旦断开就不能再连接起来，`connection` 不提供 `reconnect()` 这样的函数。但可以暂时地阻塞插槽与信号的连接，当信号发生时被阻塞的插槽将不会被调用，`connection` 对象的 `blocked()` 函数可以检测插槽是否被阻塞。但被阻塞的插槽并没有断开与信号的连接，在需要的时候可以随时解除阻塞。

`connection` 对象自身没有阻塞的功能，它需要一个辅助类 `shared_connection_block`，

它将阻塞 connection 对象，直到它被析构或者显式调用 unblock() 函数。

示范连接阻塞用法的代码如下：

```
#include <boost/signals2.hpp>
using namespace boost::signals2;
int main()
{
    signal<int(int)> sig;

    connection c1 = sig.connect(0, slots<10>());
    connection c2 = sig.connect(0, slots<20>());
    assert(sig.num_slots() == 2);           //有两个插槽连接
    sig(2);                                //调用两个插槽

    cout << "begin blocking..." << endl;
    {
        shared_connection_block block(c1); //阻塞 c1 连接
        assert(sig.num_slots() == 2);      //仍然有两个连接
        assert(c1.blocked());              //c1 被阻塞
        sig(2);                            //只有一个插槽会被调用
    }                                       //离开作用域, 阻塞自动解除
    cout << "end blocking..." << endl;
    assert(!c1.blocked());
    sig(2);                                //可以调用两个插槽
}
```

程序的运行结果如下：

```
slot10 called
slot20 called
begin blocking...
slot20 called
end blocking...
slot10 called
slot20 called
```

11.5.8 自动连接管理

在我们之前讲述的信号/插槽处理系统中存在一个问题：如果插槽在与信号建立连接后被意外地销毁了，那么信号调用将发生未定义行为。

例如下面的代码：

```
int main()
{
    signal<int(int)> sig;
```

```

sig.connect(slots<10>());           //正常连接
BOOST_AUTO( p , new slots<20>);    //创建一个指针对象

sig.connect(ref(*p));              //用 ref 包装,连接到引用
delete p;                          //指针被销毁
sig(1);                            //信号调用将发生未定义行为
}

```

这段示例代码很短,因此销毁指针引发未定义行为很可能不会造成程序崩溃,因为 `p` 指向的内存并没有被其他操作覆盖, `signal` 的调用时内存里可能还有原来的插槽数据,因此可能会正常运行。但如果在一个大型程序中,如果指针失效,那么 `signal` 的调用的后果将无法想象。^①

为了避免这种情况的出现, `signals2` 库使用 `slot` 类提供了自动连接管理的功能,能够自动跟踪插槽的生命周期,当插槽失效时会自动断开连接,以保证程序不会发生运行错误。

`slot` 的类摘要如下:

```

template<typename Signature,           //Function type R (T1,..., TN)
        typename SlotFunction = boost::function<R (T1, ..., TN)>> >
class slot : public boost::signals2::slot_base
{
public:
    template<typename Slot> slot(const Slot &);
    template<typename Func, typename Arg1,..., typename ArgN>
        slot(const Func &, const Arg1 &, ..., const ArgN &);

    result_type operator() (...);

    //跟踪功能
    slot & track(const weak_ptr<void> &);
};

```

`signals2::slot` 模板类可以自动管理插槽的连接,但通常我们不直接使用它,而是使用 `signal` 的内部 `typedef slot_type`,它已经定义好了该 `signal` 所使用的 `slot` 的模板参数。

要使用自动管理连接的功能,在信号连接时我们不能直接连接插槽,而是要用 `slot` 的构造函数包装插槽,然后再用成员函数 `track()` 来跟踪插槽使用的资源。下面的代码示范了它的用法:

```

int main()
{
    typedef signal<int(int) > signal_t;    //typedef 用于简化类型声明
    signal_t sig;

```

① 这段代码很好地演示了“未定义行为”的表现:大多数情况下它会工作正常,但关键时刻会导致程序崩溃。


```

sig.connect(slots<10>()); //连接一个普通的 slot
shared_ptr<slots<20> > p(new slots<20>); //用 shared_ptr 管理资源

//注意 slot_type 的用法
Sig.connect(signal_t::slot_type(ref(*p)).track(p));
p.reset(); //销毁插槽
assert(sig.num_slots() == 1); //一个插槽被自动断开
sig(1); //将只有一个插槽被调用
}

```

slot 的构造函数有一个有趣的地方是支持 bind 表达式相同的语法,这样可以就地绑定函数,避免使用 bind 表达式的构造成本。示范 slot 用于 bind 和 function 的情形的代码如下:

```

int main()
{
    typedef signal<int(int) > signal_t;
    typedef signal_t::slot_type slot_t;
    signal_t sig;

    //声明两个 shared_ptr
    shared_ptr<slots<10> > p1(new slots<10>);
    shared_ptr<slots<20> > p2(new slots<20>);

    function<int(int)> func = ref(*p1); //function 存储引用

    sig.connect(slot_t(func).track(p1)); //直接跟踪 function

    //使用 bind 语法,直接绑定
    sig.connect(slot_t(&slots<20>::operator(), p2.get(), _1).track(p2));

    p1.reset(); //销毁两个指针对象
    p2.reset();
    assert( sig.num_slots() == 0); //此时已经自动断开了所有连接
    sig(1); //不发生任何插槽调用
}

```

注意: 在使用 bind 语法时我们必须传递给 slot 原始指针,否则 slot 会持有一个 shared_ptr 的拷贝,导致引用计数增加,妨碍了 shared_ptr 的资源管理。

11.5.9 应用于观察者模式

本小节我们将使用 signals2 开发一个完整的观察者模式示例程序,用来演示信号/插槽的用法。这个程序将模拟一个日常生活情景:客人按门铃,门铃响,护士开门,婴儿哭闹。

首先我们要实现门铃类 ring,它是本程序中的核心类,拥有一个 signal 对象,当按门铃

时就会发出信号。

```
class ring
{
public:
    typedef signal<void()> signal_t;           //内部类型定义
    typedef signal_t::slot_type slot_t;

    connection connect(const slot_t& s)       //连接插槽
    {
        return alarm.connect(s);
    }
    void press()                               //按门铃动作
    {
        cout << "Ring alarm..." << endl;
        alarm();                             //调用 signal, 发出信号, 引发插槽调用
    }
private:
    signal_t alarm;                           //信号对象
};
```

我们决定采用随机数来让护士和婴儿的行为具有不确定性, 这样程序会更有趣些。随机数的产生使用 random 库 (9.4 小节, 357 页), 为了方便使用我们把随机数发生器定义为全局变量:

```
typedef variate_generator<rand48, uniform_smallint<> > bool_rand;
bool_rand g_rand(rand48(time(0)), uniform_smallint<>(0,100));
```

然后我们实现护士类 nurse, 它有一个 action() 函数, 根据随机数决定是惊醒开门还是继续睡觉。注意: 它的模板参数, 使用了 char const* 作为护士的名字, 因此实例化时字符串必须被声明成 extern:

```
extern char const nurse1[] = "Mary";
extern char const nurse2[] = "Kate";
template<char const *name>
class nurse                               //护士类
{
private:
    bool_rand &rand;                     //随机数发生器
public:
    nurse():rand(g_rand){}              //构造函数
    void action()
    {
        cout << name;
        if (rand() > 30)                 //70%的几率惊醒
        {
            cout << " wake up and open door." << endl;
        }
        else                             //30%几率继续睡觉
        {
            cout << " is sleeping..." << endl;
        }
    }
};
```

接下来是婴儿类，它与护士类的实现差不多：

```
extern char const baby1[] = "Tom";
extern char const baby2[] = "Jerry";
template<char const *name>
class baby
{
private:
    bool_rand &rand;
public:
    baby():rand(g_rand){}
    void action()
    {
        cout << "Baby " << name;
        if (rand() > 50)
        {   cout << " wakeup and crying loudly..." << endl;   }
        else
        {   cout << " is sleeping sweetly..." << endl;   }
    }
};
```

最后我们还需要一个客人类，它的唯一动作就是按门铃触发 `press` 事件：

```
class guest
{
public:
    void press(ring &r)
    {
        cout << "A guest press the ring." << endl;
        r.press();
    }
};
```

程序的主要功能都完成了，现在可以把它们组合起来：

```
int main()
{
    //声明门铃、护士、婴儿、客人等类的实例
    ring r;                                //门铃
    nurse<nurse1> n1;                       //护士 1
    nurse<nurse2> n2;                       //护士 2
    baby<baby1> b1;                         //婴儿 1
    baby<baby2> b2;                         //婴儿 2
    guest g;                               //访客

    //把护士、婴儿与门铃连接起来
```



```

r.connect(bind(&nurse<nurse1>::action, n1));
r.connect(bind(&nurse<nurse2>::action, n2));
r.connect(bind(&baby<baby1>::action, b1));
r.connect(bind(&baby<baby2>::action, b2));

//客人按动门铃，触发一系列的事件
g.press(r);
}

```

程序的运行结果可能是这样（比较有趣）：

```

A guest press the ring.
Ring alarm...
Mary is sleeping...
Kate is sleeping...
Baby Tom wakeup and crying loudly...
Baby Jerry is sleeping sweetly...

```

11.5.10 高级议题

本小节讨论关于 signals2 的一些高级议题。

让 signal 支持拷贝

signal 是 noncopyable 的子类，这意味着它不能被拷贝或者赋值，因此，如果我们的自定义类有 signal 成员变量（如之前的 ring 类），那么自定义类也将是不能拷贝的，企图拷贝含有 signal 成员类将会引发编译错误。

如果出于某种理由，确实需要在多个对象之间共享 signal 对象，那么可以考虑使用 shared_ptr<signal<Signature>> 作为类的成员，shared_ptr 可以很好地管理 signal 的共享语义。

下面的代码定义了一个含有 shared_ptr<signal<Signature>> 成员类 demo_class，它可以被任意拷贝或者赋值：

```

class demo_class
{
public:
    typedef signal<void()> signal_t;           //类型定义方便使用
    shared_ptr<signal_t> sig;                  //shared_ptr of signal
    int x;
    demo_class():sig(new signal_t), x(10){}    //构造函数
};
void print()                                  //一个插槽函数
{ cout << "hello sig." << endl;}

```

```
int main()
{
    demo_class obj;
    assert(obj.sig.use_count() == 1);
    demo_class obj2(obj);                //拷贝构造
    assert(obj.sig.use_count() == 2);

    obj.sig->connect(&print);             //obj 链接插槽
    (*obj2.sig)();                       //obj2 可以调用共享的信号
}
```

插槽调度

因为 `signal` 会自动把解引用操作转换为插槽调用，所以自定义合并器某种程度上也相当于一个插槽调度器。程序可以不要求所有的插槽被调用，只选择那些符合特定条件的插槽，比如当一个插槽的返回值满足要求后就终止迭代，不再调用剩余的插槽。

下面的合并器当得到一个大于 100 的返回值就停止插槽调用：

```
class combiner                                //合并器也可以不是模板类
{
public:
    typedef bool result_type;
    template<typename InputIterator>
    result_type operator()(InputIterator begin, InputIterator end) const
    {
        while(begin != end)
        {
            if(*begin > 100)
                return true;
        }
        return false;
    }
};
```

线程安全

`signal` 模板参数列表的最后一个类型参数是互斥量 `Mutex`，默认值是 `signals2::mutex`，它会检测编译器的线程支持程度，根据操作系统自动决定要使用的系统互斥量对象（Windows 下使用临界区，UNIX 下使用 `pthread_mutex`）。通常 `mutex` 都工作的很好，不需要改变它。

`signal` 对象在创建时会自动创建一个 `mutex` 保护内部状态，每一个插槽连接时也会创建出一个新的 `mutex`，当信号或插槽被调用时 `mutex` 都会自动锁定，因此 `signal` 可以很好地工作于多线程环境。

同样地, `connection` 和 `shared_connection_block` 也是线程安全的, 但用于自动连接管理的 `slot` 类不是线程安全的。

`signals2` 库中还有一个 `dummy_mutex`, 它是一个空的 `mutex` 类, 把它作为模板参数可以使 `signals2` 变成非线程安全的版本, 由于不使用锁定速度会稍微快一些。

由于 `mutex` 是 `signal` 的最后一个模板参数, 要指定它需要写出很多缺省的类型, `signals2` 使用元函数的方式可以便利地完成这个工作, 例如:

```
typedef signal_type<int(int),
    keywords::mutex_type<dummy_mutex> >::type signal_t;
```

让插槽自己管理连接

默认情况下插槽与信号 `signal`、连接 `connection` 都是无关的, 它自己无法处理与信号的连接, 只能由信号或者连接时返回的 `connection` 对象来管理, 但有的时候我们又必须让插槽自己管理连接。

`signals2` 库提供了 `connect_extended()` 函数和 `extended_slot_type` 类型定义, 可以让插槽接受一个额外的 `connection` 对象以处理连接。为了使用这个功能, 需要修改插槽的声明, 使它能够接受 `connection` 对象, 例如:

```
template<int N>
struct slots
{
    int operator()(const connection &conn, int x)
    {
        //检查连接状态
        cout << "conn=" << conn.connected() << endl;
        return x * N;
    }
};
```

在连接插槽时必须使用 `extended_slot_type` 的类 `bind` 语法, 用占位符 `_1` 向插槽传递 `connection` 对象。下面的代码示范了它的用法:

```
int main()
{
    typedef signal<int(int) > signal_t;
    typedef signal_t::extended_slot_type slot_t;
    signal_t sig;

    //_1 是 connection 对象, _2 是插槽实际使用的参数
    sig.connect_extended(slot_t(&slots<10>::operator(), slots<10>(), _1, _2));
    sig.connect_extended(slot_t(&slots<20>::operator(), slots<20>(), _1, _2));
}
```

```
sig(5);                                //整数 5 将作为 _2 的实际参数传递给插槽
}
```

关于 function

signal 内部使用 function 来存储可调用物，它的声明也与 function 很像，同样提供了 operator()。在 signal 只连接了一个插槽的时候基本上可以与 function 替换，例如，如果我们有如下的函数：

```
void f()
{ cout << "func called" << endl;}
```

那么下面的两种 function 和 signal 的调用代码功能上是等价的：

```
function<void()> func;                //function 对象
func = f;                             //存储一个可调用物
func();                               //调用函数
signal<void()> sig;                   //signal 对象
sig.connect(&f);                      //连接一个插槽
sig();                                //触发事件,产生信号,调用插槽
```

但需要注意它们的返回值，function 对象直接返回被包装函数的返回值，而 signal 则使用 optional 对象作为返回值，真正的返回值需要使用解引用操作符*才能取得。

signal 比 function 提供了用于回调更多的灵活性，但也使得它的用法比较复杂，较难掌握。

与 signals 的区别

signals 是 Boost 库中另一个信号/插槽库，实际上 signals2 的实现是基于 signals 的。signals2 与 signals 最大的区别是具有线程安全，能够用于多线程环境，而且不需要编译就可以使用。

signals2 在结构、接口、设计原理等许多方面都模仿了 signals，但也有一些很多变化，例如默认信号调用返回是一个 optional 对象、combiner() 函数改为 set_combiner()、connection 对象不再拥有 block() 方法，等等。

但大多数情况下，从 signals 库转移到 signals2 库还是相当容易的。

与 C#的区别

signals2 中的信号/插槽机制原理上类似于 c#语言的 event/delegate 机制，如果读者有 C#的经验，那么学习 signals2 库将不是很困难的。

但 C#的 delegate 的功能要比 signals2 弱，它要求精确的类型匹配，也没有合并器的概

念，只能返回一个结果。

delegate 使用 `operator+=` 来连接 event 与 delegate, `signals2` 则使用 `connect()` 函数。这是因为 `signals2` 在设计时认为 `operator+=` 并没有带来太多的好处，反而会导致连续使用 `+=` 连接、`operator-=` 等其他语义问题。

扩展实现 `operator+=`

`signal` 不支持 `operator+=`，虽然有很多合理的理由，但还是有人喜欢这种自然简单的写法。如果真的很需要，那么可以对 `signal` 做一层简单的包装，为它添加这个功能。

下面我们实现一个示范性质的 `signal` 包装类 `sig_ex`，它基本模仿了 `signal` 的形式，但简化了很多^①。`operator+=` 直接调用 `connect()` 函数，两者是等价的操作。比较麻烦的是 `operator()`，需要支持从 0 参到任意数量参数，本书仅定义了一个形式，使用了 `signal` 的内部类型定义 `arg<N>::type`，读者可以根据需要自行实现所需的 `operator()`。

```
template<typename Signature>
class sig_ex
{
public:
    typedef signal<Signature> signal_type;
    typedef typename signal_type::slot_type slot_type;
    connection connect(const slot_type& s)           //连接插槽
    {
        return sig.connect(s); }
    connection operator+=(const slot_type& s)       //操作符+=重载连接
    {
        return connect(s); }
    typename signal_type::result_type
        operator()(typename signal_type::arg<0>::type a0)
    {
        return sig(a0); }
private:
    signal_type sig;
};
```

`sig_ex` 的使用方式与 `signal` 没有什么区别，只是增加了一个方便的 `+=` 连接插槽的操作，例如：

```
int main()
{
    sig_ex<int(int)> sig;

    sig += slots<10>();           //使用类似 C# 的语法连接插槽
    sig += slots<5>();

    sig(2);                       //信号调用
}
```

① 这里使用的是对象适配器模式，也可以改用类适配器模式，使用继承来实现。

11.6 总结

本章我们讨论了 Boost 库中五个用于函数和回调编程的组件,使用它们需要对 C++ 的许多特性有深刻的理解。

我们首先讨论了 `result_of` 库。它很小但功能很强大,使用了模板元编程技术,可以帮助确定一个调用表达式的返回类型,类似 `typeof` 库,主要用于泛型编程。

`ref` 也是一个很小的库。它最初是 `tuple` 库的一部分,后来由于其重要性而被移出,成为了单独的库,而且也被收入了 TR1 标准草案。它能够包装对象的引用,变成一个可以被拷贝、赋值的普通对象,因此减少了昂贵的复制代价,标准库算法、`tuple`、`bind`、`function` 等许多库都可以从 `ref` 库受益。

但 Boost 的 `ref` 库实现有个较大的缺陷,不支持 `operator()` 重载(函数调用),本章采用 `result_of` 库做出了一个示范性质的实现,它可以配合标准库算法和其他库组件正常工作。

`bind` 是一个功能强大的函数绑定器,它远远地超越了历史上出现过的各种函数绑定器,包括标准库中的 `bind1st`、`bind2nd`、`mem_fun` 和非标准的 `select1st`、`select2nd`、`compose_f_gx`, 等等。它可以绑定任何可调用对象,搭配标准算法可以获得灵活操作容器内元素的强大功能。`bind` 还支持嵌套和操作符重载,可以组合出具有丰富表达力的函数形式。但 `bind` 过于强大也是个弱点,程序员学会 `bind` 的用法后往往会倾向于总使用 `bind` 解法,而忘记代码的清晰易读才是更重要的。如果一个绑定表达式使用了两个以上的 `bind`,那通常意味着不是一个好的解决方案。

`function` 库是函数指针的泛化,可以存储任意可调用的对象,因此 `function` 库经常配合 `bind` 使用,它可以存储 `bind` 表达式的结果,以备之后调用。`function` 具有很多的优点,它是泛型的,比普通的函数指针能够接受更多的可调用对象,因此大大增加了它的灵活性。`function` 也可以配合 `ref` 库使用,存储有内部状态的函数对象,弥补 `boost.ref` 库没有 `operator()` 的缺憾,使 `reference_wrapper<T>` 对象能够像被包装的函数对象一样被调用。

本章最后介绍的是 `signals2` 库,它综合运用了前四个组件,使用了信号/插槽机制,是观察者设计模式的一个具体应用,也是一个功能强大的回调框架。使用 `signals2` 库可以简化对象间的通信关系,降低它们的耦合性,只需要在程序开始时把它们连接起来,之后的一切都会自动处理。

`signals2` 还有许多的高级用法,可以使用合并器任意处理插槽的返回值,可以自动跟踪插槽的生命周期。它也是线程安全的,能够被安全地应用在多线程程序中,而且不需要预先编译。

本章实现的实用类如下:

- `reference_wrapper` 增强: 根据 TR1 草案,为 `boost.ref` 库增加了函数调用操作 `operator()`,使 `ref` 包装类可以被调用;
- `sig_ex`: `signals2::signal` 的包装类,增加了 `operator+=` 操作符重载,可以像 C# 语言中那样用 `+=` 连接插槽。

第 12 章

并发编程

C++中没有语言级别的并发支持，因为在 C++诞生很久以后线程等用于并发操作的概念才出现（POSIX 线程标准制定于 1995 年）。但在现在随着拥有多 CPU、多内核的计算机的大量出现，C++特别是 C++标准没有定义并发操作的规范就显得有些过时了。

作为 C++标准库的后备，Boost 实现了数个用于并发编程的库，它们是高度可用的，而且结构良好，适合于最后的标准化，无论从哪个方面来说都要强于其他非标准的第三方并发程序库。

本章讨论 Boost 库中两个用于并发编程的组件。首先是 thread 库，它为 C++增加了可移植的线程处理能力，然后是一个用于同步和异步 IO 操作的功能强大的库——asio，它使用了前摄器模式，可以处理串口、网络通信，而且有望成为 C++标准底层通信库。

并发编程是一个很广泛的话题，同时又是一门很复杂的技术，因此很难在一章的篇幅中就能够把相关的知识都介绍清楚，本章仅介绍 Boost 库在并发编程领域的一部分内容，起到入门的作用。

12.1 thread

thread 库为 C++增加了线程处理的能力，它提供简明清晰的线程、互斥量等概念，可以很容易地创建多线程应用程序。thread 库也是高度可移植的，它支持使用最广泛的 Windows 和 POSIX 线程，用它编写的代码不需要修改就可以在 Windows、UNIX 等操作系统上编译运行。

本章只讲述 thread 库的使用，而不讲述如何编写一个（好的）多线程程序，那需要另外一本书。

12.1.1 编译 thread 库

thread 库需要 date_time 库支持, 因此必须先编译 date_time 库, 请参考 2.5.1 小节。

thread 库需要编译才能使用, bjam 命令如下:

```
bjam -toolset=msvc -with-thread -build-type=complete stdlib=stlport stage
```

如果要使用工程中嵌入源码的方式, 可以直接在 cpp 文件中包含 thread 库的实现代码, 如下:

```
//tprebuild.cpp
#define BOOST_DATE_TIME_SOURCE //thread库需要使用date_time库
#define BOOST_THREAD_NO_LIB
#include <boost/thread.hpp>

#ifdef _MSC_VER //Windows 系统的线程
extern "C" void tss_cleanup_implemented(void) {} //一个必须的函数
#include <libs/thread/src/win32/thread.cpp>
#include <libs/thread/src/win32/tss_dll.cpp>
#include <libs/thread/src/win32/tss_pe.cpp>
#else //unix 系统相关的实现文件
#include <libs/thread/src/pthread/thread.cpp>
#include <libs/thread/src/pthread/once.cpp>
#endif
```

预编译源文件 tprebuild.cpp 中定义了一个 extern "C" 空函数 tss_cleanup_implemented(void)。这是因为当前 Windows 环境下的 thread 库没有实现自动 tss (线程本地存储) 清理功能, 需要用户来完成。如果链接时没有这个函数就会发生 link 错误, 因此嵌入编译需自行定义该函数来阻止 link 错误。通常可以用一个简单的空实现, 也可以自行定义其他清理代码。

12.1.2 使用 thread 库

thread 位于名字空间 boost, 为了使用 thread 组件需要包含头文件 <boost/thread.hpp>。

因为 thread 库使用了 date_time 库, 所以嵌入源码编译时需定义两个宏: BOOST_DATE_TIME_SOURCE 和 BOOST_THREAD_NO_LIB, 或者直接在工程中定义 BOOST_ALL_NO_LIB, 即:

```
#define BOOST_DATE_TIME_SOURCE
#define BOOST_THREAD_NO_LIB
#include <boost/thread.hpp>
using namespace boost;
```

在 Linux/UNIX 下链接 thread 库时还需要使用 -lpthread 选项来链接 POSIX 线程库。

12.1.3 时间功能

在多线程编程时经常要用到超时处理，需要表示时间的概念。thread 库直接利用 date_time 库 (2.5 小节, 27 页) 提供了对时间的支持，可以使用 millisec/milliseconds、microsec/microseconds 等时间长度类表示超时的时间，或者用 ptime 表示某个确定的时间点。例如：

```
this_thread::sleep(posix_time::seconds(2));          //睡眠 2 秒钟
cout << "sleep 2 seconds" << endl;
```

为了更好地表述时间的线程相关含义，thread 库重新定义了一个新的时间类型 system_time，它是 posix_time::ptime 的同义词，即：

```
typedef boost::posix_time::ptime system_time;
```

同时 thread 库也提供了一个自由函数 get_system_time()，它调用 microsec_clock 类方便地获得当前的 UTC 时间值。

此外，thread 库还有一个非常简单的 xtime 结构，它简单地包装了 ptime 类，也具有一定的时间处理能力，但功能较少，主要供 thread 库内部使用，对库用户不推荐。

12.1.4 互斥量

互斥量是一种用于线程同步的手段，它可以在多线程编程中防止多个线程同时操作共享资源（或称临界区）。一旦一个线程锁住了互斥量，那么其他线程必须等待它解锁互斥量后才能再访问共享资源。

thread 提供了七种互斥量类型（实际上只有五种），分别是：

- mutex: 独占式的互斥量，是最简单最常用的一种互斥量类型；
- try_mutex: 它是 mutex 的同义词，为了与兼容以前的版本而提供；
- timed_mutex: 它也是独占式的互斥量，但提供超时锁定功能；
- recursive_mutex: 递归式互斥量，可以多次锁定，相应地也要多次解锁；
- recursive_try_mutex: 它是 recursive_mutex 的同义词，为了与兼容以前的版本而提供；
- recursive_timed_mutex: 它也是递归式互斥量，基本功能同 recursive_mutex，但提供超时锁定功能；
- shared_mutex: multiple-reader/single-writer 型的共享互斥量（又称读写锁）。

类摘要

这些互斥量除了互斥功能不同外基本接口都很接近，具有类似下面的声明形式：

```
class mutex: boost::noncopyable
{
public:
    void lock();
    bool try_lock();
    void unlock();

    bool timed_lock(system_time const & abs_time);
    template<typename TimeDuration>
    bool timed_lock(TimeDuration const & relative_time);

    typedef unspecified-type scoped_lock;
    typedef unspecified-type scoped_try_lock;
};
```

`mutex` 对象在创建后就表示了一个互斥量，成员函数 `lock()` 用于线程阻塞等待直至获得互斥量的所有权（即锁定）；`try_lock()` 尝试锁定互斥量，如果锁定成功返回 `true`，否则返回 `false`，它是非阻塞的；当线程使用完共享资源后应该及时使用 `unlock()` 解除对互斥量的锁定。

成员函数 `timed_lock()` 只属于 `timed_mutex` 和 `recursive_timed_mutex`，它的行为结合了 `lock()` 和 `try_lock()`，阻塞等待一定的时间试图锁定互斥量，如果时间到还未锁定则返回 `false`。等待的时间可以是绝对时间（一个 UTC 时间点），也可以是从当前开始的相对时间（时间长度）。

互斥量的用法

`mutex` 的基本用法如下：

```
mutex mu;                                //声明一个互斥量对象
try
{
    mu.lock();                            //锁定互斥量
    cout << "some operations" << endl;    //临界区操作
    mu.unlock();                          //解锁互斥量
}
catch (...)                               //必须使用 try-catch 块保证解锁互斥量
{
    mu.unlock();
}
```

直接使用 `mutex` 的成员函数来锁定互斥量不够方便，而且在发生异常导致退出作用域等情况

下很可能会忘记解除锁定；因此 thread 库又提供了一系列 RAII 型的 `lock_guard` 类，由于辅助锁定互斥量。它们在构造时锁定互斥量，在析构时自动解锁，从而保证了互斥量的正确操作，避免遗忘解锁，就像是一个智能指针。

`mutex` 类使用内部类型定义 `scoped_lock` 和 `scoped_try_lock` 定义了两种 `lock_guard` 对象，分别对应执行 `lock()` 和 `try_lock()`。

使用 `lock_guard` 类可以取消麻烦的 try-catch 块，上面的代码可以改为：

```
mutex mu;
mutex::scoped_lock lock(mu);           //使用 RAII 型的 lock_guard
cout << "some operations" << endl;
```

互斥量实例

下面我们使用 `mutex` 实现一个原子操作的计数器 `basic_atom`，它可以安全地在多线程环境下正确计数^①：

```
template<typename T>
class basic_atom: noncopyable
{
private:
    T n;
    typedef mutex mutex_t;           //互斥量类型定义
    mutex_t mu;
public:
    basic_atom(T x = T()):n(x){}      //构造函数
    T operator++()                   //前置式递增操作符
    {
        mutex_t::scoped_lock lock(mu); //锁定互斥量
        return ++n;
    }
    operator T(){ return n;}         //类型转换操作符定义
};
```

`basic_atom` 是一个模板类，因此它可以配合模板参数提供不同范围的计数，并且它提供了隐式类型转换操作，用起来就像是一个普通整数，例如：

```
typedef basic_atom<int> atom_int;
atom_int x;
cout << ++x;
```

① Boost 库在头文件 `<boost/detail/atomic_count.hpp>` 也提供了一个原子计数器——`atomic_count`，它使用 `long` 进行线程安全的递增递减计数，读者可参考。

12.1.5 线程对象

`thread` 类是 `thread` 库的核心类，负责启动和管理线程对象，在概念和操作上都与 POSIX 线程很相似，它的类摘要如下：

```
class thread
{
public:

    //构造函数
    thread();
    template <class F> explicit thread(F f);
    template <class F, class A1, class A2, ...>
    thread(F f, A1 a1, A2 a2, ...);

    //线程管理
    bool joinable() const;
    void join();
    bool timed_join(const system_time& wait_until);
    template<typename TimeDuration>
    bool timed_join(TimeDuration const& rel_time);
    void detach();
    void interrupt();
    bool interruption_requested() const;

    //静态成员函数
    static void yield();
    static void sleep(const system_time& xt);
    static unsigned hardware_concurrency();

    // 支持各种比较操作符
    id get_id() const;
    bool operator==(const thread& other) const;
};
```

在使用 `thread` 对象时需要注意它是不可拷贝的，虽然它没有从 `boost::noncopyable` 继承，但 `thread` 内部把拷贝构造函数和赋值操作都声明为私有的，不能对它进行赋值或者拷贝构造。

`thread` 通过特别的机制支持转移语义，因此我们可以编写创建线程的工厂函数，封装 `thread` 的创建细节，返回一个 `thread` 对象。例如，下面的模板函数可以创建一个 `thread` 对象：

```
template<typename F>
thread make_thread(F f)
{ return thread(f); }
```


12.1.6 创建线程

从某种程度来说，线程就是在进程的另一个空间里运行的一个函数，因此线程的创建需要传递给 thread 对象一个无参的可调用物（函数或函数对象），它必须具有 operator() 以供线程执行。

如果可调用物不是无参的，那么 thread 的构造函数也支持直接传递所需的参数，这些参数将被拷贝并在发生调用时传递给函数。这是一个非常体贴方便的重载构造函数，比传统的使用 void* 来传递参数要好很多。thread 的构造函数支持最多传递九个参数，这通常足够用了。

在传递参数时需要注意，thread 使用的是参数的拷贝，因此要求可调用物和参数类型都支持拷贝。如果希望传递给线程引用值就需要使用 ref 库（参见 11.2 小节，422 页）进行包装，同时必须保证被引用的对象在线程执行期间一直存在，否则会引发未定义行为。

启动线程

当成功创建了一个 thread 对象后，线程就立刻开始执行，thread 不提供类似 start()、begin() 那样的方法。^①

假设我们有如下的一个函数，它向标准输出流打印字符串：

```
mutex io_mu;                                     //io 流是个共享资源，不是线程
                                                //安全的，需要锁定

void printing(atom_int& x, const string& str)
{
    for (int i = 0; i < 5; ++i)
    {
        mutex::scoped_lock lock(io_mu);        //锁定 io 流操作
        cout << str << ++x << endl;
    }
}
```

那么我们可以这样使用 thread 对象：

```
int main()
{
    atom_int x;                                   //原子操作的计数器

    //使用临时 thread 对象启动线程
    thread(printing, ref(x), "hello");           //向函数传递多个参数
    thread(printing, ref(x), "boost");          //使用 ref 库传递引用
}
```

① 实际上，thread 类有一个私有成员函数 start_thread()，它负责启动线程。

```
this_thread::sleep(posix_time::seconds(2)); //等待 2 秒钟
}
```

请读者注意上面的代码中的最后一句。在当线程启动后我们必须调用 `sleep()` 来等待线程执行结束，否则会因为 `main()` 的 `return` 语句导致主线程结束，而其他的线程还没有机会运行而一并结束。

通常不应该使用这种“死”等线程结束的方法，因为不可能精确地知道线程会执行多少时间，我们需要用其他更好的方法来等待线程结束。

join 和 timed_join

`thread` 的成员函数 `joinable()` 可以判断 `thread` 对象是否标识了一个可执行的线程体。如果 `joinable()` 返回 `true`，我们就可以调用成员函数 `join()` 或者 `timed_join()` 来阻塞等待线程执行结束。两者的区别如下：

- `join()` 一直阻塞等待，直到线程结束；
- `timed_join()` 阻塞等待线程结束，或者阻塞等待一定的时间段，然后不管线程是否结束都返回。注意，它不必阻塞等待指定的时间长度，如果在这段时间里线程运行结束，即使时间未到它也会返回。

使用 `join()` 可以这样操作 `thread` 对象：

```
atom_int x;
thread t1(printing, ref(x), "hello");
thread t2(printing, ref(x), "boost");

t1.timed_join(posix_time::seconds(1)); //最多等待 1 秒然后返回
t2.join();                             //等待 t2 线程结束再返回，不管执行多少时间
```

与线程执行体分离

可以使用成员函数 `detach()` 将 `thread` 与线程执行体手动分离，此后 `thread` 对象不代表任何线程体，失去对线程体的控制。例如：

```
thread t1(printing, ref(x), "hello"); //启动线程
t1.detach();                          //与线程执行体分离，但线程继续运行
```

当 `thread` 与线程执行体分离时，线程执行体将不受影响地继续执行，直到运行结束，或者随主线程一起结束。

当线程执行完毕或者 `thread` 对象被销毁时，`thread` 对象也会自动与线程执行体分离，因此，当不需要操作线程体时，我们可以使用临时对象来启动一个线程，就像之前示范的那样。

使用 bind 和 function

有时在 thread 的构造函数中写传递给调用函数的参数很麻烦，尤其是在使用大量线程对象的时候。这时我们可以使用 Boost 的 bind 和 function 库：bind 库可以把函数所需的参数绑定到一个函数对象，而 function 则可以存储 bind 表达式的结果，供程序以后使用。

例如：

```
thread t3(bind(printing, ref(x), "thread"));           //bind 表达式
function<void()> f = bind(printing, 5, "mutex");
thread(f);                                           //使用 function 对象
```

12.1.7 操作线程

通常情况下一个非空的 thread 对象唯一地标识了一个可执行的线程体，是 joinable() 的，成员函数 get_id() 可以返回线程 id 对象。

线程 id 提供了完整的比较操作符和流输出操作，因此可以被放入标准容器用来管理线程，thread 类也通过线程 id 支持线程间的比较操作。例如：

```
thread t1(...), t2(...);                          //创建两个线程对象
cout << t1.get_id() << endl;                       //输出 t1 的 id
assert(t1 != t2);                                  //比较两个线程对象
t1.detach();                                       //分离 t1 代表的线程执行体，但线程仍然继续执行
assert(t1.get_id() == thread::id()); //t1 不再标识任何线程
```

thread 类还提供了三个很有用的静态成员函数：yield()、sleep() 和 hardware_concurrency()，它们用来在线程中完成一些特殊的工作。

- yield() 函数指示当前线程放弃时间片，允许其他的线程运行。
- sleep() 让线程睡眠等待一小段时间，注意它要求参数是一个 system_time UTC 时间点而不是时间长度。
- hardware_concurrency() 可以获得硬件系统可并行的线程数量，即 CPU 数量或者 CPU 内核数量，如果无法获取信息则返回 0。

例如，下面的代码令当前线程睡眠 1 秒钟，然后输出可并行的线程数量：

```
thread::sleep(get_system_time() + posix_time::seconds(1));
cout << thread::hardware_concurrency() << endl;
```

thread 库也在子名字空间 this_thread 里提供了 3 个自由函数——get_id()、yield() 和 sleep() 用于操作当前线程。它们的功能同 thread 类的同名函数，分别用来获得线程 id、

放弃时间片和睡眠等待，但 `this_thread` 的 `sleep()` 函数不仅可以使⽤绝对的 UTC 时间点，也可以使⽤时间长度。

例如：

```
this_thread::sleep(posix_time::seconds(2));           //睡眠 2 秒钟
cout << this_thread::get_id();
this_thread::yield();
```

12.1.8 中断线程

`thread` 的成员函数 `interrupt()` 允许正在执⾏的线程被中断，被中断的线程会抛出一个 `thread_interrupted` 异常，它是一个空类，不是 `std::exception` 或者 `boost::exception` 的子类。`thread_interrupted` 异常应该在线程执⾏函数里捕获并处理，如果线程不处理这个异常，那么默认的动作是中止线程。

下面的代码定义了一个函数 `to_interrupt()`，它的行为类似之前的 `printing()` 函数，但它使⽤ `this_thread::sleep()` 睡眠 1 秒钟再输出字符串：

```
void to_interrupt(atom_int& x, const string& str)
try
{
    for (int i = 0; i < 5; ++i)
    {
        this_thread::sleep(posix_time::seconds(1)); //睡眠 1 秒钟
        mutex::scoped_lock lock(io_mu);           //锁定 io 流操作
        cout << str << ++x << endl;
    }
}
catch(thread_interrupted& ) //捕获中断异常
{
    cout << "thread_interrupted" << endl;          //显示消息
}
```

`main()` 启动这个线程，等待两秒钟，然后中断线程的执⾏：

```
int main()
{
    atom_int x;
    thread t(to_interrupt, ref(x), "hello");
    this_thread::sleep(posix_time::seconds(2)); //睡眠 2 秒钟
    t.interrupt();                               //要求线程中断执⾏
    t.join();                                     //因为线程已经中断，
                                                //所以 join() 立即返回
}
```

程序的运行的结果如下:

```
hello1
hello2
thread_interrupted
```

线程的中断点

线程不是在任意时刻都可以被中断的。如果我们将 `to_interrupt()` 函数中的 `sleep()` 睡眠等待去掉, 那么即使在主线程中调用 `interrupt()` 线程也不会被中断。

`thread` 库预定义了若干个线程的中断点, 只有当线程执行到中断点的时候才能被中断, 一个线程可以拥有任意多个中断点。

`thread` 库预定义了共 9 个中断点, 它们都是函数, 如下:

- `thread::join();`
- `thread::timed_join();`
- `condition_variable::wait();`
- `condition_variable::timed_wait();`
- `condition_variable_any::wait();`
- `condition_variable_any::timed_wait();`
- `thread::sleep();`
- `this_thread::sleep();`
- `this_thread::interruption_point();`

这些中断点中的前 8 个都是某种形式的等待函数, 表明线程在阻塞等待的时候可以被中断。而最后一个位于子名字空间 `this_thread` 的 `interruption_point()` 则是一个特殊的中断点函数, 它并不等待, 只是起到一个标签的作用, 表示线程执行到这个函数所在的语句就可以被中断。

例如, 我们把 `to_interrupt()` 函数改为使用 `interruption_point()`:

```
void to_interrupt(atom_int& x, const string& str)
try
{
    for (int i = 0; i < 5; ++i)
    {
        mutex::scoped_lock lock(io_mu); //锁定 io 流操作
    }
}
```

```

        cout << str << ++x << endl;
        this_thread::interruption_point();           //这里允许中断
    }
}
catch(thread_interrupted& )                          //捕获中断异常
{...}

```

main() 不再等待一段时间，而是启动线程后立即调用 interrupt() 来中断线程：

```

int main()
{
    atom_int x;
    thread t(to_interrupt, ref(x), "hello");          //启动线程
    t.interrupt();                                     //然后立即中断线程
    t.join();
}

```

那么线程将会输出一条“hello1”字符串后遇到中断点，并且被 thread 对象所中断执行。

启用/禁用线程中断

缺省情况下线程都是允许中断的，但 thread 库允许控制线程的中断行为。

thread 库在子名字空间 this_thread 提供了一组函数和类来共同完成线程的中断启用和禁用：

- interruption_enabled() 函数检测当前线程是否允许中断；
- interruption_requested() 函数检测当前线程是否被要求中断；
- 类 disable_interruption 是一个 RAII 类型的对象，它在构造时关闭线程的中断，析构时自动恢复线程的中断状态。

在 disable_interruption 的生命期内线程始终是不可中断的，除非使用了 restore_interruption 对象。

- restore_interruption 只能在 disable_interruption 的作用域内使用，它在构造时临时打开线程的中断状态，在析构时又关闭中断状态。

仍然以之前的 to_interrupt() 函数为例，我们为它增加中断的启用和禁用：

```

void to_interrupt(atom_int& x, const string& str)
try
{
    using namespace this_thread;                      //打开 this_thread 名字空间
    assert(interruption_enabled());                   //此时允许中断
}

```

```

for (int i = 0; i < 5; ++i)
{
    disable_interruption di;                //关闭中断
    assert(!interruption_enabled());         //此时中断不可用
    mutex::scoped_lock lock(io_mu);          //锁定 io 流操作
    cout << str << ++x << endl;
    cout << this_thread::interruption_requested() << endl;
    this_thread::interruption_point();        //中断点被禁用

    restore_interruption ri(di);              //临时恢复中断
    assert(interruption_enabled());           //此时中断可用
    cout << "can interrupted" << endl;
    cout << this_thread::interruption_requested() << endl;
    this_thread::interruption_point();        //可被中断
}                                              //离开作用域, di/ri 都被析构,
                                              //恢复线程最初的可中断状态

assert(interruption_enabled());              //此时允许中断
}
catch(thread_interrupted& )
{...}

```

main() 函数的运行结果如下:

```

hello!
1
can interrupted
1
thread_interrupted

```

运行结果中的两行“1”是函数 `this_thread::interruption_requested()` 的输出结果, 它表明线程已经收到了中断请求, 但因为第一次线程不允许中断, 故线程继续执行, 直到 `restore_interruption` 对象临时恢复了可中断的时候线程在被中断。

12.1.9 线程组

thread 库提供类 `thread_group` 用于管理一组线程, 就像是一个线程池, 它内部使用 `std::list<thread*>` 来容纳创建的 thread 对象, 类摘要如下:

```

class thread_group: private noncopyable
{
public:
    template<typename F>
    thread* create_thread(F threadfunc);
    void add_thread(thread* thrd);
    void remove_thread(thread* thrd);
    void join_all();
    void interrupt_all();

```

```
int size() const;
};
```

thread_group 的接口很小，用法也很简单。

成员函数 create_thread() 是一个工厂函数，可以创建 thread 对象并运行线程，同时加入内部的 list。我们也可以在 thread_group 外部创建线程对象，使用 add_thread() 加入到线程组。

如果不需要某个线程，remove_thread()，可以删除 list 里的 thread 对象。

join_all() 和 interrupt_all() 用来对 list 里的所有线程对象操作，等待或者中断这些线程。

例如：

```
thread_group tg;
tg.create_thread(bind(printing, ref(x), "C++"));
tg.create_thread(bind(printing, ref(x), "boost"));
tg.join_all();
```

使用 thread_group，我们可以为程序建立一个类似于全局线程池的对象，统一管理程序中使用的 thread，它可以使用 4.6 小节的单件库变成一个单件，以提供一个全局的访问点，例如：

```
typedef singleton_default<thread_group> thread_pool;
thread_pool::instance().create_thread(...);
```

不过 Boost 的单件库不提供完全的线程安全，如果你要在多线程中操作这个单件线程组需要小心，或者自己封装一个线程安全的单件对象。

12.1.10 条件变量

条件变量是 thread 库提供的另一种用于等待的同步机制，可以实现线程间的通信，它必须与互斥量配合使用，等待另一个线程中某个事件的发生（满足某个条件），然后线程才能继续执行。

thread 库提供两种条件变量对象 condition_variable 和 condition_variable_any，一般情况下我们应该使用 condition_variable_any，它能够适应更广泛的互斥量类型。

condition_variable_any 的类摘要如下：

```
class condition_variable_any
{
public:
    void notify_one();
    void notify_all();
```



```

template<typename lock_type>
void wait(lock_type& lock);

template<typename lock_type,typename predicate_type>
void wait(lock_type& lock,predicate_type predicate);

template<typename lock_type,typename duration_type>
bool timed_wait(lock_type& lock,duration_type const& rel_time);
};

```

条件变量的使用方法很简单：

拥有条件变量的线程先锁定互斥量，然后循环检查某个条件，如果条件不满足，那么就调用条件变量的成员函数 `wait()` 等待直至条件满足。其他线程处理条件变量要求的条件，当条件满足时调用它的成员函数 `notify_one()` 或 `notify_all()`，以通知一个或者所有正在等待条件变量的线程停止等待继续执行。

条件变量的用法

我们使用标准库的容器适配器 `stack` 来实现一个用于生产者-消费者模式的后进先出型缓冲区，以演示条件变量的用法。

缓冲区 `buffer` 使用了两个条件变量 `cond_put` 和 `cond_get`，分别用于处理 `put` 动作和 `get` 动作，如果缓冲区满则 `cond_put` 持续等待，当 `cond_put` 得到通知（缓冲区不满）时线程写入数据，然后通知 `cond_get` 条件变量可以取数据。`cond_get` 的处理流程与 `cond_put` 类似。具体实现代码如下：

```

#include <stack>
class buffer
{
private:
    mutex mu; //互斥量, 配合条件变量使用
    condition_variable_any cond_put; //写入条件变量
    condition_variable_any cond_get; //读取条件变量
    stack<int> stk; //缓冲区对象
    int un_read,capacity;
    bool is_full() //缓冲区满判断
    {
        return un_read == capacity;
    }
    bool is_empty() //缓冲区空判断
    {
        return un_read == 0 ;
    }
public:
    buffer(size_t n):un_read(0),capacity(n){} //构造函数
    void put(int x) //写入数据
    {

```

```

{
    mutex::scoped_lock lock(mu); //开始一个局部域
    while(is_full()) //锁定互斥量
    { //检查缓冲区是否满
        { //局部域, 锁定 cout 输出一条信息
            mutex::scoped_lock lock(io_mu);
            cout << "full waiting... " << endl;
        }
        cond_put.wait(mu); //条件变量等待
    } //条件满足, 停止等待
    stk.push(x); //压栈, 写入数据
    ++un_read;
} //解锁互斥量, 条件变量的通知不需要互斥量锁定
cond_get.notify_one(); //通知可以读取数据
}
void get(int *x) //读取数据
{
    { //局部域开始
        mutex::scoped_lock lock(mu); //锁定互斥量
        while(is_empty()) //检查缓冲区是否空
        {
            { //向 cout 输出信息
                mutex::scoped_lock lock(io_mu);
                cout << "empty waiting... " << endl;
            }
            cond_get.wait(mu); //条件变量等待
        } //条件满足, 停止等待
        --un_read;
        *x = stk.top(); //读取数据
        stk.pop(); //弹栈
    }
    cond_put.notify_one(); //通知可以写入数据
}
};

```

然后我们定义两个生产者和消费者函数:

```

buffer buf(5); //一个缓冲区对象
void producer(int n) //生产者
{
    for (int i = 0; i < n; ++i)
    {
        { //输出信息
            mutex::scoped_lock lock(io_mu);
            cout << "put " << i << endl;
        }
        buf.put(i); //写入数据
    }
}

```

```

}
void consumer( int n)                //消费者
{
    int x;
    for (int i = 0; i < n; ++i)
    {
        buf.get(&x);                //读取数据
        mutex::scoped_lock lock(io_mu);
        cout << "get " << x << endl;
    }
}

```

现在我们可以把缓冲区与生产者、消费者组合起来：

```

int main()
{
    thread t1(producer, 20);        //一个生产者线程
    thread t2(consumer, 10);        //两个消费者线程
    thread t3(consumer, 10);

    t1.join();                      //等待 t1 线程结束
    t2.join();                      //等待 t2 线程结束
    t3.join();                      //等待 t3 线程结束
}

```

程序运行后会交替出现 put、get 以及 waiting 信息，显示在多线程环境下生产者和消费者都正确地执行了预定的功能。

读者也可以使用标准库的容器适配器 queue 或 7.5 小节 (271 页) 介绍的 circular_buffer 类，来实现先进先出型和循环队列型缓冲区。

其他用法

条件变量的 wait() 函数有一个有用的重载形式：wait(lock_type& lock, predicate_type predicate)，它比普通的形式多接受一个谓词函数（或函数对象），当谓词 predicate 不满足时持续等待，相当于：

```

while(!predicate())
    wait(lock);

```

使用这个重载形式可以写出更简洁清晰的代码，通常需要配合 bind 来简化谓词函数的编写。例如，buffer 类的两个条件变量的等待可以改写成：

```

cond_put.wait(mu, !bind(&buffer::is_full, this));
cond_get.wait(mu, !bind(&buffer::is_empty, this));

```

在这里我们不得不对 bind 表达式使用逻辑非操作符（叹号），因为 wait() 的语义与定义的

`is_full()`、`is_empty()`正好相反。如果要使用更自然的语义则需要重新定义条件判断函数, 改为使用 `not_full()` 和 `not_empty()`。

`thread` 库还提供了一个 `typedef`: `condition`, 它是 `condition_variable_any` 的同义词, 用于对旧代码提供兼容。但它并不在头文件 `<boost/thread.hpp>` 中, 如果非要使用这个 `typedef`, 需要包含头文件 `<boost/thread/condition.hpp>`。

12.1.11 共享互斥量

共享互斥量 `shared_mutex` 不同于 `mutex` 和 `recursive_mutex`, 它允许线程获取多个共享所有权和一个专享所有权, 实现了读写锁的机制, 即多个读线程一个写线程。

`shared_mutex` 具有 `mutex` 的全部功能, 可以把它像 `mutex` 一样使用 `lock()` 和 `unlock()` 来获得专享所有权, 但它代价要比 `mutex` 高很多。如果要获得共享所有权需要使用 `lock_shared()` 或 `try_lock_shared()`, 相应地要使用 `unlock_shared()` 来释放共享所有权。

`shared_mutex` 没有提供内部的 `lock_guard` 类型定义, 因此在使用 `shared_mutex` 时我们必须直接使用 `lock_guard` 对象, 读锁定时使用 `shared_lock<shared_mutex>`, 写锁定时使用 `unique_lock<shared_mutex>`。

我们使用代码来示范 `shared_mutex` 的用法。首先定义一个读写数据类 `rw_data`, 它使用 `shared_mutex` 实现多个读者一个作者:

```
class rw_data
{
private:
    int m_x;                //用于读写的数据
    shared_mutex rw_mu;     //共享互斥量
public:
    rw_data():m_x(0){}     //构造函数
    void write()            //写数据
    {
        unique_lock<shared_mutex> ul(rw_mu); //写锁定
        ++ m_x;
    }
    void read(int *x)       //读数据
    {
        shared_lock<shared_mutex> sl(rw_mu); //读锁定
        *x = m_x;
    }
};
```

然后我们定义两个用于线程执行的函数, 分别执行多次的读写操作:

```
void writer(rw_data &d)    //写线程
```

```

{
    for (int i = 0; i < 20; ++i)
    {
        this_thread::sleep(posix_time::millisec(10));    //睡眠
        d.write();
    }
}

void reader(rw_data &d)    //读线程
{
    int x;
    for (int i = 0; i < 10; ++i)
    {
        this_thread::sleep(posix_time::millisec(5));    //睡眠
        d.read(&x);
        mutex::scoped_lock lock(io_mu);
        cout << "reader:" << x << endl;
    }
}

```

最后我们使用 `thread_group` 启动多个读写线程:

```

int main()
{
    rw_data d;
    thread_group pool;    //线程组

    pool.create_thread(bind(reader, ref(d)));    //读线程 1
    pool.create_thread(bind(reader, ref(d)));    //读线程 2
    pool.create_thread(bind(reader, ref(d)));    //读线程 3
    pool.create_thread(bind(reader, ref(d)));    //读线程 4
    pool.create_thread(bind(writer, ref(d)));    //写线程 1
    pool.create_thread(bind(writer, ref(d)));    //写线程 2

    pool.join_all();    //等待线程结束
}

```

运行程序可以看到多个读者可以同时获得数据。

12.1.12 future

很多情况下线程不仅仅要执行一些工作，它还可能要返回一些计算结果，一个简单的解决办法是被调线程操作一个全局变量，主调线程不断地检查是否有值或者阻塞等待，这种解法相当笨拙。

`thread` 库使用 `future` 范式提供了一种异步操作线程返回值的方法，因为这个返回值在线程开始执行时还是不可用的，是一个“未来”的“期待值”，所以被称为 `future`（期货）。

future 使用 packaged_task 和 promise 两个模板类来包装异步调用, 用 unique_future 和 shared_future 来获取异步调用结果(即 future 值)。接下来我们先使用 packaged_task 和 unique_future 这两个最常用也是最易用的类来阐述 future 的用法。

packaged_task 和 unique_future

packaged_task 好像是一个 reference_wrapper 或者 function 对象, 它提供 operator(), 包装了一个可回调物, 然后它就可以被任意的线程调用执行, 最后的 future 值可以用成员函数 get_future() 获得。

unique_future 用来存储 packaged_task 异步计算得到的 future 值, 它只能持有结果的唯一的一个引用。成员函数 wait() 和 timed_wait() 的行为类似 thread.join(), 可以阻塞等待 packaged_task 的执行, 直至获得 future 值。成员函数 is_ready()、has_value() 和 has_exception() 分别用来测试 unique_future 是否可用, 是否有值和是否发生了异常, 如果一切正常, 那么可以使用 get() 获得 future 值。

下面的代码示范了 future 特性的用法, 使用 packaged_task 和 unique_future 计算菲波拉契数列的值:

```
int fab(int n)                                //递归计算菲波拉契数列
{
    if (n == 0 || n == 1)
        return 1;
    return fab(n-1) + fab(n-2);
}
int main()
{
    //声明 packaged_task 对象, 用模板参数指明返回值类型
    //packaged_task 只接受无参函数, 因此需要使用 bind
    packaged_task<int> pt(bind(fab, 10));

    //声明 unique_future 对象, 接受 packaged_task 的 future 值,
    //同样要用模板参数指明返回值类型
    unique_future<int> uf = pt.get_future();

    //启动线程计算, 必须使用 boost::move() 来转移 packaged_task 对象
    //因为 packaged_task 是不可拷贝的
    thread(boost::move(pt));
    uf.wait();                                //unique_future 等待计算结果
    assert(uf.is_ready() && uf.has_value());
    cout << uf.get();                          //输出计算结果 89
}
```

使用多个 future 对象

为了支持多个 future 对象的使用, future 还提供 `wait_for_any()` 和 `wait_for_all()` 两个自由函数, 它们可以阻塞等待多个 future 对象, 直到任意一个或者所有 future 对象都可用 (`is_ready()`)。这两个函数有多个重载形式, 可以接受一对表示 future 容器区间的迭代器或者最多 5 个 future 对象。例如:

```
#include <boost/array.hpp>
#include <boost/foreach.hpp>
int main()
{
    typedef packaged_task<int> pti_t;           //类型定义
    typedef unique_future<int> ufi_t;          //类型定义

    array<pti_t, 5> ap;                         //使用 boost::array
    array<ufi_t, 5> au;
    for (int i = 0; i < 5; ++i)                //启动五个 future 调用
    {
        ap[i] = pti_t(bind(fab, i + 10));      //计算 future
        au[i] = ap[i].get_future();            //获取 future
        thread(move(ap[i]));                  //启动线程开始计算
    }
    wait_for_all(au.begin(), au.end());        //等待所有计算结束
    BOOST_FOREACH(ufi_t& uf, au)              //foreach 循环打印计算结果
    {
        cout << uf.get() << endl;            //输出 89,144,233,377,610
    }
}
```

如果使用 `wait_for_any()`, 那么等待和输出的代码可以是:

```
wait_for_any(au[3], au[4], au[2]);            //等待任意一个 future 值
BOOST_FOREACH(ufi_t& uf, au)
{
    if(uf.is_ready() && uf.has_value())        //检测哪个 future 有值
    {      cout << uf.get() << endl;      }
}
```

promise

promise 也用于处理异步调用返回值, 但它不同于 `packaged_task`, 不能包装一个函数, 而是包装一个值, 这个值可以作为函数的输出参数, 适用于从函数参数返回值的函数。

promise 的用法与 `packaged_task` 类似, 在线程中用 `set_value()` 设置要返回的值, 用成员函数 `get_future()` 获得 future 值赋给 future 对象。

示范 promise 用法的代码如下:

```
void fab2(int n, promise<int> *p)           //使用 promise 作为输出参数
{ p->set_value(fab(n)); }
int main(int argc, char* argv[])
{
    promise<int> p;                         //promise 变量
    unique_future<int> uf = p.get_future(); //赋值 future 对象

    thread(fab2, 10, &p);                  //启动计算线程

    uf.wait();                             //等待 future 计算结果
    cout << uf.get() << endl;
}
```

12.1.13 高级议题

本小节讨论关于 thread 库的一些高级议题。

使用 thread 库的部分功能

有的时候我们不需要使用 thread 库的全部功能,比如仅使用 mutex 提供互斥操作,把代码嵌入到其他多线程程序中。那么我们完全可以不用编译 thread 库,仅包含哪些需要功能的头文件即可,这样可以获得更好的可移植性。

thread 库中不需要编译就可以使用的头文件如下:

```
#include <boost/thread/condition_variable.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/recursive_mutex.hpp>
#include <boost/thread/thread_time.hpp>      //需编译 date_time 库
#include <boost/thread/shared_mutex.hpp>
#include <boost/thread/barrier.hpp>
```

如果仅仅想使用互斥量功能,那么还可以使用未文档化的超轻量级互斥量 `boost::detail::lightweight_mutex`, 它位于头文件 `<boost/detail/lightweight_mutex.hpp>`。`lightweight_mutex` 同样是可移植的,而且具有内部类型定义 `scoped_lock` 提供互斥量的自动锁定与释放。

锁定函数

thread 库在名字空间 boost 中提供了两个自由函数 `lock()` 和 `try_lock()`, 可以一次锁定多个互斥量,而且不会出现死锁,它们的声明如下:

```
template<typename Lockable1,...>           //最多支持 5 个互斥量
void lock(Lockable1& l1,...);
```



```
template<typename ForwardIterator>           //使用迭代器遍历互斥量容器
void lock(ForwardIterator begin, ForwardIterator end);
```

//try_lock()的形式与lock()类似

lock()不具有lock_guard的退出作用域自动解锁的特性,因此在锁定互斥量后必须手工解锁互斥量(没有对应的unlock()函数),但lock()函数保证:锁定后发生异常时解除对互斥量的锁定。

lock()函数的用法如下:

```
lock(mu1, mu2);           //锁定两个互斥量
...;                      //临界区操作
mu1.unlock();             //逐个解除锁定
mu2.unlock();
```

仅初始化一次

为了保证在多线程环境中某些仅要求被调用一次的函数被正确调用(通常是一些用于初始化的函数),thread库提供了仅初始化一次机制,使多个线程在操作初始化函数时只能有一个线程成功执行,不会发生错误。

这个机制首先需要使用一个once_flag对象,并把它初始化为值BOOST_ONCE_INIT,它将作为初始化的标志。然后再使用模板函数:call_once()来调用初始化函数,完成仅执行一次的初始化。

例如,假设我们有如下的全局变量和一个初始化函数:

```
static int g_count;           //静态全局变量,更好的方式是使用匿名名字空间
void init_count()
{
    cout << "should call once." << endl;
    g_count = 0;
}
```

然后我们在一个函数call_func()中调用call_once(),这个call_func()函数将被多线程执行,以验证初始化函数的执行情况。注意once_flag对象的赋值必须是线程安全的,在这里我们是在全局域初始化。

```
once_flag of = BOOST_ONCE_INIT; //一次初始化标志
void call_func()
{
    call_once(of, init_count); //执行一次初始化
}
```

在使用call_once()函数时我们必须预先声明once_flag对象,而不能使用临时变量,否

则会引发编译错误，即不能写如下的代码：

```
call_once(once_flag(BOOST_ONCE_INIT), init_count); //错误！
```

最后，我们在 `main()` 函数中启动多个线程来验证：仅初始化一次机制：

```
int main(int argc, char* argv[])
{
    (thread(call_func)); //我们必须用括号括住临时对象
    (thread(call_func)); //否则编译器会认为这是个空
                        //thread 对象声明

    this_thread::sleep(posix_time::seconds(1)); //等待 1 秒钟
}
```

程序的运行结果正如我们所预料的那样，`init_count()` 仅执行了一次，屏幕上将会输出：

```
should call once.
```

`call_once()` 也可以用来执行成员函数，例如 12.1.9 小节的单件线程池的取实例函数：

```
once_flag of = BOOST_ONCE_INIT;
call_once(of, &thread_pool::instance);
```

barrier

`barrier`（护栏）是 `thread` 库基于条件变量提供的另一种同步机制，可用于多个线程同步，当线程执行到 `barrier` 时必须等待，直到所有的线程都到达这个点时才能继续执行。`barrier` 的另一个名字 `rendezvous`（约会地点）更形象地描述了这种行为。

示范 `barrier` 用法的代码如下：

```
barrier br(5); //定义一个 5 个线程的 barrier
void printing(atom_int& x)
{
    {
        mutex::scoped_lock lock(io_mu); //锁定 cout 输出流
        cout << "thread"<< ++x <<" arrived barrier." << endl;
    }
    br.wait(); //在 barrier 处等待，必须 5 个
              //线程都到达这里才能继续执行

    mutex::scoped_lock lock(io_mu); //锁定 cout 流
    cout << "thread run." << endl;
}
int main(int argc, char* argv[])
{
    atom_int x;
    thread_group tg; //一个线程组
```

```

for (int i = 0; i < 5; ++i)                //创建 5 个线程
{
    tg.create_thread(bind(printing, ref(x))); //使用 bind
}
tg.join_all();                             //等待线程结束
}

```

线程本地存储

有时候函数使用了局部静态变量或者全局静态变量，因此不能用于多线程环境，因为无法保证静态变量在多线程环境下重入时的正确操作。

thread 库使用 `thread_specific_ptr` 实现了可移植的线程本地存储机制 (thread local storage, 或者是 thread specific storage (线程专有存储), 简称 tss), 使这样的变量用起来就像是每个线程独立拥有, 可以简化多线程应用, 提高性能。

`thread_specific_ptr` 是一种智能指针, 因此它的接口与 `shared_ptr` 很相似, 它重载了 `operator*` 和 `operator->`, 可以用 `get()` 获得真实指针, 也有 `reset()` 和 `release()` 函数。

`thread_specific_ptr` 的值初始时通常是空指针 (NULL), 因此需要使用 `get()` 来检测。很遗憾, `thread_specific_ptr` 没有定义隐式的 `bool` 转换, 不能直接在 `bool` 语境中检查是否为空。

示范 `thread_specific_ptr` 用法的代码如下:

```

void printing()
{
    thread_specific_ptr<int> pi;           //线程本地存储一个整数
    pi.reset(new int());                  //直接用 reset() 函数赋值

    ++(*pi);                             //递增
    mutex::scoped_lock lock(io_mu);       //锁定 io 流操作
    cout << "thread v=" << *pi << endl;
}
int main(int argc, char* argv[])
{
    (thread(printing));                   //启动两个线程
    (thread(printing));
    this_thread::sleep(posix_time::seconds(1));
}

```

程序的运行结果是:

```

thread v=1
thread v=1

```

很明显, `thread_specific_ptr` 使每个线程都拥有了一份自己的数据拷贝, 它使得编写可重入的函数的工作大大简化了。

线程结束时执行操作

`this_thread` 名字空间提供了一个 `at_thread_exit(func)` 函数, 它允许“登记”一个线程在结束的时候执行可调用物 `func`, 无论线程是否被中断。例如:

```
void end_msg(const string& msg) //定义线程结束时被调用的函数
{ cout << msg << endl; }
void printing(atom_int& x, const string& str)
{
    ...
    at_thread_exit(bind(end_msg, "end")); //使用 bind 得到函数对象
}
```

但如果线程被特定的操作系统 API 强行中止, 或者程序调用标准 C 函数 `exit()` (例如 `main()` 函数正常结束)、`abort()`, 那么线程也会强制结束, `at_thread_exit()` 登记的函数不会被调用。

lazy future

`promise` 和 `packaged_task` 都支持回调函数, 可以让 `future` 延后在需要的时候获得值, 而不必主动启动线程计算。`promise` 和 `packaged_task` 使用成员函数 `set_wait_callback()` 设置回调函数, 当 `future` 对象使用 `get_future()` 函数时启动线程调用回调函数, 计算出 `future` 值。

示范 lazy future 用法的代码如下:

```
void lazy_call_back(packaged_task<int>& task) //回调函数
try
{
    task(); //启动 task
}
catch(task_already_started&) //如果 task 已经启动会抛出异常
{}

int main()
{
    packaged_task<int> task(bind(fab, 10)); //task
    task.set_wait_callback(lazy_call_back); //设置回调
    unique_future<int> uf = task.get_future(); //future 值

    cout << uf.get(); //获取 future 值, 调用回调
                      //函数计算
}
```

12.2 asio

asio 库基于操作系统提供的异步机制，采用前摄器设计模式（Proactor）实现了可移植的异步（或者同步）IO 操作，而且并不要求使用多线程和锁定，有效地避免了多线程编程带来的诸多有害副作用（如条件竞争、死锁等）。

目前 asio 主要关注于网络通信方面，使用大量的类和函数封装了 socket API，提供了一个现代 C++ 风格的网络编程接口，支持 TCP、ICMP、UDP 等网络通信协议。但 asio 的异步操作并不局限于网络编程，它还支持串口读写、定时器、SSL 等功能，而且 asio 是一个很好的富有弹性的框架，可以扩展到其他有异步操作需要的领域。

使用 asio 不需要编译，但它依赖于其他一些 Boost 库组件，最基本的是 boost.system 和 boost.datetime 库，用来提供系统错误和时间支持。其他可选库有 regex、thread 和 serialization，如果支持 SSL，还要额外安装 OpenSSL。

本章接下来对 asio 的介绍将以基本功能为主，即不使用 system 和 datetime 以外的 Boost 组件。asio 位于名字空间 boost::asio，因此需要包含的头文件形式如下：

```
#ifdef _MSC_VER
#define _WIN32_WINNT 0x0501 //避免 VC 下的编译警告
#endif
#define BOOST_REGEX_NO_LIB
#define BOOST_DATE_TIME_SOURCE
#define BOOST_SYSTEM_NO_LIB
#include <boost/asio.hpp>
using namespace boost::asio;
```

12.2.1 概述

asio 库基于前摄器模式（Proactor）封装了操作系统的 select、poll/epoll、kqueue、overlapped I/O 等机制，实现了异步 IO 模型。它的核心类是 io_service，相当于前摄器模式中的 Proactor 角色，asio 的任何操作都需要有 io_service 的参与。

在同步模式下，程序发起一个 IO 操作，向 io_service 提交请求，io_service 把操作转交给操作系统，同步地等待。当 IO 操作完成时，操作系统通知 io_service，然后 io_service 再把结果发回给程序，完成整个同步流程。这个处理流程与多线程的 join() 等待方式很相似。

在异步模式下，程序除了要发起的 IO 操作，还要定义一个用于回调的完成处理函数。io_service 同样把 IO 操作转交给操作系统执行，但它不同步等待，而是立即返回。调用

`io_service` 的 `run()` 成员函数可以等待异步操作完成, 当异步操作完成时 `io_service` 从操作系统获取执行结果, 调用完成处理函数。

`asio` 不直接使用操作系统提供的线程, 而是定义了一个自己的线程概念: `strand`, 它保证在多线程的环境中代码可以正确地执行, 而无需使用互斥量。`io_service::strand::wrap()` 函数可以包装一个函数在 `strand` 中执行。

IO 操作会经常使用到缓冲区, `asio` 库专门用两个类 `mutable_buffer` 和 `const_buffer` 来封装这个概念, 它们可以被安全地应用在异步的读写操作中, 使用自由函数 `buffer()` 能够包装常用的 C++ 容器类型, 如数组、`array`、`vector`、`string` 等, 用 `read()`、`write()` 函数来读取缓冲区。

`asio` 库使用 `system` 库的 `error_code` 和 `system_error` 来表示程序运行的错误。基本上所有的函数都有两种重载形式, 一种形式是有一个 `error_code` 的输出参数, 调用后必须检查这个参数验证是否发生了错误; 另一种形式没有 `error_code` 参数, 如果发生了错误会抛出 `system_error` 异常, 调用代码必须使用 `try-catch` 块来捕获错误。在实际开发过程中这两种形式都可以使用, 各有利弊, 本书的代码都使用后者。

`asio` 是一个相当复杂的库, 本书只涉及其中的基本概念, 还有许多异步 IO 相关的概念在此不一一介绍。

12.2.2 定时器

定时器是 `asio` 库里最简单的一个 IO 模型示范, 提供等候时间终止的功能, 通过它我们可以快速熟悉 `asio` 的基本使用方法。

定时器功能的主要类是 `deadline_timer`, 它的类摘要如下:

```
class deadline_timer
{
public:
    explicit deadline_timer(io_service& io_service);
    deadline_timer(io_service& io_service,
        const time_type& expiry_time);

    void wait();
    template <typename WaitHandler>
    void async_wait(WaitHandler handler);
    std::size_t cancel();

    time_type expires_at() ;
    std::size_t expires_at(const time_type& expiry_time);
    duration_type expires_from_now() ;
```

```
std::size_t expires_from_now(const duration_type& expiry_time);
};
```

定时器 `deadline_timer` 有两种形式的构造函数，都要求有一个 `io_service` 对象，用于提交 IO 请求，第二个参数是定时器的终止时间，可以是 `posix_time` 的绝对时间点或者是自当前时间开始的一个时间长度。

一旦定时器对象创建，它就会立即开始计时，可以使用成员函数 `wait()` 来同步等待定时器终止，或者使用 `async_wait()` 异步等待，当定时器终止时会调用 `handler` 函数。

如果创建定时器时不指定终止时间，那么定时器不会工作，可以用成员函数 `expires_at()` 和 `expires_from_now()` 分别设置定时器终止的绝对时间和相对时间，然后再调用 `wait()` 和 `async_wait()` 等待。这两个函数也可以用于获得定时器的终止时间，只需要使用它们的无参重载形式。

定时器还有一个 `cancel()` 函数，它的功能是通知所有异步操作取消，转而等待定时器终止。

12.2.3 定时器用法

本小节将在之前介绍的 `asio` 内容基础上，使用定时器作为范例介绍 `asio` 的基本用法。

同步定时器

我们先从最简单的同步定时器开始，下面的代码示范了 `deadline_timer` 的用法：

```
#include <boost/asio.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
using namespace boost::asio;          //打开 asio 名字空间
int main()
{
    io_service ios;                    //所有 asio 程序必须要有一个 io_service 对象
    deadline_timer t(ios,              //定时器, io_service 作为构造函数的参数
        posix_time::seconds(2));      //两秒钟后定时器终止
    cout << t.expires_at() << endl;    //查看终止的绝对时间

    t.wait();                          //调用 wait() 同步等待
    cout << "hello asio" << endl;     //输出一条消息
}
```

这段代码非常简单，代码中的注释说明了大部分内容。定时器同步等待两秒钟，当等待结束后输出一条消息，然后程序结束。

读者可以把它与 `thread` 库的 `sleep()` 函数对比研究一下，两者虽然都是等待，但内部机制完全不同：`thread` 库的 `sleep()` 使用了互斥量和条件变量，在线程中等待，而 `asio` 则是调用

了操作系统的异步机制，如 select、epoll 等完成的。

同步定时器的用法很简单，但它演示了 asio 程序的基本结构和流程：一个 asio 程序首先要定义一个 io_service 对象，它是前摄器模式中最重要 proactor 角色，然后我们声明一个 IO 操作（在这里是定时器），并把它挂接在 io_service 上，再然后就可以执行后续的同步或异步操作。

异步定时器

接下来我们来研究异步定时器，代码大致与同步定时器相等，增加了回调函数，并使用 io_service.run() 和定时器的 async_wait() 方法。

首先我们要定义回调函数，asio 库要求回调函数只能有一个参数，而且这个参数必须是 const asio::error_code &类型：

```
void print(system::error_code& /*e*/)
{ cout << "hello asio" << endl;}
```

随后的异步定时器代码也同样很简单：

```
int main()
{
    io_service ios;                                //io_service 对象
    deadline_timer t(ios, posix_time::seconds(2)); //定时器

    t.async_wait(print);                           //异步等待,传入回调函数,
                                                    //立即返回

    cout << "it show before t expired." << endl;
    ios.run();                                     //很重要!异步 IO 必须!
}
```

异步定时器的代码虽然变化不多，但实现的功能却有本质的差别。

代码的前两行与同步定时器相同，这是所有 asio 程序基本的部分。重要的是异步等待 async_wait()，它通知 io_service 异步地执行 IO 操作，并且注册了回调函数，用于在 IO 操作完成时由事件多路分离器分派返回值 (error_code) 调用。

最后我们必须调用 io_service 的 run() 成员函数，它启动前摄器的事件处理循环，阻塞等待所有的操作完成并分派事件。如果不调用 run() 那么虽然操作被异步执行了，但没有一个等待它完成的机制，回调函数将得不到执行机会。

当定时器时间到终止时 io_service 将调用被注册的 print()，输出一条消息，然后程序结束。

异步定时器使用 bind

异步定时器中由于引入了回调函数，因此产生了很多的变化，我们可以增加回调函数的参数，使它能够做更多的事情。但 `async_wait()` 接受的回调函数类型是固定的，必须使用 `bind` 库来绑定参数以适配它的接口。

下面我们实现一个可以定时执行任意函数的定时器 `a_timer` (async timer)，它持有一个 `asio` 定时器对象和一个计数器，还有一个 `function` 对象用来保存回调函数：

```
class a_timer
{
private:
    int count, count_max;           //计数器成员变量
    function<void()> f;             //function 对象，持有无参无返回的可调用物
    deadline_timer t;               //asio 定时器对象
```

`a_timer` 的构造函数初始化成员变量，将计数器清零，设置计数器的上限，拷贝存储回调函数，并立即启动定时器。

之所以要“立即”启动，是因为我们必须保证在 `io_service.run()` 之前至少有一个异步操作在执行，否则 `io_service.run()` 会因为没有任何事件处理而立即不等待返回：

```
public:
    template<typename F>             //模板类型，可以接受任意可调物
    a_timer(io_service& ios, int x, F func): //初始化回调函数和计数器
        f(func), count_max(x), count(0),   //启动计时器
        t(ios, posix_time::millisec(500))
    {
        t.async_wait(bind(&a_timer::call_func, //异步等待计时器
            this, placeholders::error));      //注册回调函数
    }
```

注意在 `async_wait()` 中 `bind` 的用法，`call_func` 是 `a_timer` 的一个成员函数，因此我们需要绑定 `this` 指针，同时我们还使用了 `asio` 下子名字空间 `placeholders` 下的一个占位符 `error`，它的作用类似于 `bind` 库的占位符 `_1`、`_2`，用于传递 `error_code` 值。

接下来是 `a_timer` 的主要功能函数 `call_func()`，它符合 `async_wait()` 对回调函数的要求，有一个 `error_code` 参数，当定时器终止时它将被调用执行。

`call_func` 函数内部累加计数器，如果计数器未达到上限则调用 `function` 对象 `f`，然后重新设置定时器的终止时间，再次异步等待被调用，从而达到反复执行的目的：

```
void call_func(const system::error_code&)
{
```

```

        if (count >= count_max)           //如果计数器到达上限则退出
        {
            return; }
        ++count;
        f();                               //调用 function 对象
        t.expires_at(t.expires_at() + posix_time::millisec(500));
                                           //设置定时器的终止时间为 0.5 秒之后
        t.async_wait(bind(&a_timer::call_func, this, placeholders::error));
                                           //再次启动定时器,异步等待
    }
};

```

`a_timer` 的调用代码非常简单, 只需要一个 `io_service` 对象即可:

```

void print1()                             //第一个回调函数
{ cout << "hello asio" << endl;}
void print2()                             //第二个回调函数
{ cout << "hello boost" << endl;}
int main()
{
    io_service ios;                       //io_service 对象
    a_timer at1(ios, 5, print1);          //启动第一个定时器
    a_timer at2(ios, 5, print2);          //启动第二个定时器
    ios.run();                            //io_service 等待异步调用结束
}

```

12.2.4 网络通信简述

`asio` 库支持 TCP、UDP 和 ICMP 通信协议, 它在名字空间 `boost::asio::ip` 里提供了大量的网络通信方面的函数和类, 很好地封装了原始的 Berkeley Socket API, 展现给 `asio` 用户一个方便易用且健壮的网络通信库, 下面的论述主要针对使用最广泛的 TCP 协议。

`ip::tcp` 类是 `asio` 网络通信 (TCP) 部分主要的类, 但它本身并没有太多的功能, 而是定义了数个用于 TCP 通信的 `typedef` 类型, 用来协作完成网络通信。这些 `typedef` 包括端点类 `endpoint`、套接字类 `socket`、流类 `iostream`, 以及接受器 `acceptor`、解析器 `resolver` 等等。从某种程度上来看, `ip::tcp` 类更像是一个名字空间。

`ip::tcp` 的类摘要如下:

```

class tcp
{
public:
    ///The type of a TCP endpoint.
    typedef basic_endpoint<tcp> endpoint;
    ///The type of a resolver query.
    typedef basic_resolver_query<tcp> resolver_query;
    ///The type of a resolver iterator.

```

```

typedef basic_resolver_iterator<tcp> resolver_iterator;
///The TCP socket type.
typedef basic_stream_socket<tcp> socket;
///The TCP acceptor type.
typedef basic_socket_acceptor<tcp> acceptor;
///The TCP resolver type.
typedef basic_resolver<tcp> resolver;
///The TCP iostream type.
typedef basic_socket_iostream<tcp> iostream;
///Obtain an identifier for the type of the protocol.
int type() const;
///Obtain an identifier for the protocol.
int protocol() const;
///Obtain an identifier for the protocol family.
int family() const;

///Construct to represent the IPv4 TCP protocol.
static tcp v4();
///Construct to represent the IPv6 TCP protocol.
static tcp v6();
};

```

接下来我们将用数个小节详细介绍 asio 使用 TCP 协议进行网络通信的能力。

12.2.5 IP 地址和端点

IP 地址独立于 TCP、UDP 等通信协议，asio 库使用类 `ip::address` 来表示 IP 地址，可以同时支持 `ipv4` 和 `ipv6` 两种地址^①，它的类摘要如下：

```

class address
{
public:
    address();
    address(const address& other);

    bool is_v4() const;
    bool is_v6() const;

    ip::address_v4 to_v4() const;
    ip::address_v6 to_v6() const;

    string to_string() const;

```

① 大多数读者都应该很熟悉 `ipv4`，但可能对 `ipv6` 会比较陌生，在这里做一点简单的介绍。`ipv6` 使用 128 位来表示 ip 地址，其大小是 `ipv4` 地址的四倍。通常的 `ipv6` 地址使用 8 个 16 进制数来表示，每个 16 进制数间用冒号 (:) 分隔，但冗余的数字 0 可以被压缩表示。

```

static address (const char* str);
static address from_string(const string& str);

friend bool operator==(const address& a1, const address& a2);
...
//其他比较操作和流输出操作
};

```

address 类最重要的方法是静态成员函数 from_string(), 它是一个工厂函数, 可以从字符串产生 ip 地址, 地址的版本则可以用 is_v4() 和 is_v6() 来检测。相应地, address 也有一个 to_string() 函数, 可以把 ip 地址转换为字符串。

下面的代码示范了 address 类的用法^①:

```

ip::address addr; //声明一个 ip 地址对象
addr = addr.from_string("127.0.0.1"); //从字符串产生 ip 地址
assert(addr.is_v4()); //ipv4 的地址
cout << addr.to_string() << endl; //转换成字符串输出
addr = addr.from_string("ab::12:34:56"); //ipv6 的地址字符串
assert(addr.is_v6());

```

有了 ip 地址, 再加上通信用的端口号就构成了一个 socket 端点, 在 asio 库中用 ip::tcp::endpoint 类来表示。它的主要用法就是通过构造函数创建一个可用于 socket 通信的端点对象, 端点的地址和端口号可以用 address() 和 port() 获得, 例如:

```

ip::address addr; //ip 地址对象
addr = addr.from_string("127.0.0.1"); //一个 ipv4 的地址
ip::tcp::endpoint ep(addr, 6688); //创建端点对象, 端口为 6688
assert(ep.address() == addr);
assert(ep.port() == 6688);

```

12.2.6 同步 socket 处理

ip::tcp 的内部类型 socket、acceptor 和 resolver 是 asio 库 TCP 通信中最核心的一组类, 它们封装了 socket 的连接、断开和数据收发功能, 使用它们可以很容易地编写出 socket 程序。

socket 类是 TCP 通信的基本类, 调用成员函数 connect() 可以连接到一个指定的通信端点, 连接成功后用 local_endpoint() 和 remote_endpoint() 获得连接两端的端点信息, 用 read_some() 和 write_some() 阻塞读写数据, 当操作完成后使用 close() 函数关闭 socket。

① 代码中操作 ipv6 地址的语句在 Linux 运行完全正常, 但 Windows 下运行可能会抛出异常, 这是因为 Windows 操作系统默认不支持 ipv6。只需要在命令行里输入命令“ipv6 install”即可为 Windows 增加 ipv6 的支持, 如果今后不需要 ipv6 可以使用 uninstall 命令卸载。

如果不关闭 socket, 那么在 socket 对象析构时也会自动调用 close() 关闭。

acceptor 类对应 socket API 的 accept() 函数功能, 它用于服务器端, 在指定的端口号接受连接, 必须配合 socket 类才能完成通信。

resolver 类对应 socket API 的 getaddrinfo() 系列函数, 用于客户端解析网址获得可用的 IP 地址, 解析得到的 IP 地址可以使用 socket 对象连接。

下面我们使用 socket 类和 acceptor 类来实现一对同步通信的服务器和客户端程序。

服务器端

首先我们来实现服务器端程序, 它使用一个 acceptor 对象在 6688 端口接受连接, 当有连接时使用一个 socket 对象发送一个字符串。

```
int main()
try                                     //function-try 块
{
    cout << "server start." << endl;
    io_service ios;                    //asio 程序必需的 io_service 对象

    ip::tcp::acceptor acceptor(ios,    //创建 acceptor 对象, ipv4
                               ip::tcp::endpoint(ip::tcp::v4(), 6688)); //接受 6688 端口
    cout << acceptor.local_endpoint().address() << endl;

    while(true)                        //循环执行服务
    {
        ip::tcp::socket sock(ios);    //一个 socket 对象
        acceptor.accept(sock);        //阻塞等待 socket 连接

        cout << "client:";
        cout << sock.remote_endpoint().address() << endl;

        sock.write_some(buffer("hello asio")); //发送数据
    }
}
catch (std::exception& e)             //捕获可能发生的异常
{
    cout << e.what() << endl;
}
```

服务器端程序里要注意的是自由函数 buffer(), 它可以包装很多种类的容器成为 asio 组件可用的缓冲区类型。通常我们不能直接把数组、vector 等容器用作 asio 的读写参数, 必须使用 buffer() 函数包装。

客户端

接下来是客户端程序，为了演示方便我们使用了 12.2.3 小节（495 页）定义的 `a_timer` 类来实现定时调用功能，为此需要定义一个可被调用的函数 `client()`：

```
void client(io_service &ios)                //传入 io_service 对象
try
{
    cout << "client start." << endl;

    ip::tcp::socket sock(ios);              //创建 socket 对象
    ip::tcp::endpoint ep(ip::address::from_string("127.0.0.1"), 6688);
                                            //创建连接端点

    sock.connect(ep);                      //socket 连接到端点

    vector<char> str(100, 0);               //定义一个 vector 缓冲区
    sock.read_some(buffer(str));            //使用 buffer() 包装缓冲区接收数据
    cout << "recive from " << sock.remote_endpoint().address();
    cout << &str[0] << endl;              //输出接收到的字符串
}
catch (std::exception& e)                  //捕获可能发生的异常
{
    cout << e.what() << endl;
}
```

然后我们在 `main()` 函数中创建 `io_service` 对象，用定时器启动 `socket` 客户端：

```
int main()
{
    io_service ios;                        //io_service 对象
    a_timer at(ios, 5, bind(client, ref(ios))); //启动定时器
    ios.run();
}
```

12.2.7 异步 socket 处理

本小节我们把刚才的同步 `socket` 程序改为异步调用方式。异步程序的处理流程与同步程序基本相同，只需要把原有的同步调用函数都换成前缀是 `async_` 的异步调用函数，并增加回调函数，在回调函数中再启动一个异步调用。

服务器端

我们首先实现异步的服务器端程序，定义一个 `server` 类，它实现异步服务的所有功能，就像 12.2.3 小节（495 页）的异步定时器 `a_timer`。

```

class server
{
private:
    io_service &ios;
    ip::tcp::acceptor acceptor;
    typedef shared_ptr<ip::tcp::socket> sock_pt;

```

server 类必需的成员变量是 io_service 对象和一个 acceptor 对象，它们是 TCP 通信的必备要素，随后我们定义了一个智能指针的 typedef，它指向 socket 对象，用来在回调函数中传递。

```

public:
    server(io_service& io): ios(io),
        acceptor(ios,ip::tcp::endpoint(ip::tcp::v4(), 6688))
    {
        start();
    }

```

server 的构造函数存储 io_service 对象，使用 ios、tcp 协议和端口号初始化 acceptor 对象，并用 start() 函数立即启动异步服务。

```

void start()
{
    sock_pt sock(new ip::tcp::socket(ios)); //智能指针

    acceptor.async_accept(*sock,
        bind(&server::accept_handler, this, placeholders::error, sock)); //异步侦听服务
}

```

start() 函数用于启动异步接受连接，需要调用 acceptor 的 async_accept() 函数。为了能够让 socket 对象能够被异步调用后还能使用，我们必须使用 shared_ptr 来创建 socket 对象的智能指针，它可以在程序的整个生命周期中存在，直到没有人使用它为止。

当有 TCP 连接发生时，server::accept_handler() 函数将被调用，它使用 socket 对象发送数据。

```

void accept_handler(const system::error_code& ec, sock_pt sock)
{
    if (ec) //检测错误码
    {
        return;
    }

    cout << "client:"; //输出连接的客户端信息
    cout << sock->remote_endpoint().address() << endl;
    sock->async_write_some(buffer("hello asio"),
        bind(&server::write_handler, this, placeholders::error));

    start(); //再次启动异步接受连接
}

```

`accept_handler()` 对应同步处理中的发送数据部分，但因为是异步调用，所以代码略有不同。

首先它必须检测 `asio` 传递的 `error_code`，保证没有错误发生。然后调用 `socket` 对象的 `async_write_some()` 异步发送数据。同样，我们必须再为这个异步调用编写回调函数 `write_handler()`。当发送完数据后不要忘记调用 `start()` 再次启动服务器接受连接，否则当完成数据发送后 `io_service` 将因为没有事件处理而结束运行。

发送数据的回调函数 `write_handler()` 很简单，因为我们不需要再做更多的工作，可以直接实现一个空函数，在这里我们简单地输出一条消息，表示异步发送数据完成。

```
void write_handler(const system::error_code&)
{   cout << "send msg complete." << endl;   }
};                                              //server 类结束
```

最后我们在 `main()` 函数中创建 `io_service` 对象和 `server` 对象，调用 `run()` 方法开始异步等待：

```
int main()
try
{
    cout << "server start." << endl;
    io_service ios;                                     //io_service 对象

    server serv(ios);                                  //构造 server 对象
    ios.run();                                          //启动异步调用事件处理循环
}
catch (std::exception& e)                             //捕获可能发生的异常
{
    cout << e.what() << endl;
}
```

客户端

通常客户端不需要使用异步通信，但出于演示的目的，在这里我们也实现异步的客户端，进一步示范 `asio` 的异步用法。

与服务器端对应，我们需要定义一个 `client` 类，它实现所有异步调用功能。

```
class client
{
private:
    io_service& ios;                                     //io_service 对象
    ip::tcp::endpoint ep;                               //TCP 端点
    typedef shared_ptr<ip::tcp::socket> sock_pt;
```


client 与 server 类相似, 都必须持有一个 io_service 的引用, 也要有一个 socket 的 share_ptr, 不同的是 client 不需要 acceptor, 而是使用一个端点类直接与服务器建立连接。

```
public:
    client(io_service& io):ios(io),
        ep(ip::address::from_string("127.0.0.1"), 6688)
    {
        start(); //启动异步连接
    }
```

client 构造函数的主要作用是初始化 IP 端点对象, 并调用 start() 函数启动 TCP 连接:

```
void start()
{
    sock_pt sock(new ip::tcp::socket(ios)); //创建 socket 对象
    sock->async_connect(ep, //异步连接
        bind(&client::conn_handler, this, placeholders::error, sock));
}
```

start() 创建一个 socket 对象的智能指针以便在异步调用过程中传递, 然后使用 async_connect() 启动一个异步连接, 指定连接的处理函数是 conn_handler():

```
void conn_handler(const system::error_code& ec, sock_pt sock)
{
    if (ec) //处理错误代码
    { return; }

    cout << "recive from " << sock->remote_endpoint().address();
    shared_ptr<vector<char>> > str(new vector<char>(100, 0));
    //建立接收数据的缓冲区
    sock->async_read_some(buffer(*str), //异步读取数据
        bind(&client::read_handler, this, placeholders::error, str));

    start(); //再次启动异步连接
}
```

当异步连接成功时, conn_handler() 将被调用, 它再用 shared_ptr 包装 vector, 用 buffer() 函数把 vector 作为接收数据的缓冲区, 由 async_read_some() 异步读取, 然后再启动一个异步连接。

当异步读取结束时 read_handler() 被调用, 它直接输出 shared_ptr 指向的缓冲区内容:

```
void read_handler(const system::error_code& ec,
    shared_ptr<vector<char>> > str)
{
    if (ec) //处理错误代码
    { return; }
```

```

        cout << &(*str)[0] << endl;           //输出接收到的数据
    }
};                                              //client 类结束

```

客户端的 main() 函数代码与服务器端的完全一致:

```

int main()
try
{
    cout << "client start." << endl;
    io_service ios;
    client cl(ios);
    ios.run();
}
catch (std::exception& e)                    //捕获可能发生的异常
{
    cout << e.what() << endl;
}

```

12.2.8 查询网络地址

之前关于 TCP 通信的所有论述我们都是使用直接的 IP 地址,但在实际生活中大多数时候我们不可能知道 socket 连接另一端的地址,而只有一个域名,这时候我们就需要使用 resolver 类来通过域名获得可用的 IP,它可以实现与 IP 版本无关的网址解析。

resolver 使用内部类 query 和 iterator 共同完成查询 IP 地址的工作:首先使用网址和服务名创建 query 对象,然后由 resolve() 函数生成 iterator 对象,它代表了查询到的 ip 端点。之后就可以使用 socket 对象尝试连接,直到找到一个可用的为止。

在这里我们使用一个函数 resolv_connect() 来封装 resolver 的调用过程,代码如下:

```

#include <boost/lexical_cast.hpp>                //使用字符串转换功能
void resolv_connect(ip::tcp::socket &sock,      //socket 对象
                   const char* name, int port) //网址和端口号
{
    ip::tcp::resolver rlv(sock.get_io_service()); //resolver 对象
    ip::tcp::resolver::query qry(name, lexical_cast<string>(port));
                                                //创建一个 query 对象

    //使用 resolve() 开始迭代端点
    ip::tcp::resolver::iterator iter = rlv.resolve(qry);
    ip::tcp::resolver::iterator end;           //逾尾迭代器
    system::error_code ec = error::host_not_found;
    for ( ; ec && iter != end; ++iter)
    {
        sock.close();
    }
}

```

```

        sock.connect(*iter,ec);                //尝试连接端点
    }
    if (ec)                                    //有错误发生
    {
        cout << "can't connect." << endl;
        throw system::system_error(ec);
    }
    cout << "connect success." << endl;
}

```

`resolv_connect()` 函数中使用了 `lexical_cast`, 这是因为 `query` 对象只接受字符串参数, 所以我们需要把端口号由整数转换为字符串。

当开始 `resolver` 的迭代时, 我们需要使用 `error_code` 和逾尾迭代器两个条件来控制循环, 因为有可能迭代完所有解析到的端点都无法连接, 只有当 `error_code` 为 0 才表示连接成功。

有了 `resolv_connect()` 函数, 我们就可以不受具体 IP 地址值的限制, 以更直观更灵活的域名来连接服务器, 例如:

```

int main()
try
{
    io_service ios;
    ip::tcp::socket sock(ios);
    resolv_connect(ios, sock, "www.boost.org", 80);
    ...                                //其他操作

    ios.run();
}
catch (std::exception& e)
{
    cout << e.what() << endl;
};

```

`resolver` 不仅能够解析域名, 也支持使用 IP 地址和服务名, 例如:

```
ip::tcp::resolver::query qry ("127.0.0.1", "http");
```

12.2.9 高级议题

本小节讨论关于 `asio` 的一些高级议题。

超时处理

使用定时器, 我们可以在网络通信中实现超时处理, 只需要在异步调用后声明一个 `deadline_timer` 对象, 然后设定它的等待时间和回调函数就可以了, 用法很像 `thread` 库的

time_join() 函数。

例如,下面的代码异步连接服务器,如果超过 5 秒钟还没有完成所有操作则强制关闭 socket:

```
void conn_handler(const system::error_code&)           //连接处理函数
{...}
void time_expired(const system::error_code&,ip::tcp::socket *sock)
                                                    //超时处理函数
{
    cout << "time expired" << endl;
    sock->close();                                     //关闭 socket
}
int main()
{
    io_service ios;
    ip::tcp::socket sock(ios);
    ip::tcp::endpoint ep(ip::address::from_string("127.0.0.1"), 6688);

    sock.async_connect(ep, &conn_handler);           //异步连接
    deadline_timer t(ios, posix_time::seconds(5));    //开始计时
    t.async_wait(bind(time_expired,placeholders::error,&sock));
                                                    //异步等待超时
    ...                                                //其他操作

    ios.run();                                         //进入异步时间循环
}
```

流操作

对于有连接的 TCP 协议,asio 库专门提供了一个 ip::tcp::iostream 类来简化 socket 通信。ip::tcp::iostream 是 std::basic_iostream 的子类,可以像标准流一样操作,它内部集成了 resolver 的域名解析功能和 acceptor 的接受连接功能,能够非常简单地完成 TCP 通信。

使用流操作的客户端可以是这样:

```
int main()                                           //客户端主函数
{
    for (int i = 0;i < 5;++i)                       //循环连接 5 次
    {
        //连接到本机 6688 端口
        ip::tcp::iostream tcp_stream("127.0.0.1", "6688");
        string str;
        getline(tcp_stream,str);                    //从 tcp 流中读取一行数据
        cout << str << endl;
    }
}
```

```
}
```

使用流操作的服务器端可以是这样：

```
int main() //服务器端主函数
{
    io_service ios;

    ip::tcp::endpoint ep(ip::tcp::v4(), 6688);
    ip::tcp::acceptor acceptor(ios, ep);

    while (true)
    {
        ip::tcp::iostream tcp_stream;
        acceptor.accept(*tcp_stream.rdbuf());
        tcp_stream << "hello tcp stream";
    }
}
```

UDP 协议通信

asio 中的 udp 协议通信与 tcp 处理流程类似，但因为 udp 协议是无连接的，故不需要建立连接，使用 `send_to()` 和 `receive_from()` 就可以直接通过端点发送数据。

示范同步的 udp 客户端和服务端用法（省略了异常处理的 try-catch 块）的代码如下：

```
int main() //服务器端主函数
{
    cout << "udp server start." << endl;
    io_service ios;

    ip::udp::socket sock(ios, ip::udp::endpoint(ip::udp::v4(), 6699));
    //创建一个 udp 的 socket 对象,
    //使用 ipv4 协议,端点 6699

    for (;;) //开始服务主循环
    {
        char buf[1]; //一个临时用缓冲区
        ip::udp::endpoint ep; //要接受连接的远程端点

        system::error_code ec;
        sock.receive_from(buffer(buf), ep, 0, ec); //阻塞等待远程连接,连接的端点
        //信息保存在 ep 对象中

        if (ec && ec != error::message_size)
        {
            throw system::system_error(ec);
        }

        cout << "send to " << ep.address() << endl; //得到远程端点
    }
}
```

```

        sock.send_to(buffer("hello asio udp"), ep);           //发送数据
    }
}
int main()                                                     //客户端主函数
{
    cout << "client start." << endl;
    io_service ios;

    ip::udp::endpoint send_ep(ip::address::from_string("127.0.0.1"), 6699);
                                                                    //连接端点
    ip::udp::socket sock(ios);                                   //创建 udp socket 对象
    sock.open(ip::udp::v4());                                   //使用 ipv4 打开 socket

    char buf[1];
    sock.send_to(buffer(buf), send_ep);                         //向连接端点发送连接数据

    vector<char> v(100,0);
    ip::udp::endpoint recv_ep;
    sock.receive_from(buffer(v), recv_ep);                       //接收数据
    cout << "recv from " << recv_ep.address() << " ";
    cout << &v[0] << endl;
}

```

串口通信

asio 除了“主营业务”网络通信外也支持串口通信，需要使用 `serial_port` 类。串口通信的基本处理流程与网络通信类似，但要在通信前设置好波特率、奇偶校验位等串口通信参数。

示范串口通信基本用法的代码如下：

```

//异步处理函数
void time_exipred(system::error_code& ec, serial_port *sp)
{
    cout << "time_exipred" << endl;
    sp->close();
}
void read_handler(system::error_code& ec)
{ cout << ec.message() << endl;}
int main()                                                     //主函数
{
    io_service ios;
    serial_port sp(ios, "COM1");                               //打开串口 COM1

    //设置串口参数
    sp.set_option(serial_port::baud_rate(9600));
    sp.set_option(serial_port::flow_control(serial_port::flow_control::none));
    sp.set_option(serial_port::parity(serial_port::parity::none));
    sp.set_option(serial_port::stop_bits(serial_port::stop_bits::one));
}

```

```

sp.set_option(serial_port::character_size(8));

size_t len = sp.write_some(buffer("hello serial"));    // 向串口写数据
cout << len << endl;

vector<char> v(100);
sp.async_read_some(buffer(v), bind(read_handler, placeholders::error));
                                                                    //异步接收数据
deadline_timer t(ios, posix_time::seconds(2));          //处理超时
t.async_wait(bind(time_expired, placeholders::error, &sp));

ios.run();                                              //启动事件等待循环
}

```

上面代码中定时器的超时处理回调函数也可以直接调用 `serial_port` 的 `cancel()` 或 `close()` 方法, 如:

```
t.async_wait(bind(&serial_port::cancel, ref(sp)));
```

或:

```
t.async_wait(bind(&serial_port::close, ref(sp)));
```

12.3 总结

本章我们讨论了 Boost 库中并发编程方面的两个库, 它们是 `thread` 和 `asio`, 分别关注多线程和异步 IO 这两个领域。

`thread` 库使用多个成熟的多线程范式为我们提供了可移植的多线程处理能力。C++ 历来缺乏操纵线程的能力, 经常要求助于 C API, 而现在 `thread` 库填补了这个空白。它实现了目前多线程编程所需要的绝大部分概念, 包括互斥锁、递归锁、读写锁、条件变量、线程、`barrier`、`future` 等等, 功能丝毫不逊于 POSIX API 和其他多线程库, 而且接口非常方便易用, 并能够配合 `ref`、`bind`、`function` 等其他 Boost 程序库提供更加灵活、优雅、坚固的解决方案。

拥有了 `thread` 库这个强有力的武器, C++ 程序员只需要再了解一些多线程开发基本规则就能够轻而易举地构建出高效的多线程程序。

`asio` 库目前还处在发展完善的过程中, 它使用前摄器模式实现了同步或异步的 IO 操作, 并提供了定时器、网络通信、串口通信等许多 IO 操作。

前摄器模式非常复杂, 难以构建和应用, 而 `asio` 基于操作系统的异步 IO 能力简洁而高效地实现了前摄器模式, 并且将细节封装在 `io_service`、`basic_io_object`、`socket` 等对象中,

使库用户得到了一个高度可用的接口。asio 还很好地封装了 Berkeley Socket API，完整地支持 TCP/UDP/ICMP 等通信协议，有效地避免了误用复杂的原始 Socket API 的问题，很有可能纳入将来的 c++ 标准，成为标准的网络底层库。

thread 库和 asio 库都可以用于并发编程，但它们解决问题的途径却不一样。thread 使用的是进程内部的线程机制，很少需要操作系统内核干预，只要掌握了线程的同步方法，多线程程序的结构很容易理解也很容易实现。而 asio 使用的是异步事件处理机制，与操作系统的内核密切相关，使用它需要对操作系统的底层机制有一定的了解，比多线程程序更难于编写难于调试，但由于把异步操作的管理工作交由操作系统处理，因而能够获得更高的运行性能。

thread 和 asio 有竞争也有合作，可以在多线程环境中同步地使用 asio，获得并发处理的能力又不涉及复杂的异步调用机制，也可以在 asio 异步调用中启动多线程，使异步操作可以运行于多个线程。

本章并没有涵盖 Boost 在并发编程领域的所有内容，没有介绍另一个很重要的库——处理进程间通信 (IPC) 的 interprocess，thread 和 asio 这两个库也只是不完整的介绍，如果读者想更深入地了解它们的强大功能请阅读 Boost 文档和源代码。

本章实现的实用函数和类如下：

- `basic_atom<>`：它是一个原子操作的整数包装类，可以被安全地用在多线程环境中当做计数器；
- `a_timer`：它基于 asio 库和 function 库提供了可定时调用任意函数的功能；
- `resolv_connect()`：它封装了域名解析类 `resolver` 的调用步骤，可以直接通过域名和端口号连接 socket。

第 13 章

编程语言支持

任何程序开发语言都不可能独当一面、包打天下，总有它的长处和短处。

就 C++ 来说，它继承了 C 语言的低级能力，能够使用指针直接操作内存，也可以嵌入汇编语言以获得最快的运行速度。它还支持面向对象和泛型编程等现代程序开发技术，可以在很高的层次上对软件建模，获得高度的抽象，几乎所有领域的软件都可以使用 C++ 完成开发。

但 C++ 也有一些缺点，它不能做到便捷地跨平台开发，许多强大的功能还需要编译器和操作系统的支持。C++ 程序开发周期长，复杂的语言特性让它具有陡峭的学习曲线，C++ 也没有垃圾回收和类型反射机制，这些都限制了它的快速应用开发的能力。

因此，虽然 C++ 很强大，但它没有必要也不可能事必躬亲，与其他的编程语言配合会更好地发挥它的优点同时弥补它的缺点，Python 就是这样的一种语言。虽然表面上看 Python 与 C++ 完全不同，但基本的编程理念却很相似：具有类似的控制流语句，面向对象，支持操作符重载和异常等等，两者具有很好的互补性。

Boost 通过 `boost.python` 库提供了对 Python 语言的全面支持。

13.1 python 库概述

Boost1.42 中的 `python` 库已经发展到第 2 版，较早期的第 1 版有了很大的改进，可以更加方便和容易地在 Python 和 C++ 之间自由转换，而且功能更强大。`python` 库全面支持 C++ 和 Python 的各种特性，包括 C++ 到 Python 的异常转换、默认参数、关键字参数、引用和指针等等，让 C++ 与 Python 可以近乎完美地对接：用 C++ 很容易地为 `python` 编写扩展模块，也可以很容易地在 C++ 代码中执行 `python` 程序。

为行文方便，本章以下的描述中使用首字母大写的“Python”表示 Python 语言，首字母小写的“python”表示 boost.python 库。

13.1.1 Python 语言简介

20 世纪 80 年代末，Guido van Rossum 创建了一种新型的脚本语言，他从英国 BBC 的“Monty Python's Flying Circus”节目中摘取了一个单词作为这种新语言的名字，这就是之后风行二十余年的 Python。

Python 是一种简洁的、功能强大的、动态强类型的解释型语言。

Python 具有比 C/C++ 语言还要简洁的语法，使用代码缩进而不是分号来分隔语句，同时简化了许多传统的语法结构，从而具有优雅的代码格式。

Python 也有具有强大的功能，内建了多种高级数据结构，如列表、集合、元组和字典，使程序员只需要编写少量的代码就可以实现 C/C++ 很多行代码才能完成的工作。Python 完全支持面向过程编程和面向对象编程（实际上，Python 里的一切几乎都是对象），支持异常、名字空间、操作符重载等现代编程机制。它还有一个“包罗万象”的标准库和许多第三方库，如正则表达式、数据库、XML、电子邮件、测试、图形界面……几乎可以实现任何功能。如果想要定制功能也很容易，Python 可用 C 语言扩展底层，增添新模块。

Python 是一个动态语言，类似 Php、Perl，无需声明变量类型就可以使用。但它又是强类型的，变量一旦初始化，就不能随意改变类型。Python 不需要编译，它可以动态地解释交互运行，类似 BASIC，这大大缩短了程序的开发周期。但 Python 也能够像 Java 一样编译成字节码后运行在虚拟机（解释器）上以获得更高的效率。

Python 是可移植的，在各种操作系统上都有免费的 Python 解释器，包括 Windows、UNIX、Linux 三大主流平台。在这些平台上 Python 可以代替批处理、shell 或者 Perl，编写脚本程序方便日常工作，也可以开发非常复杂的应用程序或者服务程序，Python 能够做其他语言所能做的所有事情。

最初的 Python 是用 C 语言写成的，随着时代的发展，又逐渐出现了其他语言编写的 Python，如 Java 的 Jython 和 C# 的 IronPython，它们不仅具有 Python 的功能，还可以调用宿主语言，更为强大。因此，传统的 Python 有时又被称为 C-Python。

下面简单展示几段 Python 代码，其中的#是 Python 的注释：

```
print 'hello python'      #输出 hello python
print 2**3                #输出 2 的立方 8
```

```

for x in range(1,5):           #循环语句，输出 1 2 3 4
    print x

x = ['abc', 123, "python"]     #一个列表，可以包含任意类型
for y in x:                   #遍历列表
    print 'list[%s]' % y       #带格式的输出生

import re                     #导入正则表达式模块
reg = re.compile('a*b')       #编译一个正则表达式对象
print reg.match('ab') != None  #正则表达式匹配字符串

```

如果读者想进一步了解 Python，建议阅读推荐书目 [11]，可以从因特网上免费下载。作为一个快速的开始，它可以在一天之内阅读完并掌握。

13.1.2 安装 Python 环境

Python 目前有两个主要的版本分支：2.x 和 3.x。

2.x 稳定版是 2.62，有非常丰富的第三方库，缺点是对 unicode 支持不够好，处理中文比较麻烦。

3.x 稳定版是 3.1，对 2.x 版本做了很多语法变动（例如 print 由语句变成了函数，取消了 u'xx' 的字符串定义），完全支持 unicode，甚至可以使用中文定义变量名，但缺点是第三方库不够丰富。

本书使用的 Python 版本是 2.62，可以从 Python 的官方网站（网络资源 [7]）获取安装包 python-2.6.2.msi，默认安装位置是 c:\Python26，不用什么复杂的选项配置就可以完成安装。

13.1.3 编译 python 库

boost.python 库需编译才能使用，要求前提是已经安装了 Python（版本高于 2.2）环境。

编译 python 库的 bjam 命令如下：

```
bjam -toolset=msvc -with-python -build-type=complete stdlib=stlport stage
```

如果使用源码嵌入工程编译的方式，则需要在编译环境里指定头文件“Python.h”的包含路径（VC 在 options 中设置，gcc 使用 -I 选项指定）^①，默认情况下是“c:/Python26/include/”

① 如果读者安装的不是 Python 官方标准安装包，则可能需要做一些额外的工作才能成功编译。比如不需要 Stackless 支持，可注释掉头文件中对 stackless.h 的包含。

或“/usr/include/python2.6”，例如：

```
g++ -c pyprebuild.cpp -I/usr/include/python2.6
```

参考 python 库的 jamfile 即可完成内嵌预编译 cpp 源文件：

```
//pyprebuild.cpp
#define BOOST_PYTHON_SOURCE
//#define BOOST_PYTHON_NO_LIB

#include<libs/python/src/numeric.cpp>
#include<libs/python/src/list.cpp>
#include<libs/python/src/long.cpp>
#include<libs/python/src/dict.cpp>
#include<libs/python/src/tuple.cpp>
#include<libs/python/src/str.cpp>
#include<libs/python/src/slice.cpp>

#include<libs/python/src/converter/from_python.cpp>
#include<libs/python/src/converter/registry.cpp>
#include<libs/python/src/converter/type_id.cpp>
#include<libs/python/src/object/enum.cpp>
#include<libs/python/src/object/class.cpp>
#include<libs/python/src/object/function.cpp>
#include<libs/python/src/object/inheritance.cpp>
#include<libs/python/src/object/life_support.cpp>
#include<libs/python/src/object/pickle_support.cpp>
#include<libs/python/src/errors.cpp>
#include<libs/python/src/module.cpp>
#include<libs/python/src/converter/builtin_converters.cpp>
#include<libs/python/src/converter/arg_to_python_base.cpp>
#include<libs/python/src/object/iterator.cpp>
#include<libs/python/src/object/stl_iterator.cpp>
#include<libs/python/src/object_protocol.cpp>
#include<libs/python/src/object_operators.cpp>
#include<libs/python/src/wrapper.cpp>
#include<libs/python/src/import.cpp>
#include<libs/python/src/exec.cpp>
#include<libs/python/src/object/function_doc_signature.cpp>
```

把 pyprebuild.cpp 加入工程即可完成 python 库的编译工作。

13.1.4 使用 python 库

python 库位于名字空间 boost::python，为了使用 python 库，需要包含头文件 <boost/python.hpp>，如果使用源码嵌入工程的编译方式，还需要在头文件前加入宏 BOOST_PYTHON_SOURCE，即：

```
#define BOOST_PYTHON_SOURCE //源码嵌入工程编译方式
#include <boost/python.hpp>
using namespace boost::python;
```

13.2 嵌入 Python

我们先从 python 库最简单的用法——嵌入 Python 语句开始。这种使用方式可以调用 Python 语言的标准库和第三方库，就像拥有了一群数量庞大的库函数，让 C++ 不费任何力气就拥有了脚本语言的操纵能力。

不过目前 python 库的嵌入功能没有它的扩展功能那么强大，有的操作还需要调用 Python API，但可以满足基本的要求。

嵌入 Python 语言需要链接 Python 的运行库 python26.lib，它在 c:/Python26/Libs 目录下，可以在 VC 的工程属性中设置链接库选项，但最好是使用预处理指令放在源码中，如：

```
#pragma comment(lib, "python26.lib") //VC 系列编译器支持这个指令
#define BOOST_PYTHON_SOURCE
#include <boost/python.hpp>
using namespace boost::python;
```

13.2.1 初始化解释器

在 C 程序中执行 Python 语句有一个标准流程：

- 首先要调用 Python API 函数 `Py_Initialize()` 启动 Python 解释器；
- 解释器启动后，可以使用 `Py_IsInitialized()` 来检查解释器是否已经成功启动；
- 在完成所有的 Python 调用后，使用 `Py_Finalize()` 来清除解释器环境。

目前的 boost.python 库不完全遵循上面的流程，库文档建议不执行 `Py_Finalize()` 来清除环境，因此在 C++ 中只需要调用 `Py_Initialize()` 就可以了^①。

Python API 相当的简陋，而 python 库并没有对它进行封装，我们可以考虑自己实现一个初始化 Python 解释器的类 `pyinit`，它要比单纯的 API 函数更方便好用。

`pyinit` 使用 `Py_InitializeEx()` 初始化解释器，并提供 `isInitialized()` 检查解释器的状态，`version()` 获得 Python 解释器的版本：

① 但经作者测试，目前的 python 库似乎调用了 `Py_Finalize()` 也无问题。

```
//pyinit.hpp
#include <boost/noncopyable.hpp>
#include <boost/python.hpp>
class pyinit: boost::noncopyable
{
public:
    pyinit(int initsigs = 1)
    {
        assert(initsigs == 0 || initsigs == 1 );
        Py_InitializeEx(initsigs);
    }
    ~pyinit(){/*Py_Finalize() ;*/}

    bool isInitialized()
    {
        return Py_IsInitialized(); }
    const char* version()
    {
        return Py_GetVersion();    }
};
```

pyinit 使用了 boost::noncopyable, 但并没有实现为单件, 因为它没有内部的数据或状态, 不需要提供全局的访问点。

读者可以在 pyinit 的基础上增加更多的功能, 让它更有用。

13.2.2 封装 Python 对象

python 库使用模板类 handle 和 object 封装了 Python API 中的 PyObject 类型。handle 是一个智能指针, 一般情况下我们应当优先使用 object。

object 类封装了 PyObject, 内部也使用了引用计数, 使用起来就像 Python 语言中的原生变量, 或者是 C++ 中的 auto 和 boost.any (参见 7.7 小节, 287 页)。它的类摘要如下:

```
class object
{
public:
    object();
    object(object const&);

    //模板构造函数, 可以接受任何类型!
    template <class T> explicit object(T const& x);
    ~object();

    object& operator=(object const&);
    PyObject* ptr() const;
};
```

模板类 extract<type> 可以把 Python 对象转换成所需的值, 它的用法很像 any_cast,

如果无法转换则会抛出异常。

使用 object 及其子类，我们可以在 C++ 中编写类似 Python 的代码。示范 object 基本用法的代码如下：

```
#pragma comment(lib, "python26.lib")
#define BOOST_PYTHON_SOURCE
#include <boost/python.hpp>
#include "pyinit.hpp"
using namespace boost::python;
int main()
{
    pyinit pinit;                                //初始化 Python 环境

    object i(10);                                //一个 Python 变量,可以是任何类型
    i = 10 * i;                                    //像 Python 里一样操作
    cout << extract<int>(i) << endl;              //用 extract 转换类型

    object s("string");                           //一个 python 字符串变量,
    string str = extract<string>(s* 5);            //把字符串增加五倍,再转换成 string
    cout << str << endl;
}
```

在基本的 object 之外，python 库还提供了许多 Python 语言中类型的对应物，如 list、dict、tuple，它们分别对应 Python 语言中的列表、字典和元组结构，有许多操作函数，使用方法和 Python 基本相同。

示范这些 Python 对应数据结构基本用法的代码如下：

```
#pragma comment(lib, "python26.lib")
#define BOOST_PYTHON_SOURCE
#include <boost/python.hpp>
#include "pyinit.hpp"
using namespace boost::python;
int main()
{
    pyinit pinit;

    //list 类型,注意需要名字空间限定以避免与 std::list 冲突
    python::list l;
    l.append("zelda");                             //与 Python 类似的操作,添加数据
    l.append(2.236);
    assert(len(l) == 2);                            //也可以用 len() 获得长度
    assert(l.count("zelda") == 1);                  //count() 计算成员的数量
    cout << extract<double>(l[1]) << endl;
    l.sort();                                         //排序
}
```

```

//tuple 类型,同样需要名字空间限定以避免与 boost::tuple 冲突
python::tuple t = python::make_tuple("metroid", "samus", "ridley");
assert(len(t) == 3);
assert(string(extract<string>(t[-2])) == "samus");
                                                    //可以用 Python 里的负数进行反序索引
l.append(t);
                                                    //把 tuple 加入到 list 中
assert(len(t) == 3);

python::str s(', ');
                                                    //字符串类型
s = s.join(t);
                                                    //连接 tuple 里的字符串
cout << string(extract<string>(s)) << endl;

dict d;
                                                    //字典类型,不会引起名字冲突
d["mario"] = "peach";
                                                    //可以用任意的 key/value 值对
d[0] = "killer7";
assert(d.has_key(0));
assert(len(d) == 2);
}

```

object 还有很多用法,如用成员函数 `attr()` 访问属性,直接使用 `operator()` 调用 Python 函数,但通常这些代码都不如直接执行 Python 语句方便,把 Python 变量交给 Python 解释器去管理更好。

13.2.3 执行 Python 语句

启动 Python 解释器后,可以使用 python 库提供的 `exe()` 系列函数执行 Python 语句,这些函数的声明如下:

```

object eval(str expression, object globals, object locals)
object exec(str code, object globals, object locals)
object exec_file(str filename, object globals, object locals)

```

这三个函数的功能类似,都可以执行 Python 语句,但有小的不同: `eval()` 函数计算表达式的值并返回结果, `exec()` 执行 Python 语句并返回结果,而 `exec_file()` 则执行一个文件中的 Python 代码。

函数接口中的 `globals` 和 `locals` 参数是 Python 中的字典结构,是语句运行的全局和局部场景,通常这两个参数可以忽略,或者取 `__main__` 模块的名字空间字典。

示范简单执行 Python 语句的代码如下:

```

cout << pinit.version() << endl;
                                                    //显示 Python 的版本
cout << extract<int>(eval("3**3")) << endl;
                                                    //计算 3 的立方
exec("print 'hello python'");
                                                    //输出语句

```


如果在 Python 语句使用了变量,那么必须要指定 `globals` 参数。使用 `import()` 函数可以导入 `__main__` 模块,用成员函数 `attr()` 获取属性,如:

```
object main_module = import("__main__");
object main_namespace = main_module.attr("__dict__");
```

示范较复杂的 Python 语句执行的代码如下:

```
#pragma comment(lib, "python26.lib")
#define BOOST_PYTHON_SOURCE
#include <boost/python.hpp>
#include "pyinit.hpp"
using namespace boost::python;
int main()
{
    pyinit pinit;

    //获取运行所需的名字空间
    object main_ns = import("__main__").attr("__dict__");

    //执行 for 循环
    string str = "for x in range(1,5):\n"
                "\t\tprint x";
    exec(str.c_str(), main_ns);                                //输出 1,2,3,4

    //定义一个 Python 函数,计算 x 的 y 次方
    char *funcdef = "def power(x, y):\n"
                    "\t\treturn x**y\n"
                    "print power(5, 3)\n";
    exec(funcdef, main_ns);                                    //输出 125
    object f = main_ns["power"];                               //使用名字空间字典获得函数对象
    cout << extract<int>(f(4, 2)) << endl;                     //用 operator() 执行

    //导入 re 模块,执行正则表达式功能,输出 True
    exec("import re", main_ns);
    exec("print re.match('c*', 'ccc') != None", main_ns);
}
```

13.2.4 异常处理

如果执行 Python 语句发生错误,python 库会抛出 `error_already_set` 异常,但不含有任何信息,需要调用 API 函数 `PyErr_Print()` 向标准输出打印具体的错误信息。

为方便使用,可以给 `pyinit` 类再增加一个静态成员函数 `err_print()`:

```
class pyinit: boost::noncopyable
{
```

```
public:
    ...
    static void err_print()
    {
        PyErr_Print();
    }
}
```

可以这样使用:

```
try
{
    exec("import re"); //没有指定 globals, 将发生错误
}
catch (...)
{
    pyinit::err_print();
}
```

输出的错误信息可能是这样:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: __import__ not found
```

13.3 扩展 Python

python 库能够在 C++ 程序中调用 Python 语言,但它更重要的功能在于用 C++ 编写 Python 扩展模块,嵌入到 Python 中解释器中调用,提高 Python 的执行效率。

python 库在第 1 版的基础上做了大量的改进,充分利用了 C++ 的高级特性和新技术,封装和屏蔽了许多底层实现,展现给外界的是一个高度抽象、灵活和易于学习的接口。只要用户会编写 C++ 程序,就可以立刻为 Python 编写扩展模块。

与原始 C API 的烦琐调用步骤相比,python 库在很大程度上简化了扩展模块的编写,使编写工作就像是在写 Python 语言,而且它还很好地处理了许多原来手工编程容易出错的地方,让开发者把精力集中在模块的主要逻辑上,而不是陷入管理 Python 对象引用计数等低级细节中。

扩展 Python 我们同样需要编译 python 库,并包含头文件<boost/python.hpp>,但不必指明 Python 的运行库即可自动链接(需设定库文件搜索路径,默认是 C:\Python26\libs),即:

```
#define BOOST_PYTHON_SOURCE //源码嵌入工程编译方式
#include <boost/python.hpp>
using namespace boost::python;
```

13.3.1 最简单的例子

在本小节中我们要实现一个最简单的功能，向 Python 导出一个无参的 `hello()` 函数，它将打印出 “hello boost python” 字符串：

```
#include <string>
using std::string;
string hello_func()                //返回一个字符串
{ return "hello boost python"; }
```

python 库编写扩展模块非常容易，由于使用了模板元编程等高级技术，代码量非常少，而且看起来非常清晰易懂。

首先我们要在 VC 中建立一个 DLL 工程，名字叫 `boostpy`，是一个不使用预编译头的空工程，不要忘记设置工程字符集的 `Not Set` 和运行库多线程的 `MT` 或 `MTd` 属性，并加入宏 “`_STLP_DEBUG`” 和 “`__STL_DEBUG`”（如果使用了 `STLport`）。

然后我们在工程中新增一个 `boostpy.cpp` 文件，并加入 13.1.3 小节（515 页）定义的 python 库嵌入源码编译文件 `pyprebuild.cpp`。

接下来我们在 `boostpy.cpp` 中编写函数导出代码，使用 `BOOST_PYTHON_MODULE` 宏定义 Python 模块名，模块名必须与 `dll` 的名字相同，当然也可以在编译后改 `dll` 的名字。

宏 `BOOST_PYTHON_MODULE` 的用法很像 `test` 库的单元测试套件宏，在宏 `BOOST_PYTHON_MODULE` 定义的模块内部我们使用 `def()` 函数定义要导出的函数，需要指定导出的名字和 C++ 的函数名字，它的语法一定程度上模仿了 Python 语言的函数定义关键字 `def`。

下面列出了例子的全部代码：

```
#define BOOST_PYTHON_SOURCE          //源码嵌入工程编译方式
#include <boost/python.hpp>
using namespace boost::python;      //打开名字空间

string hello_func(){...}            //C++函数定义

BOOST_PYTHON_MODULE(boostpy)        //Python 模块定义开始
{
    //导出一个名字为 hello 的函数，其 doc string 是“函数说明字符串”
    def("hello", hello_func, "函数说明字符串");
}
```

不需要担心这个简短的程序中没有 `DllMain()`、`WINAPI` 等 Windows 编程中常见的 `dll` 导

出元素，也不需要写 def 或者 exp 文件，python 库为我们在幕后做了一切，将自动生成一个完全可用的动态链接库 boostpy.dll。

但这个 dll 不能被 Python 环境识别，必须把后缀名改成标准的 Python 模块后缀名 pyd，即 boostpy.pyd（也可以修改 VC 工程设置，直接生成后缀是 pyd 的 dll 文件），然后放置到 Python 环境可以找到的路径下——通常是 Python 主目录或者主目录下的 lib 目录（默认即 c:/Python26 和 c:/Python26/lib）。

我们首先使用 Python 交互解释器 IDLE 测试这个最小的 Python 扩展模块，下面显示结果中的“>>>”是 IDLE 的输入提示符：

```
IDLE 2.6.2
>>> import boostpy                #导入 boostpy 模块
>>> boostpy.hello()              #直接执行 hello() 函数
'hello boost python'
>>> print boostpy.hello()        #使用 print 语句输出结果
hello boost python
```

也可以使用 help() 函数来查看 boostpy 模块的信息：

```
>>> help(boostpy)
Help on module boostpy:
NAME
    boostpy
FILE
    c:\python26\boostpy.pyd
FUNCTIONS
    hello(...)
        hello() -> str :
            函数说明字符串
            C++ signature :
                class stlpd_std::basic_string<char,class stlpd_std::char_traits
                <char>,class stlpd_std::allocator<char> > hello()
```

boostpy 模块也可以在 Python 脚本中运行，使用脚本运行扩展模块时不要求 pyd 模块必须在 Python 主目录下，只要和脚本在同一个目录即可。

例如下面的 test.py 脚本：

```
#coding:utf-8
#file test.py
import boostpy                #导入 boostpy 模块
print boostpy.hello()        #使用 print 语句输出结果
```

运行这个 Python 脚本将与 IDLE 交互解释器的运行结果一样，输出“hello boost python”。

13.3.2 导出函数

python 库使用模板函数 `def()` 来导出 C++ 函数到 Python，它有多个重载形式，声明是：

```
template <class F>
void def(char const* name, F f,...);
```

`def()` 函数要求导出的名字必须是 C 字符串（以 0 结束的字符数组），不能是 `std::string` 对象，第二个参数是类型为 `F` 的函数指针，这之后可以添加文档字符串以及函数的参数列表等函数的附加信息。

向 Python 导出函数的参数需要使用 python 库中的参数关键字类 `arg`，它的类摘要如下：

```
struct arg
{
    explicit arg (char const *name);
    template <class T> arg &operator = (T const &value);
    arg operator,(char const *name) const;
    arg operator,(python::arg const &k) const;
};
```

`arg` 类的构造函数接受一个 C 字符串作为参数的导出名字，得益于 Python 的动态语言特性，我们无需指定它表示的参数类型。

为了支持 C++ 和 Python 的缺省参数特性，`arg` 重载了 `operator=`，可以指定参数的缺省值。它还重载了逗号操作符，可以使用逗号表达式把 `arg` 参数连接起来（很像 `assign` 库的用法）。但在 `def()` 函数中使用时，为了不与函数参数分隔的逗号混淆，我们需要把 `arg` 逗号表达式用圆括号括起来。

在上一小节的基础上，我们再定义两个 C++ 函数用于导出：

```
string hello_to(const string& str)           //接收一个字符串参数
{ return "hello " + str;}
string hello_x(const string& str, int x)     //接收字符串和整数参数
{
    string tmp = "hello ";
    for (int i = 0 ; i < x ; ++i)
    {
        tmp += str + " ";
    }
    return tmp;
}
```

然后我们在宏 `BOOST_PYTHON_MODULE` 定义的导出模块中导出它们，并使用 `arg` 类定义它们在 Python 中的参数：

```
def("helloto", hello_to, arg("str"));       //定义一个参数
def("hellox", hello_x, (arg("str"), "x"));  //使用逗号操作符
```

在导出函数时参数名不一定非要与 C++ 中的一致，可以是任意的名字，只要它符合 Python 的命名规则即可，比如名字可以是 “a_bit_long_name_of_arg_type_is_str”。

编译生成新的 pyd 文件后，可以使用下面的 Python 脚本调用验证：

```
print boostpy.helloto('boost')
print boostpy.hellox('C++', 5)
```

Python 脚本运行结果如下：

```
hello boost
hello C++ C++ C++ C++ C++
```

python 库另外提供了一个便捷函数 args()，可以用在不需要指定参数值的时候直接使用参数名字符串生成多个 arg 对象，例如：

```
def("hellox", hello_x, args("str", "x"));
```

13.3.3 导出重载函数

C++ 和 Python 中都有重载函数的概念，它们可以名字相同但参数和返回值不同。在向 Python 导出 C++ 重载函数的时候不能使用之前的 def() 形式，因为无法从函数名字区分出重载函数，必须使用函数指针类型定义。

使用函数指针手工导出

我们把之前的三个 hello() 系列函数都改为重载函数，名字为 hello_func()，然后我们定义三个函数指针 fp1、fp2、fp3：

```
string (*fp1)() = &hello_func;
string (*fp2)(const string&) = &hello_func;
string (*fp3)(const string&, int) = &hello_func;
```

然后就可以使用 def() 函数导出函数指针，由于函数指针定义已经包含了参数信息，因此我们无需特意使用 arg 类定义参数。当然使用 arg 也是允许的，可以更好地在 Python 中描述参数信息：

```
BOOST_PYTHON_MODULE(boostpy) //Python 模块定义开始
{
    def("hello", fp1, "doc string1");
    def("hello", fp2, "doc string2");
    def("hello", fp3, (arg("str"),arg("x")), "doc string3");
}
```

这三个函数的 Python 调用脚本如下：

```
import boostpy #导入 boostpy 模块
print boostpy.hello()
print boostpy.hello('boost')
print boostpy.hello('C++',5)
```

使用宏自动导出

如果程序中有大量的重载函数，那么手工定义函数指针的工作将会很繁琐，因此 python 库特意提供了一个方便的宏 `BOOST_PYTHON_FUNCTION_OVERLOADS`，专门用于简化重载函数的定义，它可以自动产生重载函数说明，声明如下：

```
#define BOOST_PYTHON_FUNCTION_OVERLOADS(generator_name, fname, min_args, max_args)
```

宏 `BOOST_PYTHON_FUNCTION_OVERLOADS` 使用四个参数，第一个参数 `generator_name` 用于生成一个辅助类，包含重载函数的定义，第二个参数 `fname` 指定重载函数的名字，最后两个参数指定重载函数参数的最小和最大参数个数。

使用 `BOOST_PYTHON_FUNCTION_OVERLOADS` 有一点限制，要求重载函数必须具有顺序相同的参数序列，即少的参数序列是多的参数序列的子集。例如我们的 `hello()` 系列函数的参数序列依次是：`(void)`、`(string)`、`(string,int)`，如果第三个函数的参数是 `(int,string)`，那么它就不满足 `BOOST_PYTHON_FUNCTION_OVERLOADS` 的要求，宏只能用于前两个函数。

宏 `BOOST_PYTHON_FUNCTION_OVERLOADS` 可以这样使用：

```
BOOST_PYTHON_FUNCTION_OVERLOADS(hello_overloads, hello_func, 0, 2)
```

其中 `hello_overloads` 是我们为重载函数指定的辅助类名，`hello_func` 是重载函数的名字，数字 0 和 2 表示有三个重载形式，参数最少是 0 个，最多是 3 个。

在使用 `def()` 导出前，我们还必须定义最多参数的重载函数指针类型，即：

```
typedef string (*hello_ft)(const string&, int);
```

然后我们就可以向 Python 一次性导出全部重载函数：

```
def("hello", (hello_ft)0, hello_overloads());
```

注意 `def()` 函数中第二个参数，即函数指针参数的用法，我们把一个空指针转换成了 `hello_ft` 函数指针类型，然后再用辅助类的临时对象 `hello_overloads()` 以导出所有函数。

我们也可以仍然使用函数指针，但需要用最多参数的那个函数指针：

```
def("hello", fp3, hello_overloads());
```

`BOOST_PYTHON_FUNCTION_OVERLOADS` 将把重载函数导出为一个 Python 函数，它具有

`max_args` 个缺省参数，使用 Python 的 `help()` 可以看到导出的函数说明。

`BOOST_PYTHON_FUNCTION_OVERLOADS` 的另一个用法是导出具有缺省参数值的函数，这种函数就像是有 N 个重载形式的函数，例如：

```
string hello_func(const string& str="boost", int x = 5);
typedef string (*hello_ft)(const string&, int);
def("hello", (hello_ft)0, hello_overloads());
```

混合使用手工重载和自动重载方式，我们就能够很好地减轻编写导出重载函数的工作负担，对于有相同参数序列或者缺省参数的函数使用宏 `BOOST_PYTHON_FUNCTION_OVERLOADS`，其他的则使用手工定义函数指针的方式。

13.3.4 导出类

python 库的另一个强大的功能是可以方便地把 C++ 类导出为 Python 类，这在 python 库出现前是一件非常烦琐且容易出错的工作。python 库使用一个类似 Python 语法的模板类 “`class_`” 封装了这项工作。

模板类 `class_` 有很多成员函数，因为它必须要能够支持 C++ 和 Python 两种语言的各种语义，它的类摘要如下：

```
template <class T, class Bases = bases<> >
class class_ : public object
{
    //构造函数，产生 Python 的缺省初始化函数 __init__
    class_(char const* name);

    //构造函数，指定初始化函数 __init__
    template <class Init> class_(char const* name, Init);

    //导出其他初始化函数
    template <class Init> class_& def(Init);

    //导出成员函数
    template <class F> class_& def(char const* name, F f);

    //导出 attribute
    template <class U>
    class_& setattr(char const* name, U const&);

    //导出静态成员函数
    class_& staticmethod(char const* name);
```



```

//导出成员变量
template <class D>
class_& def_readonly(char const* name, D T::*pm);
template <class D>
class_& def_readwrite(char const* name, D T::*pm);

//导出静态成员变量
template <class D>
class_& def_readonly(char const* name, D const& d);
template <class D>
class_& def_readwrite(char const* name, D& d);

//导出 property
template <class Get>
void add_property(char const* name, Get const& fget, char const* doc=0);
template <class Get, class Set>
void add_property(
    char const* name, Get const& fget, Set const& fset, char const* doc=0);
template <class Get>
void add_static_property(char const* name, Get const& fget);
template <class Get, class Set>
void add_static_property(char const* name, Get const& fget, Set const& fset);
};

```

class_类的用法与 def() 函数基本相同，它导出模板参数 T 类型为 Python 类，再用成员函数 def()、def_readonly() 等分别导出 T 的成员函数和成员变量。

例如，我们把之前的 hello() 系列函数改为一个简单的 C++ 类：

```

class demo_class
{
private:
    string msg;
public:
    static string s_hello;
    demo_class():msg("boost"){} //缺省构造函数
    string hellox(int x = 1)
    {
        string tmp = s_hello;
        for (int i = 0 ; i < x ; ++i)
        {
            tmp += msg + " ";
        }
        return tmp;
    }
};

string demo_class::s_hello = "hello "; //静态成员变量初始化

```

那么使用 class_ 可以这样导出类：

```
BOOST_PYTHON_MODULE(boostpy)
```

```
{
    class_<demo_class>("demo", "doc string")
        .def("hello", &demo_class::hellox, arg("x")=1)
        .def_readwrite("shello", &demo_class::s_hello);
}
```

class_ 首先用模板参数指定了导出类 demo_class，然后在构造函数中指定了导出名和文档字符串。随后用 def() 导出成员函数，用法与导出普通函数类似，但对于成员函数我们必须写出全名，并且使用取地址操作符&。最后我们使用 def_readwrite() 导出了成员变量。

Python 调用脚本如下：

```
from boostpy import *           #导入 boostpy 模块
d = demo()                     #使用缺省构造函数
print d.hello()                #缺省参数 1, 输出: hello boost
print d.shello                 #访问成员变量, 输出: hello
d.shello = 'goodbye '          #修改成员变量
print d.hello(2)               #输出: goodbye boost boost
```

13.3.5 导出类的更多细节

本小节将在 13.3.4 小节（528 页）的基础上讲解 python 库导出类的更多用法。

构造函数

13.3.4 小节讲述导出类时我们没有指定类的构造函数，因此 Python 在创建对象时将使用缺省构造函数。但很多情况下缺省构造函数是不够的，带参数的构造函数更加常见。

我们不能使用 def() 来导出构造函数，因为 C++ 中的构造函数不同于普通的成员函数，最重要的区别是不能取它的地址，即没有这样的语法：

```
&demo_class::demo_class
```

因此，python 库使用模板类 init<...>和 optional<...>来共同定义构造函数和构造函数中的缺省参数。它们的模板参数都是构造函数的参数类型，init 中的参数是必须出现的，而 optional 中的参数是有缺省值可以不出现的，它们的用法很像是定义重载构造函数。

假设我们为 demo_class 增加一个构造函数：

```
demo_class(const string& str = "python")
{ msg = str; }
```

那么我们既可以直接在 class_ 的构造函数中指定 init，也可以在 def() 中指定 init：

```
class_<demo_class>("demo", "doc string",
    init<optional<string> >("init doc") )
```

或:

```
class <demo_class>("demo", "doc string")  
    .def(init<optional<string> >("init doc"))
```

这两种导出形式在语义上是不同的。第一种形式指定了一个有缺省参数的构造函数，而第二种形式先指定了一个缺省构造函数，然后又指定了另外一个有缺省参数的构造函数，因此第二种形式要求被导出的类必须有缺省构造函数。

导出 property

使用 `def_readonly()` 和 `def_readwrite()` 我们可以导出 C++ 类的成员变量，同时指定它的读写属性，但这两个函数要求类的成员变量必须是 `public` 的。通常在 C++ 中很少有 `public` 的成员变量，它们总是被封装为 `private` 或者 `protected` 拒绝外界的直接访问，而使用访问函数来间接地存取值。

在 Python 语言中用 “property” 来表示类似的概念，python 库使用 `add_property()` 和 `add_static_property()` 来导出 property，前者用于操作对象 property，后者用于操作类 property（静态成员变量），它们的用法与 `def()` 基本相同。

如果我们要把 `demo_class` 的静态成员变量 `s_hello` 导出为 Python 属性，首先要定义它的访问函数：

```
static void set(const string& str)  
{ s_hello = str; }  
static string get()  
{ return s_hello ; }
```

因为这两个访问函数是静态成员函数，因此我们需要使用 `add_static_property()`，否则可以使用 `add_property()`：

```
class <demo_class>("demo", "doc string")  
    .add_static_property("rshello", &demo_class::get)  
    .add_static_property("rwshello", &demo_class::get, &demo_class::set)
```

这样我们就导出了两个 Python property，`rshello` 是只读 property，而 `rwshello` 则可读可写。

导出 attribute

`attribute` 与 `property` 在 Python 中是两个相似又有区别的概念，python 库使用 `setattr()` 函数来导出 attribute。它的用法与 `def()` 相同，但导出到 Python 中的将是一个 attribute，而不是一个普通的 Method。

例如，我们可以将 `get()` 函数改为导出成一个 attribute：

```
class_<demo_class>("demo", "doc string")
    .setattr("get", &demo_class::get);
```

staticmethod() 函数配合 setattr() 可以导出 C++ 类的静态成员函数，成为 Python 中的一个静态方法。因为静态成员函数已经被 setattr() 定义过了，因此 staticmethod() 只需要指定导出名。例如：

```
class_<demo_class>("demo", "doc string")
    .setattr("get", &demo_class::get)           //先定义 attribute
    .staticmethod("get");                       //再定义静态方法
```

重载成员函数

对于导出类的重载函数，可以使用手工定义成员函数指针的形式，也可以使用简化的工具宏，但使用的是另外一个宏 BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS，除了名字不同，它与 BOOST_PYTHON_FUNCTION_OVERLOADS 的用法完全相同。

例如，使用这个宏封装了有缺省参数的 hellox() 的代码如下：

```
BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS(hello_overloads,hellox,0,1)
class_<demo_class>("demo", "doc string")
    .def("hello",&demo_class::hellox, hello_overloads())
```

导出继承关系

class_ 也支持导出 C++ 类的继承关系，这样子类将自动获得父类导出的 Python 属性和方法，而且在 Python 中也会有多态的特性。

导出继承关系很简单，需要使用 class_ 的第二个模板参数，在这里用模板类 bases 指出基类，例如，假设我们有一个 demo_class 的子类 derived，那么它的导出继承关系代码如下：

```
class derived :public demo_class{};           //简单的派生类
class_<derived, bases<demo_class> >("derived");
```

在导出继承关系时，通常让父类有一个虚的析构函数是个好主意，这可以让 Python 能够正确地使用父类指针销毁子类对象。

13.3.6 高级议题

本小节讨论关于 python 库扩展用法的一些高级议题。

重载操作符

C++ 和 Python 都支持重载操作符，不同的是 C++ 使用 operatorX() 的形式，而 Python

则定义了若干内部方法, 如 `__add__`、`__sub__` 等。python 库使用一个 `self` 对象模仿 Python 语法提供了重载操作符的支持。

我们使用 9.2 小节 (344 页) 介绍的 `rational` 类作为导出重载操作符的例子, 代码很简单, 几乎是自说明的:

```
typedef boost::rational<int> rational;           //typedef 用来简化代码
class_<rational>("rational", "boost rational")   //缺省构造函数
    .def(init<int, int>())                       //分子分母的构造函数
    .setattr("n", &rational::numerator)         //获得分子
    .setattr("m", &rational::denominator)       //获得分母
    .def(self + int()).def(int() + self)        //重载对 int 的加法
    .def(self - int())                          //重载对 int 的加法
    .def(self + self).def(self - self)         //有理数之间的加减法
    .def(self < self);                          //比较操作符
```

对应的 Python 脚本如下:

```
from boostpy import *                           #导入 boostpy 模块
r = rational(1, 5);                             #有理数 1/5
print r.n(), r.m()                             #输出 1 5
r = r + 5                                       #重载操作符加法运算
print r.n(), r.m()                             #输出 26 5
print rational() < r                          #缺省构造有理数 0, 输出 True
```

导出枚举类型

python 库使用 `enum_` 类来导出 C++ 的枚举类型, 用法与 `class_` 基本相同, 下面的代码说明了它的用法:

```
enum demo_enum{sha1, md5, md2};                //枚举几个常见的摘要算法
BOOST_PYTHON_MODULE(boostpy)                  //Python 模块定义开始
{
    enum_<demo_enum>("digest")                //使用 value 函数导出枚举值
        .value("sha1", sha1)
        .value("md5", md5)
        .value("md2", md2);
}
```

作用域

默认情况下我们导出的所有 Python 对象都在全局作用域, 但有时候 C++ 类型会嵌套在其他类中, 而我们需要导出这种嵌套关系。

`scope` 类封装了 Python 中作用域的概念, 一个缺省构造的 `scope` 对象保存了当前的作用域。`scope` 也可以被赋值为一个 `class_` 对象, 这时作用域将变为 `class_` 对象所对应的域, 所

有接下来的导出都在这个域之内，一直持续到 `scope` 对象析构为止。例如：

```
struct demol
{
    struct demo2
    {    int f(){return 10;} };
};
BOOST_PYTHON_MODULE(boostpy)                                //Python 模块定义开始
{
    scope s = class_<demol>("demol");                        //开始一个内部作用域
    class_<demol::demo2>("demo2")
        .def("f", &demol::demo2::f);
}
```

其他功能

`python` 库还有许多其他的功能用于支持 C++ 语法到 Python 的转换，如导出虚函数、定义 Python 迭代器、转换 C++ 异常到 Python 异常、Python 的序列化模块 `Pickle` 等等，而且还有两个用 Python 编写的自动代码生成器 `pyste` 和 `py++`，可以大大简化扩展 Python 功能的代码编写工作。

这些功能在 Boost 的说明文档中都有详细描述，读者可以自行阅读参考。

13.4 总结

现实世界中存在很多种编程语言，C++ 无疑是最强大最富表现力的一种，但俗话说“一个篱笆三个桩，一个好汉三个帮”，C++ 也需要与其他的语言互相配合，共同构建复杂的软件系统。

C++ 可以为几乎所有的编程语言编写基础模块，为它们提供扩展功能，例如 Perl、Python、Java、Fortran 等等，但目前 Boost 程序库仅提供了对 Python 语言的支持，即 `python` 库。

`boost.python` 是一个功能非常强大的库，对 C++ 和 Python 的双方向转化提供了无缝的操作，它有许多复杂的用法和深入的细节，涉及 C++ 和 Python 两种语言的许多语法的细枝末节，本章只能讲述其中基本的一些知识。但相信读者有了这些基础知识，再深入学习 `python` 库将不再是难事。

`python` 库有两种使用方式：嵌入 Python 和扩展 Python。

我们首先讨论了在 C++ 中嵌入 Python 语言，这种用法需要链接 Python 运行库。`python` 库使用 `object` 类封装了 Python 对象，并有 `str`、`list`、`tuple`、`slice` 等 Python 对应的类，可以在 C++ 中编写 Python 风格的代码。`python` 库也提供了三个函数可以直接执行 Python 语句，这种方式通常更方便，使我们可以直接执行已经写好的 Python 脚本。

python 库也为 C++ 扩展 Python 提供了完善的支持，几乎所有的 C++ 语言特性都能通过 python 库翻译到对应的 Python 模块，包括重载函数、函数缺省参数、枚举、类的构造函数、嵌套类、虚函数、继承，等等。通过使用 `def()` 和 `class_`、`enum_` 等函数和类，可以编写非常简洁的描述式代码来完成导出 C++ 对象的工作，而且这种导出是非侵入的，不需要对原有的代码做任何的变动。

基于 python 库可以构建出 C++ 与 Python 的混合软件系统：

使用 Python 的动态特性、解释能力和大量的标准模块，我们能够快速构建出可用的软件原型，然后用 C++ 改写其中运行效率低的部分，作为底层模块供 Python 调用。这样我们既拥有 Python 的快速开发能力，又有了 C++ 的运行高效率。python 库为我们提供了 C++ 与 Python 的任意转换能力，可以把软件系统中的任何一个模块用任意语言实现，系统中语言所占的比例因需求不同而变化，如果侧重快速开发，那么大部分代码都会是 Python 编写的，如果侧重运行效率，那么情况就会相反。

本章实现了一个比较有用的 C++ 工具类 `pyinit`，它包装了一些 Python API 函数，提供了一个方便易用的接口，可以简化嵌入 Python 的工作。

第 14 章

其他Boost组件

第 2 章至第 13 章完整地讲述了 1.42 版 Boost 库中 50 多个组件，限于本书的篇幅（如果完全讲述可能 1000 页都不够），Boost 库中的其余组件将在本章做简要描述。

这并不意味着这些组件简单或者不重要，恰恰相反，这其中有的库非常庞大和复杂，甚至可以单独写成一本厚厚的书（如 graph、mpl）。本书不做详细阐述除篇幅的原因外还基于以下的几条理由：

第一，有的库十分复杂，限于作者的知识面和水平很难讲解清楚，如 gil、graph、preprocessor、mpl；第二，有的库过于专业，非作者的擅长领域，故不敢贸然动笔，如 MPI 和 math 中的大多数数学库；第三，有的库个人认为在实际开发工作中不是很常用，如 parameter、compressed_pair、value_initialized。

以下的分类基本依据 Boost 说明文档，但做了适当的微调。

14.1 字符串和文本处理

regex

它是 Boost 库中最早出现的正则表达式库，可用于字符串匹配、查找和替换，功能用法几乎与 xpressive 完全相同，但需要编译才能使用。

regex 已经被收入到 C++ TR1，即将出现在 C++0x 标准中。

spirit

spirit 是一个面向对象的递归下降解析器的生成框架，它使用 EBNF 语法，是一个比正则

表达式更强大的语法分析器。

语法分析器通常给人的印象是很复杂很难学习，但 `spirit` 被设计成是层次化的、可伸缩的，它提供了循序渐进的学习层次，并为初学者准备了一些预先定义好的分析对象，比如解析浮点数对象等来方便快速入门。

14.2 容器与数据结构

`gil`

`gil` 是一个通用图像库，由 Adobe 公司赞助支持开发。它为像素、色彩、通道等图像处理概念提供了泛型的、STL 式的容器和算法，可以对图像做灰度化、梯度、均值、旋转等许多运算，支持 JPG、PNG、TIFF 等文件格式。

`graph`

`graph` 库处理离散数学中的图结构，并提供图、矩阵等数据结构上的泛型算法，例如 Dijkstra 最短路径、Prim 最小生成树、连通分支等等，可以看作是 STL 在非线性容器领域的扩展。`graph` 库还特别支持 Python 语言绑定，以达到快速开发的目的。

`intrusive`

STL 和大多数 Boost 容器属于非侵入式容器，不需要对容器内的元素类型做修改即可容纳，使用起来简单方便。而 `intrusive` 库则引入了日渐被遗忘的侵入式容器和算法，其接口类似于已被熟知的 STL，提供 `list`、`tree`、`map` 等与 STL 几乎等价的容器，有的情况下能够提供更好的性能。

`pointer container`

`pointer container` 是从另一个方面对 STL 容器的扩展。STL 容器不能很好地容纳指针，使用 `shared_ptr` 可以变通处理，但有时是不合适的。`pointer container` 提供了与 STL 类似的若干种指针容器，包括 `ptr_vector`、`ptr_list`、`ptr_map` 等，性能较好且异常安全。

`multi_index`

`multi_index` 实现了具有多个 STL 兼容访问接口（索引）的容器，例如既可以用 `set` 的方式、也可以用 `list` 的方式来存取 `multi_index` 容器元素。如果使用 STL 的标准容器感到不方便时请考虑它。

14.3 迭代器

iterators

`iterators` 定义了一组基于 STL 的新的迭代器概念、构造框架和有用的适配器，能够帮助程序员更轻松地实现迭代器模式。它有可能被纳入 C++ 新标准。

range

`range` 基于 STL 迭代器提出了“范围”的概念，是一个容器的半开区间。使用 `range` 可以让代码更加简单漂亮，Boost 库的许多组件都基于 `range` 概念，例如 `foreach`、`string_algo`。

14.4 函数对象与高级编程

mem_fn

`mem_fn` 是 STL 中函数对象适配器 `mem_fun` 和 `mem_fun_ref` 的扩展，类似 `bind` 与 `bind1st`、`bind2nd` 的关系，可以完全取代 C++98 中的成员函数适配器，能够把类成员函数适配为可用于 STL 算法的函数对象。

`mem_fn` 的用法与 `bind` 很类似，仅仅是少传递了一个类实例参数，在大多数情况下 `bind` 可以完全替代 `mem_fn`。

functional

`functional` 增强了 STL 中的函数对象适配器，并解决了“引用的引用”问题。如果读者很熟悉 C++98 中诸如 `bind1st`、`mem_fun_ref`、`not1` 等函数对象适配器，那么 `boost.functional` 是个很好的选择——不需要花费太多的学习时间，它几乎是与 STL 一样的。^①

但大多数情况下，`boost.bind` 提供了更好的解决方案，读者更应该使用它。

hash

`hash` 组件实现了 TR1 中定义的散列函数，可以对 C++ 内置类型和标准库容器计算散列值，也可以拓展它以支持自定义类型。`hash` 主要用于实现散列容器如 `unordered`。

^① Boost1.43 版中 `functional` 库新增加了两个小组件：`factory` 和 `forward`。

lambda

lambda 库为 C++ 引入了 lambda 表达式和函数式编程，可以就地创建小型的函数对象，避免函数对象的定义离调用点过远，更方便代码维护。

但本书不推荐使用 boost.lambda 库。lambda 表达式是一种新的编程范式，而 lambda 表达式的语法也十分复杂，如果使用得不好很容易写出过于晦涩难懂的代码，使程序难以维护。而且，C++0x 新增了语言级别的 lambda 表达式特性，语法与 boost.lambda 很不同，简洁了许多。一旦 C++0x 发布，boost.lambda 很有可能成为“过时”的库。

但即使是 C++0x 中的 lambda 表达式也是比较复杂的，仍然有“只写”代码的倾向。本书建议使用函数内部类来就地定义小型函数对象，或者用 BOOST_FOREACH 来遍历容器。实际上，有了 BOOST_AUTO 和 BOOST_FOREACH，大部分时候我们都不需要使用 lambda 表达式。

在本书的 7.4.7 小节 (268 页) 有一个 bimap 使用 lambda 表达式的例子，可供读者参考。

signals

signals 实现了观察者模式，功能和用法与 signals2 基本相同，但不支持线程安全，而且需要编译。如果没有什么特殊理由，应该使用 signals2 库。

parameter

Python 语言提供了使用参数名来指定函数参数值的机制，使用 parameter 库，C++ 也可以获得同样的功能。在实现有多个入口参数的函数时 parameter 可以大大简化客户代码的编写，使用起来也更加方便。

14.5 泛型编程

enable_if

enable_if 库允许模板函数或者模板类在偏特化时仅针对某些特定类型有效，即启用或禁用某些特化形式，它依赖于 SFINAE (substitution failure is not an error) 原则。

call_traits

call_traits<T> 封装了可能是最好的传递参数给函数的方式，它会自动推导出最高效的传递参数的类型：内建类型传值，类实例则传引用，而且保证不会出现“引用的引用”这个非法的错误。

type_traits

`type_traits` 提供一组特征 (trait) 类, 用以在编译期确定类型是否具有某些特征, 例如类型 `T` 是否是个指针, 是否是个抽象类, 是否重载了 `new` 操作符等等, 它还包含一套可以对某个类型执行特定转换的工具类。使用 `type_traits` 可以编写出更好更高效的泛型代码。

`type_traits` 已被收入 TR1。

concept check

`C++0x` 中的一个提案是为语言增加 `concept` (概念), 但很遗憾被否决了 (直接导致了 `C++0x` 变成了 `C++1x`)。 `boost.concept_check` 库完全使用 `C++98` 标准实现了功能基本相同的概念检查, 可以在编译期检查模板函数或模板类的模板类型参数是否符合某个概念, 是否允许进行模板参数推演。如果模板参数不满足所要求的概念, 则会发生一个编译错误。

`concept_check` 主要被用来编写实现泛型算法或泛型库, 如果你是个库作者将会发现它很有用。

function_types

这个库提供了对函数、函数指针、函数引用和成员指针等类型进行分类、分解和合成的功能。

in_place_factory

`in_place_factory` 库是工厂设计模式的一种实践, 允许就地直接构造对象而不需要一个临时对象的拷贝。在本书的 4.4.6 小节 (112 页) 演示了它的用法。

proto

`proto` 库允许在 `C++` 中构建专用领域嵌入式语言, 基于表达式模板技术定义小型专用语言的“编译器”。

property map

`property map` 是一个概念库, 提供了 `key-value` 映射的属性概念定义, 为从键到值的映射定义了一个通用接口。

14.6 模板元编程

fusion

`fusion` 库提供基于 `tuple` 的编译期容器 (`vector`、`set`、`map` 等) 和算法, 是模板元编

程的强大工具，可以与 `mpl` 很好地协同工作。

`mpl`

`mpl` 是一个模板元编程框架，包含有编译期的算法、容器和函数等完整的元编程工具。运用 `mpl`，很多运行时的工作都可以在编译期完成，甚至编译结束就意味着程序的运行结束。

在本书的 6.3.12 小节（243 页）和 7.8.6 小节（300 页）有 `mpl` 库运用的简单例子。

14.7 预处理元编程

`preprocessor`

`preprocessor` 库提供预处理元编程工具，它类似于模板元编程，但发生在编译之前的预处理阶段。`preprocessor` 改变了以往人们对预处理器的看法，令人们认识到预处理也是一种强大的编程工具。`preprocessor` 也可以与模板元编程很好地配合，从而发挥更大的作用。

`wave`

`wave` 是使用 `spirit` 库开发的一个完全符合 C/C++ 标准的预处理器，你可以把它当作对 `spirit` 的一个技术演示，也可以把它当作一个现有编译器中预处理器的替代品，或者自行编写程序处理 C++ 源文件得到预处理后的代码进行分析。

14.8 并发编程

`interprocess`

`interprocess` 实现了可移植的进程间通信（IPC）的功能，包括共享内存、内存映射文件、信号量、文件锁、消息队列等许多现代操作系统的 IPC 机制，并提供了简洁易用的 STL 风格接口，大大简化了 IPC 编程工作。

`MPI`

`MPI` 库可用于高性能分布式并行计算应用的开发，它封装了标准的 `MPI`（消息传送接口）以更好地支持现代 C++ 编程风格。`boost.MPI` 不是一个完整的库，需要有底层 `MPI` 实现的支持，如 `Open MPI`、`MPICH` 等。

14.9 数学与数字

accumulators

`accumulators` 是一个用于增量统计的库，也是一个用于增量计算的可扩展的累加器框架，可以看做是 `std::accumulate` 算法的扩展。

interval

`interval` 库处理“区间”概念相关的数学问题，把一般的算术运算扩展到了区间上，可以对区间执行加减乘除算术运算、普通代数函数（平方、开方等）、初等超越函数（幂、对数等）、三角函数和反三角函数（`sin`、`cos`、`tan`、`asin` 等），还有区间特有的交、并、上下界等运算。

math

`math` 库包括 `common_factor`、`octonion`、`quaternion` 等六个组件，包含了大量数学领域的模板类和算法，如复数的反三角函数（`asin`、`acos`）、最大公约数和最小公倍数（`lcm`、`gcd`）、四元数、八元数，还有许多特殊数学函数和统计分布（拉格朗日多项式、椭圆积分、X 方分布、伯努利分布等等）。

uBLAS

`uBLAS` 是一个用于线性代数领域的数学库，比 `std::valarray` 要好得多。它支持单位向量、稀疏向量、密集矩阵、稀疏矩阵、三角矩阵等许多线性代数概念，可以对向量或矩阵进行加法和减法，与标量做乘法，执行内积、外积等许多矩阵运算。这些接口都被设计成与 STL 类似的风格，可以很容易地使用而且运算效率很高。

14.10 TR1 实现

`boost.tr1` 是对 C++ 库扩展技术报告 TR1 (Technical Report 1) 的一个实现。但实际上它并没有真正地实现 TR1，而是对其他 Boost 库进行了包装，再导入到 `std::tr1` 名字空间中。使用 `boost.tr1`，可以在 C++0x 发布时毫无困难地转移到新标准而代码无须做任何改动。

TR1 包含十多个组件，大部分都是基于 Boost 库的，如 `tuple`、`bind`、`function`、`array`、`shared_ptr`、`unordered`，这些库均已经在本书中给出了详细介绍。

不过要小心的是，`boost.tr1` 并不与标准完全一致，实现上略有不同，使用 `tr1` 应该阅读 Boost 说明文档以避开不一致的地方。但一般来说，两者几乎是相同的，唯一的重大缺陷是 `ref`。

14.11 输入输出

iostreams

`iostreams` 扩展了 C++ 标准库的流处理，建立了一个流处理框架。它定义了 `Source`、`Sink`、`Filter` 等流处理概念，使得编写流式处理更加容易。使用 `iostreams`，可以处理 `socket` 连接、数据压缩、加解密、编码转换和正则表达式等许多方面的事情。

serialization

`serialization` 库实现了 C++ 数据结构的持久化，可以把任意的 C++ 对象整编为一串二进制字节流或者文本，然后再在需要的时候解整编恢复为原来的对象。类似功能的库有很多，微软的 MFC 就提供这样的序列化功能，但 `boost.serialization` 更强大，而且支持 STL 容器。

14.12 杂项

compressed_pair

`compressed_pair` 库提供一个与 `std::pair` 非常相似的模板类 `compressed_pair`，用法也完全相同。不同之处在于 `compressed_pair` 使用了空基类优化技术，如果 `compressed_pair` 的两个成员之一是空类，那么编译器就会“压缩” `compressed_pair` 的大小以节约空间。

base_from_member

有时候基类需要由派生类的成员变量来初始化，但通常的写法因 C++ 的类初始化顺序要求而不能实现，因为基类必须在派生类之前完成初始化。解决方法是把成员移动到另一个辅助基类中，`base_from_member` 使用模板技术提供了这个用成员来初始基类的惯用法。

conversion

`conversion` 库增强了 C++ 标准中的 `static_cast<>`、`dynamic_cast<>` 等转型操作符，以相同的风格提供多态对象转型的 `polymorphic_cast<>`、`polymorphic_downcast<>` 和字面量转换的 `lexical_cast<>`。其中的 `lexical_cast<>` 已经在 5.1 节（163 页）讨论过。

flyweight

`flyweight` 库实现了享元（`flyweight`）设计模式，享元对象是不可修改的，但可以赋值。

默认情况下 `flyweight` 使用类型 `T` 作为查找的键值，也就是说根据对象的 `hash` 值查看内部是否有同样的对象。但有的时候 `T` 的构造代价太高，可以使用另一种 `key-value` 的使用方法。

numeric conversion

`numeric conversion` 库提供用于安全数字转型的一组工具，包括 `numeric_cast<>`、`bounds<>` 和 `converter<>` 等。

scope_exit

`scope_exit` 与 `shared_ptr` 的资源释放用法有些相似，它使用 `preprocessor` 库的预处理技术实现在退出作用域时的资源自动释放，也可以执行任意的代码。`scope_exit` 的用法很简单，当不想写一个类专门处理 `RAII` 问题时可以用它。

statechart

`statechart` 库提供了一个功能完善的、强大的有限状态自动机框架，完全支持 `UML` 语义，可以从 `UML` 模型很方便地转换成 `C++` 代码。比起手工构建的状态机，它可以极大地缩短开发周期，而性能也有足够的保证。

units

`units` 库实现了物理学的量纲处理，包括长度、质量、时间、电流、温度、物质的量和发光强度等七个基本量纲。`units` 库使用了模板元编程技术 (`mpl`)，支持国际标准量纲 (米、千克、秒)，也支持其他的度量单位如华氏度、达因、英尺。量纲的处理都是在编译期，没有运行时开销。

value_initialized

`C++98` 标准没有明确定义变量的初始化 (`C++0x` 对此问题有改进)，当声明一个变量时，它的初始值有可能是未定义的。`value_initialized` 库可以保证变量在声明时被正确地初始化，拥有零值或者缺省值。

utility

`utility` 库集合了很多小工具，其中的 `noncopyable`、`BOOST_BINARY`、`BOOST_CURRENT_FUNCTION` 已经在第 4 章“实用工具”做了介绍，下面描述其他的组成部分：

`checked_delete`：在编译期保证 `delete` 或 `delete[]` 操作删除的是一个完整类定义，以避免运行时出现未定义行为。

`next()` 和 `prior()`：这两个模板函数，为迭代器提供后向和前向的通用处理方式。

`addressof`: `addressof()` 函数可以获得变量的真实地址, 是取地址操作符 `&` 的增强版本, 即使是类重载了 `operator&` 也不受影响。

14.13 总结

本章介绍了 Boost 程序库中的 46 个组件, 但限于篇幅没有进行详细的阐述。读者可以把这些介绍看作是对 Boost 库的简要索引, 在需要的时候能够快速定位可用的组件, 便于进一步的学习和使用。

如果读者感兴趣, 推荐参考 Boost 文档深入研究 `pointer container`、`parameter`、`type_traits`、`iostreams`、`serialization`、`flyweight` 等库的用法, 它们在实际的开发工作中具有较高的应用价值。

第 15 章

Boost与设计模式

推荐书目[1]是软件开发领域的经典著作，它提出了设计模式的概念，系统地总结了面向对象开发的专家经验，并用简洁易懂的形式表达出来，使开发经验得以文档化。设计模式有助于软件开发开发者更好更快地理解面向对象的精髓，从面向过程、基于对象的开发过渡到面向对象的开发，设计出高内聚低耦合、结构良好、灵活健壮的程序。

什么是设计模式呢？在推荐书目[1]给出的定义是：

“设计模式针对面向对象系统中重复出现的设计问题，提出了一个通用的设计方案，并予以系统化的命名和动机解释。它描述了问题、解决方案、在什么条件下使用该解决方案及其效果。它还给出了实现要点和实例。该解决方案是解决该问题的一组精心安排的通用的类和对象，再经定制和实现就可用来解决特定上下文中的问题。”

从定义中可以知道，设计模式是一个面向对象的通用解决方案，它可以解决很多软件开发中的常见问题，理解了解决问题的基本模式，只需要再进行少量定制就可以解决自己遇到的特定问题。推荐书目[1]一共包含 23 个设计模式^①，分为创建型模式、结构型模式和行为模式三个大类，本章将结合 Boost 程序库简要介绍这些重要的模式，以便读者从设计模式的角度更深入地理解 Boost。

15.1 创建型模式

面向对象的软件开发的基础是对象。随着系统的不断演化，会出现越来越多的对象，如果单纯

① 不知道是巧合还是有意为之，这个数目恰好等于人类染色体的数量，这个数字还出现在经典科幻电影《黑客帝国》中。

使用 C++ 提供的 `new` 操作符, 将使程序到处都是硬编码的对象创建代码, 很难适应变化。而创建型模式抽象了类的实例化过程, 它封装了对象的创建动作, 使对象的创建可以独立于系统的其他部分。

抽象工厂 (Abstract Factory)

抽象工厂模式把对象的创建封装在一个类中, 这个类唯一的任务就是按需生产各种对象, 通过派生子类的方式抽象工厂可以生产不同系列的、整套的对象。工厂类通常是单件, 以保证在系统的任何地方都可以访问, 其中的每个方法都是工厂方法。在较小的软件系统中, 抽象工厂有时候会退化成一个没有子类的简单工厂。

expressive 库有一个抽象工厂 `regex_compiler` (参见 5.5.10 小节, 209 页), 可以生产各种正则表达式解析对象, 但它用模板技术而不是继承实例化出了具体的工厂。

生成器 (Builder)

生成器模式分解了复杂对象的创建过程, 创建过程可以被子类改变, 使同样的过程可以生产出不同的对象。生成器与抽象工厂不同, 它不是一次性地创建出产品, 而是分步骤逐渐地“装配”出对象, 因而可以对创建过程进行更精细的控制。

Boost 库没有生成器模式的具体应用, 因为生成器模式主要用来构造复杂的对象, 对于库来说复杂的创建过程会令库难以使用。

`multi_array` (参见 7.9 小节, 302 页) 对象的创建过程类似生成器模式, 它先用模板参数设定基本维数, 然后再逐个指定各个维度, 最后生成一个多维数组。因此, 本书中为它实现了一个生成器 `multi_builder`, 它隔离了 `multi_array` 的创建与使用, 使两者可以随意变化而互不影响。

工厂方法 (Factory Method)

工厂方法模式是另一种生产对象的方式, 它把对象的创建封装在一个方法中, 子类可以改变工厂方法的生产行为生产不同的对象。工厂方法所属的类不一定是工厂类 (抽象工厂或者生成器), 它可能是一个普通类、一个框架类, 或者是一个自由函数。^①

Boost 库中的 `make_shared()` (参见 3.4.4 小节, 74 页)、`make_optional()` (参见 4.3.5 小节, 104 页)、`make_tuple()` (参见 7.6.3 小节, 277 页)、`regex<>::compile()` (参见 5.5 小节, 196 页) 等函数都属于工厂方法模式。

^① Boost1.43 版在 `functional` 库里新增加了一个 `factory` 组件, 实现了对 `new` 操作符的完全封装, 类似于泛化的 `make_pair()`、`make_optional()`。

原型 (Prototype)

原型模式使用类的实例通过拷贝的方式创建对象，具体的拷贝行为可以定制。它最常见的用法是为类实现一个 `clone()` 成员函数，这个函数创建一个与原型相同或相似的新对象。

因为 C++ 不能高效地返回一个对象类型，因此实践中很少有完全实现的原型模式，但大多数 Boost 库组件都提供拷贝构造函数和赋值操作符重载，可以部分地实现原型模式的功能。

`weak_ptr` 的 `enable_shared_from_this` 用法（参见 3.6.3 小节，84 页）类似于一个原型模式，它创建了一个指向自身的 `shared_ptr`。exception 库提供了一个 `enable_current_exception()` 函数，它被用于线程安全地处理异常，返回一个 `clone` 的异常对象。

另一个指针容器库 `pointer container` 容纳的指针不允许共享，如果要拷贝指针容器，则需要被容纳的元素提供 `clone()` 操作。

单件 (Singleton)

单件模式保证类有且仅有一个实例，并且提供一个全局的访问点。通常的全局变量技术虽然也可以提供类似的功能，但它不能防止用户创建多个实例。单件的基本原理很简单，但有很多实现的变化。

Boost 库目前没有专门的单件库，但在 `pool` 库和 `serialization` 库提供了两个较好的实现（参见 4.6 小节，116 页）

15.2 结构型模式

结构型模式专注于如何组合类或对象进而形成更大更有用的新对象。

组合对象有两种方式，第一种是 C++ 语言本身提供的继承机制，但它在编译期就已经确定了对象的关系，无法在运行时改变，缺乏足够的灵活性。第二种方法是运行时对象组合，不同的对象之间彼此互相独立，仅通过定义良好的接口通信协同工作，它更灵活和易于模块化，但因为组合方式富有变化而较难以理解。

适配器 (Adapter)

适配器模式把一个类的接口转换（适配）为另一个接口，从而在不改变原有代码的基础上复用原代码。它的别名 `wrapper` 更清晰地说明了它的实现结构：包装了原有对象，再给出一个新的接口。

`array` 类（参见 7.1 小节，239 页）是适配器模式的一个很好例子，它把原始数组适配成了

符合 STL 标准的容器，使数组可以与其他标准库组件（算法、迭代器、其他容器）协同工作。

同样，`multi_array_ref` 和 `const_multi_array_ref`（参见 7.9 小节，302 页）把原始数组适配成了 Boost 的多维数组容器。

桥接（Bridge）

桥接模式分离了类的抽象和实现，使它们可以彼此独立地变化而互不影响。桥接模式与适配器模式有些相似，在两个对象之间加入了一个中间层次，提供间接联系增加了系统的灵活性。但两者的意图不同，适配器模式关心的是接口不匹配的问题，不关心接口的实现，只要求对象能够协同工作。桥接模式的侧重点是接口和实现，通常接口是稳定的，桥接解决实现的变化问题。

智能指针的 `pimpl` 用法（参见 3.4.6 小节，76 页）是桥接模式的一个应用，它向外界提供接口的同时并没有暴露任何内部的实现信息，因此实现可以任意地改变而不会影响到使用接口的客户代码。

桥接模式的另一个例子是随机数库 `random` 的硬件随机数发生器（参见 9.4.9 小节，367 页）。`random_device` 的对外接口不变，而内部的 `pimpl` 指针可以采用不同的硬件设备从而有不同的实现。

组合（Composite）

组合模式将小对象组合成树形结构，使用户操作组合对象如同操作一个单个对象。组合模式定义了“部分—整体”的层次结构，基本对象可以被组合成更大的对象，而且这种操作是可重复的，不断重复下去就可以得到一个非常大的组合对象，但这些组合对象与基本对象拥有相同的接口，因而组合是透明的，用法完全一致。

`xpressive` 库（参见 5.5 小节，196 页）的静态用法就利用了组合模式，它定义了许多小的正则表达式元素，通过重载操作符把它们逐个组合起来，形成一个大的正则表达式，而这些正则表达式又可以被继续组合下去。

`property_tree`（参见 7.10 小节，314 页）更好地诠释了组合模式。属性树的每一个子节点也是属性树，属性树可以有任意复杂的组合，但最终呈现给用户的还是一个 `basic_ptree` 接口，使用时完全不需要关心它内部的复杂结构，这是一个透明的接口。

`multi_array`（参见 7.9 小节，302 页）也是组合模式的一个具体应用，它是递归定义的，每个维度都是一个 `multi_array`。

装饰（Decorator）

装饰模式可以在运行时动态地给对象增加功能。它也是对对象的包装，但与适配器模式不同

的是它并没有改变被包装对象的接口，只是改变了它的能力范围，而且可以递归组合。

通过生成子类的方式也可以为对象增加功能，但它是静态的，而且大量的功能组合很容易产生“子类爆炸”现象。装饰模式可以动态、透明地给对象增加职责，并且在不需要的时候很容易去除，使用派生子类的方式无法达到这种灵活程度。

operators 库（参见 4.8 小节，125 页）的基类链技术很类似装饰模式，用后一个运算概念装饰前一个运算概念，不断组合增加了操作符重载的能力，但它是运用泛型编程技术在编译期实现的。

外观 (Facade)

外观模式为系统中的大量对象提供一个一致的对外接口，以简化系统的使用。外观是另一种形式的 wrapper，它不是包装一个对象，而是包装一组对象，简化了这组对象间的通信关系，给出一个高层次的易用接口。但外观并不屏蔽系统里的对象，如果需要，用户完全可以越过外观的包装使用底层对象以获得更灵活的功能。

随机数库 random 的变量发生器（参见 9.4.7 小节，364 页）就是一个外观模式，它屏蔽了 random 库内部的大量细节，给用户提供一个可轻松生成随机数的 operator()。

享元 (Flyweight)

享元模式使用共享的方式节约内存的使用，可以支持大量细粒度的对象。它将对象的内部状态与外部状态分离，配合工厂模式（抽象工厂或工厂方法）生成仅有内部状态的小对象，工厂内部保持小对象的引用计数从而实现共享，外部状态可以通过计算得到。

xpressive 库的 regex_compiler（参见 5.5.10 小节，209 页）不仅是一个抽象工厂模式，它同时也是享元模式，在内部保存了所有正则表达式对象从而实现共享。Boost 里还有一个 flyweight 库直接实现了享元模式。

代理 (Proxy)

代理模式与适配器模式、装饰模式很像，也包装对象，但它的意图不是改变接口插入新系统（适配），也不是为对象增加职责（装饰），而是要控制对象。外界不能直接访问对象，必须通过代理才能与被包装的对象通信。

代理模式的应用非常广泛，smart_ptr 库（参见 3.1 小节，61 页）就是代理模式的最佳应用。scoped_ptr、shared_ptr 等智能指针包装了原始指针，代理了原始指针的职能，用户只需使用智能指针的代理就可以获得原始指针同样的功能，而且不用担心资源泄漏，因为智能指针控制了原始指针的行为。

optional（参见 4.3 小节，101 页）、ref（参见 11.2 小节，422 页）、bind（参见 11.3

小节, 429 页) 和 `function` (参见 11.4 小节, 437 页) 也属于代理模式, 它们都包装了原始的函数或者对象, 为它们提供一定程度的控制, 在需要时把消息转发给原始的函数或对象完成工作。

15.3 行为模式

行为模式关注的是程序运行时的对象通信和职责分配, 跟踪动态的、复杂的控制流和消息流, 比创建型模式和结构型模式更难于掌握。通常对象一旦创建, 它们就立即联系起来, 这种联系是动态的, 很难甚至不可能从代码中看出来。行为模式以可文档化的形式描述对象通信机制, 可以帮助我们深入了解对象之间的关系。

行为模式大都采用对象组合, 封装程序的可变部分。

职责链 (Chain of Responsibility)

职责链模式把对象链成一条链, 使链上的每个对象都有机会处理请求。职责链把请求的发送者和接收者解耦, 使两者都互不知情, 而且职责链中的对象可以动态地增减, 从而增强了处理请求的灵活性。

`assign` 库 (参见 4.4 小节, 106 页) 的工作原理类似职责链模式, 但链上仅有一个对象, 它使用重载操作符 `operator()` 和 `operator`, 将赋值请求连接成一个链逐个处理, 最后完成赋值或初始化工作。

`iostreams` 库也使用了职责链模式, 它定义了 `source`、`sink`、`filter` 等概念, 对象间可以串联起来, 一个的输出作为另一个的输入, 完成流处理的功能。

命令 (Command)

命令模式把请求封装成一个对象, 使请求能够存储更多的信息拥有更多的能力。命令模式同样能够把请求的发送者和接收者解耦, 但它不关心请求将以何种方式被处理。命令模式经常与职责链模式和组合模式一起使用: 职责链模式处理命令模式封装的对象, 组合模式可以把简单的命令对象组合成复杂的命令对象。

`exception` 库 (参见 4.9 小节, 136 页) 是命令模式的一个例子。它把错误信息包装在异常中, 使用 C++ 的异常机制传递, 直到有一个 `catch` 块处理它。

解释器 (Interpreter)

解释器模式定义了一个类体系, 用于实现一个小型语言的解释器。它与组合模式很相似, 而且常常利用组合模式来实现语法树的构建。

xpressive (参见 5.5 小节, 196 页)、regex、spirit、wave 等库都使用了解释器模式, 前两者可以解析正则表达式, spirit 实现了 EBNF 语法解析器, wave 则是一个 C/C++ 的预处理解析器。

program_options (参见 10.4 小节, 400 页) 也是一个解释器模式应用, 它可以解析命令行参数这种简单的语法结构。

迭代器 (Iterator)

迭代器模式把按某种顺序访问一个集合中的元素的方式封装在一个对象中, 从而无须知道集合的内部表示就可以访问集合。

迭代器模式可能是面向对象软件开发中应用的最广泛的一个设计模式, 在 STL 中就已经有了大量的迭代器实践, 它是泛化的指针, 被用于以正序或逆序遍历容器内的元素。boost.range 和 boost.iterators 在 STL 的基础上进一步深化创新了迭代器的概念。

中介者 (Mediator)

中介者模式用一个中介对象封装了一系列对象的交互联系, 使它们不需要相互了解就可以协同工作。中介者模式在存在大量需要相互通信对象的系统中特别有用, 因为对象数量的增加会使对象间的联系非常复杂, 整个系统变得难以理解难以改动。这时中介者可以把这些对象解耦, 每个对象只需要与中介对象通信, 中介对象集中了控制逻辑, 降低了系统的通信复杂度。

中介者模式与观察者模式是相互竞争的模式, 通常观察者模式比中介者模式更容易生成可复用的对象, 中介者模式如果使用不当很容易导致中介对象过度复杂, 抵消了模式带来的好处。

Boost 库暂没有中介者模式的应用例子。

备忘录 (Memento)

备忘录模式可以捕获一个对象的内部状态, 并在对象之外保存该状态, 在之后可以随时把对象恢复到之前保存的状态。

io_state_savers 库 (参见 10.1 小节, 371 页) 的实现类似备忘录模式, 它可以保存 IO 流的各种状态, 在析构时自动恢复, 防止 IO 流因状态异常而发生错误。

观察者 (Observer)

观察者模式定义了对象间一对多的联系, 当一个对象的状态发生变化时, 所有与它有联系的观察者对象都会得到通知。观察者模式将被观察的目标和观察者解耦, 一个目标可以有任意多的观察者, 观察者也可以观察任意多的目标, 构成复杂的联系, 而每个观察者都不知道其他观察者

的存在。

观察者模式是一个非常著名、威力强大的设计模式，很多编程语言都有它不同形式的实现，如 C# 的 `event/delegate` 和 Java 的 `Observable/Observer`。Boost 库提供 `signals` 和 `signals2`（参见 11.5 小节，445 页）完全实现了观察者模式。

状态（State）

状态模式允许对象在状态发生变化时行为也同时改变。

状态转换通常的做法是对象内部有一个值来保存当前的状态，根据状态的不同使用 `if-else` 或者 `switch` 来执行不同的功能。这样会使类中存在大量结构类似的分支语句，变得难以维护和理解。状态模式消除了分支语句，把状态处理分散到了各个状态子类，每个子类集中处理一种状态，使状态的转换清晰明确。

`boost.statechart` 实现了有限状态自动机，它是状态模式的泛化。

策略（Strategy）

策略模式封装了不同的“算法”，使它们可以在运行时相互替换。它与结构型模式里的装饰模式功能接近，策略模式改变类的行为内核，而装饰模式改变类的行为外观。如果类的接口很庞大，那么装饰模式的实现代价就过高，而策略模式仅改变类的内核，可能很小。策略模式的实现结构很像状态模式，但它不改变对象的状态。

标准库和 Boost 库中的大量函数对象就是策略模式的应用。函数对象封装了各种操作，标准算法或者其他类使用传入函数对象来动态改变它的行为。

Boost 库中大部分组件的模板类型参数也可以看作是策略模式，通过配置不同的模板类型，最后实例化的模板类内部的算法都不相同。

模板方法（Template Method）

模板方法模式在父类中定义操作的主要步骤，但并不实现，而是留给子类去实现。注意这个模板与 C++ 中泛型编程用的 `template` 没有任何联系，不要引起误解。它常见的用法是“钩子操作”，父类定义了所有的公开方法，在公开方法中调用保护的钩子方法，子类实现不同的钩子方法来扩展父类的行为。

模板方法模式是一个非常基本的设计模式，也可能是最容易使用的一个设计模式，许多框架都使用模板方法定义基本的操作步骤，用户只需实现少量的具体化代码就可以利用框架的全部功能。

`boost.test`（参见 6.3 小节，219 页）定义了单元测试的框架，它使用模板方法模式定义了

许多可扩展的方法, 用户只需要依据 `test` 库的规则编写测试用例, 就可以插入到 UTF 中进行测试。

访问者 (Visitor)

访问者模式分离了类的内部元素与访问它们的操作, 可以在不改变内部元素的情况下增加作用于它们的新操作。如果一个类有很多内部数据, 因此也就有很多访问操作, 这样会使它的接口非常庞大, 难以变动难以学习。访问者模式可以做到数据的存储与使用分离, 不同的访问者可以集中不同类别的操作, 并且可以随时增加新的访问者或者新方法来增加新的操作。

`boost.variant` (参见 7.8 小节, 294 页) 提供 `static_visitor` 实现了访问者模式, 可以对一个很小的 `variant` 对象实施各种操作, 如果 `variant` 对象发生改变, `static_visitor` 也可以很容易地适应变化。

15.4 其他模式

推荐书目 [1] 成书于 1994 年, 在那以后软件界开始了广泛的模式运动, 在软件开发的各个领域各个层次都逐步发现了很多新的模式, 但最经典最基本的仍然是上面讲到的 23 个设计模式。本小节介绍一些较为常用的其他模式, 供读者参考。

空对象 (Null Object)

空对象模式又称哑对象模式 (Dumb Object), 它是一个行为模式, 扩展了空指针的涵义, 给空指针一个默认的、可接受的行为, 通常是空操作, 可以说是一个“智能空指针”。使用空对象模式, 程序就可以不必用条件语句专门处理空指针或类似的概念, 所有的对象都会有一致的、可理解的行为。

空对象模式可以和许多行为模式配合, 充当“哨兵”的角色, 完善它们的概念。例如, 策略模式有一个空策略对象, 它将不做任何事情; 职责链模式把空对象用在链的末尾, 它可以“吞下”所有无法处理的请求; 空迭代器对象则适用于叶子节点, 表示总完成遍历操作; 还可以有空命令、空观察者等等。

在 `tuple` 中的 `null_type` 就是一个空对象, 它表示了一个空的 `tuple`, 什么也不做, 但它非常重要, 没有它就无法完成 `tuple` 的部件链。

空对象模式还可以应用在 `pointer container` 库中, 使指针容器可以安全地容纳空指针, 用一致的行为处理容器元素。

空对象模式也不仅用在面向对象的软件构建中, 操作系统也有类似的概念, 如 Linux 的 `dev/null` 设备。

包装外观 (Wrapper Façade)

包装外观模式很类似外观模式,但它包装的目标不是一个面向对象子系统,而是底层的 API。包装外观模式把大量的原始 C 接口分类整理,给外界一个统一的、面向对象的易用接口,增强了原始底层接口的内聚性,同时又没有效率的损失(使用静态成员函数、名字空间和 inline 关键字)。包装外观模式可以屏蔽系统底层的细节,有利于外界不受平台变化的影响,增强可移植性。

system 库(参见 10.2 小节, 374 页)是包装外观模式的一个例子,它对 Windows 和 Unix 等操作系统的错误代码分类包装,对外提供了一个方便使用、易于理解的接口。

thread(参见 12.1 小节, 467 页)、interprocess 和 asio 库(参见 12.2 小节, 493 页)也都在不同的层次上使用了包装外观模式,提供了可移植的并发处理功能。

前摄器模式 (Proactor)

前摄器模式是应用于异步调用的设计模式,它的核心是前摄器、异步的操作处理器、异步的事件多路分离器和完成事件队列,可以不使用线程实现异步操作。

前摄器模式的基本流程可以简要描述如下:

前摄发起器创建一个完成处理器,用于在异步调用完成后的回调,然后发起一个异步操作,交给操作处理器异步执行,当异步操作完成时操作处理器将把事件放入完成事件队列。前摄器调用多路分离器从完成事件队列中获得事件,分派事件回调完成处理器执行所需的后续操作。

前摄器模式用于异步调用有很多的好处,它封装了并发机制,将并发机制与线程的执行解耦,简化了功能代码的编写,不需要考虑多线程的同步问题,能够提供高性能的异步操作。但它也有缺点,模式比较复杂,处理流程难以理解和调试。

asio 库(参见 12.2 小节, 493 页)基于操作系统的异步调用机制实现了可移植的前摄器模式,解耦了应用程序与操作系统,可以高效地实现异步 IO 操作。

15.5 总结

本章结合 Boost 程序库介绍了推荐书目[1]中的 23 个经典设计模式和 3 个常用的其他设计模式。Boost 库应用了以上几乎所有的设计模式,因此学习设计模式有助于更好更快地理解 Boost 组件的结构和用法。

设计模式通常分为三类^①：创建型模式管理面向对象系统中对象的产生，结构型模式管理面向对象系统中对象的组合，行为模式管理面向对象系统中对象的通信。

创建型模式是程序的基础，因为面向对象系统就是由许许多多的对象组成的，创建型模式能够使对象的创建独立于系统单独变化。最常用的创建型模式是抽象工厂和单件，而且这两者经常联合使用。

结构型模式把对象组合起来，以获得功能更强大更灵活的对象。使用结构型模式，我们可以避免继承滥用和“子类爆炸”现象，并在不改变原有类的基础上生成许多新的可用对象。最常用的结构型模式有适配器、组合、装饰、外观和代理。

行为模式确定了多个对象间通信与合作的最佳方式，它刻画了对象之间的合作机制，这种合作机制高效且能够适应未来的变化。最常用的行为模式有迭代器、观察者、状态、策略和访问者。

学习设计模式时理解模式的目的、意图和用途很重要，因为编程语言的语法、语义限制，很多设计模式的实现结构很相似甚至完全相同（例如装饰和代理），但不同的设计出发点（意图）和应用领域导致了模式的使用法有很大的区别。

设计模式也经常与重构（refactoring）联系在一起，重构会导致应用设计模式，而设计模式的目的是为了避免将来的重构。

① 近年来《设计模式》一书的作者们拟对模式的分类做一些调整，重新划分为“核心”、“创建”、“周边”、“其他”等四个类别，以突出模式的重要程度，但目前还没有正式定论。

第 16 章

结束语

看到这里本书已接近结束，相信此时读者已经对 Boost 库有了一个全面的了解，本章将讨论一些 Boost 程序库之外的东西。

16.1 未臻完美的 Boost

任何人都会承认，Boost 已经非常强大，它为 C++ 程序员提供了无与伦比的编程武器，让 C++ 开发工作变得更愉快、更高效。但 Boost 还谈不上尽善尽美，还缺少一些实际开发工作所需的库，还没有达到 Java 或者 Python 标准库那种“包罗万象”的程度：没有 GUI 库，没有 RPC 库，没有 COM+、CORBA 支持，没有……可以开列出一个很长的清单。

这很正常，毕竟世上无完美，我们不能过分苛求 Boost 库的作者们。Boost 库（和 C++）现在还处于持续发展的阶段，还有更多的潜力没有被释放出来。来期待一下 Boost 的新版本会在哪些方面有所改进或者增强吧，也许不久的将来就会梦想成真，具体改进应该在以下几个方面。

- 日志库。这也许目前最需要的。打印日志是程序除错的重要手段，在某些情况下甚至是唯一的手段。在 2006 年曾经有一个日志库提交 Boost 评审，但很遗憾地被否决了，这方面的空白一直延续到现在。但今年有个好消息，一个全新的日志库进入了 Boost 的评审流程，它采用了不同于 log4j 的全新架构，不仅能够支持记录日志，还能够用于其他领域。经过一个多月的严格评审，它已经在三月份被正式接纳为 Boost 的一部分，将出现在 1.43 版或稍后的版本中^①，让我们拭目以待。
- 更多的编程语言支持。增加对 Lua、Perl、Ruby 等流行脚本语言的支持，如果可能的话

① 可惜的是 1.43 版并未正式发布 log 库，Boost 社区还正在对 log 库做更进一步的测试和验证。

还有 JNI，以便使 C++ 更好地与其他系统集成。

- 数据库访问支持。目前的 C++ 语言缺乏像 Java 中 JDBC 那样的标准数据库访问接口，虽然市面上每个数据库产品都有 C/C++ API 供使用，但这些 API 都不相同，学习成本高，也造成数据库应用程序难以移植，希望 boost.dbi 库能够解决这个问题。
- 垃圾回收。这方面 C++ 社区已经取得了一些有意义的进展，已经出现了若干垃圾回收库实现。不过 Boost 已经提供了相当丰富的内存管理工具（如 smart_ptr、pool、pointer container），只要运用得当完全不需要去关心内存释放的问题，因此这个库的意义不大。
- XML 解析库。XML 是非常流行的数据交换格式，已经成为了事实上的标准，大有一统天下的趋势，Boost 库提供一个标准的 XML 处理库很有必要。虽然目前没有正式的 Boost XML 解析库，但在 1.41 版出现的 property_tree 自带了小巧快速的 rapidxml，基本功能齐全，风格与 Boost 接近，可以暂解燃眉之急。
- 高级网络通信协议库。asio 库为 C++ 提供了跨平台的底层通信协议支持，可以使用 socket、TCP/IP 等手段编写网络程序，但还缺少在这之上的高级网络协议，例如 FTP、HTTP、SOAP 等等，它们都是现在非常流行的通信协议，如果有这样一个库相信会大大增强 C++ 的网络编程能力。asio 库已经建立了一个很好的底层通信架构，虽然它还没有达到稳定的阶段，但新的网络通信库必然会以它作为实现基础。

16.2 让 Boost 工作得更好

Boost 不是 C++ 程序员唯一可用的免费开源程序库，它也不是万能的，在 Boost 能力范围之外有时候可能使用其他库会有更好的解决办法，会让 Boost 工作得更好。下面就简单介绍几个开源库，为读者开拓一下思路。

SQLite



SQLite 是一个纯 C 写成的超轻量级的数据库，能够以源码的形式轻松嵌入工程，最重要的是它的使用许可协议，比 Boost 还要宽松，允许任何人用于任何目的。

SQLite 不使用流行的服务器-客户端架构，而是使用单独的文件作为数据库，支持绝大多数 SQL 功能，可以创建索引、视图、触发器，速度很快。SQLite 的另一个特点是数据类型的自适应，可以在字段中存储任意数据而不必担心类型不符（但为了移植考虑还是应该使用标准的 SQL 语句）。

SQLite 目前还在不断发展，最新版本是 3.6.x，仅提供 C 接口，但有若干 C++ 包装类可用，

如 CppSQLite、SQLite++、WxSQLite 等。

简单示范使用 CppSQLite 来操作 SQLite 数据库的代码如下：

```
CppSQLite3DB db;                                //一个 SQLite 数据库对象

cout << "SQLite Version: " << db.SQLiteVersion() << endl;
db.open(":memory:");                            //打开内存数据库

db.execDML("create table test(id int, name char(20));");

int nRows = db.execDML("insert into test values (1, 'Andersen');");
cout << nRows << " rows inserted" << endl;

nRows = db.execDML("update test set name = 'neo the one' where id = 1;");
cout << nRows << " rows updated" << endl;

nRows = db.execDML("delete from test where id = 1;");
```

另一个 SQLite 的包装库 SQLite++ 的代码风格更接近 STL 和 Boost 的现代 C++ 风格，例如：

```
using namespace sqlitepp;
session db(":memory:");                          //打开内存数据库

db << "create table test(id int, name char(20));" //重载流操作符
db << "insert into test values (1, 'Andersen');";

statement st(db);
int n;
string name;
db << "select * from test", into(n), into(name);
```

Crypto++

Crypto++ 是由美籍华裔 Wei Dai 开发的一套密码学程序库^①，大量使用了泛型编程技术，与 Boost 颇神似，最新版本是 5.60。

Crypto++ 囊括了几乎所有已知的公开密码学算法的实现，包括：

■ 摘要算法：MD5、SHA1、SHA256 等；

① 本书即将完成之时，Boost 社区已经出现了一个要求增加密码学库 crypto 的提案。它基于另一个开源项目 Botan，同样是一个非常大且全面的密码学库，而且支持 X509 证书。

如果一切顺利，crypto 可能在 2010 年内就完成开发、评审和批准等一系列的工作，将为 asio、iostream 等库实现加密功能提供极大的便利。

- 对称加密算法: DES、AES、IDEA、XTEA 等;
- 非对称加密算法: RSA、ECC、DH、DSA 等
- 加密模式: ECB、CBC、CFB、OFB 等;

除密码学算法外, Crypto++还提供很多实用功能, 如 Base64/HEX 编码解码、随机数发生器、Socket 封装、数据压缩等等。特别值得一提的是 Crypto++的输入输出框架, IO 流手法简直就是 boost iostream 库的绝佳示范代码。

示范使用 Crypto++压缩数据文件的代码如下:

```
FileSource source("readme.txt", true,           //一个文件 source 起始
    new ZlibCompressor(                          //zlib 压缩过滤器
        new FileSink("readme.zip")));           //文件 sink 终点
```

log4cplus

log4cplus 是模仿著名的 log4j 的一个 C++日志库, 实现了较全面的日志功能, 可以记录日期时间、源代码行、信息级别等信息, 输出可以是屏幕、磁盘文件、Socket 流, 甚至是 syslogd 服务。在 boost.log 正式推出之前, log4cplus 是一个较好的替代品。

log4cplus 的用法和配置较为复杂, 因为 boost.log 即将登场, 本书不对它作进一步的介绍。同类的开源日志库还有 log4cxx、log4cpp 等, 它们基本上都使用的是 Java 日志库 log4j 的架构。

pugixml

pugixml 是一个不逊于 rapidxml 的 xml 解析器, 采用现代 C++风格实现, 速度快, 还支持 xpath/xquery, 在 boost.property_tree 处理 xml 力不从心时是个很好的选择。

下面的例子简单示范了 pugixml xpath 的使用, 选择属性 id 是 'queryFromDB' 的 select 节点:

```
xpath_query query("/select[@id='queryFromDB']");
cout << query.evaluate_string(doc) << endl;
```

wxWidgets

wxWidgets 是一个图形界面程序库, 同时也提供文件流操作、多线程、数据库访问等其他功能, 是一个非常全面的 C++程序库, 使用 wxWidgets 可以不受操作系统的限制编写出跨平台的图形界面应用程序。此外 wxWidgets 还提供 Python、Perl、Ruby 等脚本语言的绑定支持。

不过 wxWidgets 的一个缺点是缺乏简单方便的界面绘制工具, 经常需要程序员完全使用代码

来实现所有窗口控件的控制，虽然也存在一些辅助工具，但仍然没有 VisualStudio 那么方便。

16.3 工夫在诗外

程序员是一个很特殊的职业，他更多是用头脑而不是双手来创造财富，世界上最富有的人 Bill Gates 也曾经是一个程序员。成为优秀的、成功的程序员，一直是许多人（也许读者也是其中之一）孜孜以求的目标。

在本书即将结束之时，作者提出一些自己的体会和经验，也许能给您一点启迪：

- 使用良好的编程风格。不要吝啬空格、空行，总使用 TAB 缩进，为变量、函数、类起个好的名字，勤加注释，让代码更容易阅读和维护，这些都是一个专业的程序员所必须具备的素质。
- 引用一句编程忠言：有两种编写程序的方式，一种是把代码写的非常复杂，以至于看不出明显的错误；另一种是把代码写的非常简单，以至于明显看不出错误。聪明的读者显然应该选择后一种方式。
- 注重测试尤其是单元测试。测试使程序中的 bug 消灭在萌芽时期，使代码更加健壮，好的测试代码也是笔宝贵的财富。当修改代码时，回归测试会确保小改动不会导致整个系统发生错误。boost.test 库提供了完善的单元测试框架，我们应该总使用它来伴随软件的整个开发生命周期。
- 善用开源库。“不要重复发明轮子”，这句忠告已经在软件界重复了无数次，然而效果却依然微乎其微。如果在开发中遇到了什么问题，请先向 STL、Boost 寻求帮助，如果不能解决，再用因特网搜索可能的解决方案，最后才是自己动手解决。大多数开源库都实现的足够好，能够满足 80% 的需求。即使它不能完全适合你，你也拥有了一个好的起点，而不至于要白手起家。16.2 节（560 页）介绍了几个开源库，读者可以在因特网上找到更多的资源。
- 不能仅了解一门编程语言，这样很容易僵化解决问题的思路。在精通 C++ 之余也应该多了解一下其他语言的发展和知识，可以开阔你的视野，从而更好地发现 C++ 的优点和不足。作者在此推荐 Python，它的动态语言特性与 C++ 有极好的互补性，而且 boost.python 也使得两者可以很容易协同工作。
- 方法学很重要。应该熟悉编程语言之上的各种开发范式，如设计模式、重构技术、UML 建模、敏捷开发、极限编程等。它们不一定都适合你，但可以从中汲取有用的知识，帮助你在更高的层次上看待问题进而解决问题。本书的第 15 章（547 页）简要介绍了设计模式相关的基本知识，可供参考。

- 使用好的开发工具。易用的、高效率的开发工具可以节约程序员大量宝贵的时间，让他把精力集中在需要处理的问题上，而不是其他易分心的事情。对于 C++ 集成开发环境来说，应该具备语法高亮、代码提示、版本控制等功能。在 Windows 平台上 MSVC 几乎是唯一的选择，但读者也应该看看其他选择，DevC++、Code::Blocks、CodeLite 都是免费的 C++ 开发环境，试一下，也许会发现你所喜欢的特性。

最后，生活中不只有 C++、代码和编程，还有更多的东西值得我们去体味，朋友、亲人、爱人都更值得花时间与他们在一起。走出办公室，离开计算机，去亲近大自然，享受美餐，散散步，打打球……拥有美好的生活才能够创造出完美的程序。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真题解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总: **ASP.NET** 篇

.Net 技术精品资料下载汇总: **C#**语言篇

.Net 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子书下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

附录 A

推荐书目

这里列出了一些作者认为非常值得阅读的 C++ 经典书籍，它们也是作者编写本书时的案头必备参考资料，在此与读者共享。

- [1] Erich Gamma 等著，李英军等译.《设计模式 可复用面向对象软件的基础》. 北京:机械工业出版社
软件开发历史上里程碑式的著作，设计模式的开山作品，里面提出的 23 个设计模式已经成为软件界的经典，被无数其他论文或书刊引用，也被无数的软件系统（包括 STL 和 Boost）所验证并使用。本书是每一个精益求精的程序员都必须拥有的宝典和圣经，可以说是字字珠玑，值得经常翻阅以获取设计灵感。
- [2] Nicolai M. Josuttis 著，侯捷/孟岩译.《C++ 标准程序库 自修教程与参考手册》. 武汉:华中科技大学出版社
全面分析讲解 C++98 标准库，内容详细丰富，是学习 STL 的经典书籍，也可能是最好的书籍。不过较之 STL 的精妙论述，书中对 IO 流和国际化的讲解稍嫌单薄。
- [3] Bjarne Stroustrup 著，裘宗燕译.《C++ 语言的设计和演化》. 北京:机械工业出版社
C++ 语言之父所著，介绍了 C++ 语言的发展历史、设计理念，同时也详细描述了 C++ 的各种语言特性，从中可以更深刻地理解 C++ 语言的内涵。
- [4] Stanley B Lippman 等著，潘爱民等译.《C++ Primer 第三版》. 北京:中国电力出版社
C++ 入门的经典教材，不仅适合初学者，对于 C++ 熟手也有很大的参考意义。书中全面讲解了 C++98 标准定义的语言的方方面面，如果在 C++ 开发过程中对某项语言特性拿不准，可以向它求助。

- [5] Scott Meyers 著. 侯捷译. 《Effective C++ 中文版 第 3 版》. 北京: 电子工业出版社
享誉世界的“Effective 系列”, 提供了 55 个如何编写高效 C++ 代码的忠告。第 3 版较前两版变动极大, 针对 C++98 标准更新了大部分条款, 并简单介绍了 Boost 库。
- [6] Scott Meyers 著. 《Effective STL》
同属“Effective 系列”, 讲解如何高效地使用 STL, 对容器、算法等 STL 组件的使用提出了很多精辟的见解。
- [7] Douglas 等著. 於春景译. 《C++ 网络编程 卷 1 运用 ACE 和模式消除复杂性》. 武汉: 华中科技大学出版社
ACE 开发者编写的网络中间件著作, 站在领域分析的高度详细阐述了 C++ 网络编程相关的若干基本问题, 以及 ACE 是如何解决这些问题的。
- [8] Douglas 等著. 马维达译. 《C++ 网络编程 卷 2 基于 ACE 和框架的系统化复用》. 北京: 电子工业出版社
本书在卷 1 基础上深入阐述 ACE 框架的设计原理和诸多用于网络通信的设计模式。
- [9] 《C++ STL 中文版》P.J. Plauger 等著, 王昕译, 北京: 中国电力出版社
STL 四位作者联手写出的 STL 著作, 深入剖析 STL 内部实现, 想要了解 STL 实现机制、扩展 STL、学习泛型编程不容错过。
- [10] W. Richard Stevens 等著. 杨继张译. 《UNIX 网络编程 第 1 卷: 套接口 API 第 3 版》. 北京: 清华大学出版社
UNIX 网络编程的权威著作, 深入介绍 UNIX 下的网络编程的方方面面, 不过其内容完全是基于 Socket C API, C++ 程序员阅读可能存在一点困难。
- [11] Swaroop, C.H. 著, 沈洁元译. 《Python 简明教程》
帮助你快速领略 Python 的特性, 以最短的时间学会 Python 编程。

附录 B

网络资源

[1] <http://www.boost.org>

Boost 程序库的官方网站，包含了 Boost 相关的一切资料：版本更新信息、下载、文档、邮件列表、FAQ，读者应该经常访问它，以了解 Boost 的最新进展。

[2] <http://www.stlport.org>

STLport 的官方网站，但有价值的内容不是很多。

[3] <http://www.sgi.com/tech/stl/>

SGISTL 的官方网站，对每个 STL 组件都提供了详细的解说，还有一些对 STL 内部实现的技术论述。

[4] <http://www.dkuug.dk/jtc1/sc22/wg21/>

C++标准委员会官方主页，可以在这里得到 C++0x 的最新消息。

[5] <http://www.dinkumware.com>

VC 内置 STL 实现的提供商，也单独出售 STL 库。如果在使用 VC 内置 STL 时遇到了问题，在这个网站上也许可以找到一些有用的帮助。

[6] <http://www.cryptopp.com>

Crypto++的官方网站，提供了库的说明文档。

[7] <http://www.python.org>

Python 语言的官方网站。

[8] <http://www.sqlite.org>

SQLite 数据库的官方网站，有详细的在线帮助文档，包括 SQL 语法和 C 接口函数。

[9] <http://www.wxwidgets.org>

wxWidgets 的官方网站。

附录 C

C++标准简述

C++语言始于 20 世纪 70 年代末，由贝尔实验室的 Bjarne Stroustrup 博士发明。最初它被称为 C with Classes（带类的 C），1983 年末被正式命名为 C++。1990 年，C++ 标准委员会成立，C++ 标准化进程开始。1994 年，委员会完成草案登记，STL 正是在这个最后期限之前加入到了 C++ 标准。1998 年，C++ 标准表决通过并被批准，编号为 ISO/IEC 14882-1998。

C++ 把面向对象的编程方法真正引入到了现实的编程世界，是面向对象编程思想的坚定推进者，而现在则引领着泛型编程、模板元编程等新的方向。

C++98 标准

C++98 标准是 C++ 的第一个国际标准，也是目前唯一的一个 C++ 标准。与早期的 C++ 相比，它有了重大进步，包括如下：

- 新的语言特性。如模板（template）、名字空间（namespace）、布尔类型（bool）和 `const_cast` 等新的类型转换关键字。
- 异常处理体系。标准的 exception 和若干派生子类，以及异常规范。
- IO 流操作。这是对早期流库的重构标准化。
- 数值处理。在内建的整数、实数（浮点数）之外增加了复数和数值数组。
- C89 程序库。涵盖了 C 语言 1989 年标准所定义的所有内容。
- 国际化支持。`wchar_t`、`locale` 等用于国际化软件开发的特性。
- 字符串类。标准的 `string` 和 `wstring`，使程序员可以更方便地处理字符串。

- 最后，也是标准中最庞大重要的组成部分——STL。包括泛型容器（如 `vector`、`list`、`map`、`set`）、泛型容器适配器（如 `stack`）、泛型算法（如 `fill`、`sort`、`equal`、`copy`）、迭代器、函数对象、函数对象适配器、内存分配器等诸多组件。

C++98 标准是 C++ 历史上的一个重要里程碑，使编译器厂商、语言用户（程序员）都有了可遵循的统一语言规范，使程序员可以更好地写出不依赖于编译器特定扩展的“方言”的实现，极大地便利了跨平台开发。

C++0x 标准

C++98 标准并没有解决所有问题，比如遗漏了 `hash_table` 这样重要的数据结构，因此 1998 年后，标准委员会就开始为新的标准而工作。新的标准最初预计于 21 世纪的第一个十年中推出。因为日期的不确定性，新标准被称为“C++0x”。

但标准的制订工作比想象要难，很遗憾，C++0x 标准未能在 2010 年前如期出台，我们只能期待“C++1x”了。不过因为“0x”这个称呼已经叫了很多年，所以在本书中仍然用“C++0x”来称呼即将登场的 C++ 新标准。^①

C++0x 新标准较 C++98 提供了很多“与时俱进”的新语言特性，如自动推导类型声明的 `auto`/`typeof`/`decltype` 关键字、新的遍历容器的 `foreach` 循环等。库方面则增加了十余个新的组件，其中有数个基于 Boost（如 `shared_ptr`、`bind`、`tuple`、`array` 等）。目前 C++0x 的标准草案已经接近完成，但还有些讨论仍在继续中，预计可能会于 2011 年左右发布。

不过，即使 C++0x 正式公布，各个编译器厂商也不会立即推出符合新标准的产品，往往要比标准延迟几个月或者更长的时间。在它出台之前，我们最佳的选择就是使用 Boost 库来模拟实现。

^① 也有人不无幽默地认为“C++0x”中的 x 是 16 进制的。

附录 D

STL 简述

STL 即标准模板库 (Standard Template Library)，是 C++98 标准中最重要的组成部分，接下来的文字会让读者对它有个快速但比较全面的了解。

STL 的历史

STL 最初并不是 C++ 的一部分，甚至都不是用 C++ 写成的。

惠普实验室的 Alexander Stepanov (即 STL 之父) 在 20 世纪 80 年代使用 Ada 语言实现了一个泛型算法库——也就是 STL 的前身。在 C++ 增加了 `template` 特性后，Alexander Stepanov 与同在惠普实验室的 Meng Lee 合力，把 STL 用 C++ 重新改写，形成了 STL 的最初形态。1994 年，在 Bjarne Stroustrup (C++ 之父) 和 Andy Koenig (C++ 社区第二号人物) 的极力推荐下，C++ 标准草案在最后截至期限前接纳 STL 进入了 C++ 标准化进程。1998 年 C++ 标准正式通过，STL 也因此正式成为 C++ 标准的一部分。

STL 运用模板技术把数据结构与算法分离，再使用迭代器把两者粘结在一起，为程序架构提供了最大的弹性。它是一个划时代的作品，是第一个将泛型编程发挥的淋漓尽致的产品，其先进的设计思想、优雅的程序架构深刻地影响了 C++ 标准库结构和 C++ 开发范式。时至 21 世纪的第二个十年，每一个 C++ 程序员都应当像熟悉 `if`、`for` 等 C++ 基本元素一样熟悉 STL 的使用。

STL 主要有六个组成部分：容器、算法、迭代器、函数对象、适配器、内存分配器。

容器

这也许是 STL 中最引人注意的部分，它实现了计算机科学领域中数个最重要最基本的数据结构。

STL 容器可分为两大类，序列式容器和关联式容器。

序列式容器包括 `vector`、`deque` 和 `list`，分别实现了动态数组、双端队列和双向链表。关联式容器包括 `set`、`multi_set`、`map` 和 `multi_map`，分别实现了基于二叉树的集合和映射。

算法

STL 共包含近百个算法，也都是计算机科学中的经典算法，分为非变动型算法、变动型算法、移除型算法、变序型算法、排序算法、已序区间算法和数值算法共七大类。其中即有简单的 `for_each`、`copy`、`count`、`fill`，也有复杂的 `sort`、`find`。

使用 STL 算法，我们不需要编写循环语句就可以高效地完成同样的工作。

迭代器

迭代器是 C/C++ 指针的泛化，也是迭代器模式的最佳实践。通过操作符重载和模板技术，迭代器支持 `++`、`--` 等操作符，并可以如指针一样操作容器元素。

STL 中的迭代器分为输入迭代器、输出迭代器、前向迭代器、双向迭代器和随机迭代器共五类。

函数对象

早期函数对象又被称为仿函数 (functor)，是指重载了 `operator()` 操作符的类。其行为类似普通函数，但拥有普通函数不具有的优点，可以称为“智能函数”。函数对象主要用于搭配容器和算法共同工作。

适配器

适配器运用适配器模式对前述的容器、迭代器、函数对象进行适配，使程序代码可以更加灵活地组合。容器的适配器有 `stack`、`queue` 和 `priority_queue`，迭代器的适配器有逆向迭代器、插入迭代器和流迭代器等，函数对象的适配器有标准适配器和函数适配器，如 `not1`、`mem_fun` 等。

内存分配器

内存分配器用来解决不同平台的内存模型差异问题，把底层的内存分配与释放以策略模式封装，主要供 STL 内部实现使用，对 STL 用户几乎是透明的。

STL 的实现

STL 只是一个标准规范，并不是一个真正的实体，因此存在着很多个不同的具体实现，例如广泛应用在 VC 里的 Dinkumware STL、注重效率与可读性的 SGISTL、免费且高可移植的 STLport、可用于嵌入式系统的 Rogue Wave STL 等等，它们各有优缺点和适用的领域。

一个好的、符合标准的 STL 实现能够大大地提高 C++ 程序员的开发效率。

附录 E

ref_array 实现代码

这里列出本书中提出的 ref_array 和 cref_array 的完整实现代码，可以作为扩展 STL 和 Boost 的技术演示，供读者参考。

```
#include <cstddef>           //for null,size_t, ptrdiff_t
#include <algorithm>         //for swap, fill_n ...
#include <stdexcept>

#include <boost/operators.hpp>
#include <boost/assert.hpp>

#include <boost/config.hpp>

///add to boost namespace
namespace boost{

template<typename T>
class ref_array:
    private boost::totally_ordered<ref_array<T> >
{
private:
    T* elems;
    std::size_t N;
public:

    //type definitions
    typedef T          value_type;
    typedef T*         iterator;
    typedef const T*   const_iterator;
    typedef T&         reference;
    typedef const T&   const_reference;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;
```

```

//iterator support
iterator begin() { return elems; }
const_iterator begin() const { return elems; }
iterator end() { return elems+N; }
const_iterator end() const { return elems+N; }

typedef std::reverse_iterator<iterator> reverse_iterator;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

reverse_iterator rbegin() { return reverse_iterator(end()); }
const_reverse_iterator rbegin() const
{
    return const_reverse_iterator(end());
}
reverse_iterator rend() { return reverse_iterator(begin()); }
const_reverse_iterator rend() const
{
    return const_reverse_iterator(begin());
}

//operator[]
reference operator[](size_type i)
{
    BOOST_ASSERT( i < N && "out of range" );
    return elems[i];
}

const_reference operator[](size_type i) const
{
    BOOST_ASSERT( i < N && "out of range" );
    return elems[i];
}

//check range (may be private because it is static)
void rangecheck (size_type i)
{
    if (i >= size())
    {
        throw std::out_of_range("ref_array<>: index out of range");
    }
}

//at() with range check
reference at(size_type i) { rangecheck(i); return elems[i]; }
const_reference at(size_type i) const { rangecheck(i); return elems[i]; }

//front() and back()
reference front()
{

```

```

        return elems[0];
    }

    const_reference front() const
    {
        return elems[0];
    }

    reference back()
    {
        return elems[N-1];
    }

    const_reference back() const
    {
        return elems[N-1];
    }

    //size is constant
    size_type size() const { return N; }
    bool empty() const { return false; }
    size_type max_size() const { return N; }

    void resize(size_type n)
    {
        //only shrink allowed
        BOOST_ASSERT(n <= N);
        N = n;
    }

    //direct access to data (read-only)
    const T* data() const { return elems; }
    T* data() { return elems; }

    //use array as C array (direct read/write access to data)
    T* c_array() { return elems; }

    //assign one value to all elements
    void assign (const T& value)
    {
        std::fill_n(begin(), size(), value);
    }
    template<typename Cont>
    void assign (const Cont& from)
    {
        std::copy(from.begin(), from.end(), begin());
    }
    template<typename Cont>
    void operator=(const Cont& from)

```



```

    {
        assign(from);
    }

    //swap
    void swap (ref_array<T>& y) {
        std::swap(elems, y.elems);
        std::swap(N, y.N);
    }

    explicit ref_array(T *arr, std::size_t n):
        elems(arr), N(n)
    { BOOST_ASSERT(arr != NULL && "array is null"); }
    ~ref_array() {}
};

//comparisons
template<class T>
bool operator== (const ref_array<T>& x, const ref_array<T>& y)
{
    return x.size() == y.size() &&
        std::equal(x.begin(), x.end(), y.begin());
}
template<class T>
bool operator< (const ref_array<T>& x, const ref_array<T>& y)
{
    return std::lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());
}

//global swap()
template<class T>
inline void swap (ref_array<T>& x, ref_array<T>& y)
{
    x.swap(y);
}

template<typename T>
class cref_array:
    boost::totally_ordered<cref_array<T> >
{
private:
    const T* elems;
    std::size_t N;
public:
    //type definitions
    typedef T          value_type;
    typedef T*         iterator;
    typedef const T*   const_iterator;
    typedef T&         reference;

```

```
typedef const T&      const_reference;
typedef std::size_t  size_type;
typedef std::ptrdiff_t difference_type;

//const iterator support
const_iterator begin() const { return elems; }
const_iterator end() const { return elems+N; }

typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

const_reverse_iterator rbegin() const
{
    return const_reverse_iterator(end());
}
const_reverse_iterator rend() const
{
    return const_reverse_iterator(begin());
}

//operator[]
const_reference operator[](size_type i) const
{
    BOOST_ASSERT( i < N && "out of range" );
    return elems[i];
}

//check range (may be private because it is static)
void rangecheck (size_type i) const
{
    if (i >= size())
    {
        throw std::out_of_range("cref_array<>: index out of range");
    }
}

//at() with range check
const_reference at(size_type i) const { rangecheck(i); return elems[i]; }

//front() and back()
const_reference front() const
{
    return elems[0];
}

const_reference back() const
{
    return elems[N-1];
}

//size is constant
```

```

size_type size() const { return N; }
bool empty() const { return false; }
size_type max_size() const { return N; }

//direct access to data (read-only)
const T* data() const { return elems; }

//use array as C array (direct read access to data)
const T* c_array() { return elems; }

//swap
void swap (cref_array<T>& y)
{
    std::swap(elems, y.elems);
    std::swap(N, y.N);
}

explicit cref_array(const T *arr, std::size_t n):
    elems(arr), N(n)
{ BOOST_ASSERT(arr != NULL && "array is null");}
~cref_array() { }

};

//comparisons
template<class T>
bool operator== (const cref_array<T>& x, const cref_array<T>& y)
{
    return x.size() == y.size() &&
        std::equal(x.begin(), x.end(), y.begin());
}
template<class T>
bool operator< (const cref_array<T>& x, const cref_array<T>& y)
{
    return std::lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());
}

//global swap()
template<class T>
inline void swap (cref_array<T>& x, cref_array<T>& y)
{
    x.swap(y);
}

} //boost namespace

```

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真题解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余: **Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总: **ASP.NET** 篇

.Net 技术精品资料下载汇总: **C#**语言篇

.Net 技术精品资料下载汇总: **VB.NET** 篇

撼世出击: **C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总: **MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子书下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网: 精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引