The C++ Standard Library

A Tutorial and Reference

Nicolai M. Josuttis

C++ 标准链接库

C++ Standard Library

教本与手册（A Tutorial and Reference）

Nicolai M. Josuttis 着

侯捷 / 孟岩 合译

巨细靡遗 井然有序

（侯捷译序）

自从 1998 年 *C++ Standard* 定案以后，C++ 链接库便有了大幅扩充。原先为大家所熟知、标准规格定案前蕴酿已久的 STL（Standard Template Library，标准模板程式库），不再被单独对待，而是被纳入整个 C++ 标准链接库（Standard Library）。同时，原有的链接库（如 iostream）也根据泛型技术（generics）在内部上做了很大的修改。可以说，*C++ Standard* 的发布对 C++ 社群带来了翻天覆地的大变动 — 不是来自语言本身，而是来自标准链接库。这个变动，影响 C++ 程序编写风格至巨，C++之父 Bjarne Stroustrup 并因此写了一篇文章：*Learning Standard C++ as a New Language*（载于 C/C++ User's Journal, 1999/05）。

我个人于 1998 年开始潜心研究泛型技术和 STL，本书英文版《***The C++ Standard Library***》甫一出版便成为我学习 C++ 标准链接库的最重要案头工具之一。小有心得之后，我写过数篇相关技术文章，从来离不开本书的影响和帮助。我曾经把 STL（代表泛型技术目前最被广泛运用的一个成熟产品，也是 C++ 标准链接库的绝大成份）的学习比喻为三个阶段（或层次）：

- 第一境界：熟用 STL
- 第二境界：了解泛型技术的内涵与 STL 的学理乃至实作
- 第三境界：扩充 STL

不论哪一个阶段，你都能够从本书获得不同程度的帮助。

第一阶段（对最大多数程序员有立竿见影之效），我们需要一本全面而详尽的教本，附带多量而设计良好的范例，带领我们认识十数个 STL 容器（containers）、数十个 STL 算法（algorithms）、许许多多的迭代器（iteartors）、配接器（adapters）、仿函数（functors）…的各种特性和用途。这些为数繁多的组件必须经过良好的编排组织和索引，才能成就一本效果良好、富教育性又可供长久查阅的案头工具书。

在这一阶段里，本书表现极为优异。书中运用许多图表，对所有 STL 组件的成员做了极其详尽的整理。更值得称道的是书中交叉参考（cross reference）做得非常好，在许多关键的点告诉读者当下可参见哪一章哪一节哪一页，对于阅读和学习带来很大的帮助（本中文版以页页对译方式保留了所有交叉参考和索引）。

第二阶段（从 STL 的运用晋升至泛型技术的学习），我们需要一些关键的 STL 源码（或伪码，pseudo code），帮助我们理解关键的数据结构、关键的编程技术。认识这些关键源码（或伪码）同时也有助提升第一阶段的运用深度（学会使用一样东西，却不知道它的道理，不高明- ）。[1]

本书很多的方都提供了 C++ 标准链接库的关键源码。不全面，但很关键。

第三阶段（成为一位泛型技术专家；打造自己的 STL 兼容组件），我们需要深入了解 STL 的设计理念和组织架构，并深入（且全面的）了解 STL 实作手法[3]。是的，不入虎穴，不能得虎子；彻底了解 STL 如何被打造出来之后，你才能写出和 STL 水乳交融、完美整合的自定组件（user-defined components）。

本书对第三阶段的学习也有相当帮助。虽然没能提供全面的 STL 源码并分析其技术（那需要另外 800 页-），却提供了为数不少的订制型组件实作范例：p191, p213 提供了一个执行期指定排序准则并运用不同排序准则的实例，p219 提供一个自定容器（虽然只是个简单的包覆类别），p222 提供一个「*reference* 语意」示范作法，p285 提供一个针对迭代器而设计的泛型算法，p288 提供一个用于关系型容器的订制型 *inserter*，p294 有一个自定的排序准则，p441 有一个自定的（安全的）stack，p450 有一个自定的（安全的）queue，p504 有一个自定的 traits class for string，p614 有一个自定的 stream 操控器，p663 有一个自定的 stream 缓冲区，p735 有一个自定的内存配置器（allocator）。

[1] 乍见之下令人错愕的一句话。看电视需要先了解电视的原理吗？呵呵，话讲白就没意思了。这句话当然是对技术人员说的。

[2] 这方面我推荐你看《*Generic Programming and the STL - Using and Extending the C++ Standard Template Library*》, by Matthew H. Austern, Addison Wesley 1998。详见稍后说明。中译本《泛型程序设计与 STL》,侯捷/黄俊尧合译, 碁峰, 2001。

[3] 这方面我推荐你看《STL 源码剖析, *The Annotated STL Sources*》by 侯捷, 碁峰, 2002。详见稍后说明。

除了众所瞩目的 STL，本书也涵盖一般不被归类为 STL 的 String 链接库，以及一般不被视为关键的 IOStream 和 Locale 链接库[4]。三部分互有关连，以 IOStream 为主干。在 GUI（图形使用接口）和 application framework（应用程序框架）当道的今天，IOStream 提供的输出输入可能对大部份人失去了价值，但如果你希望开拓 OO 技术视野，IOStream 是一颗沉睡的珠宝。

泛型技术不仅在 C++ 被发扬光大，在 Java 上也有发展[5]，在 C# 上亦被众人期待。从目前的势头看，泛型技术（Generics）或许是面向对象（Object Oriented）技术以来程序编写方面的又一个巨大冲击。新一代 C++ 标准链接库 将[6]采用更多更复杂更具威力的泛型技术，提供给 C++ 社群更多更好更具复用价值的组件。

不论你要不要、想不想、有没有兴趣在你的程序编写过程中直接用上泛型技术，至少，在 C++ 程序编写过程中你已经不可或缺于泛型技术带来的成熟产品：C++ 标准链接库。只要你具备 C++ 语言基础，本书便可以带领你漂亮的运用 C++ 标准链接库，漂亮的提升你的编程效率和程序质量。

面对陌生，程序员最大的障碍在于心中的怯弱。To be or not to be, that is the question! 不要像哈姆雷特一样犹豫不决。面对光明的技术，必须果敢。

关于术语的处理，本书大致原则如下：

1. STL各种数据结构名称皆不译，例如 array, vector, list, deque, hast table, map, set, stack, queue, tree…。虽然其中某些已有约定俗成的中文术语，但另一些没有既标准又被普遍运用的中文名称，强译之读者瞠目以对，部分译部分不译则阅读时词性平衡感不佳（例如「面对向量和 **deque** 两种容器…」就不如「面对 **vector** 和 **deque** 两种容器…」读起来顺畅）。因此，数据结构名称全部不译。直接呈现这些简短的英文术语，可能营造更突出的视觉效果，反而有利

[4] 这方面我见过的唯一专著是《*Standard C++ IOStreams and Locales - Advanced Programmer's and Reference*》, by Angelika Langer and Klaus Kreft, Addison Wesley 2000。

[5] (1) *GJ : A Generic Java*, by Philip Wadler, Dr. Dobb's Journal February 2000.
(2) *JSR- 000014 : Adding Generics to the Java Programming Language*, http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html

[6] 请参考 http://www.boost.org/，这个链接库据称将成为下一代 C++ 标准。

阅读。技术书籍的翻译不是为了建立全中文化阅读环境，我们的读者水平也不可能受制于这些英文单字。

2. STL 六大组件的英文名称原打算全部保留，但由于处处出现，对版面的中英文比例形成视觉威胁，因此全部采用以下译名：**container** 容器，**algorithm** 算法，**iterator** 迭代器，**adapter** 配接器，**functor** 仿函数，**allocator** 配置器。[7]
3. 任何一个被保留的英文关键术语，其第一次（或前数次）出现时尽可能带上中文名称。同样的，任何关键的中文术语，我也会时而让它中英并陈。

关于编排，本书原则如下：

1. 全书按英文版页次编排，并因而得以保留原书索引。索引词条皆不译。
2. 中文版采用之程序代码字体（Courier New 8.5）比文本字体（细明体 9.5）小，英文版之程序代码字体却比其文本字体大，且行距宽。因此中文版遇有大篇幅程序行表，为保持和英文版页次相应，便会出现较多留白。根据我个人对书籍的经验，去除这些留白的最后结果亦不能为全书节省五页十页；填满每一处空白却丧失许多立即可享的好处，智者不取-。

一旦你从本书获得了对 C++ 标准链接库运用层面的全盘掌握与实践经验之后，可能希望对 STL 原理乃至实作技术做更深的研究，或甚至对泛型编程（Generic Programming）产生无比狂热。在众多相关书籍之中，下面是我认为非常值得继续进修的㆕本书：

1. 《*Generic Programming and the STL - Using and Extending the C++ Standard Template Library*》, by Matthew H. Austern, Addison Wesley 1998。本书第一篇（前五章）谈论 STL 的设计哲学、链接库背后的严密架构和严谨定义。其中对于 STL 之异于一般链接库，有许多重要立论。其余部分（第二篇、第三篇）是 STL 的完整规格（分别从 concepts 的角度和 components 的角度来阐述），并附范例程序。

[7] 原书大部份时候使用 **function object** 函数对象一词，为求精简及突出，中文版全部改用其另一个名称 **functor** 仿函数（见第 8 章译注）。

2. 《STL 源码剖析， *The Annotated STL Sources*》by 侯捷, 碁峰， 2002。本书剖析 STL 实作技法，详实揭示并批注 STL 六大组件的底层实作，并以公认最严谨的 SGI（Silicon Graphics Inc.）STL 版本为剖析对象。附许多精彩分析图，对于高度精巧的内存配置策略、各种数据结构、各种算法、乃至极为「不可思议」的配接器（adapter）实作手法，都有深入的剖析。
3. 《*Effective STL*》， by Scott Meyers, Addison Wesley 2001。本书定位为 STL 的深层运用。在深层运用的过程中，你会遇到一些难解的问题和效率的考虑，你需要知道什么该做、什么该避免。本书提供 50 个专家条款。请注意，深层运用和效率调校，可能需要读者先对底部机制有相当程度的了解。
4. 《*Modren C++ Design*》by Andrei Alexandrescu, Addison Wesley 2001。将泛型技术发挥到淋漓尽致、令人目瞪口呆的一本书籍。企图将泛型技术和设计样式（design patterns）结合在一起。领先时代开创先河的一本书。

本书由我和孟岩先生共同完成。孟岩在大陆技术论坛以 C++/OO/Generics 驰名，见解深隽文笔不凡。我很高兴和他共同完成这部作品。所谓合译，我们两人对全书都有完整的参与（而非你一半我一半的对拆法），最终由我定稿。本书同时发行繁体版和简体版，基于两岸计算器术语的岐异性，简体版由孟岩负责必要转换。

侯捷 2002/05/23 于新竹

http://www.jjhou.com（繁体网站）
http://jjhou.csdn.net（简体网站）
jjhou@jjhou.com（个人电子邮箱）

f 巨细靡遗 井然有序（侯捷译序）

孟岩译序

IT 技术书籍市场，历来是春秋战国。一般来说，同一个技术领域里总会有那么数本、十数本、甚至数十本定位相似的书籍相互激烈竞争。其中会有一些大师之作脱颖而出，面南背北，黄袍加身。通常还会有后来者不断挑战，企图以独到特色赢得自己的一片天的。比如说在算法与数据结构领域，D. E. Knuth 的那套《*The Art of Computer Programming*》一至三卷，当然是日出东方惟我独尊。但是他老人家的学生 Robert Sedgewick 凭着一套更贴近实用的《*Algorithms in C*》系列，也打出自己一片天下，成为很多推荐列表上的首选。就 C++ 运用经验类书籍来说，Scott Meyers 的《*Effective C++*》称王称霸已经多年，不过其好友 Herb Sutter 也能用一本《*Exceptional C++*》获得几乎并驾齐驱的地位。嗨，这不是很正常的事吗？技术类书籍毕竟不是诗词歌赋。苏轼一首「明月几时有，把酒问青天」，可以达到「咏中秋者，自东坡西江月后，余词尽废」的程度，但怎么可能想象一本技术著作达到「我欲乘风归去，又恐琼楼御宇，高处不胜寒」的境界！谁能够写出一本技术书，让同一领域后来者望而却步干脆死心，那才是大大的奇迹！

然而，您手上这本《*The C++ Standard Library*》，作为 C++ 标准链接库教学和参考类书籍的定音之作，已经将这个奇迹维持了三年之久。按照 IT 出版界时钟，三年的时间几乎就是半个世纪，足以锤炼又一传世经典！

1998 年 *C++ Standard* 通过之后，整个 C++ 社群面临的最紧迫任务，就是学习和理解这份标准给我们带来的新观念和新技术。而其中对于 C++ 标准链接库的学习需求，最为迫切。C++ 第二号人物 Andrew Koenig 曾经就 C++ 的特点指出：『语言设计就是链接库设计，链接库设计就是语言设计』[1]。*C++ Standard* 对链接库所做的巨大扩充和明确规范，实际上即相当于对 C++ 语言的能力做了全面提升与扩展，意味着你可以站在无数超一流专家的肩上，于轻敲键盘间，将最出色的思想、设计与技术纳入囊中，让经过千锤百炼的精美代码成为自己软件大厦的坚实基础。

[1] "Language design is library design, library design is language design"，参见 Andrew Koenig, Barbara Moo 合着《Ruminations on C++》第 25, 26 章。

可以说，对于大多数程序员来说，标准 C++ 较诸「ARM 时代」之最大进步，不是语言本身，而恰恰是标准链接库。因此，我们可以想象当时人们对 C++ 标准程式库教学类书籍的企盼，是何等热切！

一方面是已经标准化了的成熟技术，另一方面是万众期待的眼神，我们完全有理由认为，历史上理应爆发一场鱼龙混杂的图书大战。它的典型过程应该是这样：先是一批快刀手以迅雷不及掩耳盗铃之势 推出[2]一堆敛财废纸，然后在漫长的唾骂与期待中，大师之作渐渐脱颖而出。大浪淘沙，最后产生数本经典被人传颂。后虽偶有新作面世，波光点点已是波澜不兴。

然而，这一幕终究没有在「C++ 标准链接库教学与参考书籍」领域内出现。时至今日，中外技术书籍市场上这一领域内的书籍为数寥寥，与堆积如山的 C++ 语言教学类书籍形成鲜明对比。究其原因，大概有二，一是这个领域里的东西毕竟份量太重，快刀手虽然善斩乱麻，对于 C++ 标准链接库这样严整而精致的目标，一时也难以下手。更重要的原因则恐怕是 1999 年 8 月《*The C++ Standard Library*》问世，直如横刀立马，震慑天下。自推出之日起至今，本书在所有关于 C++ 标准程式库的评论与推荐列表上，始终高居榜首，在 Amazon 的销量排行榜上名列所有 C++ 相关书籍之最前列。作者仅凭一书而为天下知，成为号召力可与 Stan Lippman, Hurb Sutter 等「经典」C++ 作家比肩的人物。此书之后，虽然仍有不少著作，或深入探讨标准链接库的某些组件，或极力扩展标准库倡导的思想与技术，但是与《*The C++ Standard Library*》持同一路线的书籍，再没有出现过。所谓泰山北斗已现，后来者已然无心恋战。

于是有了这样的评论：『如果你只需要一本讲述 C++ 标准链接库和 STL 的书籍，我推荐 Nicolai Josuttis 的《*The C++ Standard Library*》。它是你能得到的唯一一本
全面讲述 C++ 标准链接库的书，也是你能想象的最好的一本书。』这种奇异情形在当今技术书坛，虽然不是绝无仅有，也是极为罕见。

究竟这本书好到什么程度，可以获得这么高的评价？

我正是带着这分疑问，接受侯捷先生的邀请，着手翻译这本经典之作。随着翻译过程的推进，我也逐渐解开了心中的疑惑。在我看来，这本书的特点有[四]：内容详实，组织严密，态度诚恳，深入浅出。

首先，作为一本链接库参考手册，内容详实全面是一项基本要求。但是，本书在

[2] 此处非笔误，而是大陆流行的一句"新俚语"，意思十分明显，就是"迅雷不及掩耳"地"掩耳盗铃"。

这方面所达到的高度可以说树立了一个典范。本书作者一开始就提出一个极高的目标，要帮助读者解决「使用 C++ 标准链接库过程中所遇到的所有问题」。众所周知，C++ 标准链接库是大家伙，每一部分又有很精深的思想和技术，既不能有所遗漏，又不能漫无边际地深入下去，何取何舍，何去何从，难度之大可想而知！作者在大局上涵盖了 C++ 标准链接库的全部内容，在此基础上又对所有组件都进行细致的、立体式的讲解。所谓立体式讲解，就是对于一个具体组件，作者首先从概念上讲解其道理，然后通过漂亮的范例说明其用法，申明其要点，最后再以图表或详解方式给出参考描述。有如钱塘江潮，层层迭迭，反反复复，不厌其烦。读完此书，我想您会和我一样感受冲击，并且完全认可作者付出的巨大心血。

C++ 标准链接库本身就是一个巨大的有机整体，加上这本书的立体讲解方式，前后组织和对应的工作如果不做好，很容易会使整部书显得散乱。令人钦佩的是，这本书在组织方面极其严密，几无漏洞。相关内容的照应、交叉索引、前后对应，无一不处理得妥善曼妙。整体上看，整本书就像一张大网，各部分内容之间组织严谨，契合密切，却又头绪清晰，脉络分明，着实难能可贵。我在阅读和翻译过程中，常常诧异于其内容组织的精密程度，简直像德国精密机械一样分毫不差 — 后来才想到，本书作者 Nicolai Josuttis 就是德国人，精密是德意志民族的性格烙印，名不虚传！

说起德意志民族，他们的另一个典型性格就是诚实坦率。这一点在这本书同样有精彩的展现。身为 C++ 标准链接库次委员会成员，作者对于 C++ 标准链接库的理解至深，不但清楚知道其优点何在，更对其缺陷、不足、不完备和不一致的地方了如指掌。可贵的是，在这些地方，作者全不避讳，开诚布公，直言不讳，事实是什么样就是什么样，绝不文过饰非，绝不含混过关。作为读者，我们不仅得以学到好东西，而且学到如何绕开陷阱和障碍。一个最典型的例子就是对于 valarray 的介绍，作者先是清清楚楚地告诉读者，由于负责该组件设计的人中途退场，这个组件没有经过细致的设计，最好不要使用。然后作者一如既往，详细介绍 valarray 的使用，完全没有因为前面的话而稍微有所懈怠。并且在必要的地方将 valarray 的设计缺陷原原本本地指出来，让读者口服心服。读到这些地方，将心比心，我不禁感叹作者的坦诚与无私，专精与严谨。

本书最具特色之处，就是其内容选取上独具匠心，可谓深入浅出。本书的目的除了作为手册使用，更是一本供学习者阅读学习的 "tutorial"（自学教本）。也就是说，除了当手册查阅，你也可以捧着它一篇一篇地阅读学习，获得系统化的坚实知识。一本书兼作 "tutorial" 和 "reference"，就好像一本字典兼作「作文指南」，没有极高的组织能力和精当的内容选择，简直难以想象会搞成什么样子。了不起的是本书不仅做到了，而且让你感觉，学习时它是一本最好的 "tutorial"，查阅时

它是一本最好的 "reference"，我要说，这是个奇迹！单从学习角度来说，本书极为实用，通过大量鲜明的例子和切中要害的讲解让你迅速入门，而且绝不仅仅浅尝辄止，而是不失时机地深入进去，把组件的实作技术和扩展方法都展现给读者。单以 STL 而论，我经常以侯捷先生提出的「STL 学习三境界」来描述一本书的定位层次，这本书就像一座金字塔，扎根于实用，尖锋直达「明理」和「扩展」层次。从中你可以学到「*reference* 语意」的 STL 容器、smart pointer（灵巧指标）的数种实现、扩充的组合型仿函数（composing function object）、STL 和 IOStream 的扩展方法、订制型的配置器（allocator）设计思路等等高级技术，也能学到大量的实践经验，比如 vector 的使用技巧，STL 容器的选择，`basic_string<>` 作为容器的注意事项等等。可以这么说，这本书足以将你从入门带到高手层次，可谓深入浅出，精彩至极！

我很高兴自己第一次进行技术书籍翻译，就能够碰到这样一本好书，这里要深深感谢侯捷先生给我一辈子都庆幸的机会。翻译过程出乎意料地艰辛，前后持续将近 10 个月。我逐字逐句地阅读原文，消化理解，译成自以为合适的中文，然后交给侯先生。侯先生接手后再逐字逐句地阅读原文，对照我的粗糙译文，进行修订和润色，反复品味形成最终译稿。作为译者，侯先生和我所追求的是，原书技术的忠实呈现加上完全中文化、中国式的表达。我们为此花费了巨大的心力，对我来说，付出的心血远远超过一般翻译一本书的范畴。虽然最终结果需要广大读者评论，但今天面对这厚厚的书稿，我问心无愧地享受这份满足感。我最大的希望是，每一位读者在学习和查阅这本中文版的时候，完全忘掉译者曾经的存在，感觉不到语言的隔阂，自由地获取知识和技术。对于一个初涉技术翻译的人来说，这个目标未免太贪心了，可是这始终会是我心里的愿望。一个译者应该是为了被忽略而努力的。

最后，感谢侯先生一直以来对我的欣赏和帮助，感谢您给我的机会，我十分荣幸！感谢华中科技大学出版社的周筠老师，您始终友好地关注着我，鼓励着我。感谢 CSDN 的蒋涛先生，您的热情鼓励始终是我的动力。感谢我的父母，弟弟，你们是我最爱的人，是我最坚强的支柱！感谢曾经帮助过我，曾经关心过我的每一个人，无论你现在怎样，我为曾经拥有过的，仍然拥有着的每一片快乐和成果，衷心地感谢你！

祝各位读书快乐！

孟岩 2002 年 5 月于北京

目录

前言

Preface

一开始，我只不过想写一本篇幅不大的有关于 C++ 标准链接库的德文书（也就 400 多页吧）。萌生这个想法是在 1993 年。而在 1999 年的今天，您看到了这个想法的成果：一本英文书，厚达 800 多页，其中包含大量的文字描述、图片和范例。我的目标是，详尽讲解 C++ 标准链接库，使你的所有（或几乎所有）编程问题都能够在你遇到之前就先给你解答。然而，请注意，这不是一种完整描述 C++ 标准链接库的所有面向的书籍，我透过「在 C++ 中利用标准链接库进行学习和程序编写」的形式，表现出最重要的主题。

每一个主题都是以一般性概念为基础而开展，然后导入日常程序编写工作所必须了解的具体细节。为了帮助你理解这些概念和细节，书中提供详尽的范例程序。

这就是我的前言 — 言简意赅！撰写此书的过程中，我得到了很多乐趣，希望你阅读本书时，能够像我一样快乐。请享用！

致谢

Acknowledgments

这本书表达的观点、概念、解决方案和范例，来源十分广泛。从这个意义上讲，封面只列我一个人的名字，未免不公平。所以我愿在此向过去数年来帮助和支持我的人和公司，表示诚挚的谢意。

我第一个要感谢的是 Dietmar Kühl。Dietmar 是一位 C++专家，尤其精通 I/O streams（资料串流）和国际化（他曾经仅仅为了好玩而写了一个 I/O stream library）。他不仅将本书的大部分从德文译为英文，还亲自动笔，发挥专长，为本书撰写了数节内容。除此之外，过去的数年里，他向我提供了很多宝贵的回馈意见。

其次，我要感谢所有检阅者和那些向我表达过意见的人。他们的努力使本书的品质获得巨大提升。由于名单太长，以下如有任何疏漏，还请见谅。英文版的检阅者包括 Chuck Allison, Greg Comeau, James A. Crotinger, Gabriel Dos Reis, Alan Ezust, Nathan Myers, Werner Mossner, Todd Veldhuizen, Chichiang Wan, Judy Ward, Thomas Wike-hult。德文版的检阅者包括 Ralf Boecker, Dirk Herrmann, Dietmar Kühl, Edda Lörke, Herbert Scheubner, Dominik Strasser, Martin Weitzel。其他投入者包括 Matt Austern, Valentin Bonnard, Greg Colvin, Beman Dawes, Bill Gibbons, Lois Goldthwaite, Andrew Koenig, Steve Rumsby, Bjarne Stroustrup, 和 David Vandevoorde。

我要特别感谢 Dave Abrahams, Janet Cocker, Catherine Ohala 和 Maureen Willard，他们对全书进行了非常细致的检阅和编辑。他们的回馈意见让本书的质量获得了难以置信的提升。

我也要特别感谢我的「活字典」Herb Sutter，他是著名的 "Guru of the Week" 的创始人，这是一个常态性的 C++ 难题讲解专栏，播出于 comp.std.c++.moderated。

我还要感谢一些公司和个人，他们的帮助使我有机会在各个不同的平台上，使用各种不同的编译程序来测试自己的范例程序。非常感谢来自 EDG 的 Steve Adamczyk，Mike Anderson 和 John Spicer，他们的编译程序真是太棒了。在 C++ 标准化过程和

本书写作过程中，提供了巨大的支持。感谢 P. J. Plauger 和 Dinkumware, Ltd，他们很早以来就持续进行与 C++ 标准规格兼容的 C++ 标准链接库实作工作。感谢 Andreas Hommel 和 Metrowerks，他们完成了深具价值的 CodeWarrior 程序开发环境。感谢 GNU 和 egcs 编译程序的所有开发者。感谢 Microsoft，他们完成了深具
价值的 Visual C++。感谢 Siemens Nixdorf Informations Systems AG 的 Roland
Hartinger，他提供了一份他们的 C++ 编译程序测试版本。感谢 Topjects GmbH，为了他那一份深具价值的 ObjectSpace library 实作品。

感谢 Addison Wesley Longman 公司里头与我共同工作过的每一个人。包括 Janet Cocker, Mike Hendrickson, Debbie Lafferty, Marina Lang, Chanda Leary, Catherine
Ohala, Marty Rabinowitz, Susanne Spitzer, 和 Maureen Willard 等等。这项工作真是
太有趣了。

此外，我还要感谢 BREDEX GmbH 的人们，感谢 C++ 社群中的所有人，特别是那些参与标准化过程的人，感谢他们的支持和耐心（有时候我问的问题确实挺傻）。

最后，也是最重要的，我要将我的感谢（附上一个亲吻）送给我的家人：Ulli, Lucas, Anica, 和 Frederic。为了这本书，我很长时间没有好好陪他们了。

但愿各位能从这本书获得乐趣，另外，请保持宽厚。
（译注：上句原文为 Have fun and be human!。请抱歉我对其精确意思毫无把握）

关于本书

1.1 缘起

C++ 问世后不久，就成为面向对象程序设计领域的实质标准（de facto standard），因此正式标准化的呼声也就浮上了台面。一旦我们有了一个可以依循的标准规格，我们才可能写出跨越 PC 乃至大型主机各种不同平台的程序。此外，如果能够建立起一个标准链接库，程序员便得以运用可移植的通用组件（general components）和更高层次的抽象性，而不必从头创造世界。

C++ 标准化过程始于 1989 年，由国际性的 ANSI/ISO 委员会负责。标准化工作以 Bjarne Stroustrup 的两本书 *The C++ Programming Language* 和 *The Annotated C++ Reference Manual* 为根基。这份标准规格于 1997 年通过后，又在数个国家进行了一些正式程序，最后于 1998 年成为国际性的 ISO/ANSI 标准。标准化过程本身包含一个任务：建立 C++ 标准链接库。作为核心语言的拓展，标准链接库提供了一些通用组件。藉由大量运用 C++ 新的抽象能力和泛型（generic types）特性，标准链接库提供了一系列共同的类别和接口。程序员藉此获得了更高层次的抽象能力。这个标准链接库提供了以下组件：

- String 型别
- 各种数据结构（例如 dynamic array、linked lists、binary trees）
- 各种算法（例如各种排序算法）
- 数值类别（numeric classes）
- 输入/输出（I/O）类别
- 国际化支持（internationalization support）类别

所有这些组件的接口都十分简单。这些组件在很多程序中都很重要。如今的资料处理工作，通常就是意味着输入、计算、处理和输出大量数据（通常是字符串）。

这个标准链接库的用法并非不言自明。想要从其强大能力中受惠，你需要一本好书；不能够仅仅列出每一个类别和其函数了事，而是必须详细解释各组件的概念和重要细节。本书正是以此为目标。本书首先从概念上介绍标准链接库及其所有

组件，然后描述实际编程（programming）所需了解的细节。为了展示组件的确切用法，书中还包括了大量实例。因此，这本书不论对初学者或是编程老手，都是极为详尽的 C++ 标准链接库文件。以本书所提供的数据来武装自己的头脑，你就能从 C++ 标准链接库中获得最大利益。

注意，我不担保本书所有内容都容易理解。标准链接库非常灵活，但这种非同寻常的灵活性是有代价的。标准链接库中有一些陷阱和缺陷，你必须小心应对。碰到它们时我会为你指出，并提出一些建议，帮助你回避问题。

1.2 阅读前的必要基础

要想读懂本书的大部分内容，你需要先了解 C++。本书讲述 C++标准组件，而不是语言本身。你应该熟悉类别（classes）、继承（inheritance）、模板（templates）和异常处理（exception handling）的概念，但不必熟知语言的每一个细节。某些重要的细节本书也会讲解（至于次要细节对链接库实作者可能很重要，对链接库使用者就不那么重要了）。注意，在标准化过程中，C++ 语言发生了很大的变化，也许你的知识已经过时了。2.2 节简单介绍了一些「使用标准链接库时，你需要了解的最新语言特性」。如果你不确定自己是否了解 C++ 的新特性（例如关键词 `typename` 以及 namespace 概念），请先阅读该节。

1.3 本书风格与结构

标准链接库内的组件有相当程度的独立性，但彼此又存在关联，所以很难在描述某一部分时全然不提其他部分。我为这本书考虑了几种不同的组织方式。一是按照 *C++ standard* 的次序，但这并非完整介绍 C++ 标准链接库的最佳选择。另一种方式是，首先纵览所有组件，再逐章详细介绍。第三个方式则是，由我依照「组件之交叉参考」程度高低，从最低者开始介绍，逐次介绍最复杂的组件。最终，我综合了三种方式：首先简短介绍标准链接库所涉及的总体概念和工具，然后分章详述各个组件，每个组件一章或数章。首当其冲的便是 STL（Standard Template Library，标准模板库）。STL 无疑是标准链接库中最强大、最复杂、最激动人心的部分，其设计深刻影响了其他组件。接下来我再讲解较易理解的组件，例如特殊容器、strings 和数值类别。再来是你或许已经使用多时的老朋友：IOStream 程式库。最后是国际化议题的讨论，这部分对于 IOStream 链接库有些影响。

讲述每个组件时，我首先给出该组件的目的、设计和范例，然后藉由各种使用方法和注意事项的描述，讲解组件的细节。最后是个参考章节，你可以在其中找到组件类别和其函数的确切标记型式（exact signature）。

以下是本书内容。最前面的四 章总体介绍了本书及 C++ 标准链接库：

z 第 1 章：关于本书
这一章（也就是你此刻正阅读的）介绍本书的主题和内容。

z 第 2 章： C++ 和其标准链接库简介
这一章对于 C++ 标准链接库的历史和背景进行简短综览。此章也包括了本书及标准链接库的技术背景的大致介绍，例如新的语言特性和所谓复杂度概念。

z 第 3 章 一般性概念（General Concepts）
本章描述标准函数库的基本概念，这些概念对于你理解和使用函数库中的所有组件都是必须的。明确地说，本章介绍了 namaspace std、头文件（headers）格式、错误和异常处理（exception handling）的一般性支持。

z 第 4 章 通用工具（Utilities）
本章描述数种提供给链接库用户和链接库本身运用的小工具，更明确地说是 max(),min(),swap()等辅助函数，和 pair,auto_ptr,numeric_limits等型
别。上述最后一个型别提供了与实作品相依的值类型相关信息。

第 5章至第 9 章分别从各个面向描述 STL：

z 第 5 章：Standard Template Library（STL，标准模板库）
STL 提供了用于处理数据的容器和算法。本章详细介绍 STL 的概念，并逐步解释其中的概念、问题、特殊编程技术，以及它们所扮演的角色。

z 第 6 章：STL 容器（Containers）
本章解释 STL 容器类别的概念和能力。首先透过详尽的例子，分别讲解 vectors、deques、lists、sets 和 maps，然后介绍它们的共通能力。最后以简明的形式列出并描述所有容器所提供的函数，做为一份上手的参考数据。

z 第 7 章：STL 迭代器（Iterators）
本章具体介绍了 STL 迭代器。解释迭代器的分类和辅助函数，以及相应的配接器（iterator adapter）如 stream iterators、reverse iterators、insert iterators。

z 第 8 章：STL 仿函数（Functors，又名 Function Objects）
本章详细讲解 STL 仿函数。

z 第 9 章：STL 算法（Algorithms）
本章罗列并描述 STL 算法。在简单介绍和比较这些算法后，藉由一个（或多个）范例，对每个算法进行详细描述。

第 10 章至 12 章描述了一些「简单的」标准类别：

- 第 10 章：特殊容器（Special Containers）
  本章描述 C++ 标准链接库的各种特殊容器，并涵盖容器配接器（container adapters）queues, stacks，以及 class `bitset`，后者可管理任意数量的 bits 或 flags。
- 第 11 章：Strings（字符串）
  本章描述 C++ 标准链接库的 string 型别（不止一种哦）。*C++ standard* 将 strings 设计为一种「能够处理不同字符类型」的基本数据型别，而且简明易用。
- 第 12 章：数值（Numerics）
  本章描述 C++ 标准链接库中的数值组件，包括复数（complex），以及一些用来处理数值数组的类别（可用于矩阵 matrices、向量 vectors 和方程式 equations）。

第 13 章和第 14 章的主题是 I/O 和国际化（两者紧密相关）：

- 第 13 章：以 Stream Classes 完成输入和输出
  本章涵盖 C++ 的 I/O 组件。该组件是众所周知的 IOStream 链接库的标准化形式。本章也讲述了一些对程序员而言可能很重要、但又鲜为人知的细节。例如如何定义及整合特殊 I/O 通道，这是一个在实务过程中经常被搞错的题目。
- 第 14 章：国际化（Internationalization, i18n）
  本章涵盖「将程序国际化」的概念和类别，包括对不同字符集（character sets）的处理，以及如何使用不同格式的浮点数和日期。

剩余部分包括：

- 第 15 章：配置器（Allocators）
  本章描述 C++ 标准链接库中内存模型（memory model）的概念。
- 附录，包括：
  — 网际网上的资源（Internet Resources）
  — 参考书目（Bibliography）
  — 索引（Index）

1.4 如何阅读本书

本书既是介绍性的使用指南，又是 C++ 标准链接库的结构化参考手册。C++ 标准链接库的各个组件在相当程度上是彼此独立的，所以读完第 2 章至第 4 章后，你可以按任意顺序阅读其他各章节。不过切记，第 5 章至第 9 章讲述的是同一组东西。要理解 STL，应该从介绍性的第 5 章开始。

如果你是一位想总体认识 C++ 标准链接库概念和其各个面向的程序员，可以简略浏览这本书，并略过其参考章节（译注：就是完整条列各个接口的那些小节）。当你需要运用某个 C++ 标准链接库组件时，最好的办法就是透过索引（index），找出相关资料。我已尽力把索引做得详尽，希望能够节省你的搜寻时间。

以我的经验来看，学习新东西的最佳方式就是阅读范例。因此，你会发现本书通篇有大量的范例，可能是几行代码，也可能是完整程序。如果是后者，批注第一行列有其文件名。你可以在我的网站 `http://www.josuttis.com/libbook/` 找到这些档案。

1.5 目前发展情势

C++ standard 在我撰写本书期间完成。请记住，有些编译程序可能还无法与之兼容。这一点很可能在不久的将来得到很大的改善。但是现在，你可能会发现，本书所谈的东西并非一定能够在你的系统上有相同的表现，或许得稍做修改才能在你的环境中正常运作。我所使用的 EGCS 编译程序 2.8 版，及其更高版本，能够编译书中所有范例程序。该编译程序几乎适用于所有计算器平台，你可以从因特网（`http://egcs.cygnus.com/`）和某些软件光盘中获得。

1.6 范例程序代码及额外信息

在我的网站 `http://www.josuttis.com/libbook/` 上，你可以获得所有范例程式码、本书及 C++ 标准链接库的其他相关信息。你也可以在因特网上找到许多其他相关信息，详见 p743。

1.7 回应

欢迎你对本书的任何响应（不论好的或坏的）。我已尽力而为，但我毕竟是凡人，而且总有结稿的时候，所以难免有一些错误或前后不一的地方，需要进一步改善。你的响应将使我的新版本有机会进步。与我联系的最佳方式是透过电子邮件：

```
libbook@josuttis.com。
```

也可以透过电话、传真、或「蜗牛般的」信件与我联系：

```
Nicolai M. Josuttis
Berggarten 9
D??38108 Braunschweig
Germany
Phone: +49 5309 5747
Fax: +49 5309 5774
```

非常感谢。

C++ 及其标准链接库简介

2.1 沿革

纜□ 的标准化过程始于 ㄖ__年，于 ㄖ__年底完成。不过，由于某些原因，最终的标准规格迟至 ㄖ恃 年 _月才公布，整个努力的成果就是国际标准化组织（鍖_，_____潮__□_条湩_____潮□_____ __滴_____考手册。这份标准被命名为___渥淜_____潮_____栝潬潧___□_潧_____涌□_涌畎__□纜□ ，文件编号 _____鰿ㄴ帆㈦ ㄖ恃，由 鍖_的各国成员机构发布。例如美国的秀__机构 [1]。

标准规格的建立，是 纜□ 的一个重要里程碑，它准确定义了 _□_的内容和行为，简化了 纜□ 的教学、使用、以及在不同平台之间的移植工作，给予用户选择不同纜□ 实作品（编译程序）的自由。它的稳定性和可移植性，对于链接库、工具库的供应者和实现者都是一桩佳音。因此，这份标准规格帮助 纜□ 应用程序开发人员更快更好地创建应用程序，减轻维护上的精力。

标准链接库是 _□_标准规格的一部分，提供一系列核心组件，用以支持 鎯_、字串（獚__涌_）、容器（数据结构）、算法（排序、搜索、合并等等）、数值计算、国别（例如不同的字符集_□_慲惯瑙爟獩瑮）等主题。

这个标准化过程竟花费了近 年的时间，未免让人奇怪。如果你了解其中一些细节，更会奇怪为什么这么久之后此一标准仍然未臻完美。十年其实不够！尽管从标准化的历史和内容来看，确实完成了许多东西，也形成了可在实务中应用的成果，但是距离完美尚远（毕竟世间无完美之物）。

†††††††††††††††††††††††††

[1] 本书写作之时，你可以花 ㄖㄈ 美元从 __卉[汶___潮楣__却_滴___□_淜_获得这份 纜准文件标（见□毁_掷糸睫□滴槮淜_。

这份标准并非某个公司花费大把钞票和大量时间后的产物。那些为制定标准而辛勤工作的人们，几乎没有从标准化组织那儿获取任何报酬。对他们而言，如果他们所处的公司对 _□_标准化漠不关心，那么他们就只能以兴趣为全部的动力了。感天谢地，有这么多勇于奉献的人，能够拿出时间和财力参与其中。

这份 纜□ 标准规格并非从零开始，它以 _□_创始人 __憖攘却牯__玲__对于这个语言所作的描述为基础。然而标准链接库并非基于某本书或某一个现成的函数库，而是将各种不同的类别（掰悜獙_）整合而成[2]，因此其结果并非十分地同质同种。你会发现不同组件背后有不同的设计原则，獗物__□污 獳和 协_ 之间的差别就是很好的例子，后者是一个数据结构和算法框架（晲憖敷_牋）。

- z st ring classes 被设计为安全易用的组件，其接口几乎不言自明，并能对许多可能的错误作检验。
- z S 的设计目标，是将不同的算法和数据结构结合在一起，并获取最佳效率，所以 协_ 并不非常便利，也不检验许多可能的逻辑错误。要运用 协_ 强大的框架和优异的效率，你就必须通晓其概念并小心运用。

标准链接库中有一个组件，在标准化之前就已经作为「准」标准而存在，那就是鋤却牴__链接库。这个东西于 ㄖ忉 年开发，ㄖ帢 年重做过一次，部分内容重新设计。由于很多程序员早就已经使用 __却牴憖，所以 __却牴憖链接库的概念没有改变，保持回溯兼容。

总体来说，整份标准规格（语言和链接库）是在来自全球各地数百位人士的大量讨论和影响下诞生的，例如日本人就是国际化（__瑙垒慑楯__汩穑莹__）组件的重要支
持者。当然，我们曾经犯下错误，曾经改弦更张，曾经意见不一。到了 _悚_年，当人们认为这个标准接近完成时，协_ 又被加了进来，此举从根本上改变了整个链接库。然而事情总要有个结束，终于有那么一天，人们决定不再考虑任何重大扩张，无论这个扩张多么有价值。就因为这样，桡__⁴_____没有被纳入标准□_尽管它作为一种常用的数据结构，理应在 协_中享有一席之地。

现有的标准并不是终极产品，每五年会有一个新版本，修正错误和不一致的地方。然而，起码在接下来的数年中，纜__程序员终于有了一个标准，有机会编写功能强大并可移植到各种平台上的 纜__程序了。

†††††††††††††††††††††††††

[2] 你可能会奇怪，为什么标准化过程中不从头设计一个新的链接库。要知道，标准化的主要目的不是为了发明新东西或发展出某些东西，而是为了让既有的东西调和共处。

2.2 新的语言特性

纜□ 语言核心和 纜□ 链接库是同时被标准化的，这么一来，链接库可以从语言的发展中受益，语言的发展也可以从链接库的实作经验中得到启发。事实上，在标准化过程中，链接库经常用到已规划但当时尚未被实现出来的语言特性。

今天的 纜□ 可不是五年前的 _□_了。如果你没有紧跟其发展，可能会对链接库所使用的语言新特性大感惊讶。本节对这些新特性进行简单的概括说明，至于细节，请查阅语言相关书籍。

我撰写本书的时候（ㄦ悚 年），并非所有编译程序都提供所有的语言新特性，这就限制了链接库的使用。我希望（并预期）这种情况能很快获得改善（绝大多数编译器厂商都参与了标准化过程）。实作一份可移植的链接库时，通常都要考虑你的电算环境是否支持你所用到的特性□_通常我们会使用一些测试程序，检查哪些语 言特性获得支 援，然后 根据检验结果设置 前处理器指 令（ 灼_____挽獳潲__牴捻窠歖）。我会在书中以脚注方式指出所有典型而重要的限制。

接下来数个小节描述和 纜__标准链接库有关的几个最重要的语言新特性。

2.2.1 template（模板）

链接库中几乎所有东西都被设计为 瑄__污瑄形式。不支持 瑄__污瑄，就不能使用标准链接库。此外链接库还需要一些新的、特殊的 瑄__污瑄特性，我将在简介之后详细说明。

所谓[4]扬_污瑄_，是针对「一个或多个尚未明确的型别」所撰写的函数或类别。使用 瑄__污瑄时，可以明白地（数__楣楴__）或隐喻地（业_汩捩鎏_）将型别当作自变量来传递。下面是一个典型例子，传回两数之中的较大数：

```
template <class T>
inline const T& max (const T& a, const T& b)
{
    // if a < b then use b else use a
    return a < b ? b : a;
}
```

在这里，第一行将 T 定义为任意数据型别，于函数被呼叫时由呼叫者指定。任何合法的识别符号都可以拿来作为参数名称，但通常以 T 表示，这差不多成了一个「准」标准。这个型别由关键词 class 引导，但型别本身不一定得是 掀悸獚霹任何数据型别只要提供 瑄__污瑄定义式内所用到的操作，都可适用于此[4]扬_污瑄[3]。

†††††††††††††††††††††††††

[3] 这里使用关键词 class，原是为了避免增加新的关键词。然而最终还是不得不引入一个新的关键词 typename。此处亦可使用关键词 typename（详见 灞_）。

遵循同样原则，你可以将□污獳_参数化，并以任意型别做为实元。这一点对容器类别非常有用。你可以实作出「有能力操控任意型别之元素」的容器。纜□ 标准链接库提供了许多_瑄涮污瑄□___悝__爟掀慳 獩_（详见第 _章和第 章）。标准函式库对于⁴扬_污瑄_掀慳獩_的使用，还有其他原因。例如可以以字符型别（舍慲惯瑄_琛__）和字符集（舍慲惯瑄爟獩_）属性，将 string c参数化（详见第 ㄱ 章）。

呥涮污瑄 并非一次编译便生出适合所有型别的代码，而是针对被使用的某个（或某组）型别进行编译。这导致一个重要的问题：实际处理 瑄__污瑄时，面对 瑄__污瑄啓愍__潮，你必须先提供它的某个实作品，然后才能呼叫，如此方可通过编译。所以目前唯一能够让「瑄__污瑄的运用」具有可移植性的方式，就是在头文件中以 楮汩__啓愍__潮实现⁴扬_污瑄_____莹[4]__。

欲实现 纜□ 标准链接库的完整功能，编译程序不仅需要提供一般的 瑄__污瑄支持，还需要很多新的 瑄__污瑄标准特性，以下分别探讨。

Nontype Templates 参数（非型别模板参数）

型别（琛__）可作为 瑄__污瑄参数，非型别（_潮____）也可以作为 瑄__污瑄参数。非型别参数因而可被看作是整个 瑄__污瑄型别的一部分。例如可以把标准类别 bitset<>（本书 □节_⁰_抖会介绍它）的 扩__数量以 瑄__污瑄参数指定之。以下述句定义了两个由 _楮_构成的容器，分别为 ファクト 个 扩__空间和 匠 个 扩__空间：

```
bitset<32> flags32;  // bitset with 32 bits
bitset<50> flags50;  // bitset with 50 bits
```

这些 _楴獩__由于使用不同的⁴扬_污瑄_参数，所以有不同的型别，不能互相赋值（慳獗__）或比较（除非提供了相应的型别转换机制）。

Default Template Parameters（预设模板参数）

呥涮污瑄□污獳敨 可以有预设自变量。例如以下宣告，允许你使用一个或两个 瑄__污瑄自变量来宣告 MyClass 物件[5]：

```
template <class T, class container = vector<T> >
class MyClass;
```

如果只传给它一个自变量，那么预设参数可作为第二自变量使用：

```
MyClass<int> x1; // equivalent to: MyClass<int,vector<int> >
```

[4] 目前 瑄__污瑄 必须定义于头文件中。为了消除这个限制，标准规格导入了一个 瑄__污瑄□_莹潮□润（模板编译模型）和一个关键词 export。可惜据我所知，目前尚无任何编译程序实现出这一特性。

[5] 注意，两个•>" 之间必须有一个空格，如果你没写空格，_>>"会被解读为移位运算子（独楦瑚溃_牡琅_导致语法错误。

注意，瑙__污瑙预设自变量可根据前一个（或前一些）自变量而定义。

关键词 typename

关键词 typename 被用来做为型别之前的标识符号。考虑下面例子：

```
template <class T>
class MyClass
   typename T::SubType * ptr;
   ...
};
```

这里，typename 指出 SubType 是□污獳_T中 定义的一个型别，因此 ptr 是一个指向 T::SubType 型别的指标。如果没有关键词 typename，SubType 会被当成一个獴慑楣成员，于是：

```
T::SubType * ptr
```

会被解释为型别 T 内的数值 SubType 与 ptr 的乘积。

SubType 成为一个型别的条件是，任何一个用来取代 T 的型别，其内都必须提供一个内隐型别（__滨_⁴_蠡）SubType 的定义。例如，将型别 Q 当做 瑙__污瑙参数：

```
MyClass<Q> x;
```

必要条件是型别 Q 有如下的内隐型别定义：

```
class Q
   typedef int SubType;
   ...
};
```

此时，MyClass<Q> 的 ptr 成员应该变成一个指向_int 型别的指标。子型别（猎扮___）也可以成为抽象数据型别（例如□污獳）：

```
class Q
   class SubType;
   ...
};
```

注意，如果要把一个 瑙__污瑙中的某个标识符号指定为一种型别，就算意图显而易见，关键词 typename 也不可或缺，因此 _□_的一般规则是，除了以 typename 修饰之外，瑙__污瑙内的任何标识符号都被视为一个实值（痴___）而非一个型别。

typename 还可在 瑙__污瑙宣告式中用来替换关键词 class：

```
template <typename T> class MyClass;
```

Member Template（成员模板）

掮慛猸浥涅敲_____莹__ ____污瑙，但这样的 浥涅敲⁴_涮____既不能是 可以是个 virtual，也不能有预设参数。例如：

```
calss MyClass
   ...
   template <class T>
   void f(T);
};
```

在这里，MyClass::f 宣告了一个成员函数集，适用任何型别参数。只要某个型别提供有 f()用到的所有操作，它就可以被当做自变量传递进去。这个特性通常用来为 瑁__污瑁□污獳敨中的成员提供自动型别转换。例如以下定义式中，assign()的参数 x，其型别必须和呼叫端所提供的对象的型别完全吻合：

```
template <class T>
class MyClass
   private:
     T value;
   public:
     void assign (const MyClass<T>& x) { // x must have same type as *this
        value = x.value;
     }
   ...
};
```

即使两个型别之间可以自动转换，如果我们对 assign() 使用不同的 瑁__污瑁别，也会出错：

```
void f()
{
   MyClass<double> d;
   MyClass<int> i;

   d.assign(d); // OK
   d.assign(i); // ERROR: i is MyClass<int>
                //       but MyClass<double> is required
}
```

如果 纜□ 允许我们为 浥浢敲□_____提供不同（一个以上）的 瑁__污瑁型别，就可以放宽「必须精确吻合」这条规则；只要型别可被赋值（*assignable*），就可以被当做上述 浥浢敲[4]_涮____□____

```
template <class T>
class MyClass
  private:
    T value;
```

```
  public:
    template <class X> // member template
    void assign (const MyClass<X>& x) { // allows different template types
      value = x.getValue();
    }
    T getValue () const
      return value;
  }
  ...
};

void f()
{
   MyClass<double> d;
   MyClass<int> i;

   d.assign(d);  // OK
   d.assign(i);  // OK (int is assignable to double)
}
```

请注意，现在，assign()的参数 x 和*this 的型别并不相同，所以你不再能够直接存取 MyClass<> 的 刈_____成员和 ____散瑙_成员，取而代之的是，此例中你必须使用类似 getValue() 之类的东西。

_____潮 獐_畣___是 浥涅敲⁴_涮____的一种特殊形式。____污瑙___玲常用于「在复制对象时实现隐式型别转换」。注意，瑙__污瑙___玲___（*hide*）业__楣楴_copy*c onstructor* 。如果型别完全吻合，业_汩捩瑛 co *py constructor* 就会被产生出来并被唤起。举个例子：

```
template <class T>
class MyClass
   public:
     //copy constructor with implicit type conversion
     // - does not hide implicit copy constructor
     template <class U>
     MyClass (const MyClass<U>& x);
     ...
};

void f()
{
   MyClass<double> xd;
```

```
    ...
    MyClass<double> xd2(xd);    // calls built-in copy constructor
    MyClass<int> xi(xd);        // calls template constructor
    ...
}
```

在这里，xd2 和 xd 的型别完全一致，所以它被内建的 *copy ctor* 初始化。xi 的型别和 xd 不同，所以它使用 瑄__污瑄□琅_进行初始化。因此，撰写 瑄__污瑄□琅_如果□_哺畬__*copy ctor* 不符合你的需要，别忘了自己提供一个 *copy ctor* 。浥涅敲瑄__污瑄的另一个例子详见 瀳ﾟ-‴__节。

Nested Template Classes

巢状的（*nested*）掰慛獙_本身也可以是个 瑄__污瑄

```
template <class T>
class MyClass
    ...
    template <class T2>
    class NestedClass;
    ...
};
```

2.2.2 基本型别的显式初始化（explicit initialization）

如果采用不含参数的、明确的 __滷__畣___（建构式）呼叫语法，基本型别会被初始化为零：

```
int i1;          // undefined value
int i2 = int();  // initialized with zero
```

这个特性可以确保我们在撰写 ____污瑄程序代码时，任何型别都有一个确切的初值。例如下面这个函数中，x 保证被初始化为零。

```
template <class T>
void f()
{
    T x = T();
    ...
}
```

2.2.3 异常处理（Exception Handling）

藉由异常处理，纔□ 标准链接库可以在不「污染」函数接口（亦即参数和回返值）的情况下处理异常。如果你遇到一个意外情况，可以藉由「丢出一个异常」来停止一般的（正常的）处理过程：

```
class Error;
void f()
{
  ...
  if (exception-condition)
     throw Error(); // create object of class Error and throw it as exception
  }
  ...
}
```

述句 throw 开始了 *stack unwinding*（堆栈辗转开解）过程，也就是说，它将导致退离任何函数区段，就像遭遇 return 述句一般，然而程序却不会跳转到任何地点。对于所有被宣告于某区段□_而该区段却因程序异常而退离□_的区域对象而言，其 搗獗____淜（解构式）会被唤起。*Stack unwinding* 的动作会持续直到退出 main() 或直到有某个 catch 子句捕捉并处理了该异常为止。如果是第一种情况，程序会结束：

```
int main()
{
   try
      ...
      f();
      ...
   }
   catch (const Error&)
      ...   // handle exception
   }
   ...
}
```

在这里，_try 区段中任何「型别为 Error 的异常」都将在 catch 子句获得处理[6] 。

†††††††††††††††††††††††††

6 异常（数捥___）会终结某个函数调用动作，而该函数正是异常发生之处。异常处理机制有能力将某个对象当作自变量，回传给函数调用者。但这并非反向函数回呼（捡__惯__）。所谓反向是指从「问题被发现的地方」到「问题被处理的地方」，也就是从下往上的方向。以此观之，异常处理和信号处理（_____□_滶汩涌 ）完全是两码事。

异常对象（数捥_莹__□__散瑳）其实就是一般类别或基本型别的对象，可以是 ints、strings，也可以是类别体系中的某个⁴扬_污瑙_掸慛獩_。通常你会设计一个特殊的敲牯_类别体系。你可以运用异常对象的状态（獚慑_），将任何信息从错误被侦测到的地点带往错误被处理的地点。

注意，这个概念叫作「异常处理（*exception handling*）」，而不是「错误处理（*error handling*）」，两者未必相同。举个例子，许多时候用户的无效输入（这经常发生）并非一种异常；这时候最好是在区域范围内采用一般的错误处理技术来处理。

你可以运用所谓的异常规格（*exception specification*）来指明某个函数可能抛出哪些异常，例如：

```
void f() throw(bad_alloc);  // f()只可能丢出 bad_alloc 异常
```

如果宣告一个空白异常规格，那就表明该函数不会丢掷任何异常：

```
void f() throw();     // f()不丢掷任何异常
```

违反异常规格将会导致特殊行为，详见 潋_关于异常类别 *bad_exception* 的描述。

纜□ 标准链接库提供了一些通用的异常处理特性，例如标准异常类别（獚_滴___数捥_莹__□污 獳歟）和 auto_ptr 类别（详见 _㈱_″__节和⁰_买_袭_节）。

2.2.4 命名空间（Namespaces）

越来越多的软件以链接库、模块（__摵___）和组件拼凑完成。各种不同事物的结合，可能导致一场名称大冲突。乡浥狞惯敨 正是用来解决此一问题。

乡浥狞惯敨 将不同的标识符号集合在一个具名范围内。如果你在 __浥狞惯_之内定义所有标识符号，则 __浥狞惯_本身名称就成了唯一可能与其他全局符号冲突的标识符号。你必须在标识符号前加上 __浥狞惯_名字，才能援引该 __浥狞惯_的符号，这和 掸慛_处理方式雷同。__浥狞惯_的名字和标识符号之间以:: 分隔开来（译注：这个符号及其意义和 掸慛_与其 浥洹敲_之间的联系有点类似）。

```
// defining identifiers in namespace josuttis
namespace josuttis
   class File;
   void myGlobalFunc();
   ...
}
...
// using a namespace identifier
josuttis::File obj;
...
josuttis::myGlobalFunc();
```

不同于 掸慛_的是，__浥狞惯敨是开放的，你可以在不同模块（浯摵___）中定义

和扩展 __浥狩惯_。因此你可以使用 __浥狩惯_来定义模块、链接库或组件，甚至在多个档案之间完成。乡浥狩惯_定义的是逻辑模块，而非实质模块。请注意，在 哼_ 及其他建模表示法（_润___涌□_____潮_）中，模块又被称为 *package*。

如果某个函数的一个或多个自变量型别，乃定义于函数所处的 __浥__惯_中，那么你不必为该函数指定 __浥狩惯_。这个规则称为 *Koenig lookup*（译注：稽__敷酏___ _是 纔□ 社群代表人物之一，对 纔□ 的形成和发展有重要贡献）。例如：

```
// defining identifiers in namespace josuttis
namespace josuttis
    class File;
    void myGlobalFunc(const File&);
    ...
}
...

josuttis::File obj;
...
myGlobalFunc(obj); // OK, lookup finds josuttis::myGlobalFunc()
```

藉由 *using declaration*，我们可以避免一再写出冗长的 __浥狩惯_名称。例如以下宣告式_：

```
using josuttis::File;      // 译注：这就是一个 using declaration
```

会使 File 成为当前范围（捵牲敲_□_溃_）内代表 josuttis::File 的一个同义字。

using directive 会使 __浥狩惯攡内的所有名字曝光。*using directive* 等于将这些名字宣告于 __浥狩惯_之外。但这么一来，名称冲突问题就可能死灰复燃。例如：

```
using namespace josuttis; // 译注：这就是一个 using directive
```

会使 File 和 MyGlobalFunc() 在当前范围内完全曝光。如果全局范围内已存在同名的 File 或 MyGlobalFunc()，而且使用者不加任何资格饰词（__慬__楣慑地使用这两个名字，编译程序将东西难辨。

注意，如果脉络环境（_潮_____）不甚清楚（例如不清楚究竟在表头文件、模块或程式库中），你不应该使用 *using directive*。这个指令可能会改变 __浥__惯_的作用范围，从而使程序代码被包含或使用于另一模块中，导致意外行为的发生。事实上在头文件中使用 *using directive* 相当不智。

纔□ 标准链接库在□_浥狩惯攡 std 中定义了它的所有标识符号，详见 _㈱_″__

2.2.5 bool 型别

为了支持布尔值（真假值），纜__增加了 bool 型别。 bool 可增加程序的可读性，并允许你对布尔值实现重载（*overloaded*）动作。两个常数 true 和 false 同时亦被引入 纜_。此外 纜□ 还提供布尔值与整数值之间的自动转换。_值相当于 false，非 _值相当于 true。

2.2.6 关键词 explicit

藉由关键词 explicit 的作用，我们可以禁止「单自变量建构式（獗涌__□_____涎_潮獗_畣___）」被用于自动型别转换。典型的例子便是群集类别（捯菉散莹__□污獳你可以将初始长度作为自变量传给建构式，例如你可以宣告一个建构式，以 獚惯_的初始大小为参数：

```
class Stack
   explicit Stack(int size); // create stack with initial size
   ...
};
```

在这里，explicit 的应用非常重要。如果没有 explicit，这个建构式有能力将一个 int 自动转型为 Stack。一旦这种情况发生，你甚至可以给 Stack 指派一个整数值而不会引起任何问题：

```
Stack s;
...
s = 40; // Oops, creates a new Stack for 40 elements and assigns it to s
```

「自动型别转换」动作会把 仜 转换为有 仜 个元素的 獚惯_，并指派给 s，这几乎肯定不是我们所要的结果。如果我们将建构式宣告为 explicit，上述赋值（指派，慳獗__）动作就会导致编译错误（那很好）。

注意，explicit 同样也能阻绝「以赋值动作进行带有转型现象之初始化行为」：

```
Stack s1(40);    // OK
Stack s2 = 40;   // ERROR
```

这是因为以下两组动作：

```
X x;
Y y(x);      // explicit conversion
```

和

```
X x;
Y y = x;     //implicit conversion
```

存在一个小差异。前者透过显式转换，根据型别 X 产生了一个型别 Y 的新对象；后者透过隐式转换，产生了一个型别 Y 的新对象。

2.2.7 新的型别转换运算符（Type Conversion Operators）

为了让你对「自变量的显式型别转换」更透彻，纔__引入以下㈣ 个新的运算符：

1. static_cast

将一个值以合逻辑的方式转型。这可看做是「利用原值重建一个临时对象，并在设立初值时使用型别转换」。唯有当上述的型别转换有所定义，整个转换才会成功。所谓的「有所定义」，可以是语言内建规则，也可以是程序员自定的转换动作。例如：

```
float x;
...
cout << static_cast<int>(x);  // print x as int
...
f(static_cast<string>("hello")); // call f() for string instead of char*
```

2. dynamic_cast

将多形型别（_____灨__⁴_灥）向下转型（__睮捡獙）为其实际静态型别（牥懂獙慑楣⁴___）。这是唯一在执行期进行检验的转型动作。你可以用它来检验某个多形实值（灯___滗灨__⁶____）的型别，例如：

```
class Car; // abstract base class (has at least one virtual function)

class Cabriolet : public Car
...
};

class Limousine : public Car
...
};

void f(Car* cp)
{
   Cabriolet* p = dynamic_cast<Cabriolet*>(cp);
   if (p == NULL)
      // did not refer to an object of type Cabriolet
      ...
   }
}
```

在这个例子中，面对实际静态型别为 Cabriolet 的对象，f()有特殊应对行为。当自变量是个 牥__牥___，而且型别转换失败时，dynamic_cast 丢出一个 *bad_cast* 异常（*bad_cast* 的描述见 潋_）。注意，以设计角度而言，你应该在运用多型技术的程序中，避免这种「程序行为取决于具体型别」的写法。

3. const_cast

   设定或去除型别的常数性（捌__璟歁_），亦可去除 volatile 饰词。除此之外不允许任何转换。

4. reinterpret_cast

   此运算符的行为由实作品（编译程序）定义。可能重新解释 _楴_意义，但也不一定如此。使用此一转型动作通常带来不可移植性。

这些运算符取代了以往小圆括号所代表的旧式转型，能够清楚阐明转型的目的。小圆括号转型可替换 dynamic_cast 之外的其他三种转型，也因此当你运用它时，你无法明确显示使用它的确切理由。这些新式转型运算符给了编译程序更多信息，让编译程序清楚知道转型的理由，并在转型失败时释出一份错误报告。

注意，这些运算符都只接受一个自变量。试看以下例子：

```
static_cast<Fraction>(15,100) // Oops, creates Fraction(100)
```

在这个例子中你得不到你想要的结果。它只用一个数值 _，将 Fraction 暂时物件设定初值，而非设定分子 ﹛、分母 _。逗号在这里并不起分隔作用，而是形成一个_comma（逗号）运算符，将两个算式组合为一个算式，并传回第二算式做为最终结果。将 ﹛ 和 _「转换」为分数的正确做法是：

```
Fraction(15,100) // fine, creates Fraction(15,100)
```

2.2.8 常数静态成员（Constant Static Members）的初始化

如今，我们终于能够在 掰悜_宣告式中对「整数型（楮瑙__慬）常数静态成员」直接赋予初值。初始化后，这个常数便可用于 掰悜_之中，例如：

```
class MyClass
   static const int num = 100;
   int elems[num];
   ...
};
```

注意，你还必须为 掰悜_之中宣告的常数静态成员，定义一个空间：

```
const int MyClass::num;    // no initialization here
```

2.2.9 main() 的定义式

我乐于澄清这个语言中一个重要而又常被误解的问题，那就是正确而可移植的 main() 的唯一写法。根据 纔_标准规格，只有两种 main() 是可移植的：

```
int main()
{
    ...
}
```

和

```
int main (int argc, char* argv[])
{
    ...
}
```

这里 argv（命令行参数数组）也可定义为 char**。请注意，由于不允许「不言而喻」的回返型别 int，所以回返型别必须明白写为 int。你可以使用 return 述句来结束 main()，但不必一定如此。这一点和 _不同，换句话说，纔□ 在 main() 的尾端定义了一个隐喻的：

```
return 0;
```

这意味如果你不采用 return 述句离开 main()，实际上就表示成功退出（传回任何一个非零值都代表某种失败）。出于这个原因，本书范例在 main() 尾端都没有 return 述句。有些编译程序可能会对此发出警告（译注：例如□棵_慬____有的甚至认为这是错误的。唔，那正是标准制定前的黑暗日子。

2.3 复杂度和 Big-O 表示法

对于 纔□ 标准链接库的某些部分（特别是 协_），算法和成员函数的效率需要严肃考虑，因此需要动用「复杂度」的概念。计算器科学家运用特定符号，比较算法的相对复杂度，如此便可以很快依据算法的运行时间加以分类，进行演算法之间的定性比较。这种衡量方法叫做⁂ 模__表示法。

籀___表示法系将一个算法的运行时间以输入量 _的函数表示。例如，当运行时间随元素个数成线性增长时（亦即如果元素个数呈倍数增长，运行时间亦呈倍数增长），复杂度为 佝__；如果运行时间独立于输入量，复杂度为 佝__。表 ㈥_列出典型的复杂度和其 籀___表示法。

请注意，籀___表示法隐藏了（忽略了）指数较小的因子（例如常数因子），这一点十分重要，更明确地说，它不关心算法到底耗用多长时间。根据这种量测法则，任何两个线性算法都被视为具有相同的接受度。甚至可能发生一种情况：带有巨大常数的线性算法竟然比带有小常数的指数算法受到欢迎（译注：因

为 籀___复杂度表示法无法显现真实的运算时间）。这是对 籀___表示法的一种合理批评。记住，这只是一种量度规则。具有最佳（最低）复杂度的算法，不一定就是最好（最快）的算法。

型别	表示法	含义
常数	佝__	运行时间与元素个数无关。
对数	佝汯______	运行时间随元素个数的增加呈对数增长。
线性	佝_	运行时间随元素个数的增加呈线性增长。
_____札_	_⠏≈__⠏__	运行时间随元素个数的增加呈「线性和对数的乘积」增长。
二次	佝 −	运行时间随元素个数的增加呈平方增长。

表 ㈎_†典型的五种复杂度

表 ㈎_列出所有复杂度分类，并以某些元素个数来说明运行时间随元素个数增长的程度。一如你所看到，当元素较少时，运行时间的差别很小，此时 籀___表示法所隐藏的常数因子可能会带来很大影响。但是当元素个数愈多，运行时间差别愈大，常数因子也就变得无关紧要了。当你考虑复杂度时，请记住，输入量必须够大才有意义。

†复杂度_		元素数目							
型别	表示法	_	_	_	匢	_	___		
常数	_____	_	┐	_	_	_	_		
对数	_______温_	_	_	_	_	_	_		
线性	__温	_	_	_	___	___	_	广〜_	
渭______	______滰__⋨	_	_	㆑	佡	ㇶ_	媟_	□__	
二次方	佝 −	_	_	___	_	㈬匢_	□__	_□__□〜	

表 ㈎_†运行时间、复杂度、元素个数_对照表

纜□ 标准手册中的某些复杂度被称为 *amortized*（分期摊还），意思是，长期而言，大量操作将如上述描述般进行，但单一操作却可能花费比平均值更长的时间。举个例子，如果你为一个 _____浩挠慲牪_追加元素，运行时间将取决于 慲牪_是否尚有备用内存。如果内存足够，就属于常数复杂度，因为在尾端加入一个新元素，总是花费相同时间。如果备用内存不足，那么就是线性复杂度，因为你必须配置足够的内存并搬动（复制）它们，实际耗用时间取决于当时的元素个数。记忆体重新配置动作并不常发生（译注：协_的 _____浩挠慲牪_容器会以某种哲学来保持备用内存），所以任何「长度充份」的序列（獥__敽捥），元素附加动作几乎可说是常数复杂度。这种复杂度我们便称为 *amortized*（分期摊还）常数时间。

3

一般概念

General Concepts

本章讲述 C++ 标准链接库中的基本概念。几乎所有 C++ 标准链接库组件都需要这些概念。

- 命名空间（namespace）std
- 头文件（headers）的名称与格式
- 错误（error）和异常（exception）处理的一般概念
- 配置器（allocator）的简单介绍

3.1 命名空间（namespace）std

当你采用不同的模块和链接库时，经常会出现名称冲突现象，这是因为不同的模组和链接库可能针对不同的对象使用相同的标识符号。namespaces（参见 p16, 2.2.4 节的介绍）用来解决这个问题。所谓 namespace，是指针识符号的某种可见范围。和 class 不同，namespace 具有扩展开放性，可以发生于任何源码档案上。因此你可以利用一个 namespace 来定义一些组件，而它们可散布于多个实质模块上。这类组件的典型例子就是 C++ 标准链接库，因为 C++ 标准链接库使用了一个 namespace。事实上，C++ 标准链接库中的所有标识符号都被定义于一个名为 std 的 namespace 中。

由于 namespace 的概念，使用 C++ 标准链接库的任何标识符号时，你有三种选择：

1. 直接指定标识符号。例如 std::ostream 而不是 ostream。完整语句类似这样：
   ```
   std::cout << std::hex << 3.4 << std::endl
   ```
2. 使用 *using declaration*（详见 p17）。例如以下程序片段使我们不必再写出范围修饰符号 std::，而可直接使用 cout 和 endl：
   ```
   using std::cout;
   using std::endl;
   ```
   于是先前的例子可以写成这样：

```
cout << std::hex << 3.4 << endl;
```

3. 使用 *using directive*（详见 p17），这是最简便的选择。如果对 namespace `std` 采用 *using directive*，便可以让 `std` 内定义的所有标识符号都有效（曝光），就好像它们被宣告为全局标识符号一样。因此，写下：

```
using namespace std;
```

之后，就可以直接写：

```
cout << hex << 3.4 << endl;
```

但请注意，由于某些晦涩的重载（overloading）规则，在复杂的程序中，这种方式可能导致意外的命名冲突，更糟的是甚至导致不一样的行为。如果场合不够明确（例如在表头文件、模块或链接库中），就应避免使用 *using directive*。

本书的例子都很小，所以，为了我自己方便，书中范例程序通常采用最后一种手法。

3.2 头文件 （Header Files）

将 C++ 标准链接库中所有标识符号都定义于 namespace `std` 里头，这种做法是标准化过程中引入的。这个作法不具回溯兼容性，因为原先的 C/C++ 头文件都将 C++ 标准链接库的标识符号定义于全局范围（global scope）。此外标准化过程中有些 classes 的接口也有了更动（当然啦，尽可能以回溯兼容为目标）。为此，特别引入了一套新的头文件命名风格，这么一来组件供货商得以藉由「提供旧的头文件」来达到回溯兼容目的。

既然有必要重新定义标准头文件的名称，正好藉此机会把头文件扩展名做个规范。以往，头文件扩展名五花八门，包括 .h, .hpp, .hxx。相较之下如今的标准头文件副档名简洁得令人吃惊：根本就没有扩展名。于是标准头文件的 `#include` 如下：

```
#include <iostream>
#include <string>
```

这种写法也适用于 C 标准头文件。但必须采用前缀字符 `c`，而不再是扩展名 `.h`：

```
#include <cstdlib>    // was: < stdlib.h>
#include <cstring>    // was: < string.h>
```

在这些头文件中，每一个标识符号都被宣告于 namespace `std`。

这种命名方式的优点之一是可以区分旧头文件中的 `char*` C 函数，和新头文件中的标准 C++ `string` class：

```
#include <string>     // C++ class string
#include <cstring>    // char* functions from C
```

注意，以操作系统角度观之，新头文件命名方式并非意味标准头文件没有扩展名。标准头文件的 `#include` 该如何处理，由编译程序决定。C++ 系统可以自动添加一个扩展名，甚至可以使用内建宣告，不读入任何档案。不过实际上大多数系统只是简单含入一个「名称与 `#include` 句中的档名完全相同」的档案。所以，在大部份系统中，C++ 标准头文件都没有扩展名。注意，「无扩展名」这一条件只适用于标准头文件。一般而言，为你自己所写的头文件加上一个良好的扩展名，仍然是个好主意，有助于轻易识别出这些档案的性质。

为了回溯兼容于 C，旧式的 C 标准头文件仍然有效，如果需要，你还是可以使用它们，例如：

```
#include <stdlib.h>
```

此时，标识符号同时宣告于全局范围和 namespace `std` 中。事实上这些头文件的行为类似于先在 `std` 中宣告所有标识符号，再悄悄使用 *using declaration* 把这些标识符号引入全局范围（参见 p17）。

至于 `<iostream.h>` 这一类 C++ 旧式表头文件，标准规格中并未加以规范（这一点在标准化过程中曾经多次改变），意味不再支持这些头文件。不过目前大多数厂商都会提供它们，以求回溯兼容。

注意，除了引入 namespace `std`，头文件还有很多改变。所以，你要嘛就采用表头档旧名，要嘛就应该完全改用新的标准名称。

3.3 错误（Error）处理和异常（Exception）处理

C++ 标准链接库由不同的成份构成。来源不同，设计与实现风格迥异。而错误处理和异常处理正是这种差异的典型表征。标准链接库中有一部分，例如 string classes，支持具体的错误处理，它们检查所有可能发生的错误，并于错误发生时丢出异常。至于其它部分如 STL 和 valarrays，效率重于安全，因此几乎不检验逻辑错误，并且只在执行期（runtime）发生错误时才丢掷异常。

3.3.1 标准异常类别（Standard Exception Classes）

语言本身或标准链接库所丢掷的所有异常，都衍生自基础类别 `exception`。这是其他数个标准异常类别的基础类别，他们共同构成一个类别体系，如图 3.1。这些标准异常类别可分为三组：

1. 语言本身支持的异常
2. C++ 标准链接库发出的异常
3. 程序作用域（scope of a program）之外发出的异常

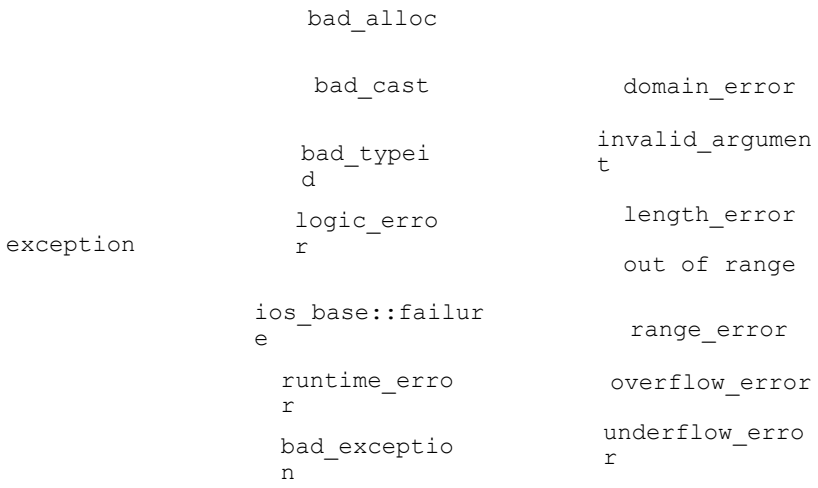

图 3.1 标准异常（Standard Exceptions）阶层体系

语言本身所支持的异常

此类异常用以支撑某些语言特性，所以，从某种角度来说它们不是标准链接库的一部分，而是核心语言的一部分。如果以下操作失败，就会丢掷这一类异常。

- 全局运算符 new 操作失败，会丢掷 *bad_alloc* 异常（若采用 new 的 nothrow 版本，另当别论）。由于这个异常可能于任何时间在任何较复杂的程序中发生，所以可说是最重要的一个异常。
- 执行期间，当一个加诸于 reference 身上的「动态型别转换动作」失败时，dynamic_cast 会丢掷 *bad_cast* 异常。p19 对于 dynamic_cast 运算符有些描述。
- 执行期型别辨识（RTTI）过程中，如果交给 typeid 的自变量为零或空指标，typeid 运算符会丢掷 *bad_typeid* 异常。
- 如果发生非预期的异常，*bad_exception* 异常会接手处理，方式如下：当函数掷出异常规格（exception specification，p16 介绍）以外的异常，*bad_exception* 就会唤起 unexpected()。例如：

```
class E1;
class E2; // not derived from E1
```

```
void f() throw(E1) // throws only exceptions of type E1
{
   ...
   throw E1(); // throws exception of type E1
   ...
   throw E2(); // calls unexpected(), which calls terminate()
}
```

f()之中丢掷出「型别为 *E2*」的异常，这种动作违反了异常规格（exception specification）的设定，于是唤起 unexpected()，后者通常会唤起 terminate()终止程序。

然而如果你在你的异常规格中列出 *bad_exception*，那么 unexpected() 总是会重新掷出（rethrows）*bad_exception* 异常。

```
class E1;
class E2;    // not derived from E1

void f() throw(E1, std::bad_exception)
                // throws exception of type E1 or
                // bad_exception for any other exception type
{
   ...
   throw E1();   // throws exception of type E1
   ...
   throw E2();   // calls unexpected(),which throws bad_exception
}
```

因此，如果异常规格罗列了 *bad_exception*，那么任何未列于规格的异常，都将在函数 unexpected()中被代之以 *bad_exception*[1] 。

C++ 标准链接库所发生的异常

C++ 标准链接库异常总是衍生自 *logic_error*。理论而言，我们能够透过一些手段，在程序中避免逻辑错误 — 例如对函数自变量进行额外测试等等。所谓逻辑错误包括违背逻辑前提或违反 class 的不变性。C++ 标准链接库提供以下逻辑错误类别：

- *invalid_argument* 表示无效自变量，例如将 bitset（array of bits）以 char 而非 '0' 或 '1' 进行初始化。
- *length_error* 指出某个行为「可能超越了最大极限」，例如对着某个字符串附加太多字符。

[1] 你可以修改 unexpected()的具体操作。然而只要宣告有异常规格，函数就绝不掷出规格中未列的异常。

- *out_of_range* 指出自变量值「不在预期范围内」，例如在诸如 array 的群集（collection）或字符串（string）中采用一个错误索引。
- *domain_error* 指出专业领域范畴内的错误。

此外，标准链接库的 I/O 部分提供了一个名为 *ios_base::failure* 的特殊异常。当资料串流（data stream）由于错误或由于到达档案尾端而发生状态改变时，就可能丢掷这个异常。此一异常的具体行为见 p602, 13.4.4 节。

程序作用域（scope of a program）之外发生的异常

衍生自 *runtime_error* 的异常，用来指出「不在程序范围内，且不容易回避」的事件。C++ 标准链接库针对执行期错误提供了以下三个 classes：

- *range_error* 指出内部计算时发生区间错误（range error）。
- *overflow_error* 指出算术运算发生上溢位（overflow）。
- *underflow_error* 指出算术运算发生下溢位（underflow）。

标准链接库所丢掷的异常

C++ 标准链接库自身可能丢掷 *range_error*、*out_of_range* 和 *invalid_argument* 异常。然而由于标准链接库会用到语言特性及客户所写的程序代码，所以也可能间接掷出任何异常。尤其是，无论何时配置储存空间，都有可能掷出 *bad_alloc* 异常。

标准链接库的任何具体实作品，都可能提供额外的异常类别（或作为兄弟类别，或衍生为子类别）。使用这些非标准类别将导致程序难以移植，因为一旦你想采用其他标准链接库实作版本，就不得不痛苦地修改你的程序。所以最好只使用标准异常。

异常类别的头文件

基础类别 *exception* 和 *bad_exception* 定义于 `<exception>`。*bad_alloc* 定义于 `<new>`。*bad_cast* 和 *bad_typeid* 定义于 `<typeinfo>`。*ios_base::failure* 定义于 `<ios>`。其他异常类别都定义于 `<stdexcept>`。

3.3.2 异常类别（Exception Classes）的成员

为了在 `catch` 子句中处理异常，你必须采用异常所提供的接口。所有标准异常的界面只含一个成员函数：`what()`，用以获取「型别本身以外的附加信息」。它传回一个以 null 结束的字符串：

```
namespace std {
   class exception {
      public:
```

```
        virtual const char* what() const throw();
        ...
    };
}
```

被传回的字符串，其内容由实作厂商定义。它很大程度（但非必然）决定了帮助的级别和信息的详细度。注意，该字符串有可能是个以 null 结尾的 "multibyte" 字符串，可被轻松转换为 `wstring`（详见 p480, 11.2.1 节）并显示出来。`what()` 传回的 C-string 在其所属的异常物件被摧毁后，就不再有效了 。[2]

标准异常中的其他成员，用来处理生成、复制、赋值、摧毁等动作。要注意的是，除了 `what()`，再没有任何异常提供任何其他成员函数，能够描述异常的种类。例如，没有可找出异常脉络（context）的一致性方法，或找出区间错误（range error）发生时的错误索引值。因此，唯一通用的异常评估手段，大概只有打印一途了：

```
try {
    ...
}
catch (const std::exception& error) {
    // print implementation-defined error message
    std::cerr << error.what() << std::endl;
    ...
}
```

唯一可能实现的另一个异常评估手段是，根据异常的精确型别，自己得出推论。例如，如果 *bad_alloc* 异常被掷出，可能是因为程序企图获得更多内存。

3.3.3 丢掷标准异常

你可以在自己的链接库或程序内部丢掷某些标准异常。允许你这般运用的各个标准异常，生成时都只需要一个 `string` 参数（第 11 章对 `string` class 有所描述），它将成为被 `what()` 传回的描述字符串。例如 *logic_error* 定义如下：

```
namespace std {
    class logic_error : public exception {
       public:
       explicit logic_error (const string& whatString);
    };
}
```

[2] C++ 标准规格并未对 `what()` 回返值的寿命加以规范，这里所讲的是被提出的一种建议解决方案。

提供这种功能的标准异常有：*logic_error* 及其衍生类别、*runtime_error* 及其衍生类别、*ios_base::failure*。你不能丢掷 *exception*，也不能丢掷任何用以支持语言核心性质的异常。

想要丢掷一个标准异常，只需生成一个描述该异常的字符串，并将它初始化，交给异常物件：

```
std::string s;
...
throw std::out_of_range(s);
```

由于 `char*` 可被隐晦转换为 `string`，所以你可以直接使用字符串字面常数：

```
throw std::out of range("out of range (somewhere, somehow)");
```

3.3.4 从标准异常类别（Exception Classes）中衍生新类别

另一个在程序中采用标准异常类别的可能情况是，定义一个直接或间接衍生自 *exception* 的特定异常类别。要这么做，首先必须确保 `what()` 机制正常运作。`what()` 是个虚拟函数，所以提供 `what()` 的方法之一就是自己实现 `what()`：

```
namespace MyLib {
  /* user-defined exception class
  * derived from a standard class for exceptions
  */
  class MyProblem : public std::exception {
    public:
    ...
    MyProblem(...) { // special constructor
    }
    virtual const char* what() const throw() { // what() function
      ...
    }
  };
  ...

  void f() {
    ...
    // create an exception object and throw it
```

```
    throw MyProblem(...);
    ...
  }
}
```

提供 what() 函数的另一种方法是，令你的异常类别衍生自 3.3.3 节所描述的标准异常：

```
namespace MyLib {
  /* user-defined exception class
  * - derived from a standard class for exceptions
  * that has a constructor for the what() argument
  */

  class MyRangeProblem : public std::out_of_range {
    public:
    MyRangeProblem (const string& whatString)
      : out_of_range(whatString) {
    }
  };
  ...

  void f() {
  ...
  //   create an exception object by using a string constructor and throw it
  throw MyRangeProblem("here is my special range problem");
  ...
  }
}
```

完整程序见 p441 的 Stack 和 p450 的 Queue。

3.4 配置器 （Allocators）

C++ 标准链接库在许多地方采用特殊对象来处理内存配置和寻址，这样的对象称为配置器（allocator）。配置器表现出一种特定的内存模型（memory model），成为一个抽象表征，表现出「内存需求」至「内存低阶呼叫」的转换。如果运用多个不同的配置器对象，你便可以在同一个程序中采用不同的内存模型。

配置器最初是作为 STL 的一部份而引进，用于处理诸如 PC 上不同指标型别（例如 near, far, huge 指标）这一类乱七八糟的问题；现在则作为一种技术方案的基础，

使得诸如共享内存（shared memory）、垃圾回收（garbage collection）、对象导向数据库（object oriented databases）等特定内存模型，能够保持一致的接口。但是这种用法还相当新颖，尚未获得广泛的接受（情况正在改变中）。

C++ 标准链接库定义了一个默认配置器（default allocator）如下：

```
namespace std {
   template <class T>
   class allocator;
}
```

默认配置器可在任何「配置器得以被当作自变量使用」的地方担任默认值。预设配置器会执行内存配置和回收的一般性手法，也就是呼叫 new 和 delete 运算符。但是 C++ 并没有对于「在什么时候以什么方式调用这些运算符」给予明确规定。所以，默认配置器甚至可能对已配置之内存施行「内部快取（internal cache）」手法。

绝大多数程序都采用默认配置器，但有时候其他链接库也可能提供某些配置器以满足特定需求。这种情况下只需简单地将它们当作自变量即可。自行设计并实作配置器的实际意义不大。实际生活中最典型的方式还是直接采用默认配置器，所以我将迟至第 15 章才详细探讨配置器。（译注：举个例子，SGI STL 对外呈现一个透通的配置器接口，内部却维护有一、二两级配置器，制作出十分精巧繁复的 memory pool 机制，对于体积小而数量极大的对象需求而言，可带来极好的时间和空间效率。详见《STL 源码剖析》第 2 章）

4

通用工具

Utilities

本章讲解 C++ 标准链接库中的通用工具。它们由短小精干的类别和函数构成，执行最一般性的工作。这些工具包括：

- 数种通用型别（general types）
- 一些重要的 C 函数
- 数值极值[1]（numeric limits）

大部分通用工具在 C++ 标准规格书第 20 款（clause）描述，定义于标准头文件 `<utility>` 内。其余工具则与标准链接库中一些比较主要的组件一起描述，其原因可能是该类工具主要便是和那些组件共同使用，抑或因为历史因素。例如某些通用辅助函数被定义于 `<algorithm>` 头文件中，但按照 STL 的定义，它们不算是算法（参见第 5 章）。

这些工具中的一部分也被运用于 C++ 标准链接库中。特别是型别 `pair`，凡需要将两个值视为一个单元的场合（例如必须「回传两个值」的某函数），就必须用到它。

4.1 Pairs（对组）

class `pair` 可以将两个值视为一个单元。C++ 标准链接库内多处使用了这个 class。尤其容器类别 `map` 和 `multimap`，就是使用 `pairs` 来管理其键值/实值（value/key）的成对元素（详见 6.6 节，p194）。任何函数需回传两个值，也需要 `pair`。

[1] 可能有些人认为数值极值应该属于第 12 章，也就是专门讲解数值的那一章，但这些数值极值在链接库的其他部分也会被用到，所以我决定把它放在这里。

Structure pair 定义于<utility>：

```
namespace std {
   template <class T1, class T2>
   struct pair {
      // type names for the values
      typedef T1 first_type;
      typedef T2 second_type;

      // member
      T1 first;
      T2 second;

      /* default constructor
      *-T1() and T2() force initialization for built-in types
      */
      pair()
        : first(T1()), second(T2()) {
      }

      // constructor for two values
      pair(const T1& a, const T2& b)
      : first(a), second(b) {
      }

      // copy constructor with implicit conversions
      template<class U, class V>
      pair(const pair<U,V>& p)
        : first(p.first), second(p.second) {
      }
   };

   // comparisons
   template <class T1, class T2>
   bool operator== (const pair<T1,T2>&, const pair<T1,T2>&);
   template <class T1, class T2>
   bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
   ... // similar: !=, < =, > , > =
```

```
    // convenience function to create a pair
    template <class T1, class T2>
    pair<T1,T2> make_pair (const T1&, const T2&);
}
```

注意，pair 被定义为 struct，而不是 class，这么一来，所有成员都是 public，我们因此可以直接存取 pair 中的个别值。

上述 *default* 建构式生成一个 pair 时，以两个「被该 *default* 建构式个别初始化」的值做为初值。根据语言规则，基本型别（如 int）的 *default* 建构式也可以引起适当的初始化动作，所以：

```
std::pair<int,float> p; // initialize p.first and p.second with zero
```

就是以 int()和 float()来初始化 p。这两个建构式都传回零值。p14 曾经讨论过基本型别的显式初始化动作。

这里之所以使用 template 形式的 *copy* 建构式，乃是因为建构过程中可能需要隐式型别转换。如果 pair 对象被复制，唤起的是由系统隐喻合成的那个 *copy* 建构式[2]。例如：

```
void f(std::pair<int,const char*>);
void g(std::pair<const int,std::string>);
...
void foo {
    std::pair<int,const char*> p(42,"hello");
    f(p);    // OK: calls built-in default copy constructor
    g(p);    // OK: calls template constructor
}
```

Pair 之间的比较

为了比较两个 pair 对象，C++ 标准链接库提供了大家惯用的运算符。如果两个 pair 对象内的所有元素都相等，这两个 pair 对象就被视为相等（equal）：

```
namespace std {
    template <class T1, class T2>
    bool operator== (const pair<T1,T2>& x, const pair<T1,T2>& y)
{
        return x.first == y.first && x.second == y.second;
    }
}
```

[2] template 形式的建构式并不会遮掩（由编译程序）隐喻合成的 *default* 建构式。详见 p13。

两个 pairs 互相比较时，第一元素具有较高的优先序。所以如果两个 pairs 的第一元素不相等，其比较结果就成为整个比较行为的结果。如果第一元素相等，才继续比较第二元素，并把比较结果当作整体比较结果。

```
namespace std {
  template <class T1, class T2>
  bool operator< (const pair<T1,T2>& x, const pair<T1,T2>& y) {
    return x.first < y.first ||
      (!(y.first < x.first) && x.second < y.second);
  }
}
```

其他的比较运算符（comparison operators）也如法炮制。

4.1.1 便捷函数 make_pair()

template 函数 make_pair() 使你无需写出型别，就可以生成一个 pair 对象[3]：

```
namespace std {
    // create value pair only by providing the values
    template <class T1, class T2>
    pair<T1,T2> make_pair (const T1& x, const T2& y) {
       return pair<T1,T2>(x, y);
    }
}
```

例如，你可以这样使用 make_pair()：

```
std::make_pair(42,'@')
```

而不必费力地这么写：

```
std::pair<int,char>(42,'@')
```

当我们有必要对一个接受 pair 自变量的函数传递两个值时，make_pair()尤其显得方便，请看下例：

```
void f(std::pair<int,const char*>);
void g(std::pair<const int,std::string>);
...
void foo {
    f(std::make_pair(42,"hello")); // pass two values as pair
    g(std::make_pair(42,"hello")); // pass two values as pair
    // with type conversions
}
```

[3] 使用 make_pair 并不会多花你任何运行时间，编译程序应该会将此一动作优化。

从例子中可以看出，make_pair()使得「将两个值当作一个 pair 自变量来传递」的动作更容易。即使两个值的型别并不准确符合要求，也能在 template 建构式提供的支持下顺利工作。当你使用 map 和 multimap，你会经常用到这个特点（详见 p203）。

注意，一个算式如果明白指出型别，便带有一个优势：产生出来的 pair 将有绝对明确的型别。例如：

```
std::pair<int,float>(42,7.77)
```

其结果与：

```
std::make_pair(42,7.77)
```

不同。后者所生成的 pair，第二元素的型别是 double（因为「无任何饰词的浮点字面常数」，其型别被视为 double）。当我们使用重载函数（overloaded function）或 template，确切的型别非常重要。例如，为了提高效率，程序员可能同时提供分别针对 float 和 double 的 function 或 template，这时候确切的型别就非常重要了。

4.1.2 Pair 运用实例

C++ 标准链接库大量运用了 pair。例如 map 和 multimap 容器的元素型别便是 pair，也就是一组键值/实值（key/value）。关于 maps 和 multimaps 的一般性描述，详见 p194, 6.6 节。p91 有一个 pair 型别的运用实例。C++ 标准链接库中凡是「必须传回两个值」的函数，也都会利用 pair 对象（实例请见 p183）。

4.2 Class auto_ptr

本节描述 auto_ptr 型别。C++ 标准链接库提供的 auto_ptr 是一种智能型指标（smart pointer），帮助程序员防止「异常被掷出时发生资源泄漏」。注意我说的是「一种」智能型指标，现实生活中还有其他许多有用的智能型指标，auto_ptr 只是针对某个特定问题而设计，对于其他问题，auto_ptr 无能为力。所以，请谨慎阅读以下内容。

4.2.1 auto_ptr 的发展动机

函数的操作经常依以下模式进行：[4]

1. 获取一些资源。

2. 执行一些动作。

3. 释放所获取的资源。

如果一开始获取的资源，被系结于区域对象（local objects）身上，当函数退出时，它们的解构式（destructor）被唤起，从而自动释放这些资源。然而事情并不总是如此顺利，如果资源是以显式手法（explicitly）获得，而且未被系结于任何对象身上，那就必须以显式手法释放。这种情形常常发生在指标身上。

一个典型的例子就是运用 new 和 delete 来产生和销毁对象:

```
void f()
{
     ClassA* ptr = new ClassA; //    create an object explicitly
     ...                       //  perform some operations
     delete ptr;               //    clean up (destroy the object explicitly)
}
```

也许你尚未意识到，这个函数其实是一系列麻烦的根源。一个显而易见的问题是，我们经常忘掉 delete 动作，特别是当函数中间存在 return 述句时更是如此。然而真正的麻烦发生于更隐晦之处，那就是当异常发生时我们所要面对的灾难。异常一旦出现，函数将立刻退离，根本不会呼叫函数尾端的 delete 述句。结果可能是内存遗失，或更一般地说是资源遗失。防止这种资源遗失的常见办法就是捕捉所有异常，例如：

[4] class auto_ptr 的推动，是以 Scott Meyers 所著《*More Effective C++*》书中的相关议题为基础（并获得他的允许）。这个问题的一般性技术最早描述于 Bjarne Stroustrup 的《*The C++ Programming Language*》2 nd Edtion 和《*The Design and Evolution of C++*》，当时的主题是 "resource allocation is initialization"。auto_ptr 被加入 C++ 标准之中，正是为了支持此一技术。

```
void f()
{
   ClassA* ptr = new ClassA;   // create an object explicitly

   try {
      ...              // perform some operations
   }
   catch (...) {       // for any exception
      delete ptr;      // - clean up
      throw;           // - rethrow the exception
   }

   delete ptr;         // clean up on normal end
}
```

你看，为了在异常发生时处理对象的删除工作，程序代码变得多么复杂和累赘！如果还有第二个对象，如果还要比照办理，如果还需要更多的 `catch` 子句，那简直是一场恶梦。这不是优良的编程风格，复杂而且容易出错，必须尽力避免。

如果使用智能型指标，情形就会大不相同。这个智能型指标应该保证，无论在何种情形下，只要自己被摧毁，就一定连带释放其所指资源。而由于智能型指标本身就是局部变量，所以无论是正常退出，还是异常退出，只要函数退出，它就一定会被销毁。`auto_ptr` 正是这种智能型指标。

`auto_ptr` 是这样一种指标：它是「它所指向的物件」的拥有者（owner）。所以，当身为对象所有者的 `auto_ptr` 被摧毁时，该对象也将遭到摧毁。`auto_ptr` 要求，一个物件只能有一个拥有者，严禁一物二主。

下面是上例改写后的版本：

```
// header file for auto_ptr
#include <memory>

void f()
{
   // create and initialize an auto_ptr
   std::auto_ptr<ClassA> ptr(new ClassA);

   ...              // perform some operations
}
```

不再需要 `delete`，也不再需要 `catch` 了。`auto_ptr` 的接口与一般指针非常相似：`operator*` 用来提领其所指物件，`operator->` 用来指向物件中的成员。然而，所有指标算术（包括 `++`）都没有定义（这可能是件好事，因为指标算术是一大麻烦根源）。

注意，auto_ptr<> 不允许你使用一般指标惯用的赋值（assign）初始化方式。你必须直接使用数值来完成初始化[5]：

```
std::auto_ptr<ClassA> ptr1(new ClassA);     // OK
std::auto_ptr<ClassA> ptr2 = new ClassA;    // ERROR
```

4.2.2 auto_ptr 拥有权（Ownership）的转移

auto_ptr 所界定的乃是一种严格的拥有权观念。也就是说，由于一个 auto_ptr 会删除其所指对象，所以这个对象绝对不能同时被其他对象「拥有」。绝对不应该出现多个 auto_ptrs 同时拥有一个对象的情况。不幸的是，这种事情可能会发生（如果你以同一个对象为初值，将两个 auto_ptrs 初始化，就会出现这种事）。程序员必须负责防范这种错误。

这个条件导致了一个问题：auto_ptr 的 *copy* 建构式和 *assignment* 运算符应当如何运作？此类操作往往是将此处数据拷贝到彼处。然而这种操作恰恰会导致上面所提的情形。解决办法很简单，但意义深远：令 auto_ptr 的 *copy* 建构式和 *assignment* 运算符将对象拥有权交出去。试看下例 *copy* 建构式的运用：

```
// initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);

// copy the auto_ptr
// - transfers ownership from ptr1 to ptr2
std::auto_ptr<ClassA> ptr2(ptr1);
```

第一个述句中，ptr1 拥有了那个 new 出来的对象。第二个述句中，拥有权由 ptr1 转交给 ptr2。此后 ptr2 就拥有了那个 new 出来的对象，而 ptr1 不再拥有它。这样，对象就只会被 delete 一次 — 在 ptr2 被销毁的时候。

[5] 下面两种情况实际上是有分别的：

```
X x;
Y y(x);    // 显式转换（explicit conversion）
```

和：

```
X x;
Y y = x;   // 隐式转换（implicit conversion）
```

前者使用显式转换，以型别 X 建构型别 Y 的一个新对象，后者使用隐式转换。

赋值（指派, *assign*）动作也差不多：

```
// initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);
std::auto_ptr<ClassA> ptr2;     // create another auto_ptr
ptr2 = ptr1; // assign the auto_ptr
             // - transfers ownership from ptr1 to ptr2
```

在这里，赋值（*assign*）动作将拥有权从 ptr1 转移至 ptr2。于是，ptr2 拥有了先前被 ptr1 所拥有的那个物件。

如果 ptr2 被赋值之前正拥有另一个对象，赋值动作发生时会呼叫 delete，将该对象删除：

```
// initialize an auto_ptr with a new object
std::auto_ptr<ClassA> ptr1(new ClassA);
// initialize another auto_ptr with a new object
std::auto_ptr<ClassA> ptr2(new ClassA);

ptr2 = ptr1;          // assign the auto_ptr
                      // - delete object owned by ptr2
                      // - transfers ownership from ptr1 to ptr2
```

注意，拥有权的转移，意味实值并非只是被简单拷贝而已。只要发生了拥有权转移，先前的拥有者（本例为 ptr1）就失去了拥有权，结果，拥有者一旦交出拥有权，就两手空空，只剩一个 null 指标在手了。在这里，*copy* 建构式更动了「用以初始化新对象」的原对象，而赋值操作也修改了右侧对象，这和程序语言中惯常的初始化动作和赋值动作可说大相径庭。那么谁来保证那个「失去了所有权、只剩一个 null 指标」的原 auto_ptr 不会再次进行提领动作呢？是的，还是程序员的责任。

只有 auto_ptr 可以拿来当做另一个 auto_ptr 的初值，普通指标是不行的：

```
std::auto_ptr<ClassA>                   // create an auto_ptr
ptr;

                                        // ERROR
ptr = std::auto_ptr<ClassA>(new ClassA); // OK, delete old object
                                        // and own new
```

起点和终站（source and sink）

拥有权的移转，使得 auto_ptrs 产生一种特殊用法：某个函数可以利用 auto_ptr 将拥有权转交给另一个函数。这种事情可能在两种情形下出现：

1. 某函数是资料的终站。如果 auto_ptr 以 *by value*（传值）方式被当做一个引数传递给某函数，就是这种情况。此时被呼叫端的参数获得了这个 auto_ptr 的拥有权，如果函数不再将它传递出去，它所指的对象就会在函数退出时被删除：

```
void sink(std::auto_ptr<ClassA>);     // sink() gets ownership
```

2. 某函数是资料的起点。当一个 auto_ptr 被传回，其拥有权便被转交给呼叫端了。见下例：

```
std::auto_ptr<ClassA> f()
{
  std::auto_ptr<ClassA> ptr(new ClassA); //  ptr owns the new object
  ...
  return ptr; // transfer ownership to calling function
}

void g()
{
  std::auto_ptr<ClassA> p;

  for (int i=0; i<10; ++i) {
    p = f(); // p gets ownership of the returned object
             // (previously returned object of f() gets deleted)
    ...
  }
}    // last-owned object of p gets deleted
```

每当 f() 被呼叫，它都 new 一个新对象，然后把该对象连同其拥有权一起传回给呼叫端。将回返值指派（*assign*）给 p，同时也完成了拥有权的移转。一旦循环再次执行这个指派动作，p 原先拥有的对象将被删除。离开 g()时，p 也会被销毁，这样就删除了 p 所拥有的最后一个对象。无论如何都不会有资源遗失之虞。即使有异常被掷出，拥有数据的 auto_ptr 也会尽职地将自己的数据删除。

缺陷

auto_ptr 的语义本身就涵盖拥有权，所以如果你无意转交你的拥有权，就不要在参数列中使用 auto_ptr，也不要以它作为回返值。下面例子是一个幼稚的作法，原本是想将 auto_ptr 所指对象打印出来，实际上却引发一场灾难：

```
// this is a bad example
template <class T>
void bad_print(std::auto_ptr<T> p) //  p gets ownership of passed argument
{
```

```
    // does p own an object ?
    if (p.get() == NULL) {
       std::cout << "NULL";
    }
    else {
       std::cout << *p;
    }
}  // Oops, exiting deletes the object to which p refers
```

只要有一个 auto_ptr 被当做自变量，放进这个 bad_print()函数，它所拥有的物件（如果有的话）就一定会被删除。因为作为自变量的 auto_ptr 会将拥有权转交给参数 p，而当函数退出时，会删除 p 所拥有的对象。这恐怕不是程序员所希望的，最终必然会引起致命的执行期错误：

```
std::auto_ptr<int> p(new int);
*p = 42;          // change value to which p refers
bad_print(p)      // Oops, deletes the memory to which p refers
;                 // RUNTIME ERROR
*p = 18;
```

你可能会认为，将 auto_ptrs 以 *pass by reference* 方式传递就万事大吉。然而这
种行为却会使「拥有权」的概念变得难以捉摸，因为面对一个「透过 reference 而获得 auto_ptr」的函数，你根本无法预知拥有权是否被转交。所以以 *by reference* 方式传递 auto_ptr 是非常糟糕的设计，应该全力避免。

考虑到 auto_ptr 的概念，我们倒是可以运用 constant reference，向函数传递拥有权。然而这十分危险，因为当你传递一个 constant reference 时，通常预期该对象不会被更动。幸好 auto_ptrs 的一个晚期设计降低了此一危险性。藉由某些实作技巧，我们可以令 constant reference 无法交出拥有权。事实上，你无法变更任何 constant reference 的拥有权：

```
const std::auto_ptr<int> p(new int);
*p = 42;          // change value to which p refers
bad_print(p)      // COMPILE-TIME ERROR
;                 // OK
*p = 18;
```

这一方案使得 auto_ptrs 比以前显得更安全一些。很多接口在需要内部拷贝时，都藉由 constant reference 获得原值。事实上，C++ 标准链接库的所有容器（例见第 6章或第 10 章）都如此，大致像这样：

```
template <class T>
void container::insert (const T& value)
{
    ...
    x = value; // assign or copy value internally
    ...
}
```

如果这一类赋值动作对 auto_ptr 有效，那么拥有权就会被转交给容器。然而正由于 auto_ptrs 的实际设计，这种行为必然会导致编译错误：

```
container<std::auto_ptr<int> > c;
const std::auto_ptr<int> p(new int);
...
c.insert(p); // ERROR
...
```

总而言之，常数型 auto_ptr 减小了「不经意转移拥有权」所带来的危险。只要一个对象藉由 auto_ptr 传递，就可以使用常数型 auto_ptr 来终结拥有权移转链，此后拥有权将不能再进行移转。

在这里，关键词 const 并非意味你不能更改 auto_ptr 所拥有的对象，而是意味你不能更改 auto_ptr 的拥有权。例如：

```
std::auto_ptr<int> f()
{
  const std::auto_ptr<int> p(new int); //  no ownership transfer possible
  std::auto_ptr<int> q(new int); // ownership transfer possible

  *p = 42;        // OK, change value to which p refers
  bad_print(p)    // COMPILE-TIME ERROR
  ;               // OK, change value to which p refers
  *p = *q;        // COMPILE-TIME ERROR
  p = q;          // COMPILE-TIME ERROR
} return p;
```

如果使用 const auto_ptr 作为自变量，对新对象的任何赋值（*assign*）动作都将导致编译期错误。就常数特性而言，const auto_ptr 比较类似常数指标（T* const p），而非指向常数的指标（const T* p）— 尽管其语法看上去比较像后者。

4.2.3 auto_ptrs 作为成员之一

在 class 中使用 auto_ptrs，你可以因而避免遗失资源。如果你以 auto_ptr 而非一般指针作为成员，当对象被删除时，auto_ptr 会自动删除其所指的成员对象，于是你也就不再需要解构式了。此外，即使在初始化期间丢掷异常，auto_ptr 也可以帮助避免资源遗失。注意，只有当对象被完整建构成功，才有可能于将来呼叫其解构式。这造成了资源遗失的隐忧：如果第一个 new 成功了，第二个 new 却失败了，就会造成资源遗失。例如：

```
class ClassB {
  private:
    ClassA* ptr1; // pointer members
    ClassA* ptr2;
  public:
    // constructor that initializes the pointers
    // - will cause resource leak if second new throws
    ClassB (ClassA val1, ClassA val2)
      : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }

    // copy constructor
    // - might cause resource leak if second new throws
    ClassB (const ClassB& x)
      : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {
    }

    // assignment operator
    const ClassB& operator= (const ClassB& x) {
      *ptr1 = *x.ptr1;
      *ptr2 = *x.ptr2;
      return *this;
    }

    ~ClassB () {
      delete ptr1;
      delete ptr2;
    }
    ...
};
```

使用 auto_ptr，你就可以轻松避免这场悲剧，：

```
class ClassB {
  private:
    const std::auto_ptr<ClassA> ptr1; // auto_ptr members
    const std::auto_ptr<ClassA> ptr2;
  public:
    // constructor that initializes the auto_ptrs
    // - no resource leak possible
```

```
    ClassB (ClassA val1, ClassA val2)
      : ptr1(new ClassA(val1)), ptr2(new ClassA(val2)) {
    }

    // copy constructor
    // - no resource leak possible
    ClassB (const ClassB& x)
      : ptr1(new ClassA(*x.ptr1)), ptr2(new ClassA(*x.ptr2)) {
    }

    // assignment operator
    const ClassB& operator= (const ClassB& x) {
      *ptr1 = *x.ptr1;
      *ptr2 = *x.ptr2;
      return *this;
    }

    // no destructor necessary
    // (default destructor lets ptr1 and ptr2 delete their objects)
    ...
};
```

然而请注意，尽管你可以略过解构式，却还是不得不亲自撰写 *copy* 建构式和 *assignment* 运算符。预设状况下，这两个操作都会转交拥有权，这恐怕并非你所愿。正如 p42 所说，为了避免拥有权的意外转交，如果你的 `auto_ptr` 在整个生命期内都不必改变其所指对象的拥有权，你可以使用 `const auto_ptr`s。

4.2.4 auto_ptrs 的错误运用

`auto_ptr` 确实解决了一个特定问题，那就是在异常处理过程中的资源遗失问题。不幸的是由于 `auto_ptr` 的具体行为方式曾经三番五次地改动，而且 C++ 标准程式库中只此一个智能型指标（smart pointer），别无分号，所以人们总是会误用 `auto_ptr`s。为了帮助你正确使用它，这里给出一些要点：

1. `auto_ptr`s 之间不能共享拥有权

一个 `auto_ptr` 千万不能指向另一个 `auto_ptr`（或其他对象）所拥有的对象。否则，当第一个指针删除该对象后，另一个指标突然间指向了一个已被销毁的对象，那么，如果再透过那个指标进行读写操作，就会引发一场灾难。

2. 并不存在针对 array 而设计的 auto_ptrs

auto_ptr 不可以指向 array，因为 auto_ptr 是透过 delete 而非 delete[] 来释放其所拥有的对象。注意，C++ 标准链接库并未提供针对 array 而设计的 auto_ptr。标准链接库另提供了数个容器类别，用来管理数据群（参见第 5 章）。

3. auto_ptrs 决非一个「四海通用」的智能型指标

并非任何适用智能型指标的地方，都适用 auto_ptr。特别请注意的是，它不是参用计数（*reference counting*）型指标 — 这种指标保证，如果有一组智慧型指针指向同一个对象，那么若且唯若（if and only if）最后一个智能型指标被销毁时，该物件才会被销毁。

4. auto_ptrs 不满足 STL 容器对其元素的要求

auto_ptr 并不满足 STL 标准容器对于元素的最基本要求，因为在拷贝（*copy*）和赋值（*assign*）动作之后，原本的 auto_ptr 和新产生的 auto_ptr 并不相等。是的，拷贝和赋值之后，原本的 auto_ptr 会交出拥有权，而不是拷贝给新的 auto_ptr。因此请绝对不要将 auto_ptr 作为标准容器的元素。幸好语言和程式库的设计本身就可以防止这种误用，如果你的工作环境符合标准，这类误用应该无法通过编译。

不幸的是，某些时候，即使误用 auto_ptr，程序仍然能够顺利运作。就此点而言，使用一个非常数（nonconstant）auto_ptr，并不比使用一个一般指标更安全。如果你的误用行为没有导致全盘崩溃，你或许会暗自庆幸，而这其实是真正的不幸，因为你或许根本就没有意识到你已经犯了错误。关于「参用计数型（reference counting）智慧指标」的讨论请见 p135, 5.10.2 节，p222, 6.8 节有一份实作码。当我们有必要在不同容器之间共享元素时，这种指针非常有用。

4.2.5 auto_ptr 运用实例

下面第一个例子展示 auto_ptrs 移转拥有权的行为：

```
// util/autoptr1.cpp
#include <iostream>
#include <memory>
using namespace std;

/* define output operator for auto_ptr
* - print object value or NULL
*/
template <class T>
ostream& operator<< (ostream& strm, const auto_ptr<T>& p)
{
   // does p own an object ?
   if (p.get() == NULL) {
```

```
        strm << "NULL";     // NO: print NULL
    }
    else {
        strm << *p;         // YES: print the object
    }
    return strm;
}

int main()
{
    auto_ptr<int> p(new int(42));
    auto_ptr<int> q;

    cout << "after initialization:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;

    q = p;
    cout << "after assigning auto pointers:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;

    *q += 13;     // change value of the object q owns
    p = q;
    cout << "after change and reassignment:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
}
```

程序输出如下：

```
after initialization:
  p: 42
  q: NULL
after assigning auto pointers:
  p: NULL
  q: 42
after change and reassignment:
  p: 55
  q: NULL
```

注意，output 运算符的第二个参数是一个 const reference，所以并没有发生拥有权的移转。

正如我在 p40 所说，请时刻铭记于心，你不能以一般指标的赋值手法来初始化一个 auto_ptr：

```
std::auto_ptr<int> p(new int(42));      // OK
std::auto_ptr<int> p = new int(42);     // ERROR

p = std::auto_ptr<int>(new int(42));    // OK
p = new int(42);                        // ERROR
```

这是因为，「根据一般指标生成一个 auto_ptr」的那个建构式，被宣告为 explicit（关于 explicit，详见 p18, 2.2.6 节）。

下面这个例子展示 const auto_ptr 的特性：

```
// util/autoptr2.cpp

#include <iostream>
#include <memory>
using namespace std;

/* define output operator for auto_ptr
* - print object value or NULL
*/
template <class T>
ostream& operator<< (ostream& strm, const auto_ptr<T>& p)
{
  // does p own an object ?
  if (p.get() == NULL) {
    strm << "NULL";    // NO: print NULL
  }
  else {
    strm << *p;        // YES: print the object
  }
  return strm;
}

int main()
{
  const auto_ptr<int> p(new int(42));
  const auto_ptr<int> q(new int(0));
  const auto_ptr<int> r;
```

```
    cout << "after initialization:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
    cout << " r: " << r << endl;

    *q = *p;
// *r = *p;   // ERROR: undefined behavior
    *p = -77;
    cout << "after assigning values:" << endl;
    cout << " p: " << p << endl;
    cout << " q: " << q << endl;
    cout << " r: " << r << endl;

// q = p;     // ERROR at compile time
// r = p;     // ERROR at compile time
}
```

程序输出如下：

```
after initialization:
  p: 42
  q: 0
  r: NULL
after assigning values:
  p: -77
  q: 42
  r: NULL
```

这个例子为 auto_ptrs 定义了一个 output 运算符，其中将 auto_ptr 以 const reference 的方式传递。根据 p43 的讨论，你不应该以任何形式传递 auto_ptr，但此处是个例外。

注意下列赋值（指派）动作是错误的：

```
*r = *p;
```

这个句子对于一个「未指向任何对象」的 auto_ptr 进行提领（*dereference*）动作。C++ 标准规格宣称，这会导致未定义行为，比如说导致程序的崩溃。从这个例子可以看出，你可以操作 const auto_ptrs 所指对象本身，但「它所拥有的究竟是哪个对象」这一事实无法改变。就算 r 不具常数性，最后一个述句也会失败，因为 p 具有常数性，其拥有权不得被更改。

4.2.6 auto_ptr 实作细目

class auto_ptr 宣告于 <memory>：

```
#include <memory>
```

auto_ptr 定义于 namespace std 中，是「可用于任何型别身上」的一个 template class。下面是 auto_ptr 的确切宣告[6]：

```
namespace std {
  // auxiliary type to enable copies and assignments
  template <class Y> struct auto_ptr_ref {};

  template<class T>
  class auto_ptr {
   public:
     // type names for the value
     typedef T element_type;

     // constructor
     explicit auto_ptr(T* ptr = 0) throw();

     // copy constructors (with implicit conversion)
     // - note: nonconstant parameter
     auto_ptr(auto_ptr&) throw();
     template<class U> auto_ptr(auto_ptr<U>&) throw();

     // assignments (with implicit conversion)
     // - note: nonconstant parameter
     auto_ptr& operator= (auto_ptr&) throw();
     template<class U>
       auto_ptr& operator= (auto_ptr<U>&) throw();

     // destructor
     ~auto_ptr() throw();

     // value access
     T* get() const throw();
     T& operator*() const throw();
     T* operator->() const throw();
```

[6] 这里所给的版本较之 C++ 标准链接库的版本作了一些小小改进，修正了几个小问题（这里的 auto_ptr_ref 是全局的，这里并且设定了从 auto_ptr_ref 到 auto_ptr 的 *assignment* 运算符，参见 p55）。

```
    // release ownership
    T* release() throw();

    // reset value
    void reset(T* ptr = 0) throw();

  // special conversions to enable copies and assignments
  public:
    auto_ptr(auto_ptr_ref<T>) throw();
    auto_ptr& operator= (auto_ptr_ref<T> rhs) throw();
    template<class U> operator auto_ptr_ref<U>() throw();
    template<class U> operator auto_ptr<U>() throw();
 };
}
```

个别成员的详细描述将在后续数节中进行。讨论过程中我把 auto_ptr<T> 简写为 auto_ptr。p56 有一份完整的 auto_ptr 实作范例。

型别定义

auto_ptr::**element_type**

z auto_ptr 所拥有之物件的型别

建构式（constructor）、赋值运算符（assign operator）、解构式（destructor）

auto_ptr::**auto_ptr()** throw()

z *default* 建构式。

z 生成一个不拥有任何物件的 auto_ptr。

z 将 auto_ptr 的值初始化为零。

explicit *auto_ptr*::**auto_ptr**(T* *ptr*) throw()

z 生成一个 auto_ptr，并拥有 ptr 所指物件。

z 此一动作完成后，*this 成为 ptr 所指对象的唯一拥有者。不允许再有其他拥有者。

- 如果 ptr 本身不是 null 指标，那就必须是个 new 回返值，因为 auto_ptr 解构式会对其所拥有的对象自动呼叫 delete。
- 不能用 new[] 所生成的 array 作为初值。当你需要 array，请考虑使用 STL 容器，p75, 5.2 节对此有些介绍。

```
auto_ptr::auto_ptr(auto_ptr& ap) throw()
template<class U> auto_ptr::auto_ptr(auto_ptr<U>& ap) throw()
```

- 针对 non-const values 而设计的一个 *copy* 建构式。
- 生成一个 auto_ptr，在入口处将 ap 所拥有的对象（如果有的话）的拥有权夺取过来。
- 此操作完毕之后，ap 不再拥有任何对象，其值变为 null 指标。所以，和一般 *copy* 建构式不同，这个操作改变了原对象。
- 注意，此函数有一个多载化的 member template（请参考 p11），使得 ap 可藉由型别自动转换，构造出合适的 auto_ptr。例如，根据一个「衍生类别的物件」，建构出一个基础类别对象的 auto_ptr。
- 拥有权移转问题，请参考 p40, 4.2.2 节。

```
auto_ptr& auto_ptr::operator= (auto_ptr& ap) throw()
template<class U> auto_ptr& auto_ptr::operator= (auto_ptr<U>& ap) throw()
```

- 针对 non-const value 而设计的一个 *assignment*（赋值）运算符
- 如果自身原本拥有对象，进入本动作时将被删除，然后获得 ap 所拥有的对象。于是，原本 ap 所拥有的对象，其拥有权就移转给了 *this。
- 此一动作完成后，ap 不再拥有任何对象。其值变为 null 指标。与一般赋值动作不同，此处这个动作改变了原对象。
- 左手边的 auto_ptr 原本所指对象将被删除（deleted）。
- 注意，此函数有一个多载化的 member template（请参考 p11）。这使得 ap 可藉由「型别自动转换」指派给合适的 auto_ptr。例如，将一个「衍生类别的对象」，指派给一个基础类别对象的 auto_ptr。
- 拥有权移转问题，请参考 p40, 4.2.2 节。

```
auto_ptr::~auto_ptr() throw()
```

- 解构式
- 如果 auto ptr 拥有某个对象，此处将呼叫 delete 删除之。

数值存取（value access）

`T* auto_ptr::get() const throw()`

- 传回 auto_ptr 所指对象的地址。
- 如果 auto_ptr 未指向任何对象，传回 null 指针。
- 这个动作并不改变拥有权。退出此函数时，auto_ptr 仍然保有对对象（如果有的话）的拥有权。

`T& auto_ptr::operator*() const throw()`

- *dereference*（提领）运算符
- 传回 auto_ptr 所拥有的物件。
- 如果 auto_ptr 并未拥有任何对象，此呼叫导致未定义行为（可能导致崩溃）。

`T* auto_ptr::operator->() const throw()`

- *member access*（成员存取）运算符
- 传回 auto_ptr 所拥有的物件中的一个成员。
- 如果 auto ptr 并未拥有任何对象，此呼叫将导致未定义行为（可能导致崩溃）。

数值操作

`T* auto_ptr::release() throw()`

- 放弃 auto_ptr 原先所拥有之对象的拥有权。
- 传回 auto_ptr 原先拥有对象（如果有的话）的地址。
- 如果 auto_ptr 原先并未拥有任何对象，传回 null 指针。

`void auto_ptr::reset(T* ptr = 0) throw()`

- 以 ptr 重新初始化 auto_ptr。
- 如果 auto_ptr 原本拥有对象，则此动作开始前先删除之。
- 呼叫结束后，*this 成为 ptr 所指对象的拥有者。注意，不应该有任何其他拥有者。
- 如果 ptr 不是 null 指标，应当是一个由 new 传回的值，因为 auto_ptr 的解构式会呼叫 delete 来删除其所拥有的对象。
- 注意，不得将透过 new[] 生成的 array 当作自变量传进来。如果需要使用 array，请考虑使用 STL 容器类别，详见 p75, 5.2 节。

Conversions（转型操作）

auto_ptr 中剩余的内容（辅助型别 auto_ptr_ref 及其相关函数）涉及非常精致的技巧，使我们得以拷贝和指派 non-const auto_ptrs，却不能拷贝和指派 const auto_ptrs（详见 p44）。下面是一份扼要解释[7]。我们有两个需求：

1. 我们需要将 auto_ptr 作为右值（rvalue）传递到函数去，或由函数中传回[8]。由于 auto_ptr 是个类别，所以这些工作应当由建构式完成。
2. 拷贝 auto_ptr 时，原指针务必放弃拥有权。这就要求拷贝动作必须修改原本的那个 auto_ptr。

一般的 *copy* 建构式当然可以拷贝右值，但为了做到这点，它必须将其参数型别宣告为一个 *reference to const object*。如果在 auto_ptr 中使用一般的 *copy* 建构式，
我们恐怕不得不将 auto_ptr 内含的实际指标宣告为 mutable，只有这样，才能在 *copy* 建构式中更改它。你以为万事大吉了吗？错，这种做法将允许用户拷贝那些宣告为 const 的对象，将其拥有权转交他人，这与其原本的常数性背道而驰。

变通作法是找出一种机制，能够将右值转化为左值。「直接转型为 reference」的那种简单的转型运算符派不上用场，因为当你实际上是把一个对象转化为自己原本的型别时，不会有任何转型操作被唤起（切记，reference 属性并非型别的一部份）。为此才有了 auto_ptr_ref 类别的引进，协助我们将右值转化为左值。这一机制的理论基础是「多载化」和「template 自变量推导规则」之间一个细微的不同处。这个差别实在太细微了，不大可能成为一般程序编写技巧而用于别处，但却足以在这里让 auto_ptr 正确运作。这就够了。

如果你的编译程序对于 non-const 和 const auto_ptrs 之间的区别尚不能做出很好的
阐释，请不必惊讶。但是请你保持清醒的头脑，如果你的编译程序尚未达到这一水平，那么 auto_ptr 的使用就会变得更加危险。因为这种情况下很容易意外地将拥有权旁落他人之手。

[7] 感谢 Bill Gibbons 指出这一点。

[8] rvalue（右值）和 lvalue（左值）的名称由来，是从赋值运算 expr1 = expr2 得来。在这种表达式中，左操作数 expr1 必须是一个（可更改的）lvalue。不过或许更贴切的
描述是：lvalue 代表 *l*ocator *value*。也就是说，这个算式藉由名字和参考值（pointer 或 reference）来指定一个对象。lvalue 并非一定「可被更改」。例如常数对象的名字就是一个不可被改动的 lvalue。所有 non-lvalues 对象，都是 rvalues。尤其显式生成 (T()) 的暂时对象和函数回返值，都是 rvalue。

类别 auto_ptr 的实作范例

以下源码展示了一个符合标准的 auto_ptr 类别的实作示范[9]：

```
// util/autoptr.hpp

/* class auto_ptr
* - improved standard conforming implementation
*/
namespace std {
  // auxiliary type to enable copies and assignments (now global)
  template<class Y>
  struct auto_ptr_ref {
    Y* yp;
    auto_ptr_ref (Y* rhs)
      : yp(rhs) {
    }
  };

  template<class T>
  class auto_ptr {
    private:
      T* ap; // refers to the actual owned object (if any)
    public:
      typedef T element_type;

      // constructor
      explicit auto_ptr (T* ptr = 0) throw()
        : ap(ptr) {
      }

      // copy constructors (with implicit conversion)
      // - note: nonconstant parameter
      auto_ptr (auto_ptr& rhs) throw()
        : ap(rhs.release()) {
      }
```

[9] 感谢 Greg Colvin 提供的这份 auto_ptr 实作内容。注意，这个实作版本并不完全合乎C++ 标准规范。事实证明，C++ 标准所规定的形式中，当利用 auto_ptr_ref 进行转型时，在某种特殊情况下仍会出现小小瑕疵。这里所给的方案很有可能彻底解决所有问题。不过，撰写本书的时候，仍有一些相关讨论正在进行。

```
template<class Y>
auto_ptr (auto_ptr<Y>& rhs) throw()
  : ap(rhs.release()) {
}

// assignments (with implicit conversion)
// - note: nonconstant parameter
auto_ptr& operator= (auto_ptr& rhs) throw() {
  reset(rhs.release());
  return *this;
}
template<class Y>
auto_ptr& operator= (auto_ptr<Y>& rhs) throw() {
  reset(rhs.release());
  return *this;
}

// destructor
~auto_ptr() throw() {
  delete ap;
}

// value access
T* get() const throw() {
  return ap;
}
T& operator*() const throw() {
  return *ap;
}
T* operator->() const throw() {
  return ap;
}

// release ownership
T* release() throw() {
  T* tmp(ap);
  ap = 0;
  return tmp;
}
```

```
    // reset value
    void reset (T* ptr=0) throw() {
      if (ap != ptr) {
        delete ap;
        ap = ptr;
      }
    }

    /* special conversions with auxiliary type to enable copies
    and
    assignments
    */
    auto_ptr(auto_ptr_ref<T> rhs) throw()
      : ap(rhs.yp) {
    }
    auto_ptr& operator= (auto_ptr_ref<T> rhs) throw() { // new
      reset(rhs.yp);
      return *this;
    }
    template<class Y>
    operator auto_ptr_ref<Y>() throw() {
      return auto_ptr_ref<Y>(release());
    }
    template<class Y>
    operator auto_ptr<Y>() throw() {
   }  return auto_ptr<Y>(release());
  };
}
```

4.3 数值极限（Numeric Limits）

一般说来，值类型（Numeric types）的极值是一个与平台相依的特性。C++ 标准链接库藉由 template `numeric_limits` 提供这些极值，取代传统 C 语言所采用的预处理器常数（preprocessor constants）。你仍然可以使用后者，其中整数常数定义于 `<climits>` 和 `<limits.h>`，浮点常数定义于 `<cfloat>` 和 `<float.h>`。新的极值概念有两个优点，第一是提供了更好的型别安全性，第二是程序员可藉此写出一些 templates 以核定（*evaluate*）这些极值。

本节的剩余部份专门讨论极值问题。注意，*C++ Standard* 规定了各种型别必须保证的最小精度，如果你能够注意并运用这些极值，就比较容易写出与平台无关的程式。这些最小值列于表 4.1。

型别	最小长度
`char`	1 byte (8 bits)
`short int`	2 bytes
`int`	2 bytes
`long int`	4 bytes
`float`	4 bytes
`double`	8 bytes
`long double`	8 bytes

表 4.1 内建型别的最小长度

Class `numeric_limits<>`

使用 template，通常是为了对所有型别一次性地撰写出一个通用解决方案。除此之外，你还可以在必要时候以 template 为每个型别提供共同接口。方法是：不但提供通用性的 template，还提供其特化（specialization）版本。`numeric_limits` 就是这项技术的一个典型例子，作法如下：

- 通用性的 template，为所有型别提供预设极值：

```
namespace std {
   /* general numeric limits as default for any type
   */
   template <class T>
   class numeric_limits {
      public:
      // no specialization for numeric limits exist
      static const bool is_specialized = false;
      ... //  other members that are meaningless for the general numeric limits
   };
}
```

这个通用性 template 将成员 is_specialized 设为 false，意思是，对型别 T 而言，无所谓极值的存在。

z 各具体型别的极值，由特化版本（specialization）提供：

```
namespace std {
    /* numeric limits for int
    * - implementation defined
    */
    template<> class numeric_limits<int> {
    public:
        // yes, a specialization for numeric limits of int does exist
        static const bool is_specialized = true;

        static T min() throw() {
            return -2147483648;
        }
        static T max() throw() {
            return 2147483647;
        }
        static const int digits = 31;
        ...
    };
}
```

这里把 is_specialized 设为 true，所有其他成员都根据特定型别的具体极值加以设定。

通用性的 numeric_limits template，及其特化版本都被放在<limits>头文件中。*C++ Standard* 所囊括的特化版本，涵盖了所有数值基本型别，包括：bool, char, signed char, unsigned char, wchar_t, short, unsigned short, int, unsigned int, long, unsigned long, float, double, long double。你可
以轻易为你自定的值类型加上补充。

表 4.2 和表 4.3 列出 class numeric_limits<> 的所有成员及其意义。最右一列显示对应的 C 常数，它们分别定义于 <climits>, <limits.h>, <cfloat>, <float.h> 内。

成员	意义	对应的 C 常数
is_specialized	型别是否有极值	
is_signed	型别带有正负号	
is_integer	整数型别	
is_exact	计算结果不产生舍/入误差（此成员对所有整数型别而言均为 true）	
is_bounded	数值集的个数有限（对所有内建型别而言，此成员均为 true）	
is_modulo	两正值相加，其结果可能因溢位而	
is_iec559	最小值（对浮点数而言，是标准化后	
min()	的最小负值）（只有当 is_bounded 成立时才有意义）	CHAR_MIN,...
max()	最大值（只有当 is_bounded 成立时才有意义）	INT_MAX, FLT_MAX,...
digits	字符和整数：不带正负号之位个数 浮点数：尾数中之 radix（见下）位	CHAR_BIT FLT_MANT_DIG,...
digits10	十进制数的个数（只有当 is_bounded 成立时才有意义）	FLT_DIG,...
radix	整数：表示式的基底（base）， 浮点数：指数表示式的基底（base）	 FLT_RADIX
min_exponent	基底 radix 的最小负整数指数	FLT_MIN_EXP,...
max_exponent	基底 radix 的最大正整数指数	FLT_MAX_EXP,...
min_exponent10	基底 10 的最小负整数指数	FLT_MIN_10_EXP,...
max_exponent10	基底 10 的最大正整数指数	FLT_MAX_10_EXP,...
epsilon()	1 和最接近 1 的值之间的差距	FLT_EPSILON,...
round_style	舍 /入（rounding）风格（见 p63）	
round_error()	最大舍/入误差量测（根据 ISO/IEC 10967-1 标准）	
has_infinity	有「正无穷大」表示式	
infinity()	表现出「正无穷大」（如有的话）	
has_quiet_NaN	本型别有不发讯号（nonsignaling）的「非数值」表述式。	
quiet_NaN()	如果可以，安静地（nonsignaling）表述出「这不是个数值」。	
has_signaling_NaN	本型别有会发出讯号（signaling）的	
signaling_NaN()	如果可以，发出讯号地（signaling）表述出「这不是个数值」。	
has_denorm	本型别是否允许非标准化数值（denormalized values，也就是 variable members of exponent bits，见 p63）	

has_denorm_loss	准确度的遗失是以一个非标准化值（denormalization）而不是以一个不精密的结果被侦测出来。
denorm_min()	最小的非标准化（denormalized）正值。
traps	已实作出 Trapping
tinyness_before	在舍/入（rounding）之前可侦测出 tinyness

表 4.2. class numeric_limits<> 的所有成员

译注 1：本表格「意义」栏中的诸多解释，涉及数字表示法专业术语。译者这方面能力有限，没把握正确译出。对于没有把握的名词，皆保留英文，望谅。

译注 2：本表格在英文版中因分页关系而被切割为表 4.2 和表 4.3。此处合而为一。后续表格从表 4.4 开始继续编号。

下面是对于 float 型别的数值限定模板特殊化的一个完全实作版，当然是和平台相依的。这里同时还给出了各成员的确切标记（signatures）：

```
namespace std {
  template<> class numeric_limits<float> {
    public:
      // yes, a specialization for numeric limits of float does exist
      static const bool is_specialized = true;

      inline static float min() throw() {
        return 1.17549435E-38F;
      }
      inline static float max() throw() {
        return 3.40282347E+38F;
      }

      static const int digits = 24;
      static const int digits10 = 6;

      static const bool is_signed = true;
      static const bool is_integer = false;
      static const bool is_exact = false;
      static const bool is_bounded = true;
      static const bool is_modulo = false;
      static const bool is_iec559 = true;

      static const int radix = 2;

      inline static float epsilon() throw() {
```

```
      return 1.19209290E-07F;
    }

    static const float_round_style round_style
      = round_to_nearest;
    inline static float round_error() throw() {
      return 0.5F;
    }

    static const int min_exponent = -125;
    static const int max_exponent = +128;
    static const int min_exponent10 = -37;
    static const int max_exponent10 = +38;

    static const bool has_infinity = true;
    inline static float infinity() throw() { return ...; }
    static const bool has_quiet_NaN = true;
    inline static float quiet_NaN() throw() { return ...; }
    static const bool has_signaling_NaN = true;
    inline static float signaling_NaN() throw() { return ...; }
    static const float_denorm_style has_denorm = denorm_absent;
    static const bool has_denorm_loss = false;
    inline static float denorm_min() throw() { return min(); }

    static const bool traps = true;
    static const bool tinyness_before = true;
  };
}
```

注意，所有数据成员如果不是 const，便是 static，这么一来其值便可在编译期间确定。至于由函数所定义的成员，在某些编译程序中恐怕无法在编译期间确定其值。因此，同一份目标码（object code）虽然可以在不同的处理器上执行，可能会得出不同的浮点值。

round_style 的值列于表 4.4，has_denorm 的值列于表 4.5。has_denorm 其实也
许应该称为 denorm_style 更贴切，可惜并非如此。这是因为 C++ 标准化后期才决定将其原本的 bool 型别改变为列举值（enumerative value）之故。不过你还是可以把 has_denorm 当成 bool 值来用；*C++ Standard* 保证，如果 denorm_absent 为 0，就等于 false，如果 denorm_present 为 1 而且 denorm_indeterminate 为 -1，那么两者都等于 true。因此你可以把 has_denorm 视为一个 bool 值，用以判定某个型别是否允许所谓的 "denormalized values"。

舍 /入（round）风格	意义
round_toward_zero	向零舍/入
round_to_nearest	向最接近的可表示值舍/入
round_toward_infinity	向正无限值舍/入
round_toward_neg_infinity	向负无限值舍/入
round indeterminate	无法确定

表 4.4 numeric_limits<> 的舍入（round）风格

舍入风格	意义
denorm_absent	此型别不允许 "denormalized values"
denorm_present	此型别允许向最接近的可表示值做 denormalized values
denorm_indeterminate	无法确定

表 4.5 numeric_limits<> 的 "denormalization style"

numeric_limits<> 使用范例

下面的例子展示某些型别极值的可能运用，例如用来了解某个型别的最大值，或确定 char 是否带正负号：

```
// util/limits1.cpp

#include <iostream>
#include <limits>
#include <string>
using namespace std;
int main()
{
  // use textual representation for bool
  cout << boolalpha;

  // print maximum of integral types
  cout << "max(short): " << numeric_limits<short>::max() << endl;
  cout << "max(int): " << numeric_limits<int>::max() << endl;
  cout << "max(long): " << numeric_limits<long>::max() << endl;
  cout << endl;
```

```
    // print maximum of floating-point types
    cout << "max(float): "
         << numeric_limits<float>::max() << endl;
    cout << "max(double): "
         << numeric_limits<double>::max() << endl;
    cout << "max(long double): "
         << numeric_limits<long double>::max() << endl;
    cout << endl;

    // print whether char is signed
    cout << "is_signed(char): "
         << numeric_limits<char>::is_signed << endl;
    cout << endl;

    // print whether numeric limits for type string exist
    cout << "is_specialized(string): "
         << numeric_limits<string>::is_specialized << endl;
}
```

程序的输出结果和执行平台有关，下面是其中一种可能：

```
max(short): 32767
max(int): 2147483647
max(long): 2147483647

max(float): 3.40282e+38
max(double): 1.79769e+308
max(long double): 1.79769e+308

is_signed(char): false

is_specialized(string): false
```

最后一行表示，型别 `string` 并没有定义数值极限。这是理所当然的，因为 `strings` 并非值类型。正如本例所示，你可以对任何型别进行询问，无论它是否定义了极值。

4.4 辅助函数

算法链接库（定义于头文件 <algorithm>）内含三个辅助函数，一个用来在两值之中挑选较大者，另一个用来在两值之中挑选较小者，第三个用来交换两值。

4.4.1 挑选较小值和较大值

「在两物之间选择较大值和较小值」的函数，定义于 <algorithm>，如下所示：

```
namespace std {
  template <class T>
  inline const T& min (const T& a, const T& b) {
    return b < a ? b : a;
  }

  template <class T>
  inline const T& max (const T& a, const T& b) {
    return a < b ? b : a;
  }
}
```

如果两值相等，通常会传回第一值。不过你的程序最好不要依赖这一点。

上述两个函数还有另一个版本，接受一个额外的 template 自变量作为「比较准则」：

```
namespace std {
  template <class T, class Compare>
  inline const T& min (const T& a, const T& b, Compare comp) {
    return comp(b,a) ? b : a;
  }

  template <class T, class Compare>
  inline const T& max (const T& a, const T& b, Compare comp) {
    return comp(a,b) ? b : a;
  }
}
```

作为「比较准则」的那个自变量应该是个函数或仿函数（functor，将于 5.9 节， p124 介绍），接受两个自变量并进行比较：在某个指定规则下，判断第一自变量是否小于第二自变量，并传回判断结果。

下面这个例子示范如何传入特定的比较函数作为自变量，以此方式来运用 max()：

```
// util/minmax1.cpp

#include <algorithm>
using namespace std;

/* function that compares two pointers by comparing the values to
which
they point
*/
bool int_ptr_less (int* a, int* b)
{
  return *a < *b;
}

int main()
{
  int x = 17;
  int y = 42;
  int* px = &x;
  int* py = &y;
  int* pmax;

  // call max() with special comparison function
  pmax = max (px, py, int_ptr_less);
  ...
}
```

注意，min()和 max()都要求它们所接受的两个自变量的型别必须一致。如果不一致，你将无法正确呼叫之：

```
int i;
long l;
...
l = std::max(i,l); // ERROR: argument types don't match
```

不过你倒是可以明白地宣告自变量型别（这样也就确定了回返值的型别）：

```
l = std::max<long>(i,l);   // OK
```

4.4.2 两值互换

函数 swap()用来交换两对象的值。其泛用型实作码定义于<algorithm>：

```
namespace std {
   template<class T>
   inline void swap(T& a, T& b) {
      T tmp(a);
      a = b;
      b = tmp;
   }
}
```

运用这个函数，你可以如此交换任意变量 x 和 y 的值：

```
std::swap(x,y);
```

当然啦，只有当 swap()所依赖的 *copy* 建构动作和 *assignment* 操作行为存在时，这个呼叫才可能有效。

swap()的最大优势在于，透过 template specialization（模板特化）或 function overloading（函数多载化），我们可以为更复杂的型别提供特殊的实作版本；我们可以交换对象内部成员，不必劳师动众地反复指派（赋值），这无疑将大大地节约时间。标准链接库中的所有容器（6.1.2 节, p147）以及 strings（11.2.8 节, p490）都运用了这项技术。举个例子，有个简单容器，仅仅内含一个 array 和一个成员（用来指示 array 元素个数量），那么为它特别实作的 swap()可以是这样：

```
class MyContainer {
   private:
     int* elems;      // dynamic array of elements
     int numElems;    // number of elements
   public:
     ...
     // implementation of swap()
     void swap(MyContainer& x) {
       std::swap(elems,x.elems);
       std::swap(numElems,x.numElems);
     }
     ...
};

// overloaded global swap() for this type
inline void swap (MyContainer& c1, MyContainer& c2)
{
   c1.swap(c2); // calls implementation of swap()
}
```

你瞧，呼叫 swap() 而非透过反复指派（赋值）动作来交换两容器的值，会带来效率上的提升。对于你自己定义的型别，如果确实能够带来效率上的改善，你就应该义不容辞地为它提供 swap() 特化版本。

4.5 辅助性的「比较运算符」（Comparison Operators）

有四个 template functions，分别定义了 !=, >, <=, >= 四个比较运算符。它们都是利用运算符 == 和 < 完成的。这四个函数定义于 <utility>：

```
namespace std {
   namespace rel_ops {
      template <class T>
      inline bool operator!= (const T& x, const T& y) {
         return !(x == y);
      }

      template <class T>
      inline bool operator> (const T& x, const T& y) {
         return y < x;
      }

      template <class T>
      inline bool operator<= (const T& x, const T& y) {
         return !(y < x);
      }

      template <class T>
      inline bool operator>= (const T& x, const T& y) {
         return !(x < y);
      }
   }
}
```

只需定义 < 和 == 运算符，你就可以使用它们。只要加上 using namespace std::rel_ops，上述四个比较运算符就自动获得了定义。例如：

```
#include <utility>

class X {
   ...
   public:
```

```
        bool operator== (const X& x) const;
        bool operator< (const X& x) const;
        ...
};

void foo()
{
    using namespace std::rel_ops; // make !=, >, etc., available
    X x1, x2;
    ...

    if (x1 != x2) {
        ...
    }
    ...

    if (x1 > x2) {
        ...
    }
    ...
}
```

注意，这些运算符都定义于 std 的次命名空间（sub-namespace）rel_ops 中。之所以如此安排，是为了防止和用户（可能）定义的全局命名空间中的同类形运算子发生冲突。于是，就算你这样使用 using directive：

```
using namespace std;  // operators are not in global scope
```

因而把 std 的全部标识符引入全局命名空间，也没问题。

另一方面，那些想向 rel_ops 借一臂之力的用户可以这么做：

```
using namespace std::rel_ops; // operators are in global scope
```

于是四 个新的运算符就轻松到手了，无需使用复杂的搜寻规则来引用它们。

某些实作版本采用两个不同的自变量型别来定义上述 template：

```
namespace std {
    template <class T1, class T2>
    inline bool operator!=(const T1& x, const T2& y) {
        return !(x == y);
    }
    ...
}
```

这么做的好处是，两个操作数的型别可以不同（只要它们之间「可以比较」就行）。但这并非 C++ 标准链接库所支持的作法。所以，如果想占这个便宜，就得付出可携性方面的代价。

4.6 头文件 <cstddef> 和 <cstdlib>

头文件 <cstddef> 和 <cstdlib> 和其 C 对应版本兼容，在 C++ 程序中经常用到。它们是 C 头文件 <stddef.h> 和 <stdlib.h> 的较新版本，定义了一些常用的常数、宏、型别和函数。

4.6.1 <cstddef> 内的各种定义

表 4.6 列出头文件 <cstddef> 的各个定义项。NULL 通常用来表明一个不指向任何对象的指针，其实就是 0（其型别可以是 int，也可以是 long）。注意，C 语言中的 NULL 通常定义为(void*)0。在 C++ 中这并不正确，NULL 的型别必须是个整数型别，否则你无法将 NULL 指派给一个指标。这是因为 C++ 并没有定义从 void* 到任何其他型别的自动转型操作[10]。NULL 同时也定义于头文件 <cstdio>, <cstdlib>, <cstring>, <ctime>, <cwchar>, <clocale> 内。

标识符	意义
NULL	指标值，用来表示「未定义」或「无值」。
size_t	一种无正负号的型别，用来表示大小（例如元素个数）。
ptrdiff_t	一种带有正负号的型别，用来表示指标之间的距离。
offsetof	表示一个成员在 struct 或 union 中的偏移量。

表 4.6 <cstddef> 中的定义项

4.6.2 <cstdlib> 内的各种定义

表 4.7 列出头文件 <cstdlib> 内最重要的一些定义。常数 EXIT_SUCCESS 和 EXIT_FAILURE 用来当做 exit() 的自变量，也可以当做 main()的回返值。

经由 atexit()登录的函数，在程序正常退出时会依登录的相反次序被一一呼叫起来。无论是透过 exit()退出或从 main()尾部退出，都会如此，不传递任何自变量。

[10] 鉴于 NULL 型别有这些晦涩的问题，有人建议 C++ 程序中最好不要使用 NULL，好直接使用 0 或用户自定的（例如）NIL 常数。不过我还是使用 NULL，所以本书范例程序中你还是可以看到它的踪迹。

定义	意义
`exit(int status)`	退出（离开，exit）程序（并清理 static 对象）
`EXIT_SUCCESS`	程序正常结束。
`EXIT_FAILURE`	程序不正常结束。
`abort()`	退出程序（在某些系统上可能导致崩溃）。
`atexit (void (*function)())`	退出（exit）程序时呼叫某些函数。

表 4.7 <cstdlib> 中的定义项

函数 `exit()`和 `abort()`可用来在任意地点终止程序运行，无需返回 `main()`：

- `exit()` 会销毁所有 static 对象，将所有缓冲区（buffer）清空（flushes），关闭所有 I/O 信道（channels），然后终止程序（之前会先呼叫经由 `atexit()`登录的函数）。如果 `atexit()`登录的函数掷出异常，就会唤起 `terminate()`。
- `abort()`会立刻终止函数，不做任何清理（clean up）工作。

这两个函数都不会销毁区域对象（local objects），因为堆栈辗转开展动作（stack unwinding）不会被执行起来。为确保所有区域对象的解构式获得呼叫，你应该运用异常（exceptions）或正常回返机制，然后再由 `main()`离开。

5

Standard

Template Library

标准模板库

STL（标准模板库）是 C++ 标准链接库的核心，它深刻影响了标准链接库的整体结构。STL 是一个泛型（generic）链接库，提供一系列软件方案，利用先进、高效的算法来管理数据。程序员无需了解 STL 的原理，便可享受数据结构和演算法领域中的这一革新成果。从程序员的角度看来，STL 是由一些可适应不同需求的群集类别（collection classes），和一些能够在这些数据群集上运作的算法构成。STL 内的所有组件都由 templates（模板）构成，所以其元素可以是任意型别。更妙的是，STL 建立了一个架构，在此架构下，你可以提供其他群集类别或演算法，与现有的组件搭配，共同运作。总之，STL 赋予 C++ 新的抽象层次。把 dynamic arrays（动态数组）、linked list（串行）、binary trees（二元树）之类的东西抛开吧，也不用再操心不同的搜寻算法了。你只需使用恰当的群集类别，然后呼叫其成员函数和（或）算法来处理数据，就万事大吉。当然，如此的灵活性并非免费午餐，代价总是有的。首要的一点是，STL 并不好懂。也正因为如此，本书倾注了好几章篇幅，为你讲解 STL 的内容。这一章介绍 STL 的总体概念，探讨其使用技术。第一个范例展示如何使用 STL，以及运用过程中有何考虑。第 6 章至第 9 章详细讨论 STL 的各个组件（包括容器 containers、迭代器 iterators、仿函数
functors、算法 algorithms），并提供更多范例。

5.1 STL 组件（STL Components）

若干精心勾画的组件共同合作，构筑起 STL 的基础。这些组件中最关键的是容器、迭代器和算法。

- 容器 **Containers**，用来管理某类对象的集合。每一种容器都有其优点和缺点，所以，为了应付程序中的不同需求，STL 准备了不同的容器类型。容器可以是 arrays 或是 linked lists，或者每个元素有一个特别的键值（key）。
- 迭代器 **Iterators**，用来在一个对象群集（collection of objects）的元素上进行巡

访动作。这个对象群集或许是个容器，或许是容器的一部分。迭代器的主要好处是，为所有容器提供了一组很小的公共接口。利用这个接口，某个操作（operations）就可以行进至群集内的下一个元素。至于如何做到，当然取决于群集的内部结构。不论这个群集是 array 或 tree，此一行进动作都能成功。为什么？因为每一种容器都提供了自己的迭代器，而这些迭代器了解该种容器的内部结构，所以能够知道如何正确行进。

迭代器的界面和一般指针差不多，以 `operator++` 累进，以 `operator*` 提领所指之值。所以，你可以把迭代器视为一种 *smart pointer*，能够把「前进至下一个元素」的意图转换成合适的操作。

z 算法 **Algorithms**，用来处理群集内的元素。它们可以出于不同的目的而搜寻、排序、修改、使用那些元素。透过迭代器的协助，我们只需撰写一次算法，就可以将它应用于任意容器之上，这是因为所有容器的迭代器都提供一致的介面。你还可以提供一些特殊的辅助性函数供算法呼叫，从而获取更佳的灵活性。这样你就可以一方面运用标准算法，一方面适应自己特殊或复杂的需求。例如，你可以提供自己的搜寻准则或元素合并时的特殊操作。

STL 的基本观念就是将数据和操作分离。数据由容器类别加以管理，操作则由可定制（configurable）的算法定义之。迭代器在两者之间充当粘合剂，使任何演算法都可以和任何容器交互运作（图 5-1）。

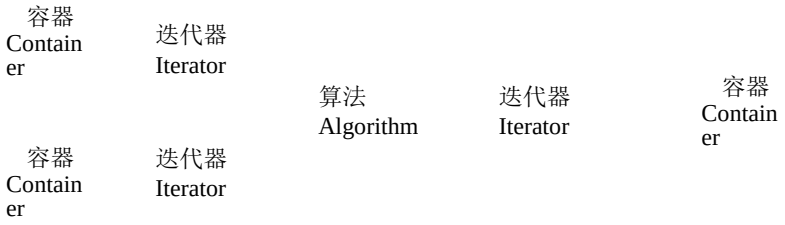

图 5.1 STL 组件之间的合作

STL 将数据和算法分开对待，而不是合并考虑。因此从某种意义上说，STL 的概念和面向对象程序编写（OOP）的最初思想是矛盾的。然而这么做有着很重要的原因。首先，你可以将各种容器与各种算法结合起来，在很小的框架（framework）内达成非常大的弹性。

STL 的一个根本特性是，所有组件都可以针对任意型别（types）运作。顾名思义，所谓 standard **template** libary 意味其内的所有组件都是「可接受任意型别」的 templates，前提是这些型别必须能够执行必要操作。因此 STL 成了泛型编程（*generic*

programming）概念下的一个出色范例。容器和算法对任意型别（types）和类别（classes）而言，都已经被一般化了。

STL 甚至提供更泛型化的组件。藉由特定的配接器（*adapters*）和仿函数（*functors*，或称 *function objects*），你可以补充、约束或订制算法，以满足特别需求。目前说这些似乎为时太早，眼下我还是先透过实例，循序渐进地讲解概念，这才是理解并熟悉 STL 的最佳方法。

5.2 容器（Containers）

容器类别（简称容器）用来管理一组元素。为了适应不同需要，STL 提供了不同的容器，如图 5.2。

图 5.2 STL 的容器种类

总的来说，容器可分为两类：

1. 序列式容器 **Sequence containers**，此乃可序（*ordered*）群集，其中每个元素均有固定位置 — 取决于插入时机和地点，和元素值无关。如果你以尾附方式对一个群集置入六个元素，它们的排列次序将和置入次序一致。STL 提供三个定义好的序列式容器：`vector`, `deque`, `list`。
2. 关系型容器 **Associative containers**，此乃已序（*sorted*）群集，元素位置取决于特定的排序准则。如果你将六个元素置入这样的群集中，它们的位置取决于元素值，和插入次序无关。STL 提供了四个关系型容器：`set`, `multiset`, `map`, `multimap`。

关系型容器也可被视为特殊的序列式容器，因为已序（*sorted*）群集正是根据某个排序准则排列（*ordered*）而成。如果你曾经用过其他群集库，例如 Smalltalk 和

NIHCL [1] 所提供者，你可能已经估计到这一点。在那些链接库中，*sorted collections* 由 *ordered collections* 衍生而来。不过请注意，STL 所提供的群集型别（collection types）彼此独立，各自实现，毫无关联（译注：意指其间并无 classes 继承关系）。

关系型容器自动对其元素排序，这并不意味它们就是用来排序的。你也可以对序列式容器的元素加以手动排序。自动排序带来的主要优点是，当你搜寻元素时，可获得更佳效率。更明确地说你可以放心使用二分搜寻法（binary search）。该演算法具有对数（logarithmic）复杂度，而非线性复杂度。这什么意思呢？如果你想在 1000 个元素中搜寻某个元素，平均而言只需 10 次比较，而非 500 次比较（参见 2.3 节, p21）。因此自动排序只不过是关系型容器的一个（有用的）副作用而已。

下面各小节详细讨论各种容器类别。其中特别讲解了容器的典型实作法。严格说来，*C++ Standard* 并未定义某一种容器的具体实作法。然而 *C++ Standard* 所规定的行为和其对复杂度的要求，让实作者没有太多变化余地。所以实际上各个实作版本之间只在细节上有所差异。第 6 章会谈到容器类别的确切行为、描述它们共有和特有的能力、并详细分析其成员函数。

5.2.1 序列式容器（Sequence Containers）

STL 内部预先定义好以下三个序列式容器：

- Vectors
- Deques
- Lists

此外你也可以将 strings 和 array 当作一种序列式容器。

Vectors

Vector 将其元素置于一个 dynamic array 中加以管理。它允许随机存取，也就是说你可以利用索引直接存取任何一个元素。在 array 尾部附加元素或移除元素均非常快速 [2]，但是在 array 中部或头部安插元素就比较费时，因为，为了保持原本的相对次序，安插点之后的所有元素都必须移动，挪出位子来。

[1] The National Institute of Health's Class Library，是最早的 C++ 类别库之一。

[2] 严格说来，元素尾附动作是一种「分摊后的（*amortized*）」高速。单一附加动作可能是缓慢的，因为 vector 可能需要重新分配内存，并将现有元素拷贝到新位置。不过这种事情不常发生，所以总体看来这个操作十分迅速。见 p22 的复杂度讨论。

以下例子针对整数型别定义了一个 vector，插入 6 个元素，然后打印所有元素：

```
// stl/vector1.cpp

#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector<int> coll;    // vector container for integer elements

  // append elements with values 1 to 6
  for (int i=1; i<=6; ++i) {
     coll.push_back(i);
  }

  // print all elements followed by a space
  for (int i=0; i<coll.size(); ++i) {
     cout << coll[i] << ' ';
  }
  cout << endl;
}
```

其中的：

```
#include <vector>
```

含入 vectors 的头文件。

以下宣告式：

```
vector<int> coll;
```

生成一个「元素型别为 int」的 vector。由于没有任何初始化参数，*default* 建构式就将它建构为空群集。

push_back()函数可为容器附加元素:

```
coll.push_back(i);
```

所有序列式容器都提供有这个成员函数。

size()成员函数回传容器中的元素个数：

```
for (int i=0; i<coll.size(); ++i) {
   ...
}
```

所有容器类别都提供有这个函数。

你可以通过 *subscript*（下标）运算符[]，存取 vector 内的某个元素：

```
cout << coll[i] << ' ';
```

在这里，元素被写至标准输出装置，所以整个程序的输出是：

```
1 2 3 4 5 6
```

Deques

所谓 deque（发音类似 "check[3]"），是 "double-ended queue" 的缩写。它是一个 dynamic array，可以向两端发展，因此不论在尾部或头部安插元素都十分迅速。在中间部份安插元素则比较费时，因为必须移动其他元素。

以下例子宣告了一个浮点数型别的 deque，并在容器头部安插 1.1 至 6.6 共 6 个元素，最后打印出所有元素。

```
// stl/deque1.cpp

#include <iostream>
#include <deque>
using namespace std;

int main()
{
  deque<float> coll; // deque container for floating-point elements

  // insert elements from 1.1 to 6.6 each at the front
  for (int i=1; i<=6; ++i) {
     coll.push_front(i*1.1); // insert at the front
  }

  // print all elements followed by a space
  for (int i=0; i<coll.size(); ++i) {
     cout << coll[i] << ' ';
  }
  cout << endl;
}
```

[3] 有时候 "deque" 听起来颇为类似 "hack"，不过这纯属巧合 -

本例之中，

```
#include <deque>
```

含入 deques 的头文件。

下面这一句：

```
deque<float> coll;
```

会产生一个空的浮点数群集。

`push_front()`函数可以用来安插元素：

```
coll.push_front(i*1.1);
```

它会将元素安插于群集前端。注意，这种安插方式造成的结果是，元素排放次序与安插次序恰好相反，因为每个元素都安插于上一个元素的前面。因此，程序输出如下：

```
6.6 5.5 4.4 3.3 2.2 1.1
```

你也可以使用成员函数 `push_back()`在 deque 尾端附加元素。vector 并未提供 `push_front()`，因为其时间效能很差（在 vector 头端安插一个元素，需要移动全部元素）。一般而言，STL 容器只提供通常具备良好时间效能的成员函数（所谓「良好」的时间效能，通常意味具有常数复杂度或对数复杂度），这可以防止程式员呼叫性能很差的函数。

Lists

List 由双向串行（doubly linked list）实作而成。这意味 list 内的每个元素都以一部分内存指示其前导元素和后继元素。List 不提供随机存取，因此如果你要存取第 10 个元素，你必须沿着串链依次走过前 9 个元素。不过，移动至下一个元素或前一个元素的行为，可以在常数时间内完成。因此一般的元素存取动作会花费线性时间（平均距离和元素数量成比例）。这比 vector 和 deque 提供的「分摊性（*amoritzed*）」常数时间，性能差很多。

List 的优势是：在任何位置上执行安插或删除动作都非常迅速，因为只需改变连结（links）就好。这表示在 list 中间位置移动元素比在 vector 和 deque 快得多。

以下例子产生一个空 list，准备放置字符，然后将 `'a'` 至 `'z'` 的所有字符插入其中，利用循环每次打印并移除群集的第一个元素，从而印出所有元素：

```
// stl/list1.cpp

#include <iostream>
#include <list>
```

```
using namespace std;

int main()
{
  list<char> coll; // list container for character elements

  // append elements from 'a' to 'z'
  for (char c='a'; c<='z'; ++c) {
     coll.push_back(c);
  }

  /* print all elements
  * - while there are elements
  * - print and remove the first element
  */
  while (! coll.empty()) {
     cout << coll.front() << ' ';
  coll.pop_front();
  }
  cout << endl;
}
```

就像先前的例子一样，头文件 <list> 内含 lists 的宣告。以下定义一个「元素型别为字符」的 list：

```
list<char> coll;
```

成员函数 empty()的回传值告诉我们容器中是否还有元素。只要这个函数回传 false（也就是说容器内还有元素），循环就继续进行：

```
while (! coll.empty()) {
   ...
}
```

循环之内，成员函数 front()会传回第一个元素：

```
cout << coll.front() << ' ';
```

pop_front()函数会删除第一个元素:

```
coll.pop_front();
```

注意，pop_front() 并不会传回被删除的元素，所以你无法将上述两个语句合而为一。

程序的输出结果取决于所用字集。如果是 ASCII 字集，输出如下 ：[4]

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

当然，为了打印 list 的所有元素而「采用循环输出并删除第一个元素」的做法实在是很奇怪。通常你只需走访所有元素即可。lists 并没有提供以 `operator[]` 直接存取元素的能力，因为 lists 不支持随机存取，如果采用 `operator[]` 会导致不良效能。运用迭代器也可以走访并打印所有元素。介绍过迭代器后我会给一个例子。如果你等不及，请跳到 p84。

Strings

你也可以将 string 当作 STL 容器来使用。这里的 strings 是指 C++ string 类别族系（`basic_string<>`, `string`, `wstring`）的对象，第 11 章对此有所介绍。Strings 跟 vectors 很相似，只不过其元素为字符。11.2.13 节, p497 对此有详细解说。

Arrays

另一种容器并非是个类别（class），而是 C/C++ 语言核心所支持的一个型别（type）：具有静态大小或动态大小的 array。但 array 并非 STL 容器，它们并没有类似 `size()` 和 `empty()` 等成员函数。尽管如此，STL 的设计允许你针对 array 呼叫 STL 演算法。当我们以 static arrays 作为初值列（initializer list）时，这一点特别有用。

Array 的运用并无新意，面对 arrays 使用算法，才是新的议题。这个议题将在 6.7.2 节 , p218 讨论。

值得注意的是，我们没有必要再直接编写 dynamic array 了。Vectors 已经具备了 dynamic array 的全部性质，并提供更安全更便捷的接口。详见 6.2.3 节, p155。

5.2.2 关系型容器（Associative Containers）

关系型容器依据特定的排序准则，自动为其元素排序。排序准则以函数形式呈现，用来比较元素值（value）或元素键（key）。预设情况下以 `operator<` 进行比较，不过你也可以提供自己的比较函数，定义出不同的排序准则。

通常关系型容器由二元树（binary tree）实作出来。在二元树中，每个元素（节点）都有一个父节点和两个子节点；左子树的所有元素都比自己小，右子树的所有元素都比自己大。关系型容器的差别主要在于元素的类型以及处理重复元素时的方

[4] 如果是 ASCII 以外的字集，输出结果可能包含非字母字符，甚至可能什么都没有（如果 'z' 不大于 'a' 的话）。

式（态度）。

下面是 STL 中预先定义好的关联容器。由于访问其中元素需要用到迭代器（iterator），所以我推迟至 p87 讨论过迭代器后再举例子。

- **Sets**
  Set 的内部元素依据其值自动排序，每个元素值只能出现一次，不允许重复。
- **Multisets**
  Multiset 和 set 相同，只不过它允许重复元素，也就是说 multiset 可包括多个数值相同的元素。
- **Maps**
  Map 的元素都是「实值/键值」所形成的一个对组（*key/value* pairs）。每个元素有一个键，是排序准则的基础。每一个键只能出现一次，不允许重复。Map 可被视为关系型数组（associative array），也就是具有任意索引型别（index type）的数组（详见 p91）。
- **Multimaps**
  Multimap 和 map 相同，但允许重复元素，也就是说 multimap 可包含多个键值（*key*）相同的元素。Multimap 可被当作「字典」（译注：*dictionary*，某种资料结构）使用。p209 有个范例。

所有关系型容器都有一个可供选择的 template 自变量，指明排序准则。预设采用 `operator<`。排序准则同时也用来测试互等性（equality）：如果两个元素都不小于对方，则两者被视为相等。

你可以将 set 视为一种特殊的 map：其元素实值就是键值。实际产品中，所有这些关系型容器通常都由二元树（binary tree）实作而成。

5.2.3 容器配接器（Container Adapters）

除了以上数个根本的容器类别，为满足特殊需求，C++ 标准链接库还提供了一些特别的（并且预先定义好的）容器配接器，根据基本容器类别实作而成。包括：

- **Stacks**
  名字说明了一切。Stack 容器对元素采取 LIFO（后进先出）管理策略。
- **Queues**
  Queue 容器对元素采取 FIFO（先进先出）管理策略。也就是说，它是个普通的缓冲区（buffer）。
- **Priority Queues**
  Priority Queue 容器中的元素可以拥有不同的优先权。所谓优先权，乃是基于程

式员提供的排序准则（预设使用 operator<）而定义。Priority queue 的效果相当于这样一个 buffer：「下一元素永远是 queue 中优先权最高的元素」。如果同时有多个元素具备最高优先权，则其次序无明确定义。

5.3 迭代器 （Iterators）

迭代器是一个「可巡访 STL 容器内全部或部分元素」的对象。一个迭代器用来指出容器中的一个特定位置。基本操作如下：

- **Operator ***
  传回当前位置上的元素值。如果该元素拥有成员，你可以透过迭代器，直接以 operator-> 取用它们[5]。
- **Operator ++**
  将迭代器前进至下一元素。大多数迭代器还可使用 operator-- 退回到前一个元素。
- **Operators == 和 Operator!=**
  判断两个迭代器是否指向同一位置。
- **Operator =**
  为迭代器赋值（将其所指元素的位置指派过去）。

这些操作和 C/C++「操作 array 元素」时的指针接口一致。不同之处在于，迭代器是个所谓的 *smart pointers*，具有走访复杂数据结构的能力。其下层运作机制取决于其所走访的数据结构。因此，每一种容器型别都必须提供自己的迭代器。事实上每一种容器都将其迭代器以巢状（nested）方式定义于内部。因此各种迭代器的接口相同，型别却不同。这直接导出了泛型程序设计的概念：所有操作行为都使用相同接口，虽然它们的型别不同。因此，你可以使用 templates 将泛型操作公式化，使之得以顺利运作那些「能够满足界面需求」的任何型别。

所有容器类别都提供有一些成员函数，使我们得以获得迭代器并以之遍访所有元素。这些函数中最重要的是：

- **begin()**
  传回一个迭代器，指向容器起始点，也就是第一元素（如果有的话）的位置。
- **end()**
  传回一个迭代器，指向容器结束点。结束点在最后一个元素之后，这样的迭代器又称作「逾尾（*past-the-end*）」迭代器。

[5] 某些老旧的 STL 环境并不对迭代器支持 operator->。

图 5.3 容器的 begin() 和 end() 成员函数

于是，begin()和 end()形成了一个半开区间（half-open range），从第一个元素开始，到最后一个元素的下一位置结束（图 5.3）。半开区间有两个优点：

1. 为「走访元素时，循环的结束时机」提供了一个简单的判断依据。只要尚未到达 end()，循环就可以继续进行。
2. 不必对空区间采取特殊处理手法。空区间的 begin()就等于 end()。

下面这个例子展现了迭代器的用法，将 list 容器内的所有元素打印出来（这就是 p79 那个 list 实例的改进版）。

```
// stl/list2.cpp

#include <iostream>
#include <list>
using namespace std;

int main()
{
  list<char> coll; // list container for character elements

  // append elements from 'a' to 'z'
  for (char c='a'; c<='z'; ++c) {
    coll.push_back(c);
  }

  /* print all elements
   * - iterate over all elements
  */
  list<char>::const_iterator pos;
  for (pos = coll.begin(); pos != coll.end(); ++pos) {
    cout << *pos << ' ';
  }
  cout << endl;
}
```

首先产生一个 list，然后填入'a' ~ 'z'字符，然后在 for 循环中印出所有元素：

```
list<char>::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
   cout << *pos << ' ';
}
```

迭代器 pos 宣告于循环之前，其型别是「指向容器中的不可变元素」的迭代器：

```
list<char>::const iterator pos;
```

任何一种容器都定义有两种迭代器型别：

1. *container*::iterator
   这种迭代器以「读/写」模式走访元素。
2. *container*::const_iterator
   这种迭代器以「只读」模式走访元素。

例如，在 class list 之中，它们的定义可能是这样：

```
namespace std {
  template <class T>
  class list {
  public:
    typedef ... iterator;
    typedef ... const_iterator;
    ...
  };
}
```

至于其中 iterator 和 const_iterator 的确切型别，则于实作中定义。

在循环中，迭代器 pos 以容器的第一个元素位置为初值：

```
pos = coll.begin()
```

循环不断进行，直到 pos 到达容器的结束点：

```
pos != coll.end()
```

在这里，pos 是在和「逾尾（*past-the-end*）」迭代器作比较。当循环内部执行 ++pos 述句，迭代器 pos 就会前进到下一个元素。

总而言之，pos 从第一个元素开始，逐一访问了每一个元素，直到抵达结束点为止（图 5.4）。如果容器内没有任何元素，coll.begin()等于 coll.end()，循环根本不会执行。

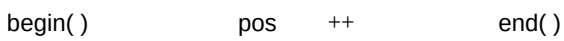

图 5.4 迭代器 pos 走访 list 的每一个元素

在循环内部，述句 *pos 代表当前（current）元素。本例将它输出之后，又接着输出了一个空格。你不能改变元素内容，因为 pos 是个 const_iterator，从迭代器的观点看去，元素是常量，不能更改。不过如果你采用非常量（nonconstant）迭代器，而且元素本身的型别也是非常量（nonconstant），那么就可以透过迭代器来改变元素值。例如：

```
// make all characters in the list uppercase
list<char>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
   *pos = toupper(*pos);
}
```

注意，这里使用「前置式递增（preincrement）」++pos，因为它比「后置式递增（postincrement）」pos++ 效率高。后者需要一个额外的临时对象，因为它必须存放迭代器的原本位置并将它回传，所以一般情况下最好使用 ++pos，不要用 pos++。也就是说，不要这么写：

```
for (pos = coll.begin(); pos != coll.end(); pos++) {
                                              ^^^^^ // OK, but slower
   ...
}
```

为了这个理由，我建议优先采用前置式递增（pre-increment）或前置式递减（pre-decrement）运算符。

5.3.1 关系型容器的运用实例

上个例子中的迭代器循环可应用于任何容器，只需调整迭代器型别即可。现在你知道如何打印关系型容器内的元素了吧。下面是使用关系型容器的一些例子。

Sets 和 Multisets 运用实例

第一个例子展示如何在 set 之中安插元素，并使用迭代器来打印它们。

```
// stl/set1.cpp

#include <iostream>
#include <set>

int main()
{
  // type of the collection
  typedef std::set<int> IntSet;

  IntSet coll; // set container for int values

  /* insert elements from 1 to 6 in arbitrary order
  *-value1 gets inserted twice
  */
  coll.insert(3);
  coll.insert(1);
  coll.insert(5);
  coll.insert(4);
  coll.insert(1);
  coll.insert(6);
  coll.insert(2);

  /* print all elements
  * - iterate over all elements
  */
  IntSet::const_iterator pos;
  for (pos = coll.begin(); pos != coll.end(); ++pos) {
     std::cout << *pos << ' ';
  }
  std::cout << std::endl;
}
```

一如以往，include 指令：

```
#include <set>
```

定义了 sets 的所有必要型别和操作。

既然容器的型别要用到好几次，不妨先定义一个短一点的名字：

```
typedef set<int> IntSet;
```

这个述句定义的 IntSet 型别，其实就是「元素型别为 int 的一个 set」。这种型别有预设的排序准则，以 operator< 为依据，对元素进行排序。这意味元素将以递增方式排列。如果希望以递减方式排列，或是希望使用一个完全不同的排序准则，你可以将该准则传入做为第二个 template 参数。下面例子即是将元素以递减方式排列[6]：

```
typedef set<int,greater<int> > IntSet;
```

以上所用的 greater<> 是一个预先定义的仿函数（functor, or function object），我将在 5.9.2 节, p131 讨论它。8.1.1 节, p294 另有一个例子，仅使用元素的部分资料（例如 ID）进行排序。

所有关系型容器都提供有一个 insert()成员函数，用以安插新元素：

```
coll.insert(3);
coll.insert(1);
...
```

新元素会根据排序准则自动安插到正确位置。注意，你不能使用序列式容器的 push_back()和 push_front()函数，它们在这里毫无意义，因为你没有权力指定新元素的位置。

所有元素（不论以任何次序）安插完毕后，容器的状态如图 5.5。元素以已序状态（*sorted*）存放于内部 tree 结构。任何一个元素（节点）的左子树的所有元素，永远小于右子树的所有元素（这里的「小于」是指就当前排序准则而言）。Sets 不允许存在重复元素，所以容器里头只有一个 "1"。

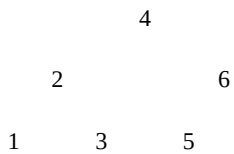

图 5.5 一个 Set，拥有 6 个元素

[6] 注意，两个 ">" 符号之间一定要有一个空格。">>" 会被编译程序视为一个右移（right-shift）运算符，从而导致语法错误。

现在，我们可以运用先前 list 例中所用的相同循环来打印 set 内的元素。以一个迭代器走访全部元素，并逐一打印出来：

```
IntSet::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
  cout << *pos << ' ';
}
```

再一次我要提醒你，由于迭代器是容器定义的，所以无论容器内部结构如何复杂，它都知道如何行事。举个例子，如果迭代器指向第三个元素，运算符++ 便会将它移动到上端的第四个元素，再一次++，便会将它移动到下方第五个元素（图 5.6）。

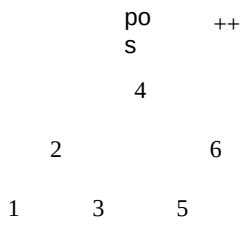

图 5.6 迭代器 pos 走访 Set 内的元素

以下是输出结果：

```
1 2 3 4 5 6
```

如果你想使用 multiset 而不是 set，唯一需要改变的就是容器的型别（set 和 multiset 的定义被置于同一个头文件）：

```
typedef multiset<int> IntSet;
```

由于 multiset 允许元素重复存在，因此其中可包含两个数值皆为 1 的元素。输出结果如下：

```
1 1 2 3 4 5 6
```

Maps 和 Multimaps 的运用实例

Map 的元素是成对的键值/实值（key/value）。因此其宣告、元素安插、元素存取皆和 set 有所不同。下面是一个 multimap 运用实例：

```
// stl/mmap1.cpp

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  // type of the collection
  typedef multimap<int,string> IntStringMMap;

  IntStringMMap coll; // container for int/string values

  // insert some elements in arbitrary order
  // - a value with key 1 gets inserted twice
  coll.insert(make_pair(5,"tagged"));
  coll.insert(make_pair(2,"a"));
  coll.insert(make_pair(1,"this"));
  coll.insert(make_pair(4,"of"));
  coll.insert(make_pair(6,"strings"));
  coll.insert(make_pair(1,"is"));
  coll.insert(make_pair(3,"multimap"));

  /* print all element values
  * - iterate over all elements
  * - element member second is the value
  */
  IntStringMMap::iterator pos;
  for (pos = coll.begin(); pos != coll.end(); ++pos) {
       cout << pos->second << ' ';
  }
  cout << endl;
}
```

程序的输出结果可能是这样：

```
this is a multimap of tagged strings
```

不过由于 "this" 和 "is" 的键值相同，两者的出现顺序也可能反过来。

拿这个例子和 p87 的 set 实例作比较，你会发现以下两点不同：

1. 这里的元素是成对的键值/实值（*key/value* pair），所以你必须首先生成这个 pair，再将它插入群集内部。辅助函数 `make_pair()` 正是为了这个目的而打造。这个问题的细节，以及其他安插方法，请见 p203，
2. 迭代器所指的是「键值/实值」对组（*key/value* pair），因此你无法一口气打印它们，你必须取出 `pair` 的成员，亦即所谓的 `first` 和 `second`（`pair` 型别在 4.1 节, p33 介绍过）。因此，以下述句：

   ```
   pos->second
   ```

   便取得了「键值/实值」对组中的第二部分，也就是 multimap 元素的实值（*value*）。和一般指标的情形一样，上述述句就是以下述句的简写方案[7]：

   ```
   (*pos).second
   ```

   同样道理，以下述句：

   ```
   pos->first
   ```

   取得「键值/实值」对组中的第一部分，也就是 multimap 元素的键值（*key*）。

Multimaps 也可以用来作为 *dictionaries*，详见 p209 实例。

将 Maps 当作关系型数组（associative arrays）

如果上述例子中以 map 取代 multimap，输出结果就不会有重复键值（*key*s），实值（*value*s）则和上述结果一样。一个「键值/实值」对组所形成的群集中，如果所有键值都是独一无二的，我们可将它视为一个关系型数组（associative array）。考虑以下例子：

```
// stl/map1.cpp

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
  /* type of the container:
  *-map: elements key/value pairs
  *-string: keys have type string
  *-float: values have type float
  */
```

[7] 某些老旧环境并没有实现出 `iterator->`，这时候你就只能使用第二个表述式了。

```
    typedef map<string,float> StringFloatMap;
    StringFloatMap coll;

    // insert some elements into the collection
    coll["VAT"] = 0.15;
    coll["Pi"] = 3.1415;
    coll["an arbitrary number"] = 4983.223;
    coll["Null"] = 0;

    /* print all elements
    * - iterate over all elements
    * - element member first is the key
    * - element member second is the value
    */
    StringFloatMap::iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
         cout << "key: \"" << pos->first << "\" "
              << "value: " << pos->second << endl;
    }
}
```

当我们宣告容器型别的时候，必须同时指定键值（*key*）和实值（*value*）的型别：

```
typedef map<string,float> StringFloatMap;
```

Maps 允许你使用 `operator[]` 安插元素：

```
coll["VAT"] = 0.15;
coll["Pi"] = 3.1415;
coll["an arbitrary number"] = 4983.223;
coll["Null"] = 0;
```

在这里，以键值为索引，键值可为任意型别。这正是关系型数组的接口。所谓关联式数组就是：索引可以采用任何型别。

注意这里的 *subscript*（下标）运算符和一般 array 所用的行为有些不同：在这里，索引可以不对应于任何元素。如果你指定了一个新索引（新键值），会导致产生一个对应的新元素，并被安插于 map。也就是说，没有任何索引是「错误」的。因此，以下述句：

```
coll["Null"] = 0;
```

其中的式子：

```
coll["Null"]
```

会产生一个新元素，键值为 "Null"。然后 *assignment*（赋值）运算符再将该元素的实值设为 0（并转化为 `float`）。6.6.3 节, p205 更详细地讨论了如何将 maps 当作关系型数组。

Multimaps 不允许我们使用 *subscript*（下标）运算符，因为 multimaps 允许单一索引对应到多个不同元素，而下标运算符却只能处理单一实值。你必须先产生一个「键值/实值」对组，然后再插入 multimap，见 p90。当然，对于 maps 也可以这么做，细节请参考 p202。

存取 multimaps 或 maps 的元素时，你必须透过 `pair` 结构的 `first` 成员和 `second` 成员，才能取得键值（*key*）和实值（*value*）。上述程序的输出如下：

```
key: "Null" value: 0
key: "Pi" value: 3.1415
key: "VAT" value: 0.15
key: "an arbitrary number" value: 4983.22
```

5.3.2 迭代器分类（Iterator Categories）

除了基本操作之外，迭代器还有其他能力。这些能力取决于容器的内部结构。STL 总是只提供效率上比较出色的操作，因此，如果容器允许随机存取（例如 vectors 或 deques），那么它们的迭代器也能进行随机操作（例如直接让迭代器指向第五元素）。

根据能力的不同，迭代器被划分为五种不同类属。STL 预先定义好的所有容器，其迭代器均属于以下两种分类：

1. 双向迭代器（**Bidirectional iterator**）
   顾名思义，双向迭代器可以双向行进：以递增（increment）运算前进或以递减（decrement）运算后退。list、set、multiset、map 和 multimap 这些容器所提供的迭代器都属此类。
2. 随机存取迭代器（**Random access iterator**）
   随机存取迭代器不但具备双向迭代器的所有属性，还具备随机访问能力。更明确地说，它们提供了「迭代器算术运算」必要的运算符（和「一般指标的算术运算」完全对应）。你可以对迭代器增加或减少一个偏移量、处理迭代器之间的距离、或是使用 < 和 > 之类的 *relational*（相对关系）运算符来比较两个迭代器。vector, deque 和 strings 所提供的迭代器都属此类。

其他迭代器类型在 7.2 节, p251 介绍。

为了撰写尽可能与容器型别无关的泛型程序代码，你最好不要使用随机存取迭代器（*random access iterator*s）的特有操作。例如以下例子，可以在任何容器上运作：

```
for (pos = coll.begin(); pos != coll.end(); ++pos) {
    ...
}
```

而下面这样的程序代码就不是所有容器都适用了：

```
for (pos = coll.begin(); pos < coll.end(); ++pos) {
    ...
}
```

两者的唯一区别在于测试循环条件时，第二例使用 operator<，第一例使用 operator!=。要知道，只有 *random access iterator*s 才支持 operator<，所以第二例中的循环对于 lists、sets 和 maps 无法运作。为了写出适用于任何容器的泛型程序代码，你应该使用 operator!= 而非 operator<。不过如此一来，程序代码的安全性可能有损，因为如果 pos 的位置在 end()的后面，你未必便能发现（关于 STL 使用上的可能错误，请见 5.11 节, p136）。究竟使用哪种方式，取决于当时情况，取决于个人经验，取决于你。

为了避免误解，我再强调一句。注意，我说的是类属、分类（categories），不是类别（classes）。所谓类属，只是定义迭代器的能力，无关乎迭代器的型别（type）。STL 的泛型概念可以以纯抽象形式工作，也就是说，任何东西只要行为「像」一个双向迭代器，那么它就是一个双向迭代器。

5.4 算法（Algorithms）

为了处理容器内的元素，STL 提供了一些标准算法，包括搜寻、排序、拷贝、重新排序、修改、数值运算等十分基本而普遍的算法。

算法并非容器类别的成员函数，而是一种搭配迭代器使用的全局函数。这么做（译注：意指搭配迭代器来使用）有一个重要优势：所有算法只需实作出一份，就可以对所有容器运作，不必为每一种容器量身订制。算法甚至可以操作不同型别（types）之容器内的元素，也可以与用户定义的容器搭配。这个概念大幅降低了程序代码的体积，提高了链接库的能力和弹性。

注意，这里所阐述的并非面向对象思维模式（OOP paradigm），而是泛型函数编程思维模式（generic functional programming paradigm）。在面向对象编程（OOP）概念里，数据与操作合为一体，在这里则被明确划分开来，再透过特定的接口彼此互动。当然这需要付出代价：首先，用法有失直观，其次，某些数据结构和演算法之间并不兼容。更有甚者，某些容器和算法虽然勉强兼容，却毫无用处（也许导致很糟的效能）。因此，深入学习 STL 的概念并了解其缺陷，显得十分重要，惟其如此，方能取其利而避其害。我将在本章剩余篇幅中，透过实例详细介绍它们。让我们从简单的 STL 算法的运用入手。以下实例展现了某些算法的使用方式：

```
// stl/algo1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   vector<int> coll;
   vector<int>::iterator pos;

   // insert elements from 1 to 6 in arbitrary order
   coll.push_back(2);
   coll.push_back(5);
   coll.push_back(4);
   coll.push_back(1);
   coll.push_back(6);
   coll.push_back(3);

   // find and print minimum and maximum elements
   pos = min_element (coll.begin(), coll.end());
   cout << "min: " << *pos << endl;
   pos = max_element (coll.begin(), coll.end());
   cout << "max: " << *pos << endl;

   // sort all elements
   sort (coll.begin(), coll.end());

   // find the first element with value 3
   pos = find (coll.begin(), coll.end(),  // range
               3);                        // value
   // reverse the order of the found element with value 3 and
   // all following elements
   reverse (pos, coll.end());

   // print all elements
   for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        cout << *pos << ' ';
   }
   cout << endl;
}
```

为了呼叫算法，首先你必须含入头文件 <algorithm>：

```
#include <algorithm>
```

最先出现的是算法 min_element()和 max_element()。呼叫它们时，你必须传入两个自变量，定义出欲处理的元素范围。如果想要处理容器内的所有元素，可以使用 begin()和 end()。两个算法都传回一个迭代器，分别指向最小或最大元素。因此，以下述句：

```
pos = min_element (coll.begin(), coll.end());
```

算法 min_element()回传最小元素的位置（如果最小元素不只一个，则回传第一个最小元素的位置）。以下述句印出该元素：

```
cout << "min: " << *pos << endl;
```

当然，你也可以合并上述两个动作于单一述句：

```
cout << *max element(coll.begin(),coll.end()) << endl;
```

接下来的算法是 sort()。顾名思义，它将「由两个自变量设定出来」的区间内的所有元素加以排序。你可以（选择性地）传入一个排序准则：预设的是 operator <。因此，本例容器内的所有元素以递增方式排列。

```
sort (coll.begin(), coll.end());
```

排序后的容器元素如下排列：

```
1 2 3 4 5 6
```

再来便是算法 find()。它在给定范围中搜寻某个值。本例在整个容器内寻找第一个数值为 3 的元素。

```
pos = find (coll.begin(), coll.end(),      // range
           3);                             // value
```

如果 find()成功了，便回传一个迭代器，指向目标元素。如果失败，回传一个「逾尾（past-the-end）」迭代器，亦即 find()所接受的第二自变量。本例在第三个元素位置上发现数值 3，因此完成后 pos 指向 coll 的第三个位置。

本例所展示的最后一个算法是 reverse()，将区间内的元素反转放置：

```
reverse (pos, coll.end());
```

于是第三个至最后一个元素之间的所有元素都被反转置放。整个程序输出如下：

```
min: 1
max: 6
1 2 6 5 4 3
```

5.4.1 区间（Ranges）

所有算法都用来处理一个或多个区间内的元素。这样的区间可以（但非强行要求）涵盖容器内的全部元素。因此，为了得以操作容器元素的某个子集，我们必须将区间首尾当做两个自变量（arguments）传给算法，而不是一口气把整个容器传递进去。

这样的接口灵活又危险。呼叫者必须确保经由两自变量定义出来的区间是有效的（valid）。所谓有效就是，从起点出发，逐一前进，能够到达终点。也就是说，程序员自己必须确保两个迭代器隶属同一容器，而且前后放置正确。否则结果难料，可能会引起无限循环，也可能会存取到内存禁区。就此点而言，迭代器就像一般指标一样危险。不过请注意，所谓「结果难料」（或说行为未有定义，*undefined behavior*）意味任何 STL 实作品均可自由选择合适的方式来处理此类错误。稍后你会发现，确保区间的有效性并不像听起来那么简单。与此相关的一些细节请参见 5.11 节， p136。所有算法处理的都是半开区间（half-open ranges） —含括起始元素位置但不含括结尾元素位置。传统的数学表示方式为：

```
[begin, end)
```

或

```
[begin, end[
```

本书采用第一种表示法。

半开区间的优点已于 p84 介绍过（主要是单纯，可避免对空群集做另外特殊处理）。当然，金无足赤，世上亦没有完美的设计。请看下面的例子：

```
// stl/find1.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
   list<int> coll;
   list<int>::iterator pos;

   // insert elements from 20 to 40
   for (int i=20; i<=40; ++i)
        coll.push_back(i);
   }
```

```
    /* find position of element with value 3
     * - there is none, so pos gets coll.end()
     */
    pos = find (coll.begin(), coll.end(),  // range
                3);                        // value

    /* reverse the order of elements between found element and the
    * end - because pos is coll.end() it reverses an empty range
    */
    reverse (pos, coll.end());

    // find positions of values 25 and 35
    list<int>::iterator pos25, pos35;
    pos25 = find (coll.begin(), coll.end(),     // range
                  25);                          // value
    pos35 = find (coll.begin(), coll.end(),     // range
                  35);                          // value

    /* print the maximum of the corresponding range
    * - note: including pos25 but excluding pos35
    */
    cout << "max: " << *max_element (pos25, pos35) << endl;

    // process the elements including the last position
    cout << "max: " << *max_element (pos25, ++pos35) << endl;
}
```

本例首先以 20 至 40 的整数做为容器初值。当搜寻元素值 3 的任务失败后，`find()` 传回区间的结束位置（本例为 `coll.end()`）并指派给 `pos`。以此 `pos` 作为稍后呼叫 `reverse()`时的区间起点，纯粹是空摆架子，因为其结果相当于：

```
reverse (coll.end(), coll.end());
```

这其实就是在逆转一个空区间，当然毫无效果了（亦即所谓的 "no-op"）。

如果使用 `find()`来获取某个子集的第一个和最后一个元素，你必须考虑一点：半开区间并不包含最后一个元素。所以上述例子第一次呼叫 `max_element()`：

```
max_element (pos25, pos35)
```

传回的是 34，而不是 35：

```
max: 34
```

为了处理最后一个元素，你必须把该元素的下一个位置传递给算法：

```
max_element (pos25, ++pos35)
```

这样才能得到正确的结果：

```
max: 35
```

注意，本例使用的是 list 容器，所以你只能以 ++ 取得 pos35 的下一个位置。如果面对的是 vectors 或 deques 的随机存取迭代器（*random access iterators*），你
可以写 pos35 + 1。这是因为随机存取迭代器允许「迭代器算术运算」（参见 p93, 5.3.2 节，p255, 7.2.5 节）。

当然，你可以使用 pos25 和 pos35 来搜寻其间的任何东西。记住，为了让搜寻动作及于 pos35，必须将元素 35 的下一位置传入，例如：

```
// increment pos35 to search with its value included
++pos35;
pos30 = find(pos25,pos35,   // range
             30);           // value
if (pos30 == pos35) {
     cout << "30 is NOT in the subrange" << endl;
}
else {
     cout << "30 is in the subrange" << endl;
}
```

本节中的所有例子都可以正常运作，但那完全是因为你很清楚 pos25 一定在 pos35 之前。否则，[pos25;pos35)就不是个有效区间。如果你对于「哪个元素在前，哪个元素在后」心中没谱儿，事情可就麻烦了，说不定会导致未定义行为。

现在假设你并不知道元素 25 和元素 35 的前后关系，甚至连它们是否存在也心存疑虑。如果你手上用的是随机存取迭代器（*random access iterators*），你可以使用 operator< 进行检查：

```
if (pos25 < pos35) {
   // only [pos25; pos35) is valid
   ...
}
else if (pos35 < pos25) {
   // only [pos35; pos25) is valid
   ...
}
else {
   // both are equal, so both must be end()
   ...
}
```

如果你手上的并非随机存取迭代器，那还真的没什么直截了当的办法可确定哪个迭代器在前。你只能在「起点和某个迭代器」之间，以及「该迭代器和终点」之间，寻找另外那个迭代器。此时你的解决方法需要一些变化：不是一口气在整个区间中搜寻两个值，而是了解哪个值先找到，哪个值后找到。例如：

```
pos25 = find (coll.begin(), coll.end(),  // range
              25);                       // value
pos35 = find (coll.begin(), pos25,       // range
              35);                       // value
if (pos35 != pos25) {
   /* pos35 is in front of pos25
   * so, only [pos35; pos25) is valid
   */
   ...
}
else {
   pos35 = find (pos25, coll.end(),      // range
                 35);                    // value
   if (pos35 != pos25) {
      /* pos25 is in front of pos35
      * so, only [pos25; pos35) is valid
      */
      ...
    }
   else {
      // both are equal, so both must be end()
      ...
   }
}
```

和前例不同的是，本例并非在 `coll` 的整个区间内搜寻 35，而是先在起点和 `pos25` 之间寻找，如果一无所获，再在 `pos25` 之后的区间寻找。其结果当然使你得以完全掌握哪个位置在前面、哪个子区间有效。

这么做并不是很有效率。当然还有其他高招，可以直接找到 25 或 35 第一次出现的位置，不过那就需要用到目前还未介绍的一些 STL 技术了：

```
pos = find_if (coll.begin(), coll.end(),          // range
               compose_f_gx_hx(logical_or<bool>()  //criterion
,                              bind2nd(equal_to<int>(),25),
                               bind2nd(equal_to<int>(),35)))
                               ;
```

```
switch (*pos) {
   case 25:
      // element with value 25 comes first
      pos25 = pos;
      pos35 = find (++pos, coll.end(),   // range
                    35);                 // value
      ...
      break;
   case 35:
      // element with value 35 comes first
      pos35 = pos;
      pos25 = find (++pos, coll.end(),   // range
                    25);                 // value
      ...
      break;
   default:
      // no element with value 25 or 35 found
      ...
      break;
}
```

这里使用了一个特别的表达式作为搜寻规则，其目的是找到数值 25 或数值 35 第一次出现的位置。这个表达式由好几个预先定义的仿函数（functors，或名 function objects）组成，我将在 5.9.2 节， p131 和 8.2 节， p305 介绍所有预先定义的仿函数。`compose_f_gx_hx` 是个灵巧的辅助型仿函数，我将在 8.3.1 节, p316 介绍它。

5.4.2 处理多个区间

有数个算法可以（或说需要）同时处理多个区间。通常你必须设定第一个区间的起点和终点，至于其他区间，你只需设定起点即可，终点通常可由第一区间的元素数量推导出来。下面例子中，`equal()`从头开始逐一比较 `coll1` 和 `coll2` 的所有元素：

```
if (equal (coll1.begin(), coll1.end(),
           coll2.begin())) {
   ...
}
```

因此，`coll2` 之中参与比较的元素数量，间接取决于 `coll1` 内的元素数量。

这使我们导出一个重要心得：如果某个算法用来处理多个区间，那么当你呼叫它时，务必确保第二（以及其他）区间所拥有的元素个数，至少和第一区间内的元素个数相同。特别是，执行涂写动作时，务必确保目标区间（destination ranges）够大。

考虑下面这个程序：

```
// stl/copy1.cpp

#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
   list<int> coll1;
   vector<int> coll2;

   // insert elements from 1 to 9
   for (int i=1; i<=9; ++i) {
        coll1.push_back(i);
   }

   // RUNTIME ERROR:
   // - overwrites nonexisting elements in the destination
   copy (coll1.begin(), coll1.end(),  // source
         coll2.begin());              // destination
   ...
}
```

这里呼叫了 `copy()`算法，将第一区间内的全部元素拷贝至目标区间。如上所述，第一区间的起点和终点都已指定，第二区间只指出起点。然而，由于该算法执行的是覆写动作（overwrites）而非安插动作（inserts），所以目标区间必须拥有足够的元素来被覆写，否则就会像这个例子一样，导致未定义的行为。如果目标区间内没有足够的元素供覆写，通常意味你会覆写 `coll2.end()`之后的任何东西，幸运的话你的程序立即崩溃 — 这起码还能让你知道出错了。你可以强制自己获得这种幸运：使用 STL 安全版本。在这个安全版本中，所有未定义的行为都会被导向一个错误处理程序（error handling procedure）。请参考 5.11.1 节，p138。

要想避免上述错误，你可以 (1) 确认目标区间内有足够的元素空间，或是 (2) 采用 *insert iterator*s。*Insert iterator*s 将在 5.5.1 节， p104 介绍。我首先解释如何修改
目标区间，俾使它有足够的空间。

要想让目标区间够大，你要不一开始就给它一个正确大小，要不就明白地改变其大小。这两个办法都只适用于序列式容器（vectors, deques, lists）。关系型容器根本不会有此问题，因为关系型容器不可能被当作覆写式算法的操作目标（原因见 5.6.2 节，p115）。以下例子展示如何增加容器的大小：

```
// stl/copy2.cpp

#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <algorithm>
using namespace std;

int main()
{
   list<int> coll1;
   vector<int> coll2;

   // insert elements from 1 to 9
   for (int i=1; i<=9; ++i) {
        coll1.push_back(i);
   }

   // resize destination to have enough room for the
   // overwriting algorithm
   coll2.resize (coll1.size());

   /* copy elements from first into second collection
   * - overwrites existing elements in destination
   */
   copy (coll1.begin(), coll1.end(),  // source
         coll2.begin())               // destination
         ;
   /* create third collection with enough room
   * - initial size is passed as parameter
   */
   deque<int>
   coll3(coll1.size());
```

```
    // copy elements from first into third collection
    copy (coll1.begin(), coll1.end(),      // source
          coll3.begin());                  // destination
}
```

在这里，resize()的作用是改变 coll2 的元素个数：

```
coll2.resize (coll1.size());
```

coll3 则是在初始化时就指明要有足够空间，以容纳 coll1 中的全部元素：

```
deque<int> coll3(coll1.size());
```

注意，这两种方法都会产出新元素并赋予初值。这些元素由 *default* 建构式初始化，没有任何自变量。你可以传递额外的自变量给建构式和 resize()，这样就可以按你的意愿将新元素初始化。

5.5 迭代器之配接器（Iterator Adapters）

迭代器（Iterators）是一个纯粹抽象概念：任何东西，只要其行为类似迭代器，它就是一个迭代器。因此，你可以撰写一些类别（classes），具备迭代器接口，但有着各不相同的行为。C++ 标准链接库提供了数个预先定义的特殊迭代器，亦即所谓迭代器配接器（iterator adapters）。它们不仅是辅助性质而已，它们赋予整个迭代器抽象概念更强大的能力。

以下数小节简介三种迭代器配接器（iterator adapters）：

1. *Insert iterator*s（安插型迭代器）
2. *Stream iterator*s（串流迭代器）
3. *Reverse iterator*s（逆向迭代器）

第 7.4 节, p264 会对它们做更详实的讲解。

5.5.1 Insert Iterators（安插型迭代器）

迭代器配接器的第一个例子是 *insert iterator*s，或称为 *inserter*s。*Inserter*s 可以使算法以安插（insert）方式而非覆写（overwrite）方式运作。使用它，可以解决算法的「目标空间不足」问题。是的，它会促使目标区间的大小按需要成长。

*Insert iterator*s 内部将接口做了新的定义：

- 如果你对某个元素设值（assign），会引发「对其所属群集的安插（insert）动作」。至于插入位置是在容器的最前或最后，或是于某特定位置上，端视三种不同的 *insert iterator*s 而定。
- 「单步前进（step forward）」不会造成任何动静（是一个 no-op）。

现在请看下面这个例子：

```
// stl/copy3.cpp

#include <iostream>
#include <vector>
#include <list>
#include <deque>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
   list<int> coll1;

   // insert elements from 1 to 9 into the first collection
   for (int i=1; i<=9; ++i) {
        coll1.push_back(i);
   }

   // copy the elements of coll1 into coll2 by appending them
   vector<int> coll2;
   copy (coll1.begin(), coll1.end(),      // source
         back_inserter(coll2));           // destination

   // copy the elements of coll1 into coll3 by inserting them at
   the
   // front - reverses the order of the elements
   deque<int> coll3;
   copy (coll1.begin(), coll1.end(),      // source
         front_inserter(coll3));          // destination

   // copy elements of coll1 into coll4
   // - only inserter that works for associative collections
   set<int> coll4;
   copy (coll1.begin(), coll1.end(),      // source
         inserter(coll4,coll4.begin()))   // destination
}  ;
```

此例运用了三种预先定义的 *insert iterators*：

1. **Back inserters**（安插于容器最尾端）

   *Back inserter*s 的内部呼叫 `push_back()`，在容器尾端插入元素（此即「尾附」动作）。以下述句完成之后，`coll1` 的所有元素都会被附加到 `coll2` 中：

   ```
   copy (coll1.begin(), coll1.end(),  // source
         back_inserter(coll2));       // destination
   ```

   当然，只有在提供有 `push_back()`成员函数的容器中，*back inserter*s 才能派上用场。在 C++ 标准链接库中，这样的容器有三：vector, deque, list。

2. **Front inserters**（安插于容器最前端）

   *Front inserter*s 的内部呼叫 `push_front()`，将元素安插于容器最前端。以下述句将 `coll1` 的所有元素插入 `coll3`：

   ```
   copy (coll1.begin(), coll1.end(),  // source
         front_inserter(coll3));      // destination
   ```

   注意，这种动作逆转了被安插元素的次序。如果你先安插 1，再向前安插 2，那么 1 会排列在 2 的后面。

   *Front inserter*s 只能用于提供有 `push_front()`成员函数的容器，在标准链接库中，这样的容器是 deque 和 list。

3. **General inserters**（一般性安插器）

   这种一般性的 inserters，简称就叫 *inserter*s，它的作用是将元素插入「初始化时接受之第二自变量」所指位置的前方。*Inserter*s 内部呼叫成员函数 `insert()`，并以新值和新位置做为自变量。所有 STL 容器都提供有 `insert()`成员函数，因此，这是唯一可用于关系型容器身上的一种预先定义好的 *inserter*。

   等等，我不是说过，在关系型容器身上安插新元素时，不能指定其位置吗？它们的位置是由它们的值决定的啊！好，我解释一下，很简单：在关系型容器中，你所给的位置只是一个提示，帮助它确定从什么地方开始搜寻正确的安插位置。如果提示不正确，效率上的表现会比「没有提示」更糟糕。7.5.2 节， p288 介绍了一个用户自定的 *inserter*，对关系型容器特别有用。

表 5.1 列出 *insert iterator*s 的功能。7.4.2 节, p271 还会介绍更多细节。

算式（expression）	Inserter 种类
`back_inserter(container)`	使用 `push_back()`在容器尾端安插元素，元素排列次序和安插次序相同。
`front_inserter(container)`	使用 `push_front()`在容器前端安插元素，元素排列次序和安插次序相反。
`inserter(container, pos)`	使用 `insert()`在 `pos` 位置上安插元素，元素排列次序和安插次序相同。

表 5.1 预先定义的三种 *Insert Iterator*s

5.5.2 Stream Iterators（串流迭代器）

另一种非常有用的迭代器配接器（iterator adapter）是 *stream iterator*，这是一种用来读写 stream[8] 的迭代器。它们提供了必要的抽象性，使得来自键盘的输入像是个群集（collection），你能够从中读取内容。同样道理，你也可以把一个算法的输出结果重新导向到某个档案或屏幕上。

下面是展示 STL 威力的一个典型例子。和一般 C 或 C++ 程序相比，本例仅用数条述句，就完成了大量复杂工作：

```
// stl/ioiter1.cpp

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
   vector<string> coll;

   /* read all words from the standard input
   * - source: all strings until end-of-file (or error)
   * - destination: coll (inserting)
   */
   copy (istream_iterator<string>(cin),  // start of source
         istream_iterator<string>(),     // end of source
         back_inserter(coll));           // destination

   // sort elements
   sort (coll.begin(), coll.end());

   /* print all elements without duplicates
   * - source: coll
   * - destination: standard output (with newline between elements)
   */
   unique_copy (coll.begin(), coll.end(),            // source
                ostream_iterator<string>(cout,"\n")); //
}  destination
```

[8] Stream（串流）是一个用来表现 I/O 信道的对象（详见第 13 章）。

这个程序只用三个述句就完成一系列工作：从标准输入设备读取所有输入文字、排序、将它们打印于屏幕。让我们逐一思考这三个述句。下面这个述句：

```
copy (istream_iterator<string>(cin),
      istream_iterator<string>(),
      back_inserter(coll));
```

用到两个 *input stream iterator*s：

1. `istream_iterator<string>(cin)`
这会产生一个可从「标准输入串流（standard input stream）`cin`」读取数据的 *stream iterator*[9]。其中的 template 自变量 `string` 表示，这个 *stream iterator* 专司读取该种型别的元素（`string` 型别将在第 11 章介绍）。这些元素透过一般的 `operator>>` 被读取进来。因此每当算法企图处理下一个元素时，*istream iterator* 就会将这种企图转化为以下行动：

```
cin >> string
```

针对 string 而执行的 *input* 运算符通常读取以空白分隔的文字（参见 p492），因此上述算法的行为将是「逐词读取（word-by-word）」。

2. `istream_iterator<string>()`
呼叫 *istream iterator*s 的 *default* 建构式，产生一个代表「串流结束符号」（end-of-stream）的迭代器，它代表的意义是：你不能再从中读取任何东西。

只要不断逐一前进的那个第一自变量不同于第二自变量，算法 `copy()` 就持续动作。这里的 end-of-stream 迭代器正是作为区间终点之用，因此这个算法便从 `cin` 读取所有 strings，直到读无可读为止（可能是因为到达了 end-of-stream，也可能是因为读入过程发生错误）。总而言之，算法的数据源是「来自 `cin` 的所有文字」。在 *back inserter* 的协助下，这些文字被拷贝并插入 `coll` 中。

接下来的 `sort()` 算法对所有元素进行排序：

```
sort (coll.begin(), coll.end());
```

最后，下面这个述句：

```
unique_copy (coll.begin(), coll.end(),
             ostream_iterator<string>(cout,"\n"));
```

将其中所有元素拷贝到目的端 `cout`。处理过程中算法 `unique_copy()` 会消除毗邻的重复值。其中的表达式：

```
ostream_iterator<string>(cout,"\n")
```

[9] 在某些老旧系统中，你必须使用 `ptrdiff_t` 作为第二个模板自变量，才能产生出 *istream iterator*（参见 7.4.3 节, p280）。

会产生一个 *output stream iterator*，透过 `operator<<` 向 `cout` 写入 strings。`cout` 之后的第二自变量（可有可无）被用来作为元素之间的分隔符。本例指定为一个换行符号，因此每个元素都被打印于独立的一行。

这个程序内的所有组件都是 templates，所以你可以轻易改变程序，对其他型别如整数或更复杂的对象进行排序。7.4.3 节， p277 对于 *iostream iterator*s 进行了更详细的介绍，并附带更多实例。

本例使用一个宣告和三个述句，对来自标准输入设备的所有文字（单词）进行排序。你还可以更进一步，只用一个宣告和一个述句就搞定一切。详见 p228。

5.5.3 Reverse Iterators（逆向迭代器）

第三种预先定义的迭代器配接器（iterator adapters）就是 *reverse iterator*s，此物像是倒转筋脉似地以逆向方式进行所有操作。它将 *increment*（递增）运算转换为 *decrement*（递减）运算，反之亦然。所有容器都可以透过成员函数 `rbegin()`和 `rend()`产生出 *reverse iterator*s。例如：

```
// stl/riter1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   vector<int> coll;

   // insert elements from 1 to 9
   for (int i=1; i<=9; ++i) {
        coll.push_back(i);
   }

   // print all element in reverse order
   copy (coll.rbegin(), coll.rend(),          // source
         ostream_iterator<int>(cout,"         // destination
"));
   cout << endl;
}
```

其中的表达式：

```
coll.rbegin(
)
```

传回一个由 coll 定义的 *reverse iterator*。这个迭代器可作为「对群集 coll 的元素逆向走访」的起点。它指向群集的结尾位置（也就是最后元素的下一位置）。
因此，表达式：

```
*coll.rbegin()
```

传回的是最后一个元素的值。

对应地，表达式：

```
coll.rend()
```

传回的 *reverse iterator*，可作为「对群集 coll 的元素逆向走访」的终点。它也是指向「逾尾」（past-the-end）位置，只不过方向相反，指的是容器内第一个元素的前一个位置。

以下表达式没有定义：

```
*coll.rend()
```

同样情况，以下表达式也没有定义：

```
*coll.end()
```

注意，当某个位置上并无合法元素时，永远不要使用 operator*或 operator->。

如果采用 *reverse iterators*，所有算法便可以不需特殊处理就以相反方向操作容器，这自然是美事一桩。使用 operator++ 前进至下一元素，被转化为使用 operator-- 后退至前一元素。本例中的 copy()，「从尾到头」地走访所有 coll 元素。程序输出如下：

```
9 8 7 6 5 4 3 2 1
```

你可以将一般迭代器转换为 *reverse iterators*，反之亦可。然而，对于具体某个迭代器而言，这样的转换会改变其所指对象。这些细节在第 7.4.1 节, p264 介绍。

5.6 更易型算法（Manipulating Algorithms）

译注：根据实质意义，我不把 manipulating algorithms 译为「操控型」算法。manipulating algorithms 是指会「删除或重排或修改元素」的算法，见 p115。该页亦出现另一个相同意义的术语：modifying algorithms。有些书籍（例如 *Generic Programming and the STL*）采用 mutating algorithms 一词。为此，我将这些相同意义的术语都译为「更易型」或「变动型」算法。我亦曾在某些书中采用「质变算法」一词。

某些算法会变更目标区间的内容，甚至会删除元素。一旦这种情况出现，请务必注意几个特殊问题。本节将对此做出解释。它们确实令人讶异，并体现了 STL「为了将容器和算法分离，以获取弹性」而付出的代价。

5.6.1 移除（Removing）元素

算法 `remove()` 自某个区间删除元素。然而如果你用它来删除容器中的所有元素，其行为肯定会让你吃惊。例如：

```
// stl/remove1.cpp
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
   list<int> coll;

   // insert elements from 6 to 1 and 1 to 6
   for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
   }

   // print all elements of the collection
   cout << "pre: ";
   copy (coll.begin(), coll.end(),            // source
         ostream_iterator<int>(cout,"         // destination
"));
   cout << endl;

   // remove all elements with value 3   // range
   remove (coll.begin(), coll.end(),     // value
           3);
```

```
    // print all elements of the collection
    cout << "post: ";
    copy (coll.begin(), coll.end(),       // source
          ostream_iterator<int>(cout," ")); // destination
    cout << endl;
}
```

缺乏 STL 深层认识的人，看了这程序，必然认为所有数值为 3 的元素都会从群集中被移除。然而，程序的输出却是这样：

```
pre: 6 5 4 3 2 1 1 2 3 4 5 6
post: 6 5 4 2 1 1 2 4 5 6 5 6
```

啊呀，remove()并没有改变群集中的元素数量。end()传回的还是当初那个终点，size()传回的还是当初那个大小。不过某些事情还是有了变化：元素的次序改变了，有些元素被删除掉了。数值为 3 的元素被其后的元素覆盖了（图 5.7）。至于群集尾端那些未被覆盖的元素，原封不动 — 但是从逻辑角度来说，那些元素已经不属于这个群集了。

图 5.7 remove() 如何运作

事实上，这个算法传回了一个新的终点。你可以利用该终点获得新区间、缩减后的容器大小，或是获得被删除元素的个数。看看下面这个改进版本：

```
// stl/remove2.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> coll;

    // insert elements from 6 to 1 and 1 to 6
```

```
    for (int i=1; i<=6; ++i) {
         coll.push_front(i);
         coll.push_back(i);
    }

    // print all elements of the collection
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    // remove all elements with value 3
    // - retain new end
    list<int>::iterator end = remove (coll.begin(), coll.end(),
                                      3);

    // print resulting elements of the collection
    copy (coll.begin(), end,
          ostream_iterator<int>(cout," "));
    cout << endl;

    // print number of resulting elements
    cout << "number of removed elements: "
         << distance(end,coll.end()) << endl;

    // remove "removed" elements
    coll.erase (end, coll.end());

    // print all elements of the modified collection
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

在这个版本中，`remove()`的回传值被设定给 `end` 迭代器：

```
list<int>::iterator end = remove (coll.begin(), coll.end(),
                                  3);
```

这个 `end` 正是「被修改之群集」经过元素移除动作后，逻辑上的新终点。接下来你便可以拿它当作新的终点使用：

```
copy (coll.begin(), end,
      ostream_iterator<int>(cout," "));
```

另一种可能用法是，藉由测定群集之「逻辑」终点和实际终点间的距离，获得「被删除元素」的数量：

```
cout << "number of removed elements: "
     << distance(end,coll.end()) << endl;
```

在这里，针对迭代器而设计的辅助函数 distance() 发挥了作用。它的功用是传回两个迭代器之间的距离。如果这两个迭代器都是随机存取迭代器（*random access iterators*），你可以使用 operator- 直接计算其距离。不过本例所用的容器是 list，只提供双向迭代器（*bidirectional iterators*）。关于 distance() 的细节，详见 7.3.2 节，p261[10]。

如果真想把那些被删除的元素斩草除根，你必须呼叫该容器的相应成员函数。容器所提供的成员函数 erase()，正适用于此目的。erase() 可以删除「自变量所指示之区间」内的全部元素：

```
coll.erase (end, coll.end());
```

下面是整个程序的完整输出：

```
6 5 4 3 2 1 1 2 3 4 5 6
6 5 4 2 1 1 2 4 5 6
number of removed elements: 2
6 5 4 2 1 1 2 4 5 6
```

如果你需要以单一述句来删除元素，可以如此这般：

```
coll.erase (remove(coll.begin(),coll.end(),
                   3),
            coll.end());
```

为何算法不自己呼叫 erase() 呢？哎，这个问题正好点出 STL 为了获取弹性而付出的代价。透过「以迭代器为接口」，STL 将数据结构和算法分离开来。然而，迭代器只不过是「容器中某一位置」的抽象概念而已。一般来说，迭代器对自己所属的容器一无所知。任何「以迭代器访问容器元素」的算法，都不得（无法）透过迭代器呼叫容器类别所提供的任何成员函数。

这个设计导致一个重要结果：算法的操作对象不一定得是「容器内的全部元素」所形成的区间，而可以是那些元素的子集。甚至算法可运作于一个「并未提供成员函数 erase()」的容器上（array 就是个例子）。所以，为了达成算法的最大弹性，不要求「迭代器必须了解其容器细节」还是很有道理的。

[10] distance() 的定义有些变化。在 STL 旧式版本中，为了使用它，你必须含入 distance.hpp，见 p263。

注意，通常并无必要删除那些「已被移除」的元素。通常，以逻辑终点来取代容器的实际终点，就足以应对。你可以以这个逻辑终点搭配任何算法演出。

5.6.2 更易型算法和关系型容器

更易型算法（指那些会移除 *remove*、重排 *resort*、修改 *modify* 元素的算法）用于关系型容器身上会出问题。关系型容器不能被当作操作目标，原因很简单：如果更易型算法用于关系型容器身上，会改变某位置上的值，进而破坏其已序（*sorted*）特性，那就推翻了关系型容器的基本原则：容器内的元素总是根据某个排序准则自动排序。因此，为了保证这个原则，关系型容器的所有迭代器均被宣告为指向常量（不变量）。如果你更易关系型容器中的元素，会导致编译错误[11]。

注意，这使你无法在关系型容器身上运用移除性（*removing*）算法，因为这类演算法实际上悄悄更易了元素：「被移除元素」被其后的「未被移除元素」覆盖。

现在问题来了，如何从关联容器中删除元素？唔，很简单：呼叫它们的成员函数！每一种关系型容器都提供用以移除元素的成员函数。例如你可以呼叫 `erase()` 来移除元素：

```
// stl/remove3.cpp

#include <iostream>
#include <set>
#include <algorithm>
using namespace std;

int main()
{
   set<int> coll;

   // insert elements from 1 to 9
   for (int i=1; i<=9; ++i) {
        coll.insert(i);
   }

   // print all elements of the collection
   copy (coll.begin(), coll.end(),
         ostream_iterator<int>(cout," "));
```

[11] 糟糕的是，有些系统提供的错误处理能力令人不敢恭维；面对错误，你无法找出原因。有些编译程序甚至连出错的源码都不列出来。希望这种状况在不久的将来获得改善。

```
    cout << endl;

    /* Remove all elements with value 3
    * - algorithm remove() does not work
    * - instead member function erase() works
    */
    int num = coll.erase(3);

    // print number of removed elements
    cout << "number of removed elements: " << num << endl;

    // print all elements of the modified collection
    copy (coll.begin(), coll.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

注意，容器类别提供了多个不同的 `erase()`成员函数。其中一种形式是以「待删除之元素值」为唯一自变量，它会传回被删除的元素个数（第 242 页）— 当然，在禁止元素重复的容器中（例如 sets 和 maps），其传回值永远只能是 0 或 1。

本节范例程序输出如下：

```
1 2 3 4 5 6 7 8 9
number of removed elements: 1
1 2 4 5 6 7 8 9
```

5.6.3 算法 vs. 成员函数

就算我们符合种种条件，得以使用某个算法，那也未必就一定是好。容器本身可能提供功能相似而性能更佳的成员函数。

一个极佳例子便是对 list 的元素呼叫 `remove()`。算法本身并不知道它工作于 list 身上，因此它在任何容器中都一样，做些㆕平八稳的工作：改变元素值，从而重新排列元素。如果它移除第一个元素，后面所有元素就会分别被设给各自的前一个元素。这就违反了 lists 的主要优点 — 藉由修改链结（links）而非实值（values）来安插、移动、移除元素。

为了避免这么糟糕的表现，list 针对所有「更易型」算法提供了一些对应的成员函数。是的，如果你使用 list，你就应该使用这些成员函数。此外请注意，这些成员函数真的移除了「被移除」的元素（译注：而不像先前所说只是某种搬移而已），一如下例所示：

```
// stl/remove4.cpp

#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main()
{
   list<int> coll;

   // insert elements from 6 to 1 and 1 to 6
   for (int i=1; i<=6; ++i) {
        coll.push_front(i);
        coll.push_back(i);
   }

   // remove all elements with value 3
   // - poor performance
   coll.erase (remove(coll.begin(),coll.end(),
                      3),
               coll.end());

   // remove all elements with value 4
   // - good performance
   coll.remove (4);
}
```

如果高效率是你的最高目标，你应该永远优先选用成员函数。问题是你必须先知道，某个容器确实存在有效率上明显突出的成员函数。面对 list 却使用 `remove()` 算法，你决不会收到任何警告讯息或错误通告。然而如果你决定使用成员函数，一旦换用另一种容器，就不得不更动程序代码。第 9 章的算法参考章节中，如果某个成员函数的性能优于某个算法，我会明白指出。

5.7 使用者自定之泛型函数（User-Defined Generic Functions）

STL 乃是一个可扩展的框架（framework）。这意味你可以撰写自己的函数和演算法，处理群集内的元素。当然，这些操作函数本身也可以是泛型的（generic）。

为了在这些操作之中宣告有效的迭代器，你必须使用容器提供的型别，因为每一种容器都有自己的迭代器。为了让我们方便写出真正的泛型函数，每一种容器都提供了一些内部的型别定义。请看下面的例子：

```
// stl/print.hpp

#include <iostream>

/* PRINT_ELEMENTS()
* - prints optional C-string optcstr followed by
* - all elements of the collection coll
* - separated by spaces
*/
template <class T>
inline void PRINT_ELEMENTS (const T& coll, const char* optcstr="")
{
   typename T::const_iterator pos;

   std::cout << optcstr;
   for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
   }
   std::cout << std::endl;
}
```

本例定义出一个泛型函数，可打印一个字符串（也可以不指定），然后打印容器的全部元素。以下宣告式：

```
typename T::const_iterator pos;
```

其中的 pos 被宣告为「传入之容器型别」内的迭代器型别，关键词 typename 在此不可或缺，用以表明 const_iterator 是型别 T 所定义的一个型别，而不是一个型别为 T 的值（请见 p11 对 typename 的介绍）。

除了 iterator 和 const_iterator，容器还提供了其他（内部定义的）型别，帮助你写出泛型函数。例如它提供了元素型别（译注：即所谓 *value type*），以便在元素暂时拷贝场合中派上用场。详见 7.5.1 节, p285。

PRINT_ELEMENTS 的第二自变量是个可有可无的前缀字，用来在打印时放于所有元素之前。你可以这样使用 PRINT_ELEMENTS()：

```
PRINT_ELEMENTS (coll, "all elements: ");
```

我之所以介绍这个函数，因为本书剩余部份会大量运用它来打印容器的所有元素。

5.8 以函数做为算法的自变量

一些算法可以接受用户定义的辅助性函数，由此提高其弹性和能力。这些函数将在算法内部被呼叫。

5.8.1「以函数做为算法的自变量」实例示范

最简单的例子莫过于 for_each()算法了。它针对区间内的每一个元素，呼叫一个由用户指定的函数。下面是个例子：

```
// stl/foreach1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// function that prints the passed argument
void print (int elem)
{
   cout << elem << ' ';
}

int main()
{
   vector<int> coll;

   // insert elements from 1 to 9
   for (int i=1; i<=9; ++i) {
        coll.push_back(i);
   }

   // print all elements
   for_each (coll.begin(), coll.end(),   // range
             print);                     // operation
   cout << endl;
}
```

这里的 for_each()函数针对[coll.begin(), coll.end())区间内的每个元素呼叫 print()函数。输出如下：

```
1 2 3 4 5 6 7 8 9
```

算法以数种态度来面对这些辅助函数：有的视之为可有可无，有的视之为必要。你可以利用它们来指定搜寻准则、排序准则、或定义某种操作，以便将某个容器内的元素转换至另一个容器。

下面是个运用实例：

```
// stl/transform1.cpp

#include <iostream>
#include <vector>
#include <set>
#include <algorithm>
#include "print.hpp"

int square (int value)
{
   return value*value;
}

int main()
{
   std::set<int> coll1;
   std::vector<int> coll2;

   // insert elements from 1 to 9 into coll1
   for (int i=1; i<=9; ++i) {
        coll1.insert(i);
   }
   PRINT_ELEMENTS(coll1,"initialized: ");

   // transform each element from coll1 to coll2
   // - square transformed values
   std::transform                              // source
   (coll1.begin(),coll1.end(),                 // destination
                  std::back_inserter(coll2)    // operation
   ,
                  square);
}
   PRINT_ELEMENTS(coll2,"squared: ");
```

此例之中，square()的作用是将 coll1 内的每一个元素予以平方运算，然后转移到 coll2（图 5.8）。输出如下：

```
initialized: 1 2 3 4 5 6 7 8 9
squared: 1 4 9 16 25 36 49 64 81
```

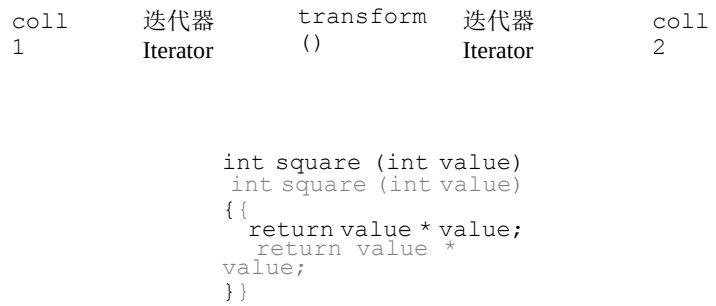

图 5.8 transform() 的运作方式

5.8.2 判断式（Predicates）

算法有一种特殊的辅助函数叫做 predicates（判断式）。所谓 predicates，就是回传布尔值（boolean）的函数。它们通常被用来指定排序准则和搜寻准则。Predicates 可能有一个或两个操作数，视具体情形而定。注意，并非任何传回布尔值的一元函数或二元函数就是合法的 predicate。STL 要求，面对相同的值，predicates 必须得出相同的结果。这条戒律将那些「被呼叫时，会改变自己内部状态」的函数清除出场。细节请见 8.1.4 节, p302。

Unary Predicates（一元判断式）

Unary predicates 会检查唯一自变量的某项特性。典型例子是像下面这样的函数，用来搜寻第一个质数：

```
// stl/prime1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include <cstdlib>     // for abs()
using namespace std;
```

```
// predicate, which returns whether an integer is a prime number
bool isPrime (int number)
{
   // ignore negative sign
   number = abs(number);

   // 0 and 1 are prime numbers
   if (number == 0 || number == 1) {
      return true;
   }

   // find divisor that divides without a remainder
   int divisor;
   for (divisor = number/2; number%divisor != 0; --divisor) {
       ;
   }

   // if no divisor greater than 1 is found, it is a prime number
   return divisor == 1;
}

int main()
{
   list<int> coll;

   // insert elements from 24 to 30
   for (int i=24; i<=30; ++i) {
       coll.push_back(i);
   }

   // search for prime number
   list<int>::iterator pos;
   pos = find_if (coll.begin(), coll.end(), // range
                  isPrime); // predicate
   if (pos != coll.end()) {
      // found
      cout << *pos << " is first prime number found" << endl;
   }
   else {
```

```
        // not found
        cout << "no prime number found" << endl;
    }
}
```

在这个例子中，find_if()算法在给定区间内寻找使「被传入之一元判断式（unary predicate）」运算结果为 true 的第一个元素。本例中的 predicate 是 isPrime()函式，它会检查某数是否为质数。透过它，这个算法可以传回给定区间内的第一个质数。如果没有任何元素能够匹配这个（质数）条件，find_if()算法就传回区间终点（也就是函数的第二自变量）。本例中，24 到 30 之间确实存在一个质数，所以程序输出：

```
29 is first prime number found
```

Binary Predicates（二元判断式）

Binary predicates 的典型用途是，比较两个自变量的特定属性。例如，为了依照你自己的原则对元素排序，你必须以一个简单的 predicate 形式提供这项原则。如果元素本身不支持 operator<，或如果你想使用不同的排序原则，这就派上用场了。

下面这个例子，根据每个人的姓名，对一组元素进行排序：

```
// stl/sort1.cpp

#include <iostream>
#include <string>
#include <deque>
#include <algorithm>
using namespace std;

class Person {
  public:
    string firstname() const;
    string lastname() const;
    ...
};

/* binary function predicate:
* - returns whether a person is less than another person
*/
```

```
bool personSortCriterion (const Person& p1, const Person& p2)
{
    /* a person is less than another person
    * - if the last name is less
    * - if the last name is equal and the first name is less
    */
    return p1.lastname()<p2.lastname() ||
           (!(p2.lastname()<p1.lastname())
&&
           p1.firstname()<p2.firstname());
}

int main()
{
    deque<Person> coll;
    ...
    sort (coll.begin(), coll.end(),  // sort range
          personSortCriterion);      // sort criterion
    ...
}
```

注意，你也可以使用仿函数（functor，或名 function object）来实作一个排序准则。这种作法的优点是，制作出来的准则将是一个型别（type），可用来做为诸如「宣告一个 set，以某种型别为排序准则」之类的事情。详见 8.1.1 节, p294。

5.9 仿函数 （Functors, Function Objects）

译注：本书英文版通篇采用的术语是 **function object**，对应之译名为「函数对象」。此物在 STL 发展初期曾经名为 **functor**，取其音义，我译为「仿函数」。考虑 STL 六大组件之译名整体性，以及「术语最好具备独特性，且不与其他名词混淆」的原则，再考虑上下文阅读的顺畅性，我认为「仿函数」较「函数对象」为佳。为此，本中文版将 function object 全以 functor 取代，并译为「仿函数」。

传递给算法的「函数型自变量」（functional arguments），并不一定得是函数，可以是行为类似函数的对象。这种对象称为 **function object**（函数物件），或称 **functor**（仿函数）。当一般函数使不上劲时，你可以使用仿函数。STL 大量运用仿函数，也提供（预先定义）了一些很有用的仿函数。

5.9.1 什么是仿函数

仿函数是泛型编程强大威力和纯粹抽象概念的又一个例证。你可以说，任何东西，只要其行为像函数，它就是个函数。因此如果你定义了一个对象，行为像函数，它就可以被当作函数来用。

好，那么，什么才算是具备函数行为（也就是行为像个函数）？所谓函数行为，是指可以「使用小括号传递自变量，藉以呼叫某个东西」。例如：

```
function(arg1,arg2); // a function call
```

如果你指望对象也可以如此这般，就必须让它们也有可能被「呼叫」 — 透过小括号的运用和自变量的传递。没错，这是可能的（在 C++ 中，很少有什么是不可能的）。你只需定义 operator()，并给予合适的参数型别：

```
class X {
public:
  // define ''function call'' operator
  return-value operator() (arguments) const;
  ...
};
```

现在，你可以把这个类别的对象当作函数来呼叫了：

```
X fo;
...
fo(arg1,arg2); // call operator () for function object fo
```

上述呼叫等同于：

```
fo.operator()(arg1,arg2); // call operator () for function object
fo
```

下面是个完整例子，是先前 p119 范例的一个仿函数版本，其行为和使用一般函数（非仿函数）完全相同：

```
// stl/foreach2.cpp

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// simple function object that prints the passed argument
class PrintInt {
  public:
    void operator() (int elem) const {
       cout << elem << ' ';
    }
};

int main()
{
  vector<int> coll;
```

```
  // insert elements from 1 to 9
  for (int i=1; i<=9; ++i) {
       coll.push_back(i);
  }

  // print all elements
  for_each (coll.begin(), coll.end(),  // range
            PrintInt())                // operation
  ;
} cout << endl;
```

PrintInt 所做的定义显示，你可以对它的对象呼叫 operator()，并传入一个 int 自变量。至于述句：

```
for_each (coll.begin(), coll.end(),
          PrintInt());
```

其中的表达式：

```
PrintInt()
```

产生出此类别的一个临时对象，当作 for_each()算法的一个自变量。for_each()算法大致如下：

```
namespace std {
  template <class Iterator, class Operation>
  Operation for_each (Iterator act, Iterator end, Operation op)
  {
    while (act != end) {     // as long as not reached the end
       op(*act);             // - call op() for actual element
       ++act;                // - move iterator to the next element
    }
    return op;
  }
}
```

for_each() 使用暂时对象 op（一个仿函数），针对每个元素呼叫 op(*act)。如果第三自变量是个一般函数，就以*act 为自变量呼叫之。如果第三自变量是个仿函数，则以*act 为自变量，呼叫仿函数 op 的 operator()。因此，本例之中，for_each()呼叫：

```
PrintInt::operator()(*act)
```

你也许不以为然，你也许认为仿函数看起来怪异、令人讨厌、甚或毫无意义。的确，它们带来更复杂的程序代码，然而仿函数有其过人之处，比起一般函数，它们有以下优点：

1. 仿函数是 "smart functions"（精灵函数，智能型函数）

「行为类似指针」的对象，我们称为 "smart pointers"。「行为类似函数」的物件呢？同样道理，我们可以称之为 "smart functions"，因为它们的能力可以超越 `operator()`。仿函数可拥有成员函数和成员变量，这意味仿函数拥有状态(state)。事实上，在同一时间里，由某个仿函数所代表的单一函数，可能有不同的状态。这在一般函数中是不可能的。另一个好处是，你可以在执行期（runtime）初始化它们 — 当然必须在它们被使用（被呼叫）之前。

2. 每个仿函数都有自己的型别

一般函数，唯有在它们的标记式（signatures）不同时，才算型别不同。而仿函式即使标记式相同，也可以有不同的型别。事实上，由仿函数定义的每一个函式行为都有其自己的型别。这对于「利用 template 实现泛型程序编写」乃是一个卓越贡献，因为如此一来，我们便可以将函数行为当做 template 参数来运用。这使得不同型别的容器可以使用同类型的仿函数作为排序准则。这可以确保你不会在排序准则不同的群集（collections）之间赋值、合并或比较。你甚至可以设计仿函数继承体系（functors hierarchies），以此完成某些特别事情，例如在一个总体原则下确立某些特殊情况。

3. 仿函数通常比一般函数速度快

就 template 概念而言，由于更多细节在编译期就已确定，所以通常可能进行更好的优化。所以，传入一个仿函数（而非一般函数），可能获得更好的性能。

这一小节的剩余部分，我会给出数个例子，展示仿函数较之于一般函数的优势所在。第 8 章专攻仿函数，有更多例子和细节。尤其该章为你展示「以函数行为作为 template 参数」这一技术带给我们的利益。

假设你需要对群集（collection）中的每个元素加上一个固定值。如果你在编译期便确切知道这个固定数，你可以使用一般函数：

```
void add10 (int& elem)
{
   elem += 10;
}

void f1()
{
   vector<int> coll;
   ...
   for_each (coll.begin(), coll.end(),     // range
             add10);                       // operation
}
```

如果你需要数个不同的固定值，而它们在编译期都已确切，你可以使用 template:

```
template <int theValue>
void add (int& elem)
{
   elem += theValue;
}

void f1()
{
   vector<int> coll;
   ...
   for_each (coll.begin(), coll.end(),     // range
             add<10>);                     // operation
}
```

如果你必须在执行时期才处理这个数值，那可就麻烦了。你必须在函数被呼叫之前先将这个数值传给该函数。这通常会导致产生一些全局变量，「算法的呼叫者」和「算法所呼叫的函数」都会用到它们。真是一团糟。

如果你两次用到该函数，每次加数不同，而都是在执行时期才处理，那么一般函式根本就无能为力。你要嘛传入一个标记（tag），要嘛干脆写两个函数。你是否有过这样的经历：握有一个函数，它有个 static 变量用以记录状态（state），而你需要这个函数在同一时间内有另一个不同状态（state）？于是你只好拷贝整份函式定义，化为两个不同的函数。这正是先前所说的问题。

如果使用仿函数，你就可以写出「更机灵」的函数，遂你所愿。对象可以有自己的状态，可以被正确初始化。下面是一个完整例子[12]：

```
// stl/add1.cpp

#include <iostream>
#include <list>
#include <algorithm>
#include "print.hpp"
using namespace std;

// function object that adds the value with which it is initialized
class AddValue {
```

1 2 辅助函数 PRINT_ELEMENTS() 已于 p118 介绍过。

```
    private:
       int theValue; // the value to add
    public:
       // constructor initializes the value to add
       AddValue(int v) : theValue(v) {
       }

    // the ''function call'' for the element adds the value
    void operator() (int& elem) const {
       elem += theValue;
    }
};

int main()
{
    list<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
       coll.push_back(i);
    }

    PRINT_ELEMENTS(coll,"initialized:
");

    // add value 10 to each element       // range
    for_each (coll.begin(), coll.end(),   // operation
              AddValue(10));
    PRINT_ELEMENTS(coll,"after adding 10: ");

    // add value of first element to each element
    for_each (coll.begin(), coll.end(),    // range
              AddValue(*coll.begin()))     // operation
    ;
    PRINT_ELEMENTS(coll,"after adding first element: ");
}
```

初始化之后，群集内含数值 1 至 9：

```
initialized: 1 2 3 4 5 6 7 8 9
```

第一次呼叫 for_each()，将每个数值加 10：

```
for_each (coll.begin(), coll.end(),   // range
          AddValue(10));              // operation
```

这里，表达式 AddValue(10)生出一个 AddValue 物件，并以 10 为初值。AddValue 建构式将这个值保存在成员 theValue 中。而在 for_each() 之内，针对 coll 的每一个元素呼叫 "()"，实际上就是对传入的那个 AddValue 暂时对象呼叫 operator()，并以容器元素作为自变量。仿函数（AddValue 对象）将每个元素加 10。结果如下：

```
after adding 10: 11 12 13 14 15 16 17 18 19
```

第二次呼叫 for_each() 亦采用相同机能，将第一元素值加到每个元素身上。首先使用第一元素值做为仿函数暂时对象的初值：

```
AddValue(*coll.begin())
```

最后结果如下：

```
after adding first element: 22 23 24 25 26 27 28 29 30
```

p335 有这个例子的改进版，其中 AddValue 仿函数的型别被改为一个 template，可接纳不同的加数。

运用此项技术，先前所说的「一个函数、两个状态」的问题就可以用「两个不同的仿函数」加以解决。例如，你可以宣告两个仿函数，然后各自运用：

```
AddValue addx(x); // function object that adds value x
AddValue addy(y); // function object that adds value y

for_each (coll.begin(),coll.end(), // add value x to each element
          addx);
...
for_each (coll.begin(),coll.end(), // add value y to each element
          addy);
...
for_each (coll.begin(),coll.end(), // add value x to each element
          addx);
```

同样道理，你也可以提供一些成员函数，在仿函数生命期间查询或改变对象状态。

注意，C++ 标准链接库并未限制算法「对着一个容器元素」呼叫仿函数的次数，因此可能导致同一个仿函数有若干副本被传给元素。如果把仿函数当做判断式（predicates）使用，这个问题会惹来一身麻烦。8.1.4 节, p302 讨论了这个问题。

5.9.2 预先定义的仿函数

C++ 标准链接库包含了一些预先定义的仿函数，涵盖许多基础运算。有了它们，很多时候你就不必费心自己去写仿函数了。一个典型的例子是作为排序准则的仿函数。operator< 之预设排序准则乃是 less<>，所以，如果你宣告：

```
set<int> coll;
```

会被扩展为[13]：

```
set<int,less<int> > coll;   // sort elements with <
```

既然如此，想必你能猜到，反向排列这些元素将不是什么难事[14]！

```
set<int,greater<int> > coll; // sort elements with >
```

类似情况，还有许多仿函数用于数值处理。下例是将群集中的全部元素都设为反相（负值）：

```
transform (coll.begin(), coll.end(),  // source
           coll.begin(),              // destination
           negate<int>());            // operation
```

其中表达式：

```
negate<int>()
```

根据预先定义好的 template class negate 生成一个仿函数，将传进来的 int 值设定为负。transform()算法使用此一运算，将第一群集的所有元素处理之后转移到第二群集。如果转移目的地就是自己，那么这段程序代码就是「对群集内的每一个元素取负值」。

同样道理，你也可以对群集内的所有元素求平方（二次方）。

```
// process the square of all elements
transform (coll.begin(), coll.end(),  // first source
           coll.begin(),              // second source
           coll.begin(),              // destination
           multiplies<int>());        // operation
```

这里运用了 transform()算法的另一种形式，以某种特定运算，将两群集内的元素处理后的结果写入第三群集。由于本例的三个群集实际上是同一个，所以其

[13] 有些系统并不支持 default template arguments，那么你只能使用后一种形式。

[14] 注意，两个 ">" 之间必须保留一个空格，否则 ">>" 会被解析为右移（right shift）运算符，因而发生语法错误。

内的每个元素都被计算了平方值，并写进群集内，改写原有值[15]。

透过一些特殊的函数配接器（function adaptors），你还可以将预先定义的仿函数和其他数值组合在一起，或使用特殊状况。下面是一个完整范例：

```
// stl/fo1.cpp

#include <iostream>
#include <set>
#include <deque>
#include <algorithm>
#include "print.hpp"
using namespace std;

int main()
{
   set<int,greater<int> > coll1;
   deque<int> coll2;

   // insert elements from 1 to 9
   for (int i=1; i<=9; ++i) {
        coll1.insert(i);
   }

   PRINT_ELEMENTS(coll1,"initialized: ");

   // transform all elements into coll2 by multiplying 10
   transform (coll1.begin(),coll1.end(),      // source
              back_inserter(coll2),           // destination
              bind2nd(multiplies<int>(),10))  // operation
   ;

   PRINT_ELEMENTS(coll2,"transformed: ");

   // replace value equal to 70 with 42
   replace_if (coll2.begin(),coll2.end(),     // range
               bind2nd(equal_to<int>(),70), // replace criterion
               42);                         // new value

   PRINT_ELEMENTS(coll2,"replaced:
   ");
```

[15] STL 早期版本中，乘法运算的仿函数名为 `times`。但这和某些操作系统（POSIX, X/Open）中用以计算时间的函数名称冲突了，所以后来改为更清楚的名称：`multiplies`。

```
    // remove all elements with values less than 50
    coll2.erase(remove_if(coll2.begin(),coll2.end(), // range
                    bind2nd(less<int>(),50)), // remove
    criterion
                coll2.end());

  }  PRINT_ELEMENTS(coll2,"removed: ");
```

其中的述句：

```
 transform                                      // source
(coll1.begin(),coll1.end(),                     // destination
            back_inserter(coll2),
            bind2nd(multiplies<int>(),10)); // operation
```

将 coll1 内的所有元素乘以 10 后转移（安插）到 coll2 中。这里使用配接器 bind2nd，使得进行 multiples<int>运算时，以源群集（source collection）的元素作为第一自变量，10 作为第二自变量。

配接器 bind2nd 的工作方式如下：transform()期望它自己的第四 自变量是个能接纳单一自变量（也就是容器实际元素）的表达式，然而我们却希望先把该元素乘以 10，再传给 transfor()。所以我们必须构造出一个表达式，接受两个自变量，并以数值 10 作为第二自变量，以此产生一个「只需单自变量」的表达式。bind2nd()正好胜任这项工作。它会把表达式保存起来，把第二自变量当作内部数值也保存起来。当演算法以实际群集元素为自变量，呼叫 bind2nd 时，bind2nd 把该元素当作第一自变量，
把原先保存下来的那个内部数值作为第二自变量，呼叫保留下来的那个表达式，并传回结果。（译注：这段繁复的文字说明可能解释效果不甚差，实际情况（源码运作）请看《STL 源码剖析》第 8 章, 侯捷着, 碁峰出版 2002）

类似情况，以下的：

```
replace_if (coll2.begin(),coll2.end(),    // range
            bind2nd(equal_to<int>(),70)    // replace criterion
,
            42);
```

其中的表达式：

```
bind2nd(equal_to<int>(),70)
```

被用来当作一项准则，判断哪些元素将被 42 替代。bind2nd以 70 作为第二自变量，呼叫二元判断式（binary predicate）equal_to，从而定义出一个一元判断式（unary predicate），处理群集内的每一个元素。

最后一个述句也一样：

```
 bind2nd(less<int>(),50)
```

它被用来判断群集内的哪些元素应当被扫地出门。所有小于 50 的元素都被移除。
程序输出如下：

```
initialized: 9 8 7 6 5 4 3 2 1
transformed: 90 80 70 60 50 40 30 20 10
replaced: 90 80 42 60 50 40 30 20 10
removed: 90 80 60 50
```

此种方式的程序编写，导致函数的组合。有趣的是，所有这些仿函数通常都宣告为 inline。如此一来，你一方面使用类似函数的表示法或抽象性，一方面又能获得出色的效能。

另外还有一些仿函数。某些仿函数可用来呼叫群集内每个元素的成员函数：

```
for_each (coll.begin(), coll.end(),       // range
          mem_fun_ref(&Person::save))     // operation
```

仿函数 `mem_fun_ref` 用来呼叫它所作用的元素的某个成员函数。因此上例就是针对 `coll` 内的每个元素呼叫 `Person::save()`。当然啦，唯有当这些元素的型别是 `Person`，或 `Person` 的衍生类别，以上程序代码才能有效运作。

8.2 节， p305 对于 STL 预先定义的仿函数、函数配接器、以及各类函数组合，有更详尽的讨论，并告诉你如何撰写你自己的仿函数。

5.10 容器内的元素

容器内的元素必须符合特定条件，因为容器乃是以一种特别方式来操作它们。本节讨论这些条件。此外，容器会在内部对其元素进行复制，我也会讨论这种行为的后果。

5.10.1 容器元素的条件

STL 的容器、迭代器、算法，都是 templates，因此可以操作任何型别 — 不论 STL 预先定义好的或用户自行定义的，都可以。然而，由于某些加诸于元素身上的操作行为，某些需求条件也就相应出现了。STL 容器元素必须满足以下三个基本要求：

1. 必须可透过 *copy* 建构式进行复制。副本与原本必须相等（equivalent），亦即所有相等测试（equality test）的结果都必须显示，原本与副本行为一致。

所有容器都会在内部生成一个元素副本，并传回该暂时性副本，因此 *copy* 建构式会被频繁地叫用。所以 *copy* 建构式的性能应该尽可能优化（这虽然不是条件之一，但可视为获得良好效能的诀窍）。如果对象的拷贝必须耗费大量时间，你可以选用「reference 语义」来使用容器，因而避免拷贝任何对象。详见 6.8 节, p222。

2. 必须可以透过 *assignment* 运算符完成赋值动作。容器和算法都使用 *assignment* 运算符，才能以新元素改写（取代）旧元素。
3. 必须可以透过解构式完成销毁动作。当容器元素被移除（*removed*），它在容器内的副本将被销毁。因此解构式绝不能被设计为 `private`。此外，依 C++ 惯例，解构式绝不能丢掷异常（throw exceptions），否则没戏唱了。

这三个条件对任何 class 而言其实都是隐喻成立的。如果某个 class 既没有为上述动作定义特殊版本，也没有定义任何「可能破坏这些动作之健全性」的特殊成员，那么它自然而然也就满足了上述条件。

下面几个条件，也应当获得满足[16]：

- 对序列式容器而言，元素的 *default* 建构式必须可用。
  我们可以在没有给予任何初值的情况下，创建一个非空容器，或增加容器的元素个数。这些元素都将以 *default* 建构式完成。
- 对于某些动作，必须定义 `operator==` 以执行相等测试。如果你有搜寻需求，这一点特别重要。
- 在关系型容器中，元素必须定义出排序准则。预设情况下是 `operator<`，透过仿函数 `less<>` 被叫用。

5.10.2 Value 语意 vs. Reference 语意

所有容器都会建立元素副本，并回传该副本。这意味容器内的元素与你放进去的物件「相等（equal）」但非「同一（identical）」。如果你修改容器中的元素，实际上改变的是副本而不是原先对象。这意味 STL 容器所提供的是「*value* 语意」。它们所容纳的是你所安插的对象值，而不是对象本身。然而实用上你也许需要用到「*reference* 语意」，让容器容纳元素的 reference。

STL 只支援 *value* 语意，不支援 *reference* 语意。这当然是利弊参半。好处是：

- 元素的拷贝很简单。
- 使用 references 时容易导致错误。你必须确保 reference 所指向的对象仍然健在，并需小心对付偶尔出现的循环引用（circular references）状态。

缺点是：

- 「拷贝元素」可能导致不好的效能；有时甚至无法拷贝。

16 在某些老式系统中，即使你未用到这些额外条件，也必须满足它们。例如某些 vector 实作版本无论如何用到元素的 *default* 建构式。另一些实作版本则要求 *comparison*（比较）运算符必须存在。然而根据标准，这些要求是错误的，所以它们终将逐渐被取消。

z 无法在数个不同的容器中管理同一份对象。

实用上你同时需要两种作法。你不但需要一份独立（于原先对象）的拷贝（此乃 *value* 语意），也需要一份代表原数据、俾能相应改变原值的拷贝（此乃 *reference* 语意）。不幸的是，C++ 标准链接库不支持 *reference* 语意。不过我们可以利用 *value* 语意来实现 *reference* 语意。

一个显而易见的方法是以指针作为元素[17]。然而一般指标有些常见问题。例如它们指向的对象也许不复存在，「比较」行为也未必如你所预期，因为实际比较的是指针而非指针所指对象。所以使用一般指针作为容器元素，必须非常谨慎。

好一点的办法是使用某种智能型指标（smart pointers），所谓智能型指标，是一种对象，有着类似指针的接口，但内部作了一些额外检查和处理工作。这里有一个重要的问题：它们需要多么智慧？C++ 标准链接库确实提供了一个智能型指标，名为 `auto_ptr`（详见 4.2 节，p38），乍见之下用于此处似乎颇为合适。然而，你可千万别使用 `auto_ptr`s，因为它们不符合作为容器元素所需的基本要求。当 `auto_ptr` 执行了拷贝（*copy*）或赋值（*assign*）动作后，标的物与原物并不相等：原来的那个 `auto_ptr` 发生了变化，其值并不是被拷贝了，而是被移转了（见 p43 和 p47）。这意味即使对容器中的元素进行排序和打印，也会摧毁它们！所以，千万别在容器内放置 `auto_ptr`s（如果你的 C++ 系统符合标准规范，当你企图将 `auto_ptr` 当作容器元素，你应该会收到错误讯息）。详见 p43。

想要获得适用于 STL 容器的 *reference* 语意，你必须自己写个合适的智能型指标。但请注意：就算你使用带有参用计数（reference counting）功能的智能型指针（译注：可参考《*More Effective C++*》条款 28），也就是那种「当最后一个指向对象的 reference 不复存在后，能够自动摧毁对象」的智能型指针，仍然很麻烦。举个例子，如果你拥有直接存取元素的能力，你就可以更改元素值，而这在关系型容器中却会打破元素顺序关系。你肯定不想那样是吧！6.8 节 p222 更细致地探讨了容器的 *reference* 语意。尤其棒的是该处展示了一种作法，通过「参用计数」智能型指标，实现 STL 容器的 *reference* 语意。

5.11 STL 内部的错误处理和异常处理

错误是无可避免的，可能是程序（程序员）引起的逻辑性错误（logical error），也可能是程序运行时的环境或背景（例如内存不足）所引起的执行期错误（runtime error）。这两种错误都能够被异常机制（exceptions）处理（p15 有一个关于异常的简短介绍）。本节讨论 STL 内部如何处理错误（error）和异常（exceptions）。

[17] C 程序员或许很能认可「以指标实现 reference 语意」的手法。因为在 C 语言中函数自变量只能 *passed by value*（传值），因此需要藉由指标才能实现所谓的 *call by reference*。

5.11.1 错误处理（Error Handling）

STL 的设计原则是效率优先，安全次之。错误检查相当花时间，所以几乎没有。如果你能正确无误地编写程序，自然很好。如果你不行，那就大难临头了。C++ 标准链接库接纳 STL 之前，对于是否应该加入更多的错误检验，曾有过一些讨论。大部分人决定不加入，原因有二：

1. 错误检验会降低效率，而速度始终是程序的总体目标。刚刚提过，良好的效率是 STL 的设计目标之一。
2. 如果你认为安全重于效率，你还是可以如愿：或增加一层包装（wrapper），或使用 STL 特殊版本。但是，一旦错误检验被放进所有基本动作内，再想消除它们以获得高效率，可就没门了。举个例子，如果每一个 *subscript*（下标）运算子都对索引范围进行合法性检验，你就无法撰写不作检验的版本。反过来则可以。

所以，错误检验是可行的，但并不是 STL 的内在条件。

C++ 标准链接库指出，对于 STL 的任何运用，如果违反规则，将会导致未定义的行为。因此，如果索引、迭代器、或区间范围不合法，结果将未有定义。如果你使用的 STL 并非安全版本，就会导致未定义的内存存取，这可能导致难缠的副作用，甚至导致全盘崩溃。从这个意义上说，STL 和 C 指标一样容易引发错误。寻找这样的错误是非常困难的，尤其当你缺乏一个 STL 安全版本时，更是如此。

具体地说，使用 STL，必须满足以下要求：

- 迭代器务必合法而有效。例如你必须在使用它们之前先将它们初始化。注意，迭代器可能会因为其他动作的副效应而变得无效。例如当 vectors 和 deques 发生元素的安插、删除或重新配置时，迭代器可能因此失效。
- 一个迭代器如果指向「逾尾（past-the-end）」位置，它并不指向任何对象，因此不能对它呼叫 `operator*` 或 `operator->`。这一点适用于任何容器的 `end()` 和 `rend()` 所传回的迭代器。
- 区间（range）必须是合法的：
  - 用以指出某个区间的前后两迭代器，必须指向同一个容器。
  - 从第一个迭代器出发，必须可以到达第二个迭代器所指位置。
- 如果涉及的区间不只一个，第二区间及后继各区间必须拥有「至少和第一区间一样多」的元素。
- 覆盖（overwritten）动作中的「标的区间」（destination ranges）必须拥有足够元素，否则就必须采用 *insert iterator*s（插入型迭代器）。

以下实例展示了一些可能的错误：

```
// stl/iterbug1.cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   vector<int> coll1;       // empty collection
   vector<int> coll2;       // empty collection

   /* RUNTIME ERROR:
   * - beginning is behind the end of the range
   */
   vector<int>::iterator pos = coll1.begin();
   reverse (++pos, coll1.end());

   // insert elements from 1 to 9 into coll2
   for (int i=1; i<=9; ++i) {
        coll2.push_back (i);
   }

   /* RUNTIME ERROR:
   * - overwriting nonexisting elements
   */
   copy (coll2.begin(), coll2.end(),  // source
         coll1.begin());              // destination

   /* RUNTIME ERROR:
   * - collections mistaken
   *-begin() and end() mistaken
   */
   copy (coll1.begin(), coll2.end(),  // source
         coll1.end());                // destination
}
```

注意，这些错误发生在执行期间而非编译期间，因而导致未定义的行为。

误用 STL 的方法百百种，STL 没有义务预防你的各种可能不慎。因此，在软件开发阶段使用「安全版本」的 STL 是个好主意。第一个 STL 安全版本由 Cay Horstmann 开发[18]。不幸的是大部分 STL 开发厂商所供应的 STL，都是植基于 STL 最原始版本，其中并未包含错误处理。但是情况正在好转，有一个带有警戒能力的 STL 版本，名为 "STLport"，几乎适用于任何平台，可自 http://www.stlport.org/ 免费下载。

[18] 你可以从 www.horstmann.com/safestl.html 获得一份由 Cay Horstmann 开发的 STL "safe"。

5.11.2 异常处理（Exception Handling）

STL 几乎不检验逻辑错误。所以逻辑问题几乎不会引发 STL 产生异常。事实上 *C++ Standard* 只要求唯一一个函数调用动作必要时直接引发异常：`vector` 和 `deque` 的成员函数 `at()`（它是下标运算符的受验版本）。此外，*C++ Standard* 要求，只有一般的（标准的）异常才可以发生，像是因内存不足而引发的 *bad_alloc* 或是因客户自定之操作行为而引发的异常。

异常何时发生？异常一旦发生对 STL 组件有何影响？在标准化过程中，很长一段时间里，并未对此问题定义出相关的行为规范。事实上每一个异常都会引发未定义的行为。如果执行某项动作的过程中丢掷出异常，那么即使容器马上解构，也会导致未定义行为，例如程序整个崩解。因此如果你需要的是有担保的、确定的行为，STL 无能为力，它甚至不可能正确地将堆栈辗转开解（所谓 stack unwinding）。

如何处理异常，这是标准化过程中最晚的几个讨论议题之一。找到好的解决方法可不容易，而且花了很长时间，因为：

1. 很难确定 C++ 标准链接库究竟应该提供怎样的安全程度。你大概认为应该尽可能提供最佳安全性。例如你可能觉得，对着 `vector` 中的任何位置插入一个新元素，要嘛成功，要嘛应该不生任何效果。然而把后继元素向后移动以空出位置容纳新元素，这种行为通常会导致异常，而且无法复原。如果想要达成上述提出的目标，安插动作就必须把 `vector` 的每一个元素拷贝到新位置去，这对效率是莫大的折损！如果优异效能是设计目标之一（就像 STL），你绝对无法完美处理所有异常状况，你必须在效率和安全之间寻求某种妥协。
2. 还有一种考虑：处理异常的程序代码本身，也会对效能带来负面影响。这与「尽可能获得最佳效能」的设计目标抵触。然而编译程序实作者指出，原则上，异常处理的实作方案应该可以免除任何明显的效能负荷（许多编译程序也确实做到了这一点）。毫无疑问，如果效能没有明显损耗，又能在异常发生时拥有确定、有保障的行为（而非当机了事），那当然比较好。

经过种种讨论，C++ 标准链接库就「异常处理问题」提供了以下基本保证[19]：C++ 标准链接库在面对异常时，保证不会发生资源泄漏（resources leak），也不会与容器的恒常特性（container invariants）发生抵触。

遗憾的是很多时候这还不够，你需要更强的保证，保证当异常被丢出时，进行中的操作不产生任何影响。以异常的观点来看，这种操作可被视为「不可切割的」（**atomic**）。借用数据库领域的一个术语，这些操作支持所谓「交付或回复，二择一」（**commit-or-rollback**）行为，又称为「安全交易行为」（**transaction safe**）。

[19] 特别感谢 Dave Abrahams 和 Greg Colvin 对于 C++ 标准链接库的异常安全问题所的贡献，以及在这个主题上对我的帮助。

考虑到这种强烈需求，C++ 标准链接库如今做出以下保证：

- 对于所有「以节点为构造基础」（node-based）的容器如 lists, sets, multisets, maps 和 multimaps，如果节点建构失败，容器保持不变。移除节点的动作保证不会失败（当然你得保证解构式不得丢出异常）。然而，如果是对关系型容器插入多个元素，为保证已序性（*sorted*），失败时无法完全恢复原状。所有对关系型容器「插入单一元素」的操作，支持 **commit-or-rollback** 行为。也就是说，要不成功，要不没有任何影响。此外，所有擦拭（erase）操作，无论是针对单一元素或针对多重元素，肯定会成功。

  面对 lists，就算同时插入多个元素，这个操作也是属于「安全交易行为」（**transaction-safe**）。事实上 list 的所有操作，除了 remove(), remove_if(), merge(), sort() 和 unique() 之外，要不成功，要不没有任何影响（也就是 **commit-or-rollback**）。至于上述各函数，C++ 标准链接库也提供了有条件的保证（见 p172）。所以如果你需要一个 **transaction-safe** 容器，就用 list 吧。

- 所有「以 array 为构造基础」（array-based）的容器如 vectors 和 deques，安插元素时如果失败，都不可能做到完全回复。要达到完全回复，就必须在安插动作之前拷贝所有（安插点之后的）后继元素。而且为了实现拷贝动作的完全回复性，需要耗费大量时间。不过由于 push 和 pop 这两个动作在容器尾端执行，不需拷贝任何既有元素，所以万一发生异常，这两个动作可以保证容器会回复原状。此外，如果元素的型别能够保证拷贝动作（也就是 *copy* 建构式和 *assignment* 运算符）不丢出异常，则所有加诸于该种元素身上的操作，都能够保证「要不成功，要不毫无影响」的行为。

6.10.10 节， p248 有一份详细整理，让你对「异常发生时，拥有较强烈的保证」的各种容器操作，有一份了解。

注意，所有这些保证都有一个前提：解构式不得丢掷异常（C++ 中通常如此）。C++ 标准链接库做了这个承诺，身为应用程序员的你，也得做出相同承诺。

如果你需要具备「完全 **commit-or-rollback** 能力」的容器，你应当使用 list（但不要呼叫它的 sort 和 unique），或使用任何关系型容器（但不要对它安插多个元素）。当你使用它们，可以确保数据不会损失，也确保不会在任何「修改动作」之前先拷贝元素 — 要知道，对一个容器而言，拷贝动作极可能代价高昂。

如果你不使用「以节点为构造基础」（node-based）的容器，但又希望获得「完全 **commit-or-rollback** 能力」，只好自己动手为每一个关键操作提供一份包装（wrapper）了。举个例子，以下函数对任何容器而言，几乎都可以安全地将元素安插于某个特定位置上：

```
template <class T, class Cont, class Iter>
void insert (Cont& coll, const Iter& pos, const T& value)
{
```

```
    Cont tmp(coll);          // copy container and all elements
    tmp.insert(pos,value)    // modify the copy
    coll.swap(tmp)       // use copy (in case no exception was thrown)
}   ;
```

注意我的用词，我说「几乎」，因为这个函数仍然未臻完美。这是因为，当 `swap()` 针对关联性容器复制「比较准则（comparison criterion）」时如果发生异常，那么 `swap()` 便会丢掷异常。这下你明白了吧，想完美处理异常是多么不容易！

5.12 扩展 STL

STL 被设计成一个框架（framework），可以向任何方向扩展。你可以提供自己的容器、迭代器、算法、仿函数…，只要你满足条件即可。事实上很多有用的扩展都没有出现在 C++ 标准链接库中。不能非难他们，C++ 标准委员会必须在某个时刻停止加入新特性，将精力集中于现有特性的完善上，否则标准化工作永无完结之日。STL 遗漏的最重要组件是 hash table（容器类）。这完全是因为它太晚被提出之故。新的标准链接库很可能包含数种不同形式的 hash table。大部分 C++ 标准链接库实作版本已经提供了 hash table，只可惜彼此之间有些差异。详见 6.7.3 节, p221。

另一些有用的扩展是额外的仿函数（8.3 节, p313）、迭代器（7.5.2 节, p288）、容器（6.7 节, p217）和算法（7.5.1 节, p285）。

6

STL 容器

STL container

本章延续第 5 章以来的讨论，详细讲解 STL 容器。首先对所有容器共通的能力和操作进行巡礼，然后详细讲解每一个容器，包括内部数据结构、操作（operations）、性能，以及各种操作的运用。如果某些操作值得深述，我还会给出相应的实例。每个容器的讲解都以一个典型运用实例作为结束。本章还讨论一个有趣的问题：各种容器的使用时机。比较各种容器的能力、优点、缺点之后，你便会了解如何选择最符合需求的容器。最后，本章详细介绍了每一个容器的所有成员。这一部分可视为参考手册，你可以在其中找到容器接口的细节和容器操作的确切标记式（signature）。必要的时候我会列出交叉索引，帮助你了解相似或互补的算法。

C++ 标准链接库还提供了一些特殊的容器类别 — 所谓的「容器配接器」（container adapters，包括 stack, queue, priority queue），以及 bitsets 和 valarrays。这些容器都有一些特殊界面，并不满足 STL 容器的一般要求，所以本书把它们放在其他章节讲解[1]。容器配接器和 bitsets 安排在第 10 章，valarrays 安排在 12.2 节, p547。

[1] 从历史沿革来说，容器配接器是 STL 的一部分。然而，从概念角度观之，它们并属不 STL framework，它们只不过是「使用」STL。

6.1 容器的共通能力和共通操作

6.1.1 容器的共通能力

本节讲述 STL 容器的共通能力。其中大部分都是必要条件，所有 STL 容器都必须满足那些条件。三个最核心的能力是：

1. 所有容器提供的都是「*value* 语意」而非「*reference* 语意」。容器进行元素的安插动作时，内部实施的是拷贝动作，置于容器内。因此 STL 容器的每一个元素都必须能够被拷贝。如果你意图存放的对象不具有 public *copy* 建构式，或者你要的不是副本（例如你要的是被多个容器共同容纳的元素），那么容器元素就只能是指针（指向对象）。5.10.2 节, p135 对此有所描述。
2. 总体而言，所有元素形成一个次序（order）。也就是说，你可以依相同次序一次或多次巡访每个元素。每个容器都提供「可传回迭代器」的函数，运用那些迭代器你就可以巡访元素。这是 STL 算法赖以生存的关键界面。
3. 一般而言，各项操作并非绝对安全。呼叫者必须确保传给操作函数的自变量符合需求。违反这些需求（例如使用非法索引）会导致未定义的行为。通常 STL 自己不会丢掷异常。如果 STL 容器所呼叫的使用者自定操作（user-defined operations）掷出异常，会导致各不相同的行为。参见 5.11.2 节, p139。

6.1.2 容器的共通操作

以下操作为所有容器共有，它们均满足上述核心能力。表 6.1 列出这些操作。后续各小节分别探讨这些共通操作。

初始化（initialization）

每个容器类别都提供了一个 *default* 建构式，一个 *copy* 建构式和一个解构式。你可以以某个已知区间的内容做为容器初值 — 是的，负责此一行为的建构式专门用来从另一个容器或 array 或标准输入设备（standard input）得到元素并建构出容器。这些建构式都是 member templates（p11），所以如果提供了从「来源端」到「标的端」的元素型别自动转换，那么不光是容器型别可以不同，元素型别也可以不同[2]。下面是个实例：

[2] 如果系统本身不支持 member templates，那就只能接受相同型别。你可以换用算法 copy，参见 p188 范例。

操作	效果
ContType c	产生一个未含任何元素的空容器
ContType c1(c2)	产生一个同型容器
ContType c(beg,end)	复制 [beg;end) 区间内的元素，做为容器初值
c.~ContType()	删除所有元素，释放内存
c.size()	传回容器中的元素数量
c.empty()	判断容器是否为空（相当于 size()==0, 但可能更快）
c.max_size()	传回元素的最大可能数量
c1 == c2	判断是否 c1 等于 c2
c1 != c2	判断是否 c1 不等于 c2，相当于 !(c1 == c2)
c1 < c2	判断是否 c1 小于 c2
c1 > c2	判断是否 c1 大于 c2，相当于 c2 < c1
c1 <= c2	判断是否 c1 小于等于 c2，相当于 !(c2 < c1)
c1 >= c2	判断是否 c1 大于等于 c2，相当于 !(c1 < c2)
c1 = c2	将 c2 的所有元素指派给 c1
c1.swap(c2)	交换 c1 和 c2 的数据
swap(c1,c2)	同上，是个全局函数
c.begin()	传回一个迭代器，指向第一元素
c.end()	传回一个迭代器，指向最后元素的下一位置
c.rbegin()	传回一个逆向迭代器，指向逆向巡访时的第一元素
c.rend()	传回一个逆向迭代器，指向逆向巡访时的最后元素的下一位置
c.insert(pos,elem)	将 elem 的一份副本安插于 pos 处。回返值和 pos 的意义并不相同。
c.erase(beg,end)	移除 [beg;end) 区间内的所有元素。某些容器会传回未被移除的第一个接续元素。
c.clear()	移除所有元素，令容器为空
c.get_allocator()	传回容器的内存模型（memory model）

表 6.1 容器类别（Container Classes）的共通操作函数

z 以另一个容器的元素为初值，完成初始化动作：

```
std::list<int> l; // l is a linked list of ints
...
// copy all elements of the list as floats into a vector
std::vector<float> c(l.begin(),l.end());
```

z 以某个 array 的元素为初值，完成初始化动作：

```
int array[] = { 2, 3, 17, 33, 45, 77 };
...
// copy all elements of the array into a set
std::set<int> c(array,array+sizeof(array)/sizeof(array[0]));
```

z 以标准输入设备完成初始化动作：

```
// read all integer elements of the deque from standard input
std::deque<int> c((std::istream_iterator<int>(std::cin)),
                  (std::istream_iterator<int>()));
```

注意，不要遗漏了涵括「初始化自变量」的那对「多余的」括号，否则这个算式的意义会迥然不同，肯定让你匪夷所思，你会得到一堆奇怪的警告或错误。看看不写括号的情形：

```
std::deque<int> c(std::istream_iterator<int>(std::cin),
                  std::istream_iterator<int>());
```

这种情况下 c 被视为一个函数，回返值是 deque<int>，第一参数的型别是 istream_iterator<int>，参数名为 cin，第二参数无名称，型别是「一个函式，不接受任何自变量，回返值型别为 istream_iterator<int>」。以上结构不论作为宣告式或算式，语法上都正确。根据 C++ 规则它被视为宣告式。只要加上一对括号，便可使自变量

```
(std::istream_iterator<int>(std::cin))
```

不再符合宣告式语法[3]，也就消除了歧义。

原则上还有一些操作，可支持从另一区间获取数据、指派、插入元素。不过这些操作的确切接口在各容器中彼此不同，有不同的附加自变量。

与大小相关的操作函数（Size Operations）

所有容器都提供了三个和大小相关的操作函数：

1. **size()**
   传回当前容器的元素数量。

2. **empty()**
   这是 size()==0 算式的一个快捷形式。empty()的实作可能比 size()==0 更有效率，所以你应该尽可能使用它。

3. **max_size()**
   传回容器所能容纳的最大元素数量。其值因实作版本的不同而异。例如 vector 通常保有一个内存区块的全部元素，所以在 PCs 上可能会有相关限定。max_size()通常传回索引型别的最大值。

[3] 感谢 EDG 的 John H. Spicer 给予的说明。

比较（Comparisons）

包括常用的比较运算符==, !=, <, <=, >, >=。它们的定义依据以下三个规则：

1. 比较动作的两端（两个容器）必须属于同一型别。

2. 如果两个容器的所有元素依序相等，那么这两个容器相等。采用 `operator==` 检查元素是否相等。

3. 采用字典式（lexicographical）顺序比较原则来判断某个容器是否小于另一个容器。参见 p360。

比较两个不同型别的容器，必须使用「比较」算法，参见 9.5.4 节, p356。

指派（Assignments）和 swap()

当你对着容器指派元素时，源容器的所有元素被拷贝到标的容器内，后者原本的所有元素全被移除。所以，容器的指派（赋值）动作代价比较高昂。

如果两个容器型别相同，而且拷贝后源容器不再被使用，那么我们可以使用一个简单的优化方法：`swap()`。`swap()`的性能比上述优异得多，因为它只交换容器的内部数据。事实上它只交换某些内部指针（指向实际数据如元素、配置器、排序准则 — 如果有的话），所以时间复杂度是「常数」，不像实际指派（赋值）动作的复杂度为「线性」。

6.2 Vectors

vector 模塑出一个 dynamic array。因此，它本身是「将元素置于 dynamic array 中加以管理」的一个抽象概念（图 6.1）。不过请注意，*C++ Standard* 并未要求必须以 dynamic array 实作 vector，只是规定了相应条件和操作复杂度。

图 6.1 vector 的结构

使用 vector 之前，必须含入头文件 `<vector>`[4]

```
#include <vector>
```

其中，型别 `vector` 是一个定义于 namespace `std` 内的 template：

```
namespace std {
  template <class T,
            class Allocator = allocator<T> >
  class vector;
}
```

vector 的元素可以是任意型别 `T`，但必须具备 *assignable* 和 *copyable* 两个性质。第二个 template 参数可有可无，用来定义内存模型（memory model，参见 15 章）。默认的内存模型是 C++ 标准链接库提供的 `allocator`。[5]

6.2.1 Vectors 的能力

vectors 将其元素复制到内部的 dynamic array 中。元素之间总是存在某种顺序，所以 vectors 是一种有序群集（ordered collection）。vector 支持随机存取，因此只要知道位置，你可以在常数时间内存取任何一个元素。vector 的迭代器是随机存取迭代器，所以对任何一个 STL 算法都可以奏效。

在末端附加或删除元素时，vector 的性能相当好。可是如果你在前端或中部安插或删除元素，性能就不怎么样了，因为动作点之后的每一个元素都必须移到另一个位置，而每一次移动都得呼叫 *assignment*（赋值）运算符。

[4] 早期的 STL 中，vectors 的定义头文件是 `<vector.h>`

[5] 在不支持 default template parameters 的系统中，第二参数通常就没有了。

大小（Size）和容量（Capacity）

vector 优异性能的密诀之一，就是配置比其所容纳的元素所需的更多内存。为了能够高效运用 vectors，你应该了解大小和容量之间的关系。

vectors 之中用于操作大小的函数有 `size()`,`empty()`,`max_size()`（ 6.1.2 节，p144）。另一个与大小有关的函数是 `capacity()`，传回 vector 实际能够容纳的元素数量。如果超越这个数量，vector 就有必要重新配置内部存储器。

vector 的容量之所以很重要，有以下两个原因：

1. 一旦记忆体重新配置，和 vector 元素相关的所有 references、pointers、iterators 都会失效。

2. 记忆体重新配置很耗时间。

所以如果你的程序管理了和 vector 元素相关的 references、pointers、iterators，或如果执行速度对你而言至关重要，那么就必须考虑容量问题。

你可以使用 `reserve()`保留适当容量，避免一再重新配置内存。如此一来，只要保留的容量尚有余裕，就不必担心 references 失效。

```
std::vector<int> v;   // create an empty vector
v.reserve(80);        // reserve memory for 80 elements
```

另一种避免重新配置内存的方法是，初始化期间就向建构式传递附加自变量，建构出足够的空间。如果你的自变量是个数值，它将成为 vector 的起始大小。

```
std::vector<T> v(5);    // creates a vector and initializes it with five values
                        // (calls five times the default constructor of type T)
```

当然，要获得这种能力，此种元素型别必须提供一个 *default* 建构式。请注意，如果型别很复杂，就算提供了 *default* 建构式，初始化动作也很耗时。如果你这么做只不过是为了保留足够的内存，那倒不如使用 `reserve()`。

vectors 的容量，概念上和 strings 类似（参见 11.2.5 节，p485）。不过有一个大不同点：vector 不能使用 `reserve()`来缩减容量，这一点和 strings 不同。如果呼叫 `reserve()`所给的自变量比当前 vector 的容量还小，不会引发任何举动。此外，如何达到时间和空间的最佳效率，系由实作版本决定。因此具体实作版本中，容量的增长幅度可能比你我料想的还大。事实上为了防止内存破碎，在许多实作方案中即使你不呼叫 `reserve()`，当你第一次安插元素时也会一口气分配整块记忆体（例如 2K）。如果你有一大堆 vectors，每个 vector 的实际元素却寥寥无几，那么浪费的内存相当可观。

既然 vectors 的容量不会缩减，我们便可确定，即使删除元素，其 references、pointers、iterators 也会继续有效，继续指向动作发生前的位置。然而安插动作却可能使 references、pointers、iterators 失效（译注：因为安插可能导致 vector 重新配置）。

这里有一个间接缩减 vector 容量的小窍门。注意，两个 vectors 交换内容后，两者的容量也会互换，因此下面的例子虽然保留了元素，却缩减了容量：

```
template <class T>
void shrinkCapacity(std::vector<T>& v)
{
   std::vector<T> tmp(v); // copy elements into a new vector
   v.swap(tmp);           // swap internal vector data
}
```

你甚至可以利用下面的述句直接缩减容量：

```
// shrink capacity of vector v for type T
std::vector<T>(v).swap(v);
```

不过请注意，swap()之后原先所有的 references、pointers、iterators 都换了指涉对象；它们仍然指向原本位置。换句话说上述的 shrinkCapacity() 使所有 references、pointers、iterators 失效。

6.2.2 Vector 的操作函数

建构、拷贝和解构

表 6.2 列出 vectors 的所有建构式和解构式。你可以在建构时提供元素，也可以不。如果只指定大小，系统便会呼叫元素的 *default* 建构式一一制造新元素。记住，即使对基本型别如 int，显式呼叫 *default* 建构式进行初始化，也是一样可行（这个特性在 p14 介绍过）。请参考 6.1.2 节, p144 对于初始化的介绍。

操作	效果
`vector<Elem> c`	产生一个空 vector，其中没有任何元素
`vector<Elem> c1(c2)`	产生另一个同型 vector 的副本（所有元素都被拷贝）
`vector<Elem> c(n)`	利用元素的 *default*建构式生成一个大小为 n 的 vector
`vector<Elem>c(n,elem)`	产生一个大小为 n 的 vector，每个元素值都是 elem
`vector<Elem> c(beg,end)`	产生一个 vector，以区间[beg;end)做为元素初值
`c.~vector<Elem>()`	销毁所有元素，并释放内存

表 6.2 Vectors 的建构式和解构式

[6] 你（或你的编译程序）大概会认为这个述句实在荒谬，居然针对一个暂时对象呼叫个—non-const 成员函数。然而标准 C++确实允许我们这么做。

非变动性操作（Nonmodifying Operations）

表 6.3 列出 vectors 的所有非变动性操作。参见 6.1.2 节, p14 附注和 6.2.1 节, p149。

操作	效果
`c.size()`	传回当前的元素数量
`c.empty()`	判断大小是否为零。等同于 `size()==0`，但可能更快
`c.max_size()`	传回可容纳的元素最大数量
`capacity()`	传回重新分配空间前所能容纳的元素最大数量
`reserve()`	如果容量不足，扩大之 [7]
`c1 == c2`	判断 `c1` 是否等于 `c2`
`c1 != c2`	判断 `c1` 是否不等于 `c2`，等同于 `!(c1==c2)`。
`c1 < c2`	判断 `c1` 是否小于 `c2`
`c1 > c2`	判断 `c1` 是否大于 `c2`，等同于 `c2<c1`。
`c1 <= c2`	判断 `c1` 是否小于等于 `c2`，等同于 `!(c2<c1)`。
`c1 >= c2`	判断 `c1` 是否大于等于 `c2`，等同于 `!(c1<c2)`。

表 6.3 Vectors 的非变动性操作

赋值（指派, Assignments）

操作	效果
`c1 = c2`	将 `c2` 的全部元素指派给 `c1`
`c.assign(n,elem)`	复制 `n` 个 `elem`，指派给 `c`
`c.assign(beg,end)`	将区间`[beg;end)`内的元素指派给 `c`
`c1.swap(c2)`	将 `c1` 和 `c2` 元素互换
`swap(c1,c2)`	同上。此为全局函数。

表 6.4 Vectors 的指派（赋值）操作

表 6.4列出「将新元素指派给 vectors，并将旧元素全部移除」的方法。一系列`assign()`函数和建构式一一对应。你可以采用不同的内容指派方式（来自容器、array 和标准输入设备），这和 p144 的建构式情况类似。所有指派（赋值）操作都可能会呼叫元素型别的 *default* 建构式、*copy* 建构式、*assignment* 运算符和/或解构式，视元素数量的变化而定。例如：

```
std::list<Elem> l;
std::vector<Elem> coll;
```

[7] `reserve()`的确会更易（变动，*modify*）vector。因为它造成所有 references、pointers 和 iterators 失效。但是从逻辑内容来说，容器并没有变化，所以还是把它列在这里。

```
...
// make coll be a copy of the contents of l
coll.assign(l.begin(),l.end());
```

元素存取（Element Access）

表 6.5 列出用来直接存取 vector 元素的全部操作函数。按照 C 和 C++ 的惯例，第一元素的索引为 0，最后元素的索引为 size()-1。所以第 n 个元素的索引是 n-1。对于 non-const vectors，这些函数都传回元素的 reference。也就是说你可以使用这些操作函数来更改元素内容（如果没有其他妨碍因素的话）。

操作	效果
c.at(idx)	传回索引 idx 所标示的元素。如果 idx 越界，丢掷 *out_of_range*
c[idx]	传回索引 idx 所标示的元素。不进行范围检查。
c.front()	传回第一元素。不检查第一个元素是否存在。
c.back()	传回最后一个元素。不检查最后一个元素是否存在。

表 6.5 直接用来存取 vectors 元素的各项操作

对呼叫者来说，最重要的事情莫过于搞清楚这些操作是否进行范围检查。只有 at() 会那么做。如果索引越界，at()会丢掷一个 *out_of_range* 异常（详 3.3 节, p25）。其他函数都不作检查。如果发生越界错误，会引发未定义行为。对着一个空 vector 呼叫 operator[],front(),back()，都会引发未定义行为。

```
std::vector<Elem> coll;    // empty!

coll[5] = elem;            // RUNTIME ERROR  Î undefined behavior
std::cout << coll.front(); // RUNTIME ERROR   Î undefined behavior
```

所以，呼叫 operator[]时，你必须心里有数，确定索引有效；呼叫 front()或 back()
时必须确定容器不空：

```
std::vector<Elem> coll;   // empty!
if (coll.size() > 5) {
   coll[5] = elem;        // OK
}
if (!coll.empty()) {
   cout << coll.front();  // OK
}
coll.at(5) = elem;        // throws out_of_range exception
```

迭代器相关函数（Iterator Functions）

vectors 提供了一些常规函数来获取迭代器，如表 6.6。vector 迭代器是 *random access iterator*s（随机存取迭代器；关于迭代器分类详见 7.2 节， p251），因此从理论上讲，你可以藉此迭代器操作所有 STL 算法。

操作	效果
`c.begin()`	传回一个随机存取迭代器，指向第一元素。
`c.end()`	传回一个随机存取迭代器，指向最后元素的下一位置。
`c.rbegin()`	传回一个逆向迭代器，指向逆向迭代的第一元素。
`c.rend()`	传回一个逆向迭代器，指向逆向迭代的最后元素的下一位置。

表 6.6 Vectors 的迭代器相关函数

这些迭代器的确切型别由实作版本决定。对 vectors 来说，通常就是一般指标。一般指标就是随机存取迭代器，而 vector 内部结构通常也就是个 array，所以指标行为可以适用。不过你可不能仰仗这一点。例如也许有个 STL 安全版本，对所有区间范围和其他潜在错误实施检查，那么其 vector 迭代器可能就是个辅助类别。7.2.6 节 , p258 展示了「以指标实作迭代器」和「以类别实作迭代器」之间的差异所引起的麻烦问题。

vector 迭代器持续有效，除非发生两种情况：(1) 用户在一个较小索引位置上安插或移除元素，(2) 由于容量变化而引起记忆体重新分配（详见 6.2.1 节, p149）。

安插（insert）和移除（remove）元素

表 6.7 列出 vector 元素的安插、移除操作函数。依 STL 惯例，你必须保证传入的自变量合法：(1) 迭代器必须指向一个合法位置、(2) 区间的起始位置不能在结束位置之后、(3) 绝不能从空容器中移除元素。

关于性能，以下情况你可以预期安插动作和移除动作会比较快些：

- 在容器尾部安插或移除元素
- 容量一开始就够大
- 安插多个元素时，「呼叫一次」当然比「呼叫多次」来得快

安插元素和移除元素，都会使「作用点」之后的各元素的 references、pointers、iterators 失效。如果安插动作甚至引发记忆体重新分配，那么该容器身上的所有 references、pointers、iterators 都会失效。

操作	效果
`c.insert(pos,elem)`	在 pos 位置上插入一个 elem 副本，并传回新元素位置
`c.insert(pos,n,elem)`	在 pos 位置上插入 n 个 elem 副本。无回传值。
`c.insert(pos,beg,end)`	在 pos 位置上插入区间[beg;end)内的所有元素的副本。无回传值。
`c.push_back(elem)`	在尾部添加一个 elem 副本。
`c.pop_back()`	移除最后一个元素（但不回传）。
`c.erase(pos)`	移除 pos 位置上的元素，传回下一元素的位置。
`c.erase(beg,end)`	移除[beg, end)区间内的所有元素，传回下一元素的位置。
`c.resize(num)`	将元素数量改为 num（如果 size()变大了，多出来的新元素都需以 *default* 建构式建构完成）
`c.resize(num,elem)`	将元素数量改为 num（如果 size()变大了，多出来的新元素都是 elem 的副本）
`c.clear()`	移除所有元素，将容器清空。

表 6.7 vector 的安插、移除相关操作

vectors 并未提供任何函数可以直接移除「与某值相等」的所有元素。这是算法发挥威力的时候。以下述句可将所有其值为 val 的元素移除：

```
std::vector<Elem> coll;
...
// remove all elements with value val
coll.erase(remove(coll.begin(),coll.end(),
                  val),
           coll.end());
```

具体解释详见 5.6.1 节, p111。

如果只是要移除「与某值相等」的第一个元素，可以这么做：

```
std::vector<Elem> coll;
...
// remove first element with value val
std::vector<Elem>::iterator pos;
pos = find(coll.begin(),coll.end(),
           val);
if (pos != coll.end()) {
   coll.erase(pos);
}
```

6.2.3 将 Vectors 当作一般 Arrays 使用

C++ 标准链接库并未明确要求 vector 的元素必须分布于连续空间中。但是一份标准规格缺陷报告显示，这个缺点将获得弥补，标准规格书中将明确保证上述论点。如此一来你可以确定，对于 vector v 中任意一个合法索引 i，以下算式肯定为 true：

```
&v[i] == &v[0] + i
```

保证了这一点，就可推导出一系列重要结果。简单地说，任何地点只要你需要一个 dynamic array，你就可以使用 vector。例如你可以利用 vector 来存放常规的 C 字符串（型别为 char* 或 const char*）：

```
std::vector<char> v;       // create vector as dynamic array of chars

v.resize(41);         //make room for 41 characters (including '\0')
strcpy(&v[0],"hello,          // copy a C-string into the vector
world!");
printf("%s\n", &v[0]); // print contents of the vector as C-string
```

不过，这么运用 vector 你可得小心（和使用 dynamic array 一样小心），例如你必须确保上述 vector 的大小足以容纳所有数据，如果你用的是 C-String，记住最后有 '\0' 元素。这个例子说明，不管出于什么原因（例如为了和既有的 C 链接库库溫打交道），只要你需要一个元素型别为 T 的 array，就可以采用 vector<T>，然后传递第一元素的地址给它。

注意，千万不要把迭代器当作第一元素的地址来传递。vector 迭代器是由实作版本定义的，也许并不是个一般指标。

```
printf("%s\n", v.begin()); // ERROR (might work, but not portable)
printf("%s\n", &v[0]);     // OK
```

6.2.4 异常处理（Exception Handling）

vector 只支持最低限度的逻辑错误检查。*subscript*（下标）运算符的安全版本 at()，是唯一被标准规格书要求可能丢掷异常的一个函数（p152）。此外标准规格书也规定，只有一般标准异常（例如内存不足时丢掷 *bad_alloc*），或用户自定操作函数的异常，才可能发生。

如果 vector 呼叫的函数（元素型别所提供的函数，或使用者提供的函数）掷出异常，C++ 标准链接库作出如下保证：

1. 如果 push_back()安插元素时发生异常，该函数不生效用。
2. 如果元素的拷贝动作（包括 *copy* 建构式和 *assignment* 运算符）不丢掷异常，那么 insert()要嘛成功，要嘛不生效用。
3. pop_back()决不会丢掷任何异常。

4. 如果元素拷贝动作（包括 *copy*建构式和 *assignment*运算符）不丢掷异常，`erase()` 和 `clear()`就不丢掷异常。
5. `swap()`不丢掷异常。
6. 如果元素拷贝动作（包括 *copy* 建构式和 *assignment* 运算符）绝对不会丢掷异常，那么所有操作不是成功，就是不生效用。这类元素可被称为 **POD**（plain old data，简朴的老式资料）。POD 泛指那些无 C++ 特性的型别，例如 C structure 便是。

所有这些保证都基于一个条件：解构式不得丢掷异常。参见 5.11.2 节, p139 对于 STL 异常处理的一般性讨论。6.10.10 节, p249 列出对于异常给予特别保证的所有容器操作函数。

6.2.5 Vectors 运用实例

下面例子展示了 vectors 的简单用法：

```
// cont/vector1.cpp

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  // create empty vector for strings
  vector<string> sentence;

  // reserve memory for five elements to avoid reallocation
  sentence.reserve(5);

  // append some elements
  sentence.push_back("Hello,");
  sentence.push_back("how");
  sentence.push_back("are");
  sentence.push_back("you");
  sentence.push_back("?");

  // print elements separated with spaces
  copy (sentence.begin(), sentence.end(),
        ostream_iterator<string>(cout," "));
  cout << endl;
```

```
  // print  "technical data"
  cout << " max_size(): " << sentence.max_size() << endl;
  cout << " size(): " << sentence.size() << endl;
  cout << " capacity(): " << sentence.capacity() << endl;

  // swap second and fourth element
  swap (sentence[1], sentence[3]);

  // insert element "always" before element "?"
  sentence.insert
  (find(sentence.begin(),sentence.end(),"?"),
                 "always");

  // assign "!" to the last element
  sentence.back() = "!";

  // print elements separated with spaces
  copy (sentence.begin(), sentence.end(),
        ostream_iterator<string>(cout," "));
  cout << endl;

  // print  "technical data" again
  cout << " max_size(): " << sentence.max_size() << endl;
  cout << " size(): " << sentence.size() << endl;
} cout << " capacity(): " << sentence.capacity() << endl;
```

程序的输出可能像这样：

```
Hello, how are you ?
  max_size(): 268435455
  size(): 5
  capacity(): 5
Hello, you are how always !
  max_size(): 268435455
  size(): 6
  capacity(): 10
```

注意我说「可能」。是的，`max_size()`和 `capacity()`的结果由实作版本决定。从这个例子中你可以看到，当容量不足时，此一实作版本将容量扩充一倍。

6.2.6 Class vector<bool>

C++ 标准链接库专门针对元素型别为 bool 的 vector 设计了一个特殊版本，目的是获取一个优化的 vector。其耗用空间远远小于以一般 vector 实作出来的 bool vector。一般 vector 的实作版本会为每个元素至少分配一个 byte 空间，而 vector<bool>特殊版本内部只用一个 bit 来存储一个元素。所以通常小 8 倍之多。不过这里有个小麻烦：C++ 的最小可寻址值通常以 byte 为单位。所以上述的 vector 特殊版本需针对 references 和 iterators 做特殊考虑。

考虑结果是，vector<bool> 无法满足其 他 vectors 必须的 所有条件（例如 vector<bool>::reference 并不传回真正的 lvalue，
vector<bool>::iterator
不是个真正的随机存取迭代器）。所以某些 template 程序代码可能适用于任何型别的 vector，唯独无法应付 vector<bool>。此外 vector<bool>可能比一般 vectors 慢一些，因为所有元素操作都必须转化为 bit 操作。不过 vector<bool> 的具体方案也是由实作版本决定，所以性能（包括速度和空间消耗）也可能都有不同。

注意，vector<bool> 不仅仅是个特殊的 bool 版本，它还提供某些特殊的 bit 操作。你可以利用它们更方便操作 bit 或旗标（flags），而且由于 vector<bool> 的大小可动态改变，你还可以把它当成动态大小的 bitfield（位场）。如此一来你便可以添加或移除 bits。如果你需要静态大小的 bitfield，应当使用 bitset，而不是 vector<bool>。bitset 详见 10.4 节, p460。

操作	效果
c.flip()	将所有 bool 元素值取反值，亦即求补码。
m[idx].flip( )	将索引 idx 的 bit 元素取反值
m[idx]=val	令索引 idx 的 bit 元素值为 val（指定单一 bit）
m[idx1]=m[idx2]	令索引 idx1 的 bit 元素值为索引 idx2 的 bit 元素值

表 6.8 vector<bool> 的特殊操作

表 6.8 列出 vector<bool> 的特殊操作。flip()对 vector 中的每一个 bit 取补码。注意，你竟然可以对单一 bool 元素呼叫 flip()。是不是很惊讶？也许你觉得让 *subscript* 运算符传回 bool，再对如此基本型别呼叫 flip()是不可能的。然而这里 vector<bool>用了一个常见技巧，称作 **proxy**[8]，对于 vector<bool>，*subscript* 运算符（及其他传回单一元素的运算符）的回返型别实际上是个辅助类别，一旦

[8] proxy 可让你控制一般无法控制的东西，通常用来获取更好的安全性。上述情形中，此技术施行某种控制，使某种操作成为可能。原则上其传回的对象行为类似 bool。

你要求回返值为 bool，便会触发一个自动型别转换函数。表 6.8 的其他操作由成员函数支援。vector<bool> 的相关宣告如下：

```
namespace std {
class vector<bool> {
   public:
      // auxiliary type for subscript operator
      class reference {
         ...
         public:
            // automatic type conversion to bool
            operator bool() const;

            // assignments
            reference& operator= (const bool);
            reference& operator= (const reference&);

            // bit complement
            void flip();
      }
      ...

      // operations for element access
      // - return type is reference instead of bool
      reference operator[](size_type n);
      reference at(size_type n);
      reference front();
      reference back();
      ...
   };
}
```

你会发现，所有用于元素存取的函数，传回的都是 reference 型别。所以，你可以使用以下述句：

```
c.front().flip(); // negate first Boolean element
c[5] = c.back();  // assign last element to element with index
5
```

一如往常，为了避免未定义的行为，呼叫者必须确保第一、第六和最后一个元素存在。

只有在 non-const vector<bool> 容器中才会用到内部型别 reference。存取元素用的 const member function 会传回型别为 bool 的普通数值。

6.3 Deques

容器 deque（发音为 "deck"）和 vector 非常相似。它也采用 dynamic array 来管理元素，提供随机存取，并有着和 vector 几乎一模一样的接口。不同的是 deque 的 dynamic array 头尾都开放，因此能在头尾两端进行快速安插和删除（图 6.2）。

图 6.2 deque 的逻辑结构

为了获取这种能力，deque 通常实作为一组独立区块，第一区块朝某方向扩展，最后一个区块朝另一方向扩展，如图 6.3。

图 6.3 deque 的内部结构

使用 deque 之前，必须先含入头文件 `<deque>`[9]：

```
#include <deque>
```

在其中，`deque` 型别是定义于命名空间 `std` 内的一个 class template：

```
namespace std {
   template <class T,
```

[9] 早期的 STL 中，deque 的头文件是 `<deque.h>`。

```
                class Allocator = allocator<T> >
    class deque;
}
```

和 vector 相同，第一个 template 参数用来表明元素型别 — 只要是 *assignable* 和 *copyable* 都可以胜任。第二个 template 参数可有可无，用来指定内存模型（memory model），预设为 `allocator`（详见第 15 章）[10]。

6.3.1 Deques 的能力

与 vectors 相比，deques 功能上的不同处在于：

- 两端都能快速安插元素和移除元素（vector 只在尾端逞威风）。这些操作可以在分期摊还的常数时间（amortized constant time）内完成。
- 存取元素时，deque 的内部结构会多一个间接过程，所以元素的存取和迭代器的动作会稍稍慢一些。
- 迭代器需要在不同区块间跳转，所以必须是特殊的智能型指标，非一般指标。
- 在对内存区块有所限制的系统中（例如 PC 系统），deque 可以内含更多元素，因为它使用不止一块内存。因此 deque 的 `max_size()` 可能更大。
- deque 不支持对容量和内存重分配时机的控制。特别要注意的是，除了头尾两端，在任何地方安插或删除元素，都将导致指向 deque 元素的任何 pointers、references、iterators 失效。不过，deque 的内存重分配优于 vectors，因为其内部结构显示，deques 不必在内存重分配时复制所有元素。
- deque 的内存区块不再被使用时，会被释放。deque 的内存大小是可缩减的。不过，是不是这么做，以及究竟怎么做，由实作版本定义之。

deques 的下述特性跟 vectors 差不多：

- 在中段部分安插、移除元素的速度相对较慢，因为所有元素都需移动以腾出或填补空间。
- 迭代器属于 *random access iterator*（随机存取迭代器）。

总之，如果是以下情形，最好采用 deque：

- 你需要在两端安插和移除元素（这是 deque 的拿手好戏）。
- 无需引用（*refer to*）容器内的元素。
- 要求容器释放不再使用的元素（不过，标准规格上并没有保证这一点）。

vectors 和 deques 的接口几乎一样，所以如果无需什么特殊性质，两者都可试试。

10 在尚未支持 default template parameters 的系统中，第二参数通常会被省略。

6.3.2 Deque 的操作函数

表 6.9 至表 6.11 列出了 deque 的所有操作函数：

操作	效果
`deque<Elem> c`	产生一个空的 deque
`deque<Elem> c1(c2)`	针对某个 deque 产生同型副本（所有元素都被拷贝）
`deque<Elem> c(n)`	产生一个 deque，含有 n 个元素，这些元素均以 *default* 建构式产生出来。
`deque<Elem> c(n,elem)`	产生一个 deque，含有 n 个元素，这些元素均是 elem 的副本。
`deque<Elem> c(beg,end)`	产生一个 deque，以区间[beg;end)内的元素为初值
`c.~deque<Elem>()`	销毁所有元素，释放内存

表 6.9 deques 的建构式和解构式

操作	效果
`c.size()`	传回容器的实际元素个数
`c.empty()`	判断容器大小是否为零。等同于 size()==0，但可能更快。
`c.max_size()`	传回可容纳的最大元素数量。
`c1 == c2`	判断是否 c1 等于 c2
`c1 != c2`	判断是否 c1 不等于 c2。等同于 !(c1 == c2)。
`c1 < c2`	判断是否 c1 小于 c2。
`c1 > c2`	判断是否 c1 大于 c2。等同于 c2 < c1。
`c1 <= c2`	判断是否 c1 小于等于 c2 。等同于 !(c2 < c1) 。
`c1 >= c2`	判断 c1 是否大于等于 c2 。等同于 !(c1 < c2) 。
`c.at(idx)`	传回索引 idx 所标示的元素。如果 idx 越界，丢掷 *out_of_range*
`c[idx]`	传回索引 idx 所标示的元素，不进行范围检查。
`c.front()`	传回第一个元素。不检查元素是否存在。
`c.back()`	传回最后一个元素。不检查元素是否存在。
`c.begin()`	传回一个随机迭代器，指向第一元素。
`c.end()`	传回一个随机迭代器，指向最后元素的下一位置。
`c.rbegin()`	传回一个逆向迭代器，指向逆向迭代时的第一个元素。
`c.rend()`	传回一个逆向迭代器，指向逆向迭代时的最后元素的下一位置。

表 6.10 deques 的非变动性操作（nonmodifying operations）

操作	效果
`c1 = c2`	将 c2 的所有元素指派给 c1
`c.assign(n,elem)`	将 n 个 elem 副本指派给 c
`c.assign(beg,end)`	将区间[beg;end)中的元素指派给 c
`c1.swap(c2)`	将 c1 和 c2 的元素互换
`swap(c1,c2)`	同上。此为全局函数。
`c.insert(pos,elem)`	在 pos 位置插入一个 elem 副本，并传回新元素的位置
`c.insert(pos,n, elem)`	在 pos 位置插入 elem 的 n 个副本，无回返值。
`c.insert(pos,beg, end)`	在 pos 位置插入在区间[beg;end)所有元素的副本，无回返值。
`c.push_back(elem)`	在尾部添加 elem 的一个副本
`c.pop_back()`	移除最后一个元素（但不回传）
`c.push_front(elem)`	在头部插入 elem 的一个副本
`c.pop_front()`	移除头部元素（但不回传）
`c.erase(pos)`	移除 pos 位置上的元素，传回下一元素位置
`c.erase(beg,end)`	移除[beg, end)区间内的所有元素，传回下一元素位置
`c.resize(num)`	将大小（元素个数）改为 num。如果 size()增长了，新增元素都以 *default* 建构式产生出来。
`c.resize(num,elem)`	将大小（元素个数）改为 num。如果 size()增长了，新增元素都是 elem 的副本。
`c.clear()`	移除所有元素，将容器清空。

表 6.11 deques 的变动性操作（modifying operations）

deques 的各项操作只在以下数点和 vectors 不同：

1. deques 不提供容量操作（capacity()和 reserve()）。
2. deque 直接提供函数，用以完成头部元素的安插和删除（push_front()和 pop_front()）。

其他操作都相同，所以这里不重复。它们的具体描述请见 p150, 6.2.2 节。

还有一些值得考虑的事情：

1. 除了 at()，没有任何成员函数会检查索引或迭代器是否有效。
2. 元素的插入或删除可能导致记忆体重新分配，所以任何插入或删除动作都会使所有指向 deque 元素的 pointers、references 和 iterators 失效。唯一例外是在头部或尾部插入元素，动作之后，pointers 和 references 仍然有效（但 iterators 就没这么幸运）。

6.3.3 异常处理（Exception Handling）

原则上 deque 提供的异常处理和 vectors 提供的一样（p155）。新增的操作函数 push_front()和 pop_front()分别对应于 push_back()和 pop_back()。因此，
C++ 标准链接库保证下列行为：

- 如果以 push_back()或 push_front()安插元素时发生异常，则该操作不带来任何效应。
- pop_back()和 pop_front()不会丢掷任何异常。

STL 的异常处理一般原则请见 p139, 5.11.2 节。异常发生时，提供特殊保障的所有容器操作函数均列于 p248, 6.10.10 节。

6.3.4 Deques 运用实例

以下程序以简单的例子说明 deque 的功用：

```
// cont/deque1.cpp

#include <iostream>
#include <deque>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
   // create empty deque of strings
   deque<string> coll;

   // insert several elements
   coll.assign (3, string("string"));
   coll.push_back ("last string");
   coll.push_front ("first string");

   // print elements separated by newlines
   copy (coll.begin(), coll.end(),
         ostream_iterator<string>(cout,"\n"));
   cout << endl;

   // remove first and last element
   coll.pop_front();
   coll.pop_back();
```

```
    // insert "another" into every element but the first
    for (int i=1; i<coll.size(); ++i) {
         coll[i] = "another " + coll[i];
    }

    // change size to four elements
    coll.resize (4, "resized string");

    // print elements separated by newlines
    copy (coll.begin(), coll.end(),
          ostream_iterator<string>(cout,"\n"));
}
```

程序输出如下：

```
first string
string
string
string
last string

string
another string
another string
resized string
```

6.4 Lists

List 使用一个 doubly linked list（双向串行）来管理元素，如图 6.4。按惯例，C++ 标准链接库并未明定实作方式，只是遵守 list 的名称、限制和规格。

图 6.4 List 的结构

使用 list 时必须含入头文件 <list>[1] ：

```
#include <list>
```

其中 list 型别系定义于 namespace std 中，是个 class template：

```
namespace std {
   template <class T,
             class Allocator = allocator<T> >
   class list;
}
```

任何型别 T 只要具备 *assignable* 和 *copyable* 两性质，就可以作为 list 的元素。第二个 template 自变量可有可无，用来指定内存模型（详见第 15 章）。默认的记忆体模型是 C++ 标准链接库所提供的 allocator[12] 。

6.4.1 Lists 的能力

List 的内部结构和 vector 或 deque 截然不同，所以在几个主要方面与前述二者存在明显区别：

- List 不支援随机存取。如果你要存取第 5 个元素，就得顺着串链一一爬过前 4 个元素。所以，在 list 中随机巡访任意元素，是很缓慢的行为。
- 任何位置上（不只是两端）执行元素的安插和移除都非常快，始终都是常数时间内完成，因为无需移动任何其他元素。实际上内部只是进行了一些指标操作而已。

[1] 早期 STL 中，list 的定义头文件是 <list.h>

[12] 如果系统不支持 default template parameters，通常省略第二自变量。

- 安插和删除动作并不会造成指向其他元素的各个 pointers、references、iterators 失效。
- List 对于异常有着这样的处理方式：要嘛操作成功，要嘛什么都不发生。你决不会陷入「只成功一半」这种前不着村后不巴店的尴尬境地。

Lists 所提供的成员函数反映出它和 vectors 以及 deques 的不同：

- 由于不支持随机存取，lists 既不提供 *subscript*（下标）运算符，也不提供 `at()`。
- Lists 并未提供容量、空间重新分配等操作函数，因为全无必要。每个元素都有自己的内存，在被删除之前一直有效。
- Lists 提供了不少特殊的成员函数，专门用于移动元素。较之同名的 STL 通用算法，这些函数执行起来更快，因为它们无需拷贝或移动，只需调整若干指标即可。

6.4.2 List 的操作

生成（Creation），复制（Copy）和销毁（Destroy）

lists 的生成、复制和销毁动作，和所有序列式容器相同，详见表 6.12。关于初值来源（initialization sources）的若干注意事项，可参考 6.1.2 节, p144。

操作	效果
`list<Elem> c`	产生一个空的 list
`list<Elem> c1(c2)`	产生一个与 `c2` 同型的 list（每个元素都被复制）
`list<Elem> c(n)`	产生拥有 `n` 个元素的 list，这些元素都以 *default* 建构式初始化
`list<Elem> c(n,elem)`	产生拥有 `n` 个元素的 list，每个元素都是 `elem` 的副本。
`list<Elem> c(beg,end)`	产生一个 list 并以 `[beg;end)` 区间内的元素为初值
`c.~list<Elem>()`	销毁所有元素，释放内存

表 6.12 Lists 的建构式和解构式

非变动性操作（Nonmodifying Operations）

Lists 也提供诸如「询问大小」和「两相比较」等等一般性操作。详见表 6.13 和 6.1.2 节 , p144。

操作	效果
c.size()	传回元素个数
c.empty()	判断容器大小是否为零。等同于 size()==0，但可能更快。
c.max_size()	传回元素的最大可能数量
c1 == c2	判断是否 c1 等于 c2
c1 != c2	判断是否 c1 不等于 c2。等同于!(c1 == c2)。
c1 < c2	判断是否 c1 小于 c2
c1 > c2	判断是否 c1 大于 c2。等同于与 c2 < c1 相同。
c1 <= c2	判断是否 c1 小于等于 c2。等同于 !(c2 < c1)。
c1 >= c2	判断是否 c1 大于等于 c2。等同于 !(c1 < c2)。

表 6.13 Lists 的非变动性操作（Nonmodifying Operations）

赋值（指派, Assignment）

和其他序列式容器一样，lists 也提供了一般常用的赋值（指派）动作，如表 6.14。

操作	效果
c1 = c2	将 c2 的全部元素指派给 c1
c.assign(n,elem)	将 elem 的 n 个拷贝指派给 c
c.assign(beg,end)	将区间[beg;end)的元素指派给 c
c1.swap(c2)	将 c1 和 c2 的元素互换
swap(c1,c2)	同上。此为全局函数。

表 6.14 Lists 的 *assignment*（指派, 赋值）操作函数

一如往常，安插动作和建构式一一匹配，如此一来就有能力提供不同的初值来源（initialization sources），详见 6.1.2 节, p144。

元素存取（Element Access）

list 不支持随机存取，只有 front()和 back()能够直接存取元素，如表 6.15。

操作	效果
c.front()	传回第一个元素。不检查元素存在与否。
c.back()	传回最后一个元素。不检查元素存在与否。

表 6.15 Lists 元素的直接存取

一如以往，这些操作并不检查容器是否为空。对着空容器执行任何操作，都会导致未定义的行为。所以呼叫者必须确保容器至少含有一个元素。例如：

```
std::list<Elem> coll;      // empty!

std::cout << coll.front(); // RUNTIME ERROR   Î undefined behavior

if (!coll.empty()) {
    std::cout << coll.back();  // OK
}
```

迭代器相关函数（Iterator Functions）

只有运用迭代器，才能够存取 list 中的各个元素。Lists 提供的迭代器函数如表 6.16。然而由于 list 不能随机存取，这些迭代器只是双向（而非随机）迭代器。所以凡是用到随机存取迭代器的算法（所有用来操作元素顺序的算法 — 特别是排序算法 — 都归此类）你都不能呼叫。不过你可以拿 list 的特殊成员函数 `sort()` 取而代之，详见 p245。

操作	效果
`c.begin()`	传回一个双向迭代器，指向第一元素。
`c.end()`	传回一个双向迭代器，指向最后元素的下一位置。
`c.rbegin()`	传回一个逆向迭代器，指向逆向迭代的第一个元素。
`c.rend()`	传回一个逆向迭代器，指向逆向迭代的最后元素的下一位置。

表 6.16 Lists 的迭代器相关函数

元素的安插（Inserting）和移除（Removing）

表 6.17 列出 lists 元素的安插和移除操作。Lists 提供 deques 的所有功能，还增加了 `remove()` 和 `remove_if()` 算法应用于 list 身上的特殊版本。

和一般运用 STL 时相似，你必须确保自变量的正确。迭代器必须指向合法位置，区间终点不能位于区间起点的前头；还有，你不能从空容器中移除元素。

如果想要安插或移除多个元素，你可以对它们进行单一呼叫，比多次呼叫来得快。

为了移除元素，Lists 特别配备了 `remove()` 算法（9.7.1 节， p378）的特别版本。这些成员函数比 `remove()` 算法的速度更快，因为它们只进行内部指标的操作，无需顾及元素。所以，面对 list，你应该呼叫成员函数 `remove()`，而不是像面对 vectors 和 deques 那样呼叫 STL 算法（如 p154 所示）。

操作	效果
c.insert(pos,elem)	在迭代器 pos 所指位置上安插一个 elem 副本，并传回新元素的位置。
c.insert(pos,n,elem)	在迭代器 pos 所指位置上安插 n 个 elem 副本，无回返值。
c.insert(pos,beg,end)	在迭代器 pos 所指位置上安插 [beg;end)区间内的所有元素的副本，无回返值。
c.push_back(elem)	在尾部安插一个 elem 副本。
c.pop_back()	移除最后一个元素（但不传回）。
c.push_front(elem)	在头部安插一个 elem 副本。
c.pop_front()	移除第一元素（但不传回）。
c.remove(val)	移除所有其值为 val 的元素。
c.remove_if(op)	移除所有「造成 op(elem)结果为 true」的元素。
c.erase(pos)	移除迭代器 pos 所指元素，传回下一元素位置。
c.erase(beg,end)	移除区间[beg;end)内的所有元素，传回下一元素位置
c.resize(num)	将元素容量变为 num。如果 size()变大，则以 *default* 建构式建构所有新增元素。
c.resize(num,elem)	将元素容量变为 num。如果 size()变大，则以 elem副本做为新增元素的初值。
c.clear()	移除全部元素，将整个容器清空。

表 6.17 Lists 的安插、移除操作函数

想要将所有「与某值相等」的元素移除，可以这么做（进一步细节详见 5.6.3 节, p116）：

```
std::list<Elem> coll;
...
// remove all elements with value val
coll.remove(val);
```

如果只是想移除「与某值相等」的第一个元素，你得使用诸如 p154 中针对 vectors 所用的算法。

如果使用 remove_if()，你可以藉由一个函数或仿函数[13]来定义元素移除原则；它可以将每一个「令传入之操作结果为 true」的元素移除：

```
list.remove_if (not1(bind2nd(modulus<int>(),2)));
```

如果你对以上述句感到头晕，别急，前进 p306 看看详细解释。关于 remove()和 remove_if()的其他例子，详见 p378。

[13] 一个不支持 member templates 的系统，通常不会提供 remove_if()成员函数。

Splice 函数

Linked lists 的一大好处就是不论在任何位置，元素的安插和移除都只需要常数时间。如果你有必要将若干元素从 A 容器转放到 B 容器，那么上述好处就更见其效了，因为你只需要重新定向某些指标即可，如图 6.5。

图 6.5 Splice 操作函数用以改变 list 元素的次序

为了利用这个优势，lists 不仅提供 `remove()`，还提供其他一些成员函数，用来改变元素和区间次序，或是用来重新串链。我们不仅可以呼叫这些函数，移动单一 list 内的元素，也可以移动两个 lists 之间的元素 — 只要 lists 的型别一致即可。表 6.18 列举出这些函数。6.10.8 节, p244 另有详细说明，实例可见 6.4.4 节, p172。

操作	效果
`c.unique()`	如果存在若干相邻而数值相同的元素，就移除重复元素，只留下一个。
`c.unique(op)`	如果存在若干相邻元素，都使 `op()` 的结果为 `true`，则移除重复元素，只留下一个。
`c1.splice(pos,c2)`	将 `c2` 内的所有元素转移到 `c1` 之内、迭代器 `pos` 之前。
`c1.splice(pos,c2, c2pos)`	将 `c2` 内的 `c2pos` 所指元素转移到 `c1` 内的 `pos` 所指位置上（`c1` 和 `c2` 可相同）。
`c1.splice(pos,c2, c2beg,c2end)`	将 `c2` 内的 `[c2beg;c2end)` 区间内所有元素转移到 `c1` 内的 `pos` 之前（`c1` 和 `c2` 可相同）。
`c.sort()`	以 `operator<` 为准则，对所有元素排序。
`c.sort(op)`	以 `op()` 为准则，对所有元素排序。
`c1.merge(c2)`	假设 `c1` 和 `c2` 容器都包含已序（*sorted*）元素，将 `c2` 的全部元素转移到 `c1`，并保证合并后的 list 仍为已序。
`c1.merge(c2,op)`	假设 `c1` 和 `c2` 容器都包含 `op()` 原则下的已序（*sorted*）元素，将 `c2` 的全部元素转移到 `c1`，并保证合并后的 list 在 `op()` 原则下仍为已序。
`c.reverse()`	将所有元素反序（reverse the order）。

表 6.18 Lists 的特殊变动性操作（Special Modifying Operations）

6.4.3 异常处理（Exception Handling）

所有 STL 标准容器中，lists 对于异常安全性（**exception safety**）提供了最佳支持。几乎所有操作都是不成功便成仁：要嘛成功，要嘛无效。仅有少数几个操作没有如此保证，包括指派（赋值）运算和成员函数 `sort()`，不过它们也有基本保证：异常发生时不会泄漏资源，也不会与容器恒常特性（invariants）发生冲突。`merge()`, `remove()`,`remove_if()`,`unique()` 提供的保证是有前提的，那就是元素间的比较动作（采用 `operator==` 或判断式 *predicate*）并不会丢掷异常。用数据库编程术语来说，只要你不呼叫赋值操作或 `sort()`，并保证元素相互比较时不丢掷异常，那么 lists 便可说是「交易安全（**transaction safe**）」。表 6.19 列出异常状况下提供特殊保证的所有操作函数。STL 异常处理的一般性讨论，请见 5.11.2 节, p139。

操作	保证
`push_back()`	如果不成功，就是无任何作用。
`push_front()`	如果不成功，就是无任何作用。
`insert()`	如果不成功，就是无任何作用。
`pop_back()`	不丢掷异常。
`pop_front()`	不丢掷异常。
`()`	不丢掷异常。
`erase()`	不丢掷异常。
`clear()`	如果不成功，就是无任何作用。
`resize()`	只要元素比较动作不丢掷异常，它就不丢掷异常。
`remove()`	只要判断式 *predicate* 不丢掷异常，它就不丢掷异常。
`remove_if()`	只要元素比较动作不丢掷异常，它就不丢掷异常。
`()`	不丢掷异常。
`unique()`	只要元素比较时不丢掷异常，它便保证「要不成功，要不无任何作用」。
`splice()`	不丢掷异常。
`merge()`	不丢掷异常。
`swap()`	不丢掷异常。

表 6.19 Lists 的各种操作在异常发生时提供的特殊保证

6.4.4 Lists 运用实例

下面这个例子突显出 list 特殊成员函数的用法:

```
// cont/list1.cpp

#include <iostream>
#include <list>
#include <algorithm>
```

```
using namespace std;

void printLists (const list<int>& l1, const list<int>& l2)
{
   cout << "list1: ";
   copy (l1.begin(), l1.end(), ostream_iterator<int>(cout," "));
   cout << endl << "list2: ";
   copy (l2.begin(), l2.end(), ostream_iterator<int>(cout," "));
   cout << endl << endl;
}

int main()
{
   // create two empty lists
   list<int> list1, list2;

   // fill both lists with elements
   for (int i=0; i<6; ++i) {
        list1.push_back(i);
        list2.push_front(i);
   }
   printLists(list1, list2);

   //   insert all elements of list1 before the first element with value 3 of list2
   //   - find() returns an iterator to the first element with value 3
   list2.splice(find(list2.begin(),list2.end(), //  destination
position
                3),list1);           // source list
   printLists(list1, list2);

   // move first element to the end
   list2.splice(list2.end(),      // destination position
                list2,            // source list
                list2.begin());   // source position
   printLists(list1, list2);

   // sort second list, assign to list1 and remove duplicates
   list2.sort();
   list1 = list2;
```

```
    list2.unique();
    printLists(list1, list2);

    // merge both sorted lists into the first list
    list1.merge(list2);
    printLists(list1, list2);
}
```

程序输出如下:

```
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0

list1:
list2: 5 4 0 1 2 3 4 5 3 2 1 0

list1:
list2: 4 0 1 2 3 4 5 3 2 1 0 5

list1: 0 0 1 1 2 2 3 3 4 4 5 5
list2: 0 1 2 3 4 5

list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
list2:
```

6.5 Sets 和 Multisets

Set 和 multiset 会根据特定的排序准则，自动将元素排序。两者不同处在于 multisets 允许元素重复而 sets 不允许（请参考 6.6 节和第 5 章关于本主题的讨论）。

图 6.6 Sets 和 Multisets

使用 set 或 multiset 之前，必须先含入头文件 `<set>`[14]：

```
#include <set>
```

在这个头文件中，上述两个型别都被定义为命名空间 `std` 内的 class templates：

```
namespace std {
   template <class T,
             class Compare = less<T>,
             class Allocator = allocator<T> >
   class set;

   template <class T,
          class Compare = less<T>,
          class Allocator = allocator<T> >
   class multiset;
}
```

只要是 *assignable*、*copyable*、*comparable*（根据某个排序准则）的型别 `T`，都可以成为 set 或 multiset 的元素型别。可有可无的第二个 template 自变量用来定义排序准则。如果没有传入特别的排序准则，就采用预设准则 `less`[15]— 这是一个仿函数，以 `operator<` 对元素进行比较，以便完成排序（`less` 的内容详见 p305）。可有可无的第三自变量用来定义内存模型（参见第 15 章）。默认的内存模型是

14 早期 STL 中，sets 的头文件是 `<set.h>`，multisets 的头文件是 `<mulitiset.h>`

15 在不支持 default template arguments 的系统中，第二自变量通常会被省略。

allocator，由 C++ 标准链接库提供[16]。

所谓「排序准则」，必须定义 **strict weak ordering**，其意义如下：

1. 必须是「非对称的（**antisymmetric**）」。

对 operator< 而言，如果 x < y 为真，则 y < x 为假。

对判断式 *predicate* op() 而言，如果 op(x,y) 为真，则 op(y,x) 为假。

2. 必须是「可递移的（**transitive**）」。

对 operator< 而言，如果 x < y 为真且 y < z 为真，则 x < z 为真。

对判断式 op() 而言，如果 op(x,y) 为真且 op(y,z) 为真，则 op(x,z) 为真。

3. 必须是「非反身的（**irreflexive**）」。

对 operator< 而言，x < x 永远为假。

对判断式 *predicate* op() 而言，op(x,x) 永远为假。

基于这些特性，排序准则也可用于相等性检验，也就是说，如果两个元素都不小于对方（或说 op(x,y) 和 op(y,x) 都为假），则两个元素相等。

译注：以上种种性质（及其他各种相关性质）是 STL 学术理论的一部分。STL 先建立起一个抽象概念阶层体系，形成一个软件组件分类学，最后再以实际工具（C++ template）将各个概念实作出来。这些理论架构的最佳描述书籍是《 *Generic Programming and the STL*》,by Matthew H. Austern, Addison Wesley 1998；中译本《泛型程序设计与 STL》，侯捷/黄俊尧合译，碁峰 2000。

6.5.1 Sets 和 Multisets 的能力

和所有标准关系型容器类似，sets 和 multisets 通常以平衡二元树（balanced binary tree，图 6.7）完成。C++ 标准规格书并未明定之，但由 sets 和 multisets 各项操作的复杂度可以得出这样的结论[17]。

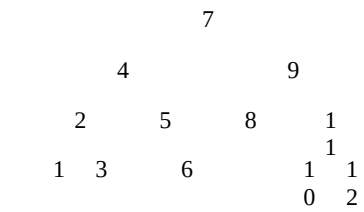

图 6.7 Sets 和 Multisets 的内部结构

[16] 在不支持 default template arguments 的系统中，第三自变量通常会被省略。

[17] 事实上 sets 和 multisets 通常以红黑树（red-black tree）实作而成。红黑树在改变元素数量和元素搜寻方面都很出色，它保证节点安插时最多只会作两个重新链结（relink）动作，而且到达某一元素的最长路径深度，最多只是最短路径深度的两倍。

自动排序的主要优点在于使二元树于搜寻元素时具有良好性能。其搜寻函数演算法具有对数（logarithmic）复杂度。在拥有 1000 个元素的 sets 或 multisets 中搜寻元素，二元树搜寻动作（由成员函数执行）的平均时间为线性搜寻（由 STL 演算法执行）的 1/50。关于复杂度的讨论，详见 p21, 2.3 节。

但是，自动排序造成 sets 和 multisets 的一个重要限制：你不能直接改变元素值，因为这样会打乱原本正确的顺序。因此，要改变元素值，必须先删除旧元素，再插入新元素。这里提供的接口正反映了这种行为：

- sets 和 multisets 不提供用来直接存取元素的任何操作函数。
- 通过迭代器进行元素间接存取，有一个限制：从迭代器的角度来看，元素值是常数。

6.5.2 Sets 和 multisets 的操作函数

生成（Create）、复制（Copy）和销毁（Destroy）

表 6.20 列出 sets 和 multisets 的建构式和解构式。

操作	效果
set `c`	产生一个空的 set/multiset，其中不含任何元素。
set `c(op)`	以 `op` 为排序准则，产生一个空的 set/multiset。
set `c1(c2)`	产生某个 set/multiset 的副本，所有元素均被复制。
set	以区间`[beg;end)`内的元素产生一个 set/multiset。
set`(beg,end,op)`	以 `op`为排序准则，利用`[beg;end)`内的元素生成一个 set/multiset
`c.~`***set***`()`	销毁所有元素，释放内存。

其中 ***set*** 可为下列形式：

set	效果
`set<Elem>`	一个 set，以 `less<>`（`operator<`）为排序准则
`set<Elem,Op>`	一个 set，以 `op` 为排序准则
`multiset<Elem>`	一个 multiset，以 `less<>`（`operator<`）为排序准则
`multiset<Elem,Op>`	一个 multiset，以 `op` 为排序准则

表 6.20 Sets 和 Multisets 的建构式和解构式

有两种方式可以定义排序准则：

1. 以 template 参数定义之。

例如[18]：

```
std::set<int, std::greater<int> > coll;
```

这种情况下，排序准则就是型别的一部分。因此型别系统确保「只有排序准则相同的容器才能被合并」。这是排序准则的通常指定法。更精确地说，第二参数是排序准则的型别，实际的排序准则是容器所产生的函数对象（*function object*，或称 *functor*）。为了产生它，容器建构式会呼叫「排序准则型别」的 *default* 建构式。p294 有一个「使用者自定之排序准则」的运用实例。

2. 以建构式参数定义之。

这种情况下，同一个型别可以运用不同的排序准则，而排序准则的初始值或状态也可以不同。如果执行期才获得排序准则，而且需要用到不同的排序准则（但数据型别必须相同），此一方式可派上用场。p191 有个完整例子。

如果使用者没有提供特定的排序准则，那么就采用预设准则 — 仿函数 `less<>`。`less<>` 系透过 `operator<` 对元素进行排序[19]。

请注意，排序准则也被用于元素相等性检验工作。当采用预设排序准则时，两元素的相等性检验语句如下：

```
if (! (elem1 < elem2 || elem2 < elem1))
```

这样做有三点好处：

1. 只需传递一个自变量作为排序准则。

2. 不必针对元素型别提供 `operator==`。

3. 你可以对「相等性」有截然相反的定义（即使算式中 `operator==` 的行为有所不同，也无关紧要）。不过当心造成混淆。

这种相等性检验方式会花费比较长的时间，因为评估上述算式可能需要两次比较。注意，如果第一次比较结果为 true，就不用进行第二次比较了。

看到这里，如果容器型别名称让你很烦，采用「型别定义式」不失为一个好办法。在使用容器型别（以及迭代器型别）的任何地方都可以采用这种便捷之道，例如：

18 注意，两个 > 之间需加上一个空格，因为 >> 会被编译程序视为移位运算符，导本处致语法错误。

19 在不支持 default template parameters 的系统中，通常必须这么设定排序准则：`set<int,less<int> > coll;`

```
typedef std::set<int,std::greater<int> > IntSet;
...
IntSet coll
IntSet::iterator pos;
```

「利用区间的起点和终点来建构容器」的建构式，可以从其他型别的容器中，或是从 array、或是从标准输入设备（standard input）中接受元素来源。详见 6.1.2 节, p144。

非变动性操作（Nonmodifying Operations）

Sets 和 multisets 提供常见的非变动性操作，用来查询大小、相互比较。

操作	效果
`c.size()`	传回容器的大小
`c.empty()`	判断容器大小是否为零。等同于 `size()==0`，但可能更快
`c.max_size()`	传回可容纳的最大元素数量
`c1 == c2`	判断是否 c1 等于 c2
`c1 != c2`	判断是否 c1 不等于 c2。等同于 `!(c1 == c2)`。
`c1 < c2`	判断是否 c1 小于 c2
`c1 > c2`	判断是否 c1 大于 c2。等同于 `c2 < c1`。
`c1 <= c2`	判断是否 c1 小于等于 c2。等同于 `!(c2 < c1)`。
`c1 >= c2`	判断是否 c1 大于等于 c2。等同于 `!(c1 < c2)`。

表 6.21 Sets 和 Multisets 的非变动性操作（Nonmodifying Operations）

元素比较动作只能用于型别相同的容器。换言之，元素和排序准则必须有相同的型别，否则编译时期会产生型别方面的错误。

```
std::set<float> c1;   // sorting criterion: std::less<>
std::set<float,std::greater<float> > c2;
...
if (c1 == c2) {       // ERROR: different types
   ...
}
```

比较动作系以「字典（lexicographical）顺序」来检查某个容器是否小于另一个容器。如果要比较不同型别（拥有不同排序准则）的容器，你必须采用 p356, 9.5.4 节的「比较算法（comparing algorithms）」。

特殊的搜寻函数（Special Search Operations）

Sets 和 multisets 在元素快速搜寻方面有优化设计，所以提供了特殊的搜寻函数，如表 6.22。这些函数是同名的 STL 算法的特殊版本。面对 sets 和 multisets，你应该优先采用这些优化算法，如此可获得对数复杂度，而非 STL 算法的线性复杂度。举个例子，在 1,000 个元素中搜寻，平均 10 次比较之后便可得出结果，如果是线性复杂度，平均 500 次比较才能有结果（参见 2.3 节, p21）。

操作	效果
`count(elem)`	传回「元素值为 elem」的元素个数。
`find(elem)`	传回「元素值为 elem」的第一个元素，如果找不到就传回 end()。
`lower_bound(elem)`	传回 elem 的第一个可安插位置，也就是「元素值 >= elem」的第一个元素位置。
`upper_bound(elem)`	传回 elem 的最后一个可安插位置，也就是「元素值 > elem」的第一个元素位置。
`equal_range(elem)`	传回 elem 可安插的第一个位置和最后一个位置，也就是「元素值 == elem」的元素区间。

表 6.22 Sets 和 Multisets 的搜寻操作函数

成员函数 `find()` 搜寻出与自变量值相同的第一个元素，并传回一个迭代器，指向该位置。如果没找到这样的元素，就传回容器的 `end()`。

`lower_bound()` 和 `upper_bound()` 分别传回元素可安插点的第一个和最后一个位置。换言之，`lower_bound()` 传回大于等于自变量值的第一个元素所处位置，`upper_bound()` 传回大于自变量值的第一个元素位置。`equal_range()` 则是将 `lower_bound()` 和 `upper_bound()` 的回返值做成一个 pair 传回（型别 pair 在 p33, 4.1 节介绍），所以它传回的是「与自变量值相等」的元素所形成的区间。如果 `lower_bound()` 或「`equal_range()` 的第一值」等于「`equal_range()` 的第二值」或 `upper_bound()`，则此 sets 或 multisets 内不存在相同数值的元素。这是当然啦，同值区间中至少也得包含一个元素嘛！

下面例子说明如何使用 `lower_bound()`, `upper_bound()` 和 `equal_range()`：

```
// cont/set2.cpp

#include <iostream>
#include <set>
using namespace std;
```

```
int main ()
{
  set<int> c;

  c.insert(1);
  c.insert(2);
  c.insert(4);
  c.insert(5);
  c.insert(6);

  cout << "lower_bound(3): " << *c.lower_bound(3) << endl;
  cout << "upper_bound(3): " << *c.upper_bound(3) << endl;
  cout << "equal_range(3): " << *c.equal_range(3).first << " "
                             << *c.equal_range(3).second << endl;

  cout << endl;
  cout << "lower_bound(5): " << *c.lower_bound(5) << endl;
  cout << "upper_bound(5): " << *c.upper_bound(5) << endl;
  cout << "equal_range(5): " << *c.equal_range(5).first << " "
                             << *c.equal_range(5).second << endl;
}
```

程序输入如下：

```
lower_bound(3): 4
upper_bound(3): 4
equal_range(3): 4 4

lower_bound(5): 5
upper_bound(5): 6
equal_range(5): 5 6
```

上例如果使用 multisets 而不是 sets，程序输出相同。

指派（赋值， Assignments）

sets 和 multisets 只提供所有容器都提供的基本赋值操作（表 6.23），详见 p147。

这些操作函数中，赋值操作的两端容器必须具有相同型别。尽管「比较准则」本身可能不同，但其型别必须相同。p191 列出一个「排序准则不同，但型别相同」的例子。如果准则不同，准则本身也会被指派（assigned）或交换（swapped）。

操作	效果
`c1 = c2`	将 `c2` 中所有元素指派给 `c1`
`c1.swap(c2)`	将 `c1` 和 `c2` 的元素互换。
`swap(c1,c2)`	同上，此为全局函数。

表 6.23 Sets 和 Multisets 的指派（赋值）操作

迭代器相关函数（Iterator Functions）

Sets 和 multisets 不提供元素直接存取，所以只能采用迭代器。Sets 和 multisets 也提供了一些常见的迭代器函数（表 6.24）。

操作	效果
`c.begin()`	传回一个双向迭代器（将元素视为常数），指向第一元素
`c.end()`	传回一个双向迭代器（将元素视为常数），指向最后元素的下一位置
`c.rbegin()`	传回一个逆向迭代器，指向逆向巡访时的第一个元素
`c.rend()`	传回一个逆向迭代器，指向逆向巡访时的最后元素的下一位置

表 6.24 Sets 和 multisets 的迭代器相关操作函数

和其他所有关系型容器类似，这里的迭代器是双向迭代器（参见 p255, 7.2.4 节）。所以，对于只能用于随机存取迭代器的 STL 算法（例如排序或随机乱序 *random shuffling* 算法），sets 和 multisets 就无福消受了。

更重要的是，对迭代器操作而言，所有元素都被视为常数，这可确保你不会人为改变元素值，从而打乱既定顺序。然而这也使得你无法对 sets 或 multisets 元素呼叫任何变动性算法（modifying algorithms）。例如你不能对它们呼叫 `remove()`，因为 `remove()` 算法实际上是以一个自变量值覆盖被移除的元素（详细讨论见 p115, 5.6.2 节）。如果要移除 sets 和 multisets 的元素，你只能使用它们所提供的成员函式。

元素的安插（Inserting）和移除（Removing）

表 6.25 列出 sets 和 multisets 的元素安插和删除函数。

按 STL 惯例，你必须保证自变量有效：迭代器必须指向有效位置、序列起点不能位于终点之后、不能从空容器中删除元素。

安插和移除多个元素时，单一呼叫（一次处理）比多次呼叫（逐一处理）快得多。

操作	效果
`c.insert(elem)`	安插一份 `elem` 副本，传回新元素位置（不论是否成功 — 对 sets 而言）
`c.insert(pos,elem)`	安插一份 `elem` 副本，传回新元素位置（`pos` 是个提示，指出安插动作的搜寻起点。如果提示恰当，可大大加快速度）
`c.insert(beg,end)`	将区间`[beg;end)`内所有元素的副本安插到 `c`（无回返值）
`c.erase(elem)`	移除「与 `elem` 相等」的所有元素，传回被移除的元素个数
`c.erase(pos)`	移除迭代器 `pos` 所指位置上的元素，无回返值。
`c.erase(beg,end)`	移除区间`[beg;end)`内的所有元素，无回返值。
`c.clear()`	移除全部元素，将整个容器清空。

表 6.25 Sets 和 Multisets 的元素安插和移除

注意，安插函数的回返值型别不尽相同：

- Sets 提供如下接口：

```
pair<iterator,bool> insert(const value_type& elem);
iterator            insert(iterator pos_hint,
                           const value_type& elem);
```

- Multisets 提供如下接口：

```
iterator insert(const value_type& elem);
iterator insert(iterator pos_hint,
                const value_type& elem);
```

回返值型别不同的原因是：multisets 允许元素重复，而 sets 不允许。因此如果将某元素安插至一个 set 内，而该 set 已经内含同值元素，则安插动作将告失败。所以 set 的回返值型别是以 `pair` 组织起来的两个值（关于 `pair` 详见 p33, 4.1 节）：

1. pair 结构中的 `second` 成员表示安插是否成功。
2. pair 结构中的 `first` 成员传回新元素的位置，或传回现存的同值元素的位置。

其他任何情况下，函数都传回新元素位置（如果 sets 已经内含同值元素，则传回同值元素的位置）。

以下例子把数值 3.3 的元素安插到 set `C` 中，藉此说明如何使用上述接口：

```
std::set<double> c;
...
if (c.insert(3.3).second) {
    std::cout << "3.3 inserted" << std::endl;
}
else {
    std::cout << "3.3 already exists" << std::endl;
}
```

如果你还想处理新位置或旧位置，程序代码得更复杂些：

```
// define variable for return value of insert()
std::pair<std::set<float>::iterator,bool> status;

// insert value and assign return value
status = c.insert(value);

// process return value
if (status.second) {
    std::cout << value << " inserted as element "
}
else {
    std::cout << value << " already exists as element "
}
std::cout << std::distance(c.begin(),status.first) + 1
          << std::endl;
```

对于此一序列的两次呼叫结果可能如下：

```
8.9 inserted as element 4
7.7 already exists as element 3
```

注意，所有拥有「位置提示参数」的安插函数，其回返值型别都一样，不论是 sets 或 multisets，这些函数都只传回一个迭代器。这些函数的效果与「无位置提示参数」的函数一样，只不过性能略有差异。你可以传进一个迭代器，该位置将作为一个提示，用来提升性能。事实上如果被安插元素的位置恰好紧贴于提示位置之后，那么时间复杂度就会从「对数」一变而为「分期摊还常数（amortized constant）」（复杂度的介绍请见 p21, 2.3 节）。和「单自变量安插函数」不同的是，带有「额外提示位置」的若干安插函数，都具有相同的回返值型别，这就确保你至少有了一个通用型安插函数，在各种容器中有共同接口。事实上通用型安插迭代器（*general inserters*）就是靠这个接口的支持才得以实现。

要删除「与某值相等」的元素，只需呼叫 `erase()`：

```
std::set<Elem> coll;
...
// remove all elements with passed value
coll.erase(value);
```

和 lists 不同的是，`erase()`并非取名为 `remove()`（后者的讨论请见 p170）。是的，它的行为不同，它传回被删除元素的个数，用在 sets 身上，回返值非 0 即 1。

如果 multisets 内含重复元素，你不能使用 `erase()`来删除这些重复元素中的第一个。你可以这么做：

```
std::multiset<Elem> coll;
...
// remove first element with passed value
std::multiset<Elem>::iterator pos;
pos = coll.find (elem);
if (pos != coll.end()) {
   coll.erase(pos);
}
```

这里应该采用成员函数 `find()`，而非 STL 算法 `find()`，因为前者速度更快（参见 p154 的例子）。

注意，还有一个回返值不一致的情况。作用于序列式容器和关系型容器的 `erase()`函数，其回返值有以下不同：

1. 序列式容器提供下面的 `erase()`成员函数：

```
iterator erase(iterator pos);
iterator erase(iterator beg, iterator end);
```

2. 关系型容器提供下面的 `erase()`成员函数：

```
void erase(iterator pos);
void erase(iterator beg, iterator end);
```

存在这种差别，完全是为了性能。在关系型容器中「搜寻某元素并传回后继元素」可能颇为耗时，因为这种容器的底部是以二元树完成，所以如果你想编写对所有容器都适用的程序代码，你必须忽略回返值。

6.5.3 异常处理（Exception Handling）

Sets 和 multisets 是「以节点（nodes）为基础」的容器。如果节点建构失败，容器仍保持原样。此外，由于解构式通常并不抛掷异常，所以节点的移除不可能失败。

然而，面对多重元素安插动作，「保持元素次序」这一条件会造成「异常抛出时能够完全复原」这一需求变得不切实际。因此只有「单一元素安插动作」才支持「成功，否则无效」的操作策略。至于「多元素删除动作」总是能够成功。如果排序准则之复制/赋值动作会丢掷异常，则 `swap()`也会丢掷异常。

STL 异常处理的一般性讨论见于 p139, 5.11.2 节。p248 的 6.10.10 节列出「异常出现时会给予特殊保证」的所有容器操作函数。

6.5.4 Sets 和 multisets 运用实例

以下程序展示 sets 的一些能力[20]：

```
// cont/set1.cpp

#include <iostream>
#include <set>
using namespace std;

int main()
{
   /* type of the collection:sets:
   * - no duplicates
   * - elements are integral values
   * - descending order
   */
   typedef set<int,greater<int> > IntSet;

   IntSet coll1; // empty set container

   // insert elements in random order
   coll1.insert(4);
   coll1.insert(3);
   coll1.insert(5);
   coll1.insert(1);
   coll1.insert(6);
   coll1.insert(2);
   coll1.insert(5);

   // iterate over all elements and print them
   IntSet::iterator pos;
   for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
        cout << *pos << ' ';
   }
   cout << endl;
```

[20] distance()的定义已有改变。早期 STL 版本中，你必须含入 distance.hpp p263 见

```
    // insert 4 again and process return value
    pair<IntSet::iterator,bool> status = coll1.insert(4);
    if (status.second) {
       cout << "4 inserted as element "
            << distance(coll1.begin(),status.first) + 1
            << endl;
    }
    else {
       cout << "4 already exists" << endl;
    }

    // assign elements to another set with ascending order
    set<int> coll2(coll1.begin(),
             coll1.end());

    // print all elements of the copy
    copy (coll2.begin(), coll2.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;

    // remove all elements up to element with value 3
    coll2.erase (coll2.begin(), coll2.find(3));

    // remove all elements with value 5
    int num;
    num = coll2.erase (5);
    cout << num << " element(s) removed" << endl;

    // print all elements
    copy (coll2.begin(), coll2.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

首先，以下的型别定义：

```
typedef set<int,greater<int> > IntSet;
```

定义了一个 set，其中容纳降序（递减）排列的 ints。产生一个空的 sets 之后，首先利用 insert() 安插数个元素：

```
IntSet coll1;

coll1.insert(4);
...
```

注意数值为 5 的元素被安插两次，但第二次安插动作会被程序忽略，因为 sets 不允许数值重复的元素。

打印所有元素后，程序再次安插元素 4。这次按照 p183 的方法来处理 `insert()` 回返值。

以下述句：

```
set<int> coll2(coll1.begin(),coll1.end());
```

产生一个新的 set，其中容纳升序（递增）排列的 `int`s，并以原本那个 sets 的元素做为初值[21]。

两个容器有不同的排序准则，所以它们的型别不同，不能直接相互指派（赋值）或比较。但只要元素型别相同或彼此可以转型，你就可以使用某些「有能力处理不同容器型别」的算法来达成目的。

以下述句：

```
coll2.erase (coll2.begin(), coll2.find(3));
```

移除了数值为 3 的元素之前的所有元素。注意数值为 3 的元素位于序列尾端，所以没被移除。

最后，所有数值为 5 的元素都被移除：

```
int num;
num = coll2.erase (5);
cout << num << " element(s) removed" << endl;
```

程序输出如下：

```
6 5 4 3 2 1
4 already exists
1 2 3 4 5 6
1 element(s) removed
3 4 6
```

[21] 这行述句需要两个语言新性质：member templates 和 default template arguments。如果系统不支持，你必须改成这样：

```
set<int,less<int> > coll2;
copy (coll1.begin(), coll1.end(),
      inserter(coll2,coll2.begin()))
;
```

对于 multisets，上一个程序需要些微改变，并产生不同结果：

```
// cont/mset1.cpp

#include <iostream>
#include <set>
using namespace std;

int main()
{
   /* type of the collection: sets
   * - duplicates allowed
   * - elements are integral values
   * - descending order
   */
   typedef multiset<int,greater<int> > IntSet;

   IntSet coll1; // empty multiset container

   // insert elements in random order
   coll1.insert(4);
   coll1.insert(3);
   coll1.insert(5);
   coll1.insert(1);
   coll1.insert(6);
   coll1.insert(2);
   coll1.insert(5);

   // iterate over all elements and print them
   IntSet::iterator pos;
   for (pos = coll1.begin(); pos != coll1.end(); ++pos) {
        cout << *pos << ' ';
   }
   cout << endl;

   // insert 4 again and process return value
   IntSet::iterator ipos = coll1.insert(4);
   cout << "4 inserted as element "
        << distance(coll1.begin(),ipos) + 1
        << endl;
```

```
    // assign elements to another multiset with ascending order
    multiset<int> coll2(coll1.begin(),
                        coll1.end());

    // print all elements of the copy
    copy (coll2.begin(), coll2.end(),
    ostream_iterator<int>(cout," "));
    cout << endl;

    // remove all elements up to element with value 3
    coll2.erase (coll2.begin(), coll2.find(3));

    // remove all elements with value 5
    int num;
    num = coll2.erase (5);
    cout << num << " element(s) removed" << endl;

    // print all elements
    copy (coll2.begin(), coll2.end(),
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

这个程序把所有的 set 都改为 multiset，此外 `insert()`回返值的处理也有所不同：

```
IntSet::iterator ipos = coll1.insert(4);
cout << "4 inserted as element "
     << distance(coll1.begin(),ipos) + 1
     << endl;
```

由于 multisets 可能包含重复元素，所以安插动作只在异常被抛出时才失败。因此，回返值型别只是一个迭代器，指向新元素位置。

程序输出如下：

```
6 5 5 4 3 2 1
4 inserted as element 5
1 2 3 4 4 5 5 6
2 element(s) removed
3 4 4 6
```

6.5.5 执行期指定排序准则

无论是将排序准则作为第二个 template 自变量传入，或是采用预设的排序准则 less<>，通常你都会将排序准则定义为型别的一部分。但有时必须在执行期处理排序准则，或者你可能需要对同一种数据型别采用不同的排序准则。此时你就需要一个「用来表现排序准则」的特殊型别，使你能够在执行期间传递某个准则。以下范例程序说明了这种做法：

```
// cont/setcmp.cpp

#include <iostream>
#include <set>
#include "print.hpp"
using namespace std;

// type for sorting criterion
template <class T>
class RuntimeCmp {
   public:
      enum cmp_mode {normal, reverse};

   private:
      cmp_mode mode;

   public:
      // constructor for sorting criterion
      // - default criterion uses value normal
      RuntimeCmp (cmp_mode m=normal) : mode(m) {
      }

      // comparision of elements
      bool operator() (const T& t1, const T& t2) const {
         return mode == normal ? t1 < t2 : t2 < t1;
      }

      // comparision of sorting criteria
      bool operator== (const RuntimeCmp& rc) {
         return mode == rc.mode;
   }
};

// type of a set that uses this sorting criterion
typedef set<int,RuntimeCmp<int> > IntSet;
```

```
// forward declaration
void fill (IntSet& set);

int main()
{
   // create, fill, and print set with normal element order
   // - uses default sorting criterion
   IntSet coll1;
   fill(coll1);
   PRINT_ELEMENTS (coll1, "coll1: ");

   // create sorting criterion with reverse element order
   RuntimeCmp<int> reverse_order(RuntimeCmp<int>::reverse);

   // create, fill, and print set with reverse element order
   IntSet coll2(reverse_order);
   fill(coll2);
   PRINT_ELEMENTS (coll2, "coll2: ");

   // assign elements AND sorting criterion
   coll1 = coll2;
   coll1.insert(3);
   PRINT_ELEMENTS (coll1, "coll1: ");

   // just to make sure...
   if (coll1.value_comp() == coll2.value_comp()) {
      cout << "coll1 and coll2 have same sorting criterion"
           << endl;
   }
   else {
      cout << "coll1 and coll2 have different sorting criterion"
           << endl;
   }
}

void fill (IntSet& set)
{
   // fill insert elements in random order
   set.insert(4);
   set.insert(7);
   set.insert(5);
   set.insert(1);
   set.insert(6);
   set.insert(2);
   set.insert(5);
}
```

在这个程序中，RuntimeCmp<> 是一个简单的 template，提供「执行期间面对任意型别定义一个排序准则」的泛化能力。其 *default* 建构式采用默认值 normal，按升幂排序；你也可以将 RuntimeCmp<>::reverse 传递给建构式，便能按降序排序。

程序输出入下：

```
coll1: 1 2 4 5 6 7
coll2: 7 6 5 4 2 1
coll1: 7 6 5 4 3 2 1
coll1 and coll2 have same sorting criterion
```

注意，coll1 和 coll2 拥有相同型别，该型别即 fill() 函数的参数型别。再请注意，*assignment* 运算符不仅指派了元素，也指派了排序准则（否则任何一个指派动作岂不轻易危及排序准则！）

6.6 Maps 和 Multimaps

Map 和 multimap 将 *key/value* pair（键值/实值 对组）当作元素，进行管理。它们可根据 *key* 的排序准则自动将元素排序。multimaps 允许重复元素，maps 不允许，见图 6.8。

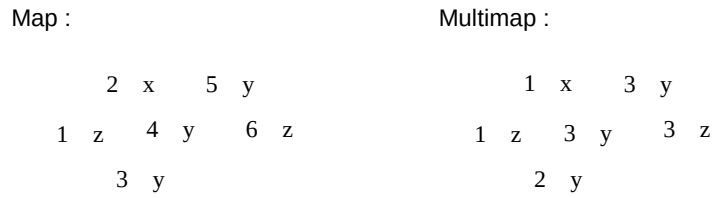

图 6.8 Maps 和 Multimaps

使用 map 和 multimap 之前，你必须先含入头文件 `<map>`[22] ：

```
#include <map>
```

在其中，map 和 multimap 被定义为命名空间 `std` 内的 class templates：

```
namespace std {
   template <class Key, class T,
             class Compare = less<Key>,
             class Allocator = allocator<pair<const Key,T> > >
   class map;

   template <class Key, class T,
             class Compare = less<Key>,
             class Allocator = allocator<pair<const Key,T> > >
   class multimap;
}
```

第一个 template 自变量被当做元素的 *key*，第二个 template 自变量被当做元素的 *value*。Map 和 multimap 的元素型别 `Key` 和 `T`，必须满足以下两个条件：

[22] 在早期 STL 中，maps 被定义于 `<map.h>` 而 mltimaps 被定义于 `<multimap.h>`。

1. *key/value* 必须具备 *assignable*（可赋值的）和 *copyable*（可复制的）性质。

2. 对排序准则而言，*key* 必须是 *comparable*（可比较的）。

第三个 template 自变量可有可无，用来定义排序准则。和 sets 一样，这个排序准则必须定义为 **strict weak ordering**（参见 p176）。元素的次序由它们的 *key* 决定，和 *value* 无关。排序准则也可以用来检查相等性：如果两个元素的 *key* 彼此都不小于对方，则两个元素被视为相等。如果使用者未传入特定排序准则，就使用预设的 `less` 排序准则 — 以 `operator<`来进行比较[23]（`less` 的详细数据请见 p305）。

第四个 template 自变量也是可有可无，用来定义内存模型（详见第 15 章）。预设的内存模型是 `allocator`，由 C++ 标准链接库提供[24]。

6.6.1 Maps 和 Multimaps 的能力

和所有标准的关系型容器一样，maps/multimpas 通常以平衡二元树完成，如图 6.9。

标准规格书并未明定这一点，但是从 map 和 multimap 各项操作的复杂度可以得出这一结论。典型情况下，set, multisets, map, multimaps 使用相同的内部数据结构。因此你可以把 set 和 multisets 分别视为特殊的 map 和 multimaps，只不过 sets 元素的 *value* 和 *key* 是指同一物件。因此 map 和 multimaps 拥有 set 和 multisets 的所有
能力和所有操作函数。当然某些细微差异还是有的：首先，它们的元素是 *key/value* pair，其次，map 可作为关系型数组来运用。

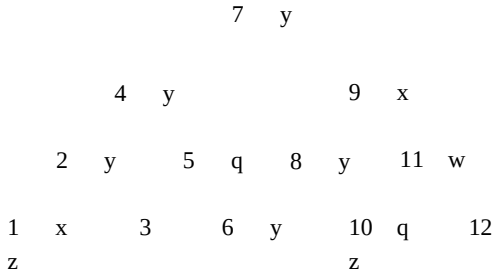

图 6.9 Maps 和 Multimaps 的内部结构

[23] 在不支持 default template parameters 的系统中，第三个参数通常会被省略。

[24] 在不支持 default template parameters 的系统中，第四个参数通常会被省略。

Map 和 multimaps 根据元素的 *key* 自动对元素进行排序。这么一来，根据已知的 *key* 搜寻某个元素时，就能够有很好的性能，而根据已知 *value* 搜寻元素时，性能就很糟糕。「自动排序」这一性质使得 map 和 multimaps 身上有了一条重要的限制：你不可以直接改变元素的 *key*，因为这会破坏正确次序。要修改元素的 *key*，你必须先移除拥有该 *key* 的元素，然后插入拥有新的 *key*/*value* 的元素（详见 p201）。 从迭代器的观点来看，元素的 *key* 是常数。至于元素的 *value* 倒是可以直接修改，当然，前提是 *value* 并非常数型态。

6.6.2 Map 和 Multimap 的操作

生成（Create）、复制（Copy）和销毁（Destroy）

表 6.26 列出 maps 和 multimaps 的生成、复制、销毁等各项操作。

操作	效果
map `c`	产生一个空的 map/multimap，其中不含任何元素。
map `c(op)`	以 `op` 为排序准则，产生一个空的 map/multimap。
map `c1(c2)`	产生某个 map/multimap 的副本，所有元素均被复制。
map	以区间`[beg;end)`内的元素产生一个 map/multimap。
map `c(beg,end,op)`	以 `op` 为排序准则，利用`[beg;end)`内的元素生成一个 map/multimap。
`c.~`***map***`()`	销毁所有元素，释放内存。

其中，***map*** 可为下列型式：

map	效果
`map<Key,Elem>`	一个 map，以 `less<>`（`operator<`）为排序准则。
`map<Key,Elem,Op>`	一个 map，以 `op` 为排序准则。
`multimap<Key,Elem>`	一个 multimap，以 `less<>`（`operator<`）为排序准则。
`multimap<Key,Elem,Op>`	一个 multimap，以 `op` 为排序准则。

表 6.26 Maps 和 Multimaps 的建构式和解构式

有两种方式可以定义排序准则：

1. 以 template 自变量定义之。

例如[25]：

```
std::map<float, std::string, std::greater<float> > coll;
```

这种情况下，排序准则就是型别的一部分。因此型别系统确保「只有排序准则相同的容器才能被合并」。这是比较常见的排序准则指定法。更精确地说，第三参数是排序准则的型别。实际的排序准则是容器所产生的函数对象（*function object*，或称 *functor*）。为了产生它，容器建构式会呼叫「排序准则型别」的 *default* 建构式。p294 有一个「使用者自定之排序准则」的运用实例。

2. 以建构式参数定义之。

在这种情况下，你可以有一个「排序准则型别」并为它指定不同的排序准则（也就是说让该型别所产生出来的对象（代表一个排序准则）的初值或状态不同）。如果执行期才获得排序准则，而且程序需要用到不同的排序准则（但其数据型别必须相同），此一方式可派上用场。一个典型的例子是在执行期指定「*key* 的型别为 `string`」的排序准则。完整例子见 p213。

如果使用者没有提供特定排序准则，就采用预设准则 — 仿函数 `less<>`。`less<>` 系透过 `operator<` 对元素进行排序[26]。

你应当做一些型别定义（typedef），从而简化繁琐的型别表示式：

```
typedef std::map<std::string,float,std::greater<std::string> >
                 StringFloatMap;
...
StringFloatMap coll;
```

某些建构式使用区间起点和终点作为自变量，它们可以使用不同型别的容器、array、标准输入设备（standard input）来进行初始化，详见 6.1.2 节，p144。然而由于元素是 *key/value* pair，因此你必须确定来自源区间的元素型别也是 `pair<`*key*, *value* `>`，或至少可转化成 `pair<`*key*, *value*`>`。

非变动性操作（Nonmodifying Operations）

map 和 multimaps 提供常见的非变动性操作，用来查询大小、相互比较。如表 6.27。

[25] 注意，两个 > 之间需加上一个空格，因为 >> 会被编译程序视为移位运算符，导本处语法错误。

[26] 在不支持 default template parameters 的系统中，通常必须这么设定排序准则：

```
map<float, string, less<float> > coll;
```

操作	效果
`c.size()`	传回容器的大小。
`c.empty()`	判断容器大小是否为零。等同于 `size()==0`，但可能更快。
`c.max_size()`	传回可容纳的最大元素数量。
`c1 == c2`	判断是否 `c1` 等于 `c2`。
`c1 != c2`	判断是否 `c1` 不等于 `c2`。等同于 `!(c1 == c2)`。
`c1 < c2`	判断是否 `c1` 小于 `c2`。
`c1 > c2`	判断是否 `c1` 大于 `c2`。等同于 `c2 < c1`。
`c1 <= c2`	判断是否 `c1` 小于等于 `c2`。等同于 `!(c2 < c1)`。
`c1 >= c2`	判断是否 `c1` 大于等于 `c2`。等同于 `!(c1 < c2)`。

表 6.27 Maps 和 Multimaps 的非变动性操作（Nonmodifying Operations）

元素比较动作只能用于型别相同的容器。换言之，容器的 *key*、*value*、排序准则都必须有相同的型别，否则编译期会产生型别方面的错误。例如：

```
std::map<float,std::string> c1;     // sorting criterion: less<>
std::map<float,std::string,std::greater<float> > c2;
...
if (c1 == c2) { // ERROR: different types
    ...
}
```

比较动作系以「字典（lexicographical）顺序」来检查某个容器是否小于另一个容器（详见 p360）。如果要比较不同型别（拥有不同排序准则）的容器，你必须采用 p356, 9.5.4 节的「比较算法（comparing algorithms）」。

特殊的搜寻动作（Special Search Operations）

就像 set 和 multisets 一样，map 和 multimaps 也提供特殊的搜寻函数，以便利用内部树状结构获取较好的性能，见表 6.28。

成员函数 `find()`用来搜寻拥有某个 *key* 的第一个元素，并传回一个迭代器，指向该位置。如果没找到这样的元素，就传回容器的 `end()`。你不能以 `find()`搜寻拥有某特定 *value* 的元素，你必须改用通用算法如 `find_if()`，或干脆写一个显式循环。下面这个例子便是利用一个简单循环，对拥有特定 *value* 的所有元素进行某项操作:

```
std::multimap<std::string,float> coll;
...
// do something with all elements having a certain value
std::multimap<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
```

操作	效果
count(key)	传回「键值等于 key」的元素个数
find(key)	传回「键值等于 key」的第一个元素，找不到就传回 end()。
lower_bound(key)	传回「键值为 key」之元素的第一个可安插位置，也就是「键值 >= key」的第一个元素位置。
upper_bound(key)	传回「键值为 key」之元素的最后一个可安插位置，也就是「键值 > key」的第一个元素位置。
equal_range(key)	传回「键值为 key」之元素的第一个可安插位置和最后一个可安插位置，也就是「键值 == key」的元素区间。

表 6.28 Maps 和 Multimaps 的特殊搜寻操作函数

```
    if (pos->second == value) {
       do_something();
    }
 }
```

当你要以此类循环来移除元素时，请特别当心。因为可能会发生一些意料之外的事。细节详见 p204。如果使用 find_if() 算法做类似的搜寻动作，会比写一个循环更复杂，因为你必须提供仿函数（*functor*，亦即函数物件 *function object*），将元素的 *value* 拿来和某个 *value* 比较。详见 p211 实例。

至于 lower_bound(), upper_bound(), equal_range()，其行为和 sets（见 p180）的相应函数十分相似，唯一的不同就是：元素是个 *key*/*value* pair。

指派（赋值，Assignments）

Maps 和 multimaps 只支持所有容器都提供的基本指派操作（表 6.29），详见 p147。

操作	效果
c1 = c2	将 c2 中所有元素指派给
c1.swap(c2)	将 c1 和 c2 的元素互换。
swap(c1,c2)	同上。此为全局函数。

表 6.29 Maps 和 Multimaps 的指派（赋值）操作

这些操作函数中，赋值动作的两端容器必须具有相同型别。尽管「比较准则」本身可能不同，但其型别必须相同。p213 列出一个「排序准则不同，但型别相同」的例子。如果准则不同，准则本身也会被指派（assigned）或交换（swapped）。

迭代器函数（Iterator Functions）和元素存取（Element Access）

Map 和 multimaps 不支持元素直接存取，因此元素的存取通常是经由迭代器进行。不过有个例外：map 提供 *subscript*（下标）运算符，可直接存取元素，详见 6.6.3 节, p205。表 6.30 列出 maps 和 multimaps 所支持的迭代器相关函数。

操作	效果
`c.begin()`	传回一个双向迭代器（*key* 被视为常数），指向第一元素
`c.end()`	传回一个双向迭代器（*key* 被视为常数），指向最后元素的下一位置
`c.rbegin()`	传回一个逆向迭代器，指向逆向巡访时的第一个元素
`c.rend()`	传回一个逆向迭代器，指向逆向巡访时的最后元素的下一位置

表 6.30 Maps 和 Multimaps 的迭代器相关操作函数

和其他所有关系型容器类似，这里的迭代器是双向迭代器（参见 p255, 7.2.4 节）。所以，对于只能用于随机存取迭代器的 STL 算法（例如排序或随机乱序算法 *random shuffling*），maps 和 multimaps 就无福消受了。

更重要的是，在 map 和 multimap 中，所有元素的 *key* 都被视为常数。因此元素的
实质型别是 `pair<const key,T>`。这个限制是为了确保你不会因为变更元素的 *key* 而破坏业已排好的元素次序。所以你不能针对 map 或 multimap 呼叫任何变动性演算法（modifying algorithms）。例如你不能对它们呼叫 `remove()`，因为 `remove()` 算法实际上是以一个自变量值覆盖被移除的元素（详细讨论见 p115，5.6.2 节）。如果要移除 map 和 multimaps 的元素，你只能使用它们所提供的成员函数。

下面是 map 迭代器运用实例：

```
std::map<std::string,float> coll;
...
std::map<std::string,float>::iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos) {
     std::cout << "key: " << pos->first << "\t"
               << "value: " << pos->second << std::endl;
}
```

其中迭代器 `pos` 巡访了 `string/float` pair 所组成的序列。以下算式：

```
pos->first
```

获得元素的 *key*，而以下算式：

```
pos->second
```

获得元素的 *value*[27] 。

如果你尝试改变元素的 *key*，会引发错误：

```
pos->first = "hello"; // ERROR at compile time
```

不过如果 *value* 本身的型别并非 const，改变 *value* 没有问题：

```
pos->second = 13.5; // OK
```

如果你一定得改变元素的 *key*，只有一条路：以一个「*value* 相同」的新元素替换掉旧元素。下面是个泛化函数：

```
// cont/newkey.hpp

namespace MyLib {
   template <class Cont>
   inline
   bool replace_key (Cont& c,
                     const typename Cont::key_type& old_key,
                     const typename Cont::key_type& new_key)
   {
      typename Cont::iterator pos;
      pos = c.find(old_key);

      if (pos != c.end()) {
         // insert new element with value of old element
         c.insert(typename Cont::value_type(new_key,
                                            pos->second));
         // remove old element
         c.erase(pos);
         return true;
      }
      else {
         // key not found
         return false;
      }
   }
}
```

关于 insert()和 erase()成员函数，请见下一节讨论。

[27] pos->first 是(*pos).first 的简写形式。有些链接库只支持后一种形式。

这个泛型函数的用法很简单，把旧的 *key* 和新的 *key* 传递进去就行。例如：

```
std::map<std::string,float> coll;
...
MyLib::replace_key(coll,"old key","new key");
```

如果你面对的是 multimaps，情况也一样。

注意，maps 提供了一种非常方便的手法，让你改变元素的 *key*。只需如此这般：

```
// insert new element with value of old element
coll["new_key"] = coll["old_key"];
// remove old element
coll.erase("old_key");
```

关于 maps 的 *subscript*（下标）运算符使用细节，详见 6.6.3 节, p205。

元素的安插（Inserting）和移除（Removing）

表 6.31 列出 maps 和 multimaps 所支持的元素安插和删除函数。

操作	效果
`c.insert(elem)`	安插一份 `elem` 副本，传回新元素位置（不论是否成功 — 对 maps 而言）。
`c.insert(pos,elem)`	安插一份 `elem` 副本，传回新元素位置（`pos` 是个提示，指出安插动作的搜寻起点。如果提示恰当，可大大加快速度）
`c.insert(beg,end)`	将区间`[beg;end)`内所有元素的副本安插到 `c`（无回返值）
`c.erase(elem)`	移除「实值（*value*）与 `elem` 相等」的所有元素，传回被移除的元素个数。
`c.erase(pos)`	移除迭代器 `pos` 所指位置上的元素，无回返值。
`c.erase(beg,end)`	移除区间`[beg;end)`内的所有元素，无回返值。
`c.clear()`	移除全部元素，将整个容器清空。

表 6.31 Maps 和 Multimaps 的元素安插和移除

p182 之中关于 set 和 multisets 的说明，此处依然适用。上述操作函数的回返值型别有些差异，其情况与 set 和 multisets 的情况完全相同。当然，这里的元素是 *key*/*value* pair。所以这里的用法更复杂些。

安插一个 *key*/*value* pair 的时候，你一定要记住，在 map 和 multimaps 内部，*key* 被视为常数。你要不得提供正确型别，要不就得提供隐式或显式型别转换。有三个不同的方法可以将 *value* 传入 map：

1. 运用 value_type

为了避免隐式型别转换，你可以利用 value_type 明白传递正确型别。value_type 是容器本身提供的型别定义。例如：

```
std::map<std::string,float> coll;
...
coll.insert(std::map<std::string,float>::value_type("otto",
                                                    22.3));
```

2. 运用 pair<>

另一个作法是直接运用 pair<>。例如：

```
std::map<std::string,float> coll;
...
// use implicit conversion:
coll.insert(std::pair<std::string,float>("otto",22.3));
// use no implicit conversion:
coll.insert(std::pair<const
std::string,float>("otto",22.3));
```

上述第一个 insert()述句内的型别并不正确，所以会被转换成真正的元素型别。为了做到这一点，insert()成员函数被定义为 member template [28]。

3. 运用 make_pair()

最方便的办法是运用 make_pair()函数（详见 p36）。这个函数根据传入的两个自变量建构出一个 pair 物件：

```
std::map<std::string,float> coll;
...
coll.insert(std::make_pair("otto",22.3));
```

和作法 2 一样，也是利用 member template insert()来执行必要的型别转换。

下面是个简单例子，对着 map 安插一个元素，然后检查是否成功：

```
std::map<std::string,float> coll;
...
if (coll.insert(std::make_pair("otto",22.3)).second) {
    std::cout << "OK, could insert otto/22.3" << std::endl;
}
else {
    std::cout << "Oops, could not insert otto/22.3 "
              << "(key otto already exists)" << std::endl;
}
```

[28] 如果你的系统不支持 member template，你必须传递型别正确的元素，你通常必须因此进行显式型别转换（explicit conversions）。

关于 insert() 回返值的讨论，请见 p182，那儿有更多例子，也适用于 maps。注意此处仍然透过 map 的 *subscript*（下标）运算符提供较为方便的元素安插和设定动作。这一点将在 6.6.3 节, p205 讨论。

如果要移除「拥有某个 *value*」的元素，呼叫 erase() 即可办到：

```
std::map<std::string,float> coll;
...
// remove all elements with the passed key
coll.erase(key);
```

erase() 传回移除元素的个数。对 maps 而言其回返值非 0 即 1。

如果 multimap 内含重复元素，你不能使用 erase() 来删除这些重复元素中的第一个。你可以这么做：

```
typedef std::multimap<std::string,float> StringFloatMMap;
StringFloatMMap coll;
...
// remove first element with passed key
StringFloatMMap::iterator pos;
pos = coll.find(key);
if (pos != coll.end()) {
   coll.erase(pos);
}
```

这里应该采用成员函数 find()，而非 STL 算法 find()，因为前者速度更快（参见 p154 的例子）。然而你不能使用成员函数 find() 来移除「拥有某个 *value*（而非某个 *key*）」的元素。详细讨论请见 p198。

移除元素时，当心发生意外状况。当你移除迭代器所指对象时，有一个很大的危险，看看这个例子：

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ++pos) {
   if (pos->second == value) {
       coll.erase(pos); // RUNTIME ERROR !!!
   }
}
```

对 pos 所指元素实施 erase()，会使 pos 不再成为一个有效的 coll 迭代器。如果
此后你未对 pos 重新设值就径行使用 pos，前途未卜！事实上只要一个 ++pos 动作就会导致未定义的行为。

如果 erase() 总是传回下一元素的位置，那就好办了：

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
for (pos = coll.begin(); pos != coll.end(); ) {
   if (pos->second == value) {
      pos = coll.erase(pos); // would be fine, but COMPILE TIME
ERROR
   }
   else {
      ++pos;
   }
}
```

可惜 STL 设计过程中否决了这种想法，因为万一用户并不需要这一特性，就会耗费不必要的运行时间。我个人不太赞成这项决定，因为这么一来代码会变得更复杂，更容易出错，就时间而言，恐怕得不偿失！

下面是移除「迭代器所指元素」的正确作法：

```
typedef std::map<std::string,float> StringFloatMap;
StringFloatMap coll;
StringFloatMap::iterator pos;
...
// remove all elements having a certain value
for (pos = coll.begin(); pos != coll.end(); ) {
   if (pos->second == value) {
      coll.erase(pos++);
   }
   else {
      ++pos;
   }
}
```

注意，pos++ 会将 pos 移向下一元素，但传回其原始值（指向原位置）的一个副本。因此，当 erase() 被唤起，pos 已经不再指向那个即将被移除的元素了。

6.6.3 将 Maps 视为关系型数组（Associated Arrays）

通常，关系型容器并不提供元素的直接存取，你必须依靠迭代器。不过 maps 是个例外。Non-const maps 提供下标运算符，支持元素的直接存取，如表 6.32。不过，

下标运算符的索引值并非元素整数位置，而是元素的 *key*。也就是说，索引可以是任意型别，而非局限为整数型别。这种接口正是我们所说的关系型数组（associative array）。

操作	效果
`m[key]`	传回一个 reference，指向键值为 `key` 的元素。如果该元素尚未存在，就安插该元素。

表 6.32 Maps 的直接元素存取（透过 `operator[]`）

和一般 array 之间的区别还不仅仅在于索引型别。其他的区别包括：你不可能用上一个错误索引。如果你使用某个 *key* 作为索引，而容器之中尚未存在对应元素，那么就会自动安插该元素。新元素的 *value* 由 *default* 建构式建构。如果元素的 *value* 型别没有提供 *default* 建构式，你就没这个福分了。再次提醒你，所有基本数据型别都提供有 *default* 建构式，以零为初值（见 p14）。

关系型数组的行为方式可说是毁誉参半：

z 优点是你可以透过更方便的接口对着 map 安插新元素。例如：

```
std::map<std::string,float> coll; // empty collection
/* insert "otto"/7.7 as key/value pair
* - first it inserts "otto"/float()
* - then it assigns 7.7
*/
coll["otto"] = 7.7;
```

其中的述句：

```
coll["otto"] = 7.7;
```

处理如下：

1. 处理 `coll["otto"]`：
   - 如果存在键值为 `"otto"` 的元素，以上式子传回该元素的 reference。
   - 如果没有任何元素的键值是 `"otto"`，以上式子便为 map 自动安插一个新元素，键值 *key* 为 `"otto"`，实值 *value* 则以 *default* 建构式完成，并传回一个 reference 指向新元素。

2. 将 7.7 指派给 *value*：
   - 紧接着，将 7.7 指派给上述刚刚诞生的新元素。

这样，map 之内就包含了一个键值（*key*）为 `"otto"`的元素，其实值（*value*）为 7.7。

- 缺点是你可能会不小心误置新元素。例如下面的述句可能会做出一些意想不到的事情：

```
std::cout << coll["ottto"];
```

它会安插一个键值为 "otto" 的新元素，然后打印其实值，预设情况下是 0。然而，按道理它应该产生一条错误讯息，告诉你你把 "otto" 拼写错了。

同时亦请注意，这种元素安插方式比一般的 maps 安插方式来得慢，p202 曾经谈过这个主题。原因是新元素必须先使用 *default* 建构式将实值（*value*）初始化，而这个初值马上又被真正的 *value* 给覆盖了。

6.6.4 异常处理（Exception Handling）

就异常处理而言，Maps 和 multimaps 的行为与 sets 和 multisets 一样。参见 p185。

6.6.5 Maps 和 Multimaps 运用实例

将 Map 当作关系型数组

下面这个例子将 map 当成一个关系型数组来使用。这个 map 用来反映股票行情。元素的键值（*key*）是股票名称，实值（*value*）是股票价格：

```
// cont/map1.cpp

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{
   /* create map / associative array
   * - keys are strings
   * - values are floats
   */
   typedef map<string,float> StringFloatMap;

   StringFloatMap stocks; // create empty container

   // insert some elements
   stocks["BASF"] = 369.50;
```

```
    stocks["VW"] = 413.50;
    stocks["Daimler"] = 819.00;
    stocks["BMW"] = 834.00;
    stocks["Siemens"] = 842.20;

    // print all elements
    StringFloatMap::iterator pos;
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
         cout << "stock: " << pos->first << "\t"
              << "price: " << pos->second << endl;
    }
    cout << endl;

    // boom (all prices doubled)
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
         pos->second *= 2;
    }

    // print all elements
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
         << "price: " << pos->second << endl;
    }
    cout << endl;

    /* rename key from "VW" to "Volkswagen"
    * - only provided by exchanging element
    */
    stocks["Volkswagen"] = stocks["VW"];
    stocks.erase("VW");

    // print all elements
    for (pos = stocks.begin(); pos != stocks.end(); ++pos) {
    cout << "stock: " << pos->first << "\t"
         << "price: " << pos->second << endl;
    }
}
```

程序输出如下:

```
stock: BASF price: 369.5
stock: BMW price: 834
stock: Daimler price: 819
stock: Siemens price: 842.2
stock: VW price: 413.5

stock: BASF price: 739
stock: BMW price: 1668
stock: Daimler price: 1638
stock: Siemens price: 1684.4
stock: VW price: 827

stock: BASF price: 739
stock: BMW price: 1668
stock: Daimler price: 1638
stock: Siemens price: 1684.4
stock: Volkswagen price: 827
```

将 Multimap 当作字典

下面例子展示如何将 multimap 当成一个字典来使用:

```
// cont/mmap1.cpp

#include <iostream>
#include <map>
#include <string>
#include <iomanip>
using namespace std;

int main()
{
   // define multimap type as string/string dictionary
   typedef multimap<string,string> StrStrMMap;

   // create empty dictionary
   StrStrMMap dict;

   // insert some elements in random order
   dict.insert(make_pair("day","Tag"));
```

```
    dict.insert(make_pair("strange","fremd"));
    dict.insert(make_pair("car","Auto"));
    dict.insert(make_pair("smart","elegant"));
    dict.insert(make_pair("trait","Merkmal"));
    dict.insert(make_pair("strange","seltsam"));
    dict.insert(make_pair("smart","raffiniert"));
    dict.insert(make_pair("smart","klug"));
    dict.insert(make_pair("clever","raffiniert"));

    // print all elements
    StrStrMMap::iterator pos;
    cout.setf (ios::left, ios::adjustfield);
    cout << ' ' << setw(10) << "english "
         << "german " << endl;
    cout << setfill('-') << setw(20) << ""
         << setfill(' ') << endl;
    for (pos = dict.begin(); pos != dict.end(); ++pos) {
         cout << ' ' << setw(10) << pos->first.c_str()
              << pos->second << endl;
    }
    cout << endl;

    // print all values for key "smart"
    string word("smart");
    cout << word << ": " << endl;
    for (pos = dict.lower_bound(word);
         pos != dict.upper_bound(word); ++pos) {
         cout << "     " << pos->second << endl;
    }

    // print all keys for value "raffiniert"
    word = ("raffiniert");
    cout << word << ": " << endl;
    for (pos = dict.begin(); pos != dict.end(); ++pos) {
         if (pos->second == word) {
              cout << " " << pos->first << endl;
         }
    }
}
```

程序输出如下:

```
  english    german
------------------------
- car        Auto
  clever     raffinier
  day        t
  smart      Tag
  smart      elegant
  smart      raffinier
  strange    t
  strange    klug
  trait      fremd
             seltsam
smart:       Merkmal
   elegant
   raffinier
t
   klug
raffiniert:
   clever
   smart
```

搜寻具有某特定实值（values）的元素

下面例子展示如何使用全局的 `find_if()`算法来搜寻具有某特定 *value* 的元素:

```
// cont/mapfind.cpp

#include <iostream>
#include <algorithm>
#include <map>
using namespace std;

/* function object to check the value of a map element
*/
template <class K, class V>
class value_equals {
   private:
   V value;

   public:
   // constructor (initialize value to compare with)
```

```
    value_equals (const V& v)
      : value(v) {
    }

    // comparison
    bool operator() (pair<const K, V> elem) {
    return elem.second == value;
    }
};

int main()
{
    typedef map<float,float> FloatFloatMap;
    FloatFloatMap coll;
    FloatFloatMap::iterator pos;

    // fill container
    coll[1]=7;
    coll[2]=4;
    coll[3]=2;
    coll[4]=3;
    coll[5]=6;
    coll[6]=1;
    coll[7]=3;

    // search an element with key 3.0
    pos = coll.find(3.0); // logarithmic complexity
    if (pos != coll.end()) {
       cout << pos->first << ": "
            << pos->second << endl;
    }

    // search an element with value 3.0
    pos = find_if(coll.begin(),coll.end(), // linear complexity
                  value_equals<float,float>(3.0));
    if (pos != coll.end()) {
         cout << pos->first << ": "
              << pos->second << endl;
    }
}
```

程序输出如下：

```
3: 2
4: 3
```

6.6.6 综合实例：运用 Maps, Strings 并于执行期指定排序准则

这里再示范一个例子。此例针对高级程序员而非 STL 初学者。你可以把它视为展现 STL 威力与障碍的一个范例。更明确地说，这个例子展现了以下技巧：

- 如何使用 maps
- 如何撰写和使用仿函数（functor, 或名 function object）
- 如何在执行期定义排序准则
- 如何在「不在乎大小写」的情况下比较字符串（strings）

```
// cont/mapcmp.cpp

#include <iostream>
#include <iomanip>
#include <map>
#include <string>
#include <algorithm>
using namespace std;

/* function object to compare strings
* - allows you to set the comparison criterion at runtime
* - allows you to compare case insensitive
*/
class RuntimeStringCmp {
  public:
    // constants for the comparison criterion
    enum cmp_mode {normal, nocase};

  private:
    // actual comparison mode
    const cmp_mode mode;

    // auxiliary function to compare case insensitive
    static bool nocase_compare (char c1, char c2)
    {
       return toupper(c1) < toupper(c2);
    }
```

```
  public:
    // constructor: initializes the comparison criterion
    RuntimeStringCmp (cmp_mode m=normal) : mode(m) {
    }

    // the comparison
    bool operator() (const string& s1, const string& s2) const {
       if (mode == normal) {
          return s1 < s2;
       }
       else {
          return lexicographical_compare(s1.begin(), s1.end(),
                                         s2.begin(), s2.end(),
                                         nocase_compare);
       }
    }
};

/* container type:
* - map with
* -string keys
* -string values
* - the special comparison object type
*/
typedef map<string, string, RuntimeStringCmp> StringStringMap;

// function that fills and prints such containers
void fillAndPrint(StringStringMap& coll);

int main()
{
   // create a container with the default comparison criterion
   StringStringMap coll1;
   fillAndPrint(coll1);
```

```
    // create an object for case-insensitive comparisons
    RuntimeStringCmp
ignorecase(RuntimeStringCmp::nocase);

    // create a container with the case-insensitive
    // comparisons criterion
    StringStringMap coll2(ignorecase);
    fillAndPrint(coll2);
}

void fillAndPrint(StringStringMap& coll)
{
    // fill insert elements in random order
    coll["Deutschland"] = "Germany";
    coll["deutsch"] = "German";
    coll["Haken"] = "snag";
    coll["arbeiten"] = "work";
    coll["Hund"] = "dog";
    coll["gehen"] = "go";
    coll["Unternehmen"] = "enterprise";
    coll["unternehmen"] = "undertake";
    coll["gehen"] = "walk";
    coll["Bestatter"] = "undertaker";

    // print elements
    StringStringMap::iterator pos;
    cout.setf(ios::left, ios::adjustfield);
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
       cout << setw(15) << pos->first.c_str() << " "
            << pos->second << endl;
    }
    cout << endl;
}
```

`main()`建构出两个容器，并对它们呼叫 `fillAndPrint()`。这个函数以相同的元素值填充上述两个容器，然后打印其内容。两个容器的排序准则不同：

1. `coll1` 使用一个型别为 `RuntimeStringCmp` 的预设仿函数。这个仿函数以元素的 `operator<` 来执行比较动作。
2. `coll2` 使用一个型别为 `RuntimeStringCmp` 的仿函数，并以 `nocase` 为初值。`nocase` 会令这个仿函数以「大小写无关」模式来完成字符串的比较和排序。

程序输出如下：

```
Bestatter   undertake
Deutschland r
Germany     snag
Haken       dog
Hund        work
Unternehmen German
arbeiten    walk
deutsch
gehen
unternehmen
undertake
arbeiten    work
Bestatter   undertake
deutsch     r
Deutschland German
Germany     walk
gehen       snag
Haken       dog
Hund
Unternehmen
undertake
```

第一部分打印第一个容器的内容，该容器以 `operator<` 进行排序。首先输出所有键值为大写的字符串，然后是键值为小写的字符串。

第二部分以「大小写无关」模式打印所有字符串，次序和第一部分不同。请注意，第二部分少列了一个元素，因为在大小写无关的情况下 `"Unternehmen"`和 `"unternehmen"` 被视为两个相同字符串[29]，而我们使用的 map 并不接纳重复元素。很不幸，打印结果乱七八糟。原本「*value* 应为 `"enterprise"`」的那个 *key*（是个德文字），其 *value* 却变成 `"undertake"`。看来这里应该使用 multimap。没错，multimap 的确是用来表现字典的一个典型容器。

[29] 德语中的所有名词，第一个字母皆大写。动词全部小写。

6.7 其他 STL 容器

STL 是个框架，除了提供标准容器，它也允许你使用其他数据结构作为容器。你可以使用 string 或 array 作为 STL 容器，也可以自行撰写特殊容器以满足特殊需求。如果你自行撰写容器，仍可从诸如排序、合并等算法中受益。这样的框架正是「开放性封闭（*Open-Closed*）」原则的极佳范例[30]：允许扩展，谢绝修改。

下面是使你的容器「STL 化」的三种不同方法：

1. **The invasive approach**[31]（侵入性作法）
   直接提供 STL 容器所需接口。特别是诸如 `begin()`和 `end()`之类的常用函数。这种作法需以某种特定方式编写容器，所以是侵入性的。
2. **The noninvasive approach**[31]（非侵入性作法）
   由你撰写或提供特殊迭代器，作为算法和特殊容器间的界面。此一作法是非侵入性的，它所需要的只是「巡访容器所有元素」的能力 — 这是任何容器都能以某种形式展现的能力。
3. **The wrapper approach**（包装法）
   将上述两种方法加以组合，我们可以写一个外套类别（wrapper class）来包装任何数据结构，并显暴出与 STL 容器相似的接口。

本节首先将 string 视为标准容器来讨论，当作侵入性作法的一个例子，然后再以非侵入性作法讨论重要的标准容器：array。当然你也可以使用包装法来存取 array 的资料。本节最后概略讨论了一个目前尚未被涵盖于标准规格中的容器：**hash table**。

任何 STL 容器都应该能够以不同的配置器（allocator）加以参数化。C++ 标准程式库提供了一些特殊函数和类别，帮助你撰写配置器并对付尚未初始化的内存。
详见 15.2 节, p728。

6.7.1 Strings 可被视为一种 STL 容器

C++ 标准链接库的 `string` 类别，乃是「以侵入性作法编写 STL 容器」的一个好例子（关于 `string` 类别的详尽讨论，请见第 11 章）。Strings 可被视为以字符（characters）为元素的一种容器；字符构成序列，你可以在序列上来回移动巡访。因此，标准的 `string` 类别提供了 STL 容器接口。Strings 也提供成员函数 `begin()`和 `end()`，传回随机存取迭代器，可用来巡访整个 string。同时，为了支持迭代器和迭代器配接器（iterator adapters），strings也提供了一些操作函数，例如`push_back()`用以支持 *back inserters*。

30 我从 Robert C. Martin 那儿头一次听说这个名称，他是从 Bertrand Meyer 那儿听来的。

31 有时也说成 **intrusive** 和 **nonintrusive**

从 STL 角度来思考，string 的处理有点不寻常，因为我们通常将 string 当作一个物件来处理（我们可以传递、复制或设定 string）。但如果要对单个字符进行处理，采用 STL 算法将大有裨益。例如可以采用 istream 迭代器读取字符，或转换 string 内的字符（譬如转成大写或小写）。此外，透过 STL 算法，可以对 string 采取特殊的比较规则 — 标准 string 接口并不提供这种能力。

p497, 11.2.13 节是 string 完整章节的一部分，在那里我详细讨论了 string 的 STL 相关特性，并给出一些实例。

6.7.2 Arrays 可被视为一种 STL 容器

我们也可以把 array当成 STL 容器来使用，但 array并不是类别，所以不提供 begin() 和 end()等成员函数，也不允许存在任何成员函数。在这里，我们只能采用非侵入性作法或包装法。

直接运用 array

采取非侵入性作法很简单，你只需要一个对象，它能够透过 STL 迭代器接口，巡访 array 的所有元素。事实上这样的对象早就恭候多时了，就是一般指标。STL 设计之初就决定让迭代器拥有和一般指针相同的接口，于是你可以将一般指标当成迭代器来使。这又一次展示了纯粹抽象的泛化概念：「行为类似迭代器」的任何东西就是一种迭代器。事实上指标正是一个随机存取迭代器（参见 p255, 7.2.5 节）。以下例子示范如何以 array 作为 STL 容器：

```
// cont/array1.cpp

#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;

int main()
{
   int coll[] = { 5, 6, 2, 4, 1, 3 };

   // square all elements
   transform (coll, coll+6,       // first source
              coll,               // second source
              coll,               // destination
              multiplies<int>())  // operation
   ;
```

```
    // sort beginning with the second element
    sort (coll+1, coll+6);

    // print all elements
    copy (coll, coll+6,
          ostream_iterator<int>(cout," "));
    cout << endl;
}
```

千万注意，一定要正确传递 array 尾部位置，这里是 `coll+6`。记住，一定要确保区间尾端是最后元素的下一个位置。

程序输出如下：

```
25 1 4 9 16 36
```

p382 和 p421 还有一些例子。

一个 array 外包装

Bjarne Stroustrup 的《*The C++ Programming Language*》第三版中，介绍了一个很有用的 array 包装类别，性能不输一般的 array，而且更安全。这是「使用者自行定义 STL 容器」的一个好例子。该容器所使用的，就是包装法：在 array 之外包装一层常用的容器界面。

Class `carray`（这是 "C array" 或 "constant size array" 的缩写）定义如下[32]：

```
// cont/carray.hpp

#include <cstddef>

template<class T, std::size_t thesize>
class carray {
   private:
   T v[thesize]; // fixed-size array of elements of type T

   public:
   typedef T value_type;
   typedef T* iterator;
```

[32] 原始例子名为 `c_array`，定义于 Bjarne Stroustrup 的《*The C++ Programming Language*》第三版 17.5.4 节。这里我做了一些改动。

```
    typedef const T*      const_iterator;
    typedef T&            reference;
    typedef const T&      const_reference
    typedef std::size_t   ;
    typedef std::ptrdiff_t difference_type;

    // iterator support
    iterator begin() { return v; }
    const_iterator begin() const { return v; }
    iterator end() { return v + thesize; }
    const_iterator end() const { return v+thesize; }

    // direct element access
    reference operator[](std::size_t i) { return v[i]; }
    const_reference operator[](std::size_t i) const { return v[i]; }

    // size is constant
    size_type size() const { return thesize; }
    size_type max_size() const { return thesize; }

    // conversion to ordinary array
    T* as_array() { return v; }
};
```

下面是 carray 的一个运用实例：

```
// cont/carray1.cpp

#include <algorithm>
#include <functional>
#include "carray.hpp"
#include "print.hpp"
using namespace std;

int main()
{
   carray<int,10> a;

   for (unsigned i=0; i<a.size(); ++i) {
        a[i] = i+1;
   }
```

```
    PRINT_ELEMENTS(a);

    reverse(a.begin(),a.end());
    PRINT_ELEMENTS(a);

    transform(a.begin(),a.end()     // source
    ,                               // destination
              a.begin(),            // operation
              negate<int>());
}   PRINT_ELEMENTS(a);
```

如你所见，你可以使用一般容器接口（`begin()`, `end()`, `operator[]`）来直接操作这个容器。这么一来你也就可以使用那些需要呼叫 `begin()`和 `end()`的各项操作了，例如某些 STL 算法，以及 p118 所介绍的辅助函数 `PRINT_ELEMENTS()`。

程序输出如下：

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1
```

（译注：关于这个 class，更细致的实作手法请参考 http://www.boost.org/ 的 Boost 链接库）

6.7.3 Hash Tables

有一个数据结构可用于群集（collection）身上，非常重要，却未包含于 C++ 标准链接库内，那就是 hash table。最初的 STL 并未涵盖 hash table，然而确实曾有提
案要求，将 hash table 并入标准规格。但是标准委员会觉得这份提议来得太晚，没有采纳。（我们必须在某个时间点中止引入新功能，开始关注细节，否则工作永无止境）

不过，C++ 社群早已经有了数种可用的 hash table 实作版本。一般而言链接库会提供㈣ 种 hash table：hash_set, hash_multiset, hash_map, hash_multimap。和其他关联式容器一样，"multi" 版允许元素重复，"map" 版的元素是个 *key*/*value* pair。Bjarne Stroustrup 在其《*The C++ Programming Language*》第三版 17.6 节中曾经实作一个
`hash_map` 容器作为示范，并进行详细讨论。关于 hash table 的具体实现，可参见 STLport（http:://www.stlport.org）。当然，由于 hash table 尚未正规化，所以不同的实作版本可能在细节上有所不同。（译注：如果你对 hash table 的运用和设计原理感兴趣，请参考《STL 源码剖析》by 侯捷, 碁峰 2002，5.7 节~5.11 节。该处讨论的是 SGI STL 实作版本，涵括底层 hash table 和外显接口 hash_set, hash_multiset, hash_map, hash_multimap）

6.8 动手实现 Reference 语意

通常，STL 容器提供的是「*value* 语意」而非「*reference* 语意」，后者在内部建构了元素副本，任何操作传回的也是这些副本。p135, 5.10.2 节讨论了这种作法的优劣，并分析了产生后果。总之，要在 STL 容器中用到「*reference* 语意」（不论是因为元素的复制代价太大，或因为需要在不同群集中共享同一个元素），就要采用智能型指标，避免可能的错误。这里有一个解决办法：对指针所指之对象采用 **reference counting**（参用计数）智能型指标[33]。

```
// cont/countptr.hpp

#ifndef COUNTED_PTR_HPP
#define COUNTED_PTR_HPP

/* class for counted reference semantics
* - deletes the object to which it refers when the last CountedPtr
* that refers to it is destroyed
*/
template <class T>
class CountedPtr {
  private:
    T* ptr;      // pointer to the value
    long* count;  // shared number of owners

  public:
    // initialize pointer with existing pointer
    // - requires that the pointer p is a return value of new
    explicit CountedPtr (T* p=0)
      : ptr(p), count(new long(1)) {
    }

    // copy pointer (one more owner)
    CountedPtr (const CountedPtr<T>& p) throw()
      : ptr(p.ptr), count(p.count) {
        ++*count;
    }
```

[33] 感谢 Greg Colvin 和 Beman Dawes 对这个 class 的实作内容所给予的回应。

```
    // destructor (delete value if this was the last owner)
    ~CountedPtr () throw() {
       dispose();
    }

    // assignment (unshare old and share new value)
    CountedPtr<T>& operator= (const CountedPtr<T>& p) throw() {
       if (this != &p) {
          dispose();
          ptr = p.ptr;
          count = p.count;
          ++*count;
       }
       return *this;
    }

    // access the value to which the pointer refers
    T& operator*() const throw() {
       return *ptr;
    }
    T* operator->() const throw() {
       return ptr;
    }

  private:
    void dispose() {
       if (--*count == 0) {
          delete count;
          delete ptr;
       }
    }

};

#endif /*COUNTED_PTR_HPP*/
```

这个 class 有点类似标准规格书所提供的 `auto_ptr` class（参见 p38, 4.2 节）。用
来初始化智能型指标的值，应当是 `operator new` 的回返值。但是和 `auto_ptr` 不同的是，这种智能型指标一旦被复制，原指标和新的副本指标都是有效的。只有当指向同一对象的最后一个智能型指标被摧毁，其所指对象才会被删除。

你可以改善这个 class，例如你可以为它实现自动型别转换，或是提供「将拥有权由智能型指标交至呼叫端手上」的能力。

以下程序说明如何使用这个 class：

```
// cont/refsem1.cpp

#include <iostream>
#include <list>
#include <deque>
#include <algorithm>
#include "countptr.hpp"
using namespace std;

void printCountedPtr (CountedPtr<int> elem)
{
   cout << *elem << ' ';
}

int main()
{
   // array of integers (to share in different containers)
   static int values[] = { 3, 5, 9, 1, 6, 4 };

   // two different collections
   typedef CountedPtr<int> IntPtr;
   deque<IntPtr> coll1;
   list<IntPtr> coll2;

   /* insert shared objects into the collections
   * - same order in coll1  coll1
   * - reverse order in coll2 coll2
   */
   for (int i=0; i<sizeof(values)/sizeof(values[0]); ++i) {
        IntPtr ptr(new int(values[i]));
        coll1.push_back(ptr);
        coll2.push_front(ptr);
   }
```

```
    // print contents of both collections
    for_each (coll1.begin(), coll1.end(),
              printCountedPtr);
    cout << endl;
    for_each (coll2.begin(), coll2.end(),
              printCountedPtr);
    cout << endl << endl;

    /* modify values at different places
    * - square third value in coll1
    * - negate first value in coll1
    * - set first value in coll2 to 0
    */
    *coll1[2] *= *coll1[2];
    (**coll1.begin()) *= -1;
    (**coll2.begin()) = 0;

    // print contents of both collections again
    for_each (coll1.begin(), coll1.end(),
              printCountedPtr);
    cout << endl;
    for_each (coll2.begin(), coll2.end(),
              printCountedPtr);
    cout << endl;
}
```

程序输出如下：

```
3 5 9 1 6 4
4 6 1 9 5 3

-3 5 81 1 6 0
0 6 1 81 5 -3
```

注意，如果你呼叫一个辅助函数，而它在某处保存了群集（collection）内的某个元素（一个 `IntPtr`），那么即使群集被销毁，或其元素全被删除，那个智能型指标所指的元素依然有效。

关于其他智能型指标类别，请参考 http://www.boost.org/ 的 Boost 链接库，该程序库是 C++ 标准链接库的扩充（在那儿 `CountedPtr<>` 名为 `shared_ptr<>`）。

6.9 各种容器的运用时机

C++ 标准链接库提供了各具特长的不同容器。现在的问题是：该如何选择最佳的容器类别？表 6.33 作了一番概述，但其中有些描述可能不一定实际。例如，如果你需要处理的元素数量很少，可以忽略复杂度，因为线性算法通常对元素本身的处理过程比较快，这种情况下，「线性复杂度搭配快速的元素处理」要比「对数复杂度搭配缓慢的元素处理」来得划算。

以下规则做为表 6.33 的补充，可能对你有所帮助：

- 预设情况下应该使用 vector。vector 的内部结构最简单，并允许随机存取，所以数据的存取十分方便灵活，数据的处理也够快。
- 如果经常要在序列头部和尾部安插和移除元素，应该采用 deque。如果你希望元素被移除时，容器能够自动缩减内存，那么你也应该采用 deque。此外，由于 vectors 通常采用一个内存区块来存放元素，而 deque 采用多个区块，所以后者可内含更多元素。
- 如果需要经常在容器的中段执行元素的安插、移除和移动，可考虑使用 list。List 提供特殊的成员函数，可以在常数时间内将元素从 A 容器转移到 B 容器。但由于 list 不支持随机存取，所以如果只知道 list 的头部却要造访 list 的中段元素，性能会大打折扣。

  和所有「以节点为基础」的容器相似，只要元素还是容器的一部分，list 就不会令指向那些元素的迭代器失效。vectors 则不然，一旦超过其容量，它的所有 iterators、pointers、references 都会失效；执行安插或移除动作时，也会令一部分 iterators、pointers、references失效。至于 deque，当它的大小改变，所有 iterators、pointers、references 都会失效。
- 如果你要的容器是这种性质：「每次操作若不成功，便无效用」（并以此态度来处理异常），那么你应该选用 list（但是不保证其 *assignment* 运算符和 `sort()`；而且如果元素比较过程中会掷出异常，那就不要呼叫 `merge()`,`remove()`, `remove_if()`,`unique()`，参见 p172），或是采用关联式容器（但是不保证多元素安插动作，而且如果比较准则（comparision criterion）的复制/指派动作都可能丢掷异常，那么也不保证 `swap()`）。STL 的异常处理通论请见 p139，5.11.2 节。6.10.10 节， p249 提供了一个表，列举出「异常发生时提供特别保障」的所有容器操作函数。
- 如果你经常需要根据某个准则来搜寻元素，那么应当使用「以该排序准则对元素进行排序」的 set 或 multiset。记住，理论上，面对 1,000 个元素的排序，对数复杂度比线性复杂度好 10 倍。此时正是二元树拿手好戏的发挥时机。

	Vector	Deque	List	Set	Multiset	Map	Multimap
典型内部结构	dynamic array	array of arrays	doubly linked list	binary tree	biary tree	binary tree	binary tree
元素 value	value	value	value	value	value	*key*/*value* pair	*key*/*value* pair
元素可重复	是	是	是	否	是	对 *key* 而言否	是
可随机存取	是	是	否	否	否	对 *key* 而言是	否
迭代器类型	随机存取	随机存取	双向	双向 元素被视为常数	双向 元素被视为常数	双向 *key* 被视为常数	双向 *key* 被为常数
元素搜寻速度	慢	慢	非常慢	快	快	对 *key* 而言快	对 *key* 而言快
快速安插移除	尾端	头尾两端	任何位置	—	—	—	—
安插移除导致除效 iterators, pointers, references	重新配置时	总是如此	绝不会	绝不会	绝不会	绝不会	绝不会
释放被移除元素之内存	绝不会	有时会	总是如此	总是如此	总是如此	总是如此	总是如此
允许保留内存	是	否	—	—	—	—	—
交易安全若失败不带来任何影响	尾端 push/pop 时	头尾两端 push/pop 时	任何时候除了排序和赋值	任何时候除了多元素安插	任何时候除了多元素安插	任何时候除了多元素安插	任何时候除了多元素安插

表 6.33 STL 容器能力一览表

就搜寻速度而言，hash table 通常比二元树还要快 5~10 倍。所以如果有 hash table 可用，就算它尚未标准化，也应该考虑使用。但是 hash table 的元素并未排序，所以如果元素必须排序，它就用不上了。由于 hash table 不是 C++ 标准链接库的一员，如果你要保证可移植性，就必须拥有其源码。

- 如想处理 *key*/*value* pair，请采用 map 或 multimap（可以的话请采用 hash table）。
- 如果需要关系型数组，应采用 map。
- 如果需要字典结构，应采用 multimap。

有一个问题比较棘手：如何根据两种不同的排序准则对元素进行排序？例如存放元素时，你希望采用客户提供的排序准则，搜寻元素时，希望使用另一个排序准则。这和数据库的情况相同，你需要在数种不同的排序准则下进行快速存取。这时候你可能需要两个 sets 或 maps，各自拥有不同的排序准则，但共享相同的元素。注意，数个群集共享相同的元素，乃是一项特殊技术，6.8 节, p222 对此有所阐述。

关系型容器拥有自动排序能力，并不意味它们在排序方面的执行效率更高。事实上由于关系型容器每安插一个新元素，都要进行一次排序，所以速度反而不及序列式容器经常采用的手法：先安插所有元素，然后呼叫 9.2.2 节, p328 介绍的排序算法进行一次完全排序。

下面两个简单的程序分别使用不同的容器，从标准输入读取字符串，进行排序，然后打印所有元素（去掉重复字符串）：

1. 运用 **set**：

```
// cont/sortset.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <set>
using namespace std;

int main()
{
   /* create a string set
   * - initialized by all words from standard input
   */
   set<string>
   coll((istream_iterator<string>(cin)),
                  (istream_iterator<string>()));

   // print all elements
   copy (coll.begin(), coll.end(),
}        ostream_iterator<string>(cout, "\n"));
```

2. 运用 vector：

```
// cont/sortvec.cpp

#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
using namespace std;

int main()
{
   /* create a string vector
   * - initialized by all words from standard input
   */
   vector<string>
   coll((istream_iterator<string>(cin)),
                     (istream_iterator<string>()));

   // sort elements
   sort (coll.begin(), coll.end());

   // print all elements ignoring subsequent duplicates
   unique_copy (coll.begin(), coll.end(),
}               ostream_iterator<string>(cout, "\n"));
```

我在我的系统上使用大约 150,000 个字符串来测试这两个程序，我发现 vectors 版本快 10% 左右。如果使用 `reserve()`，vectors 版本还可以再快将近 5%。如果允许重复元素（改用 multiset 取代 set，呼叫 `copy()` 取代 `unique_copy()`），则情况发生剧烈变化：vectors 版本领先超过 40%！这些比较虽然不具代表性，但至少证实了一点：对各种不同的元素处理方法多加尝试是值得的。

现实中预测哪种容器最好，往往相当困难。STL 的一大优点就是你可以轻而易举地尝试各种版本。主要工作 — 各种数据结构和算法 — 已经就位，你只需依照对自己最有利的方式将它们组合运用就行了。

6.10 细说容器内的型别和成员

本节讨论各种 STL 容器，阐述 STL 容器所支持的一切操作函数。型别和成员一律按功能分组。针对每一种型别定义和操作，本节描述其标记式（signature）、行为、支援者（容器）。本节涉及的容器包括 vectors、deques、lists、sets、multisets、maps、multimaps 和 strings。后续数节中 *container* 指的是「支援该成员」的某容器型别。

6.10.1 容器内的型别

container::**value_type**
- 元素型别。
- 用于 sets 和 multisets 时是常数。
- 用于 maps 和 multimaps 时是 pair <const *key-type*, *value-type*>。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 之中都有定义。

container::**reference**
- 元素的引用型别（reference type）。
- 典型定义：*container*::value_type&。
- 在 vector<bool> 中其实是个辅助类别（参见 p158）。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**const_reference**
- 常数元素的引用型别（reference type）。
- 典型定义：const *container*::value_type&。
- 在 vector<bool> 中是 bool。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**iterator**
- 迭代器型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**const_iterator**
- 常数迭代器的型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**reverse_iterator**
- 反向迭代器型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps 中都有定义。

container::**const_reverse_iterator**
- 常数反向迭代器的型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**size_type**
- 无正负号整数型别，用以定义容器大小。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**difference_type**
- 有正负号整数型别，用以定义距离。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

container::**key_type**
- 用以定义关系型容器的元素内的 *key* 型别。
- 用于 sets 和 multisets 时，相当于 `value_type`。
- 在 sets, multisets, maps, multimaps 中都有定义。

container::**mapped_type**
- 用以定义关系型容器的元素内的 *value* 型别。
- 在 maps 和 multimaps 中都有定义。

container::**key_compare**
- 关系型容器内的「比较准则」的型别。
- 在 sets, multisets, maps, multimaps 中都有定义。

container::**value_compare**
- 用于整个元素之「比较准则」的型别。
- 用于 sets 和 multisets 时，相当于 `key_compare`。
- 在 maps 和 multimaps 中，它是「比较准则」的辅助类别，仅比较两元素的 *key*。
- 由 sets, multisets, map, multimap 中都有定义。

container::**allocator_type**
- 配置器型别。
- 在 vectors, deques, lists, sets, multisets, maps, multimaps, strings 中都有定义。

6.10.2 生成（Create）、复制（Copy）、销毁（Destroy）

STL 容器支持下列建构式和解构式，并且大多数建构式允许将配置器作为一个附加自变量传递（参见第 6.10.9 节, p246）。

container::*container* ()
- *default* 建构式
- 产生一个新的空容器
- vectors, deques, lists, sets, multisets, maps, multimaps, strings 都支援

```
explicit container::container (const CompFunc& op)
```
- 以 op 为排序准则，产生一个空容器（参见 p191 和 p213 实例）。
- 排序准则必须定义一个 **strict weak ordering**（参见 p176）。
- sets, multisets, maps, multimaps 支援。

```
explicit container::container (const container& c)
```
- *copy* 建构式。
- 产生既有容器的一个副本。
- 针对 c 中的每一个元素呼叫 *copy* 建构式。
- vectors, deques, lists, sets, multisets, maps, multimaps, strings 都支援。

```
explicit container::container (size_type num)
```
- 产生一个容器，可含 *num* 个元素。
- 元素由其 *default* 建构式创建。
- vectors, deques, lists 都支援。

```
container::container (size_type num, const T& value)
```
- 产生一个容器，可含 *num* 个元素。
- 所有元素都是 *value* 的副本。
- T 是元素型别。
- 对于 strings，*value* 并非 *pass by reference*。
- vectors、deques、lists 和 strings 都支援。

```
container::container (InputIterator beg, InputIterator end)
```
- 产生容器，并以区间[beg;end)内的所有元素为初值。
- 此函数为一个 member template（参见 p11）。因此只要源区间的元素型别可转换为容器元素型别，此函数即可派上用场。
- vectors、deques、lists、sets、multisets、maps、multimaps、strings 都支援。

```
container::container (InputIterator beg, InputIterator end,
                      const CompFunc& op)
```
- 产生一个排序准则为 *op* 的容器，并以区间[beg;end)内的所有元素进行初始化。
- 此函数为一个 member template（参见 p11）。因此只要源区间的元素型别可转换为容器元素型别，此函数即可派上用场。
- 排序准则必须定义一个 **strict weak ordering**（参见 p176）。
- sets、multisets、maps 和 multimaps 都支援。

container::~*container* ()
- 解构式。
- 移除所有元素，并释放内存。
- 对每个元素呼叫其解构式。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

6.10.3 非变动性操作（Nonmodifying Operations）

大小相关操作（Size Operations）

size_type *container*::**size** () const
- 传回现有元素的数目。
- 欲检查容器是否为空，应使用 empty()，因为 empty()可能更快。
- vectors、deques、lists、sets、multisets、maps、multimasp 和 strings 都支援。

bool *container*::**empty** () const
- 检验容器是否为空，并传回检查结果。
- 相当于 *container*::size()==0，但是可能更快（尤其对 lists 而言）。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

size_type *container*::**max_size** () const
- 传回容器可包含的最大元素个数。
- 这是一个技术层次的数值，可能取决于容器的内存模型。尤其 vectors 通常使用一个内存区段（segment），所以 vector 的这个值往往小于其他容器。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

容量操作（Capacity Operations）

size_type *container*::**capacity** () const
- 传回重配置内存之前所能容纳的最多元素个数。
- vectors 和 strings 都支援。

void *container*::**reserve** (size_type *num*)
- 在内部保留若干内存，至少能够容纳 num 个元素。
- 如果 *num* 小于实际容量，对 vectors 无效，对 strings 则是一个非固定的缩减请求（nonbinding shrink request）。

- vectors 的容量如何缩小，请见 p149 例子。
- 每次重新配置都会耗用相当时间，并造成所有 references、pointers、iterators 失效。因此 `reserve()`可以提高速度，保持 references、pointers、iterators 的有效性。详见 p149。
- vectors 和 strings 都支援。

元素间的比较（Comparison Operations）

```
bool comparison (const container& c1, const container& c2)
```

- 传回两个同型容器的比较结果。
- *comparison* 可以是下面之一：

```
operator ==
operator !=
operator <
operator >
operator <=
operator >=
```

- 如果两个容器拥有相同数量的元素，且元素顺序相同，而且所有相应元素两两相比之结果为 true，我们便说这两个容器相等。
- 要检验 A 容器是否小于 B 容器，需使用「字典顺序」来比较。关于「字典顺序」，请见 p360 `lexicographical_compare()`的描述。
- vectors、deques、lists、sets、multisets、maps、multimaps, strings 都支援。

关系型容器特有的非变动性操作

这里所介绍的成员函数都是对应于 p338, 9.5 节和 p397, 9.9 节所讨论的 STL 算法的特殊实作版本。这些函数利用了关系型容器的元素已序性，提供更好的性能。例如在 1,000 个元素中进行搜寻，所需的比较平均不超过 10 次（参见 p21, 2.3 节）。

```
size_type container::count (const T& value) const
```

- 传回与 *value* 相等的元素个数。
- 这是 p338 所讨论的 `count()`算法的特殊版本。
- `T` 是被排序值的型别
  - — 在 sets 和 multisets 中，`T` 是元素型别。
  - — 在 maps 和 multimaps 中，`T` 是 *key* 的型别。
- 复杂度：线性。
- sets、multisets、maps 和 multimaps 都支援。

```
iterator container::find (const T& value)
const_iterator container::find (const T& value) const
```

- 传回「实值等于 *value*」的第一个元素位置。
- 如果找不到元素就传回 `end()`。
- 这是 p341 所讨论的 `find()` 算法的特殊版本。
- `T` 是被排序值的型别：
  - 在 sets 和 multisets 中，`T` 是元素型别。
  - 在 maps 和 multimaps 中，`T` 是 *key* 的型别。
- 复杂度：对数。
- sets、multisets、maps 和 multimaps 都支援。

```
iterator container::lower_bound (const T& value)
const_iterator container::lower_bound (const T& value) const
```

- 传回一个迭代器，指向「根据排序准则，可安插 *value* 副本的第一个位置」。
- 传回之迭代器指向「实值大于等于 *value* 的第一个元素」（有可能是 `end()`）。
- 如果找不到就传回 `end()`。
- 这是 p413 所讨论的 `lower_bound()` 算法的特殊版本。
- `T` 是被排序值的型别：
  - 在 sets 和 multisets 中，`T` 是元素型别。
  - 在 map 和 multimap 中，`T` 是 *key* 的型别。
- 复杂度：对数。
- sets、multisets、maps 和 multimaps 都支援。

```
iterator container::upper_bound (const T& value)
const_iterator container::upper_bound (const T& value) const
```

- 传回一个迭代器，指向「根据排序准则，可安插 *value* 副本的最后一个位置」。
- 传回之迭代器指向「实值大于 *value* 的第一个元素」（有可能是 `end()`）。
- 如果找不到就传回 `end()`。
- 这是 p413 所讨论的 `upper_bound()` 算法的特殊版本。
- `T` 是被排序值的型别：
  - 在 sets 和 multisets 中，`T` 是元素型别。
  - 在 map 和 multimap 中，`T` 是 *key* 的型别。
- 复杂度：对数。
- sets、multisets、maps 和 multimaps 都支援。

```
pair<iterator,iterator> container::equal_range (const T& value)
pair<const_iterator,const_iterator>
                        container::equal_range(const T& value) const
```

- 传回一个区间（一对迭代器），指向「根据排序准则，可安插 *value* 副本的第一个位置和最后一个位置」。
- 传回一个区间，其内的元素实值皆等于 *value*。
- 相当于：

```
make_pair(lower_bound(value),upper_bound(value))
```

- 这是 p415 所讨论的 `equal_range()` 算法的特殊版本。
- `T` 是被排序值的型别：
  - 在 sets 和 multisets 中，`T` 是元素型别。
  - 在 map 和 multimap 中，`T` 是 *key* 的型别。
- 复杂度：对数。
- sets、multisets、maps 和 multimaps 都支援。

```
key_compare container::key_comp ()
```

- 传回一个「比较准则」。
- sets、multisets、maps 和 multimaps 都支援。

```
value_compare container::value_comp ()
```

- 传回一个作为比较准则的对象。
- 在 sets 和 multisets 中，它相当于 `key_comp()`。
- 在 maps 和 multimaps 中，它是一个辅助类别，用来比较两元素的 *key*。
- sets、multisets、maps 和 multimaps 都支援。

6.10.4 赋值（指派，Assignments）

```
container& container::operator = (const container& c)
```

- 将 `c` 的所有元素指派给现有容器，亦即以 `c` 的元素替换所有现有元素。
- 这个运算符会针对被覆盖的元素呼叫其 *assignment* 运算符，针对被附加的元素呼叫其 *copy* 建构式，针对被移除的元素呼叫其解构式。
- vectors、deques、lists、sets、multisets、maps 和 multimaps 都支援。

```
void container::assign (size_type num, const T& value)
```

- 指派 *num* 个 *value*，亦即以 *num* 个 *value* 副本替换掉所有现有元素。
- `T` 必须是元素型别。
- sets、multisets、maps 和 multimaps 都支援。

`void container::assign (InputIterator beg, InputIterator end)`

- 指派区间[beg;end)内的所有元素，亦即以[beg;end)内的元素副本替换掉所有现有元素。
- 此函数为一个 member template（参见 p11）。因此只要源区间的元素型别可转换为容器元素型别，此函数即可派上用场。
- vectors、deques、lists 和 strings 都支援。

`void container::swap (container& c)`

- 和 c 交换内容。
- 两个容器互换：
  - 元素
  - 排序准则（如果有的话）。
- 此函数拥有常数复杂度。如果不再需要容器中的老旧元素，则应使用本函数来取代赋值动作（参见 p147, 6.1.2 节）。
- 对于关系型容器，只要「比较准则」进行复制或指派时不丢掷异常，本函数就不丢掷异常。对于其他所有容器，此函数一律不丢掷异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

`void swap (container& c1, container& c2)`

- 相当于 c1.swap(c2)（参见稍早前的描述）。
- 对于关系型容器，只要「比较准则」进行复制或赋值时，不丢掷异常，本函数就不丢掷异常。对于其他所有容器，此函数一律不丢掷异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

6.10.5 直接元素存取

`reference container::at (size_type idx)`
`const_reference container::at (size_type idx) const`

- 二者都传回索引 *idx* 所代表的元素（第一个元素的索引为 0）。
- 如果传入一个无效索引（< 0 或 >= size()），会导致 *out_of_range* 异常。
- 后续的修改或记忆体重新配置，可能会导致传回的 reference 无效。
- 如果呼叫者保证索引有效，那么最好使用速度更快的 operator[]。
- vectors、deques 和 strings 都支援。

```
reference container::operator[] (size_type idx)
const_reference container::operator[] (size_type idx) const
```

- 二者都传回索引 *idx* 所代表的元素（第一个元素的索引为 0）。
- 如果传入一个无效索引（< 0 或 >= size()），会导致未定义的行为。所以呼叫者必须确保索引有效，否则应该使用 at()。
- (1) 修改 strings 或 (2) 记忆体重新配置，可能会导致 non-const strings 传回的 reference 失效（详见 p487）。
- vectors、deques 和 strings 都支援。

```
T& map::operator[] (const key_type& key)
```

- 关系型数组的 operator[]。
- 在 map 中，会传回 *key* 所对应的 *value*。
- 注意：如果不存在「键值为 *key*」的元素，则本操作会自动生成一个新元素，其初值由 *value* 型别的 *default* 建构式给定。所以不存在所谓的无效索引。例如：

```
map<int,string> coll;
coll[77] = "hello";  // insert key 77 with value "hello"
cout << coll[42];   // Oops, inserts key 42 with value "" and
                    // prints the value
```

详见 p205, 6.6.3 节。

- T 是元素的 *value* 型别。
- 相当于：

```
(*((insert(make_pair(x,T()))).first)).seco
nd
```

- 只有 map 支持此一操作。

```
reference container::front ()
const_reference container::front () const
```

- 都传回第一个元素（第一个元素的索引为 0）。
- 呼叫者必须确保容器内有元素（size()>0），否则会导致未定义的行为。
- vectors、deques 和 lists 都支援。

```
reference container::back ()
const_reference container::back () const
```

- 都传回最后一个元素（索引为 size()-1）。
- 呼叫者必须确保容器内拥有元素（size()>0）；否则会导致未定义的行为。
- vectors、deques 和 lists 都支援。

6.10.6「会产出迭代器」的各项操作

本节各个成员函数都会传回迭代器，凭借这些迭代器你可以巡访容器中的所有元素。表 6.34 列出各种容器所提供的迭代器类型（参见 p251, 7.2 节）。

容器	迭代器类型（iterator category）
Vector	随机存取
Deque	随机存取
List	双向
Set	双向，元素为常量
Multiset	双向，元素为常量
Map	双向，*key* 为常量
Multimap	双向，*key* 为常量
String	随机存取

表 6.34 各种容器提供的迭代器类型

```
iterator container::begin ()
const_iterator container::begin () const
```

- 传回一个迭代器，指向容器起始处（第一元素的位置）。
- 如果容器为空，则此动作相当于 `container::end()`。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

```
iterator container::end ()
const_iterator container::end () const
```

- 传回一个迭代器，指向容器尾端（最后元素的下一位置）。
- 如果容器为空，则此动作相当于 `container::begin()`。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

```
reverse_iterator container::rbegin ()
const_reverse_iterator container::rbegin () const
```

- 传回一个逆向迭代器，指向逆向迭代时巡访的第一个元素。
- 如果容器为空，则此动作相当于 `container::rend()`。
- 关于逆向迭代器，详见 p264, 7.4.1 节。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

```
reverse_iterator container::rend ()
const_reverse_iterator container::rend () const
```

- 传回一个逆向迭代器，指向逆向迭代时巡访的最后一个元素的下一位置。
- 如果容器为空，则此动作相当于 `container::rbegin()`。
- 关于逆向迭代器，详见 p264, 7.4.1 节。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

6.10.7 元素的安插（Inserting）和移除（Removing）

```
iterator container::insert (const T& value)
pair<iterator,bool> container::insert (const T& value)
```

- 安插一个 *value* 副本于关系型容器。
- 元素可重复者（multisets 和 multimap）采用第一形式。传回新元素的位置。
- 元素不可重复者（sets 和 map）采用第二形式。如果有「具备相同 *key*」的元素已经存在，导致无法安插，会传回现有元素的位置和一个 `false`。如果安插成功，传回新元素的位置和一个 `true`。
- `T` 是容器元素的型别，对 map 和 multimap 而言那是一个 *key*/*value* pair。
- 函数如果不成功，不带来任何影响。
- sets、multisets、maps、multimaps 和 strings 都支援。

```
iterator container::insert (iterator pos, const T& value)
```

- 在迭代器 *pos* 的位置上安插一个 *value* 副本。
- 传回新元素的位置。
- 对于关系型容器（sets、multisets、maps 和 multimaps），*pos* 只作为一个提示，指向安插时必要的搜寻动作的起始建议位置。如果 *value* 刚好可安插于 *pos* 之后，则此函数具有「分期摊还之常数时间」复杂度，否则具有对数复杂度。
- 如果容器是 sets 或 maps，并且已内含一个「实值等于 *value*（意即两者的 *key* 相等）」的元素，则此呼叫无效，并传回现有元素的位置。
- 对于 vectors 和 deques，这个操作可能导致指向其他元素的某些 iterators 和 references 无效。
- `T` 是容器元素的型别，在 maps 和 multimaps 中是一个 *key*/*value* pair。
- 对于 strings，*value* 并不采用 *pass by reference*。
- 对于 vectors 和 deques，如果元素的复制动作（*copy* 建构式和 `operator=`）不丢掷异常，则此函数一旦失败并不会带来任何影响。对于所有其他容器，函数一旦失败并不会带来任何影响。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

`void container::insert (iterator pos, size_type num, const T& value)`

- 在迭代器 *pos* 的位置上安插 *num* 个 *value* 副本。
- 对于 vectors 和 deques，此操作可能导致指向其他元素的 iterators 和 references 失效
- T 是容器元素的型别，在 maps 和 multimaps 中是一个 *key*/*value* pair。
- 对于 strings，*value* 并不采用 *pass by reference*。
- 对于 vectors 和 deques，如果元素复制动作（*copy* 建构式和 operator=）不丢掷异常，则函数失败亦不会带来任何影响。对于 lists，函数若失败不会带来任何影响。
- vectors、deques、lists 和 strings 都支援。

`void container::insert (InputIterator beg, InputIterator end)`

- 将区间[*beg*,*end*)内所有元素的副本安插于关系型容器内。
- 此函数是个 member template（参见 p11），因此只要源区间的元素可转换为容器元素的型别，本函数就可派上用场。
- sets、multisets、maps、multimaps 和 strings 都支援。

```
void container::insert (iterator pos, InputIterator beg,
                                      InputIterator end)
```

- 将区间[*beg*,*end*)内所有元素的副本安插于迭代器 *pos* 所指的位置上。
- 此函数是个 member template（参见 p11），因此只要源区间的元素可转换为容器元素的型别，本函数就可派上用场。
- 对于 vectors 和 deques，此操作可能导致指向其他元素的 iterators 和 references 失效。
- 对于 lists，此函数若失败不会带来任何影响。
- vectors、deques、lists 和 strings 都支援。

`void container::push_front (const T& value)`

- 安插 *value* 的副本，使成为第一个元素。
- T 是容器元素的型别。
- 相当于 `insert(begin(),value)`。
- 对于 deques，此一操作会造成「指向其他元素」的 iterators 失效，而「指向其他元素」的 references 仍保持有效。
- 此函数若失败不会带来任何影响。
- deques 和 lists 都支援。

`void container::push_back (const T& value)`

- 安插 *value* 的副本，使成为最后一个元素。
- T 是容器元素的型别。
- 相当于 `insert(end(),value)`。
- 对于 vectors，如果造成记忆体重新配置，此操作会造成「指向其他元素」的 iterators 和 references 失效。

- 对于 deques，此一操作造成「指向其他元素」的 iterators 失效，而「指向（或说代表）其他元素」的 reference 始终有效。
- 此函数若失败不会带来任何影响。
- vectors、deques、lists 和 strings 都支援。

```
void list::remove (const T& value)
void list::remove_if (UnaryPredicate op)
```

- remove()会移除所有「实值等于 *value*」的元素。
- remove_if()会移除所有「使判断式 *op*(*elem*)结果为 true」的元素。
- 注意在函数调用过程中，*op* 不应改变状态。详见 p302, 8.14 节。
- 两者都会呼叫被移除元素的解构式。
- 剩余元素的相对次序保持不变（stable）。
- 这是 p378 所讨论的 remove()算法的特殊版本。
- T 是容器元素的型别。
- 细节和范例见 p170。
- 只要元素的比较动作不丢掷异常，此函数也不丢掷异常。
- 只有 lists 支持这个成员函数。

```
size_type container::erase (const T& value)
```

- 从关系型容器中移除所有和 *value* 相等的元素。
- 传回被移除的元素个数。
- 呼叫被移除元素的解构式。
- T 是已序（*sorted*）元素的型别。
  - 在 sets 和 multisets 中，T 是元素型别。
  - 在 map 和 multimap 中，T 是 *key* 的型别。
- 此函数不丢掷异常。
- sets、multisets、maps 和 multimaps 都支援。

```
void container::erase (iterator pos)
iterator container::erase (iterator pos)
```

- 将迭代器 *pos* 所指位置上的元素移除。
- 序列式容器（vectors、deques、lists 和 strings）采用第二形式，传回后继元素的位置（或传回 end()）。
- 关系型容器（sets、multisets、maps 和 multimaps）采用第一形式，无回返值。
- 两者都呼叫被移除元素的解构式。

- 注意，呼叫者必须确保迭代器 *pos* 有效。例如：
  ```
  coll.erase(coll.end()); // ERROR  Î undefined behavior
  ```
- 对于 vectors 和 deques，此操作可能造成「指向其他元素」的 iterators 和 references 无效。
- 对于 vectors 和 deques，只要元素复制动作（*copy* 建构式和 operator=）不丢掷异常，此函数就不丢掷异常。对于其他容器，此函数不丢掷异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

`void container::erase (iterator beg, iterator end)`
`iterator container::erase (iterator beg, iterator end)`

- 移除区间[*beg*,*end*)内的所有元素。
- 序列式容器（vectors、deques、lists 和 strings）采用第二形式，传回被移除的最后一个元素的下一位置（或传回 end()）。
- 关系型容器（sets、multisets、maps 和 multimaps）采用第一形式，无回返值。
- 一如区间惯例，始于 *beg*（含）终于 *end*（不含）的所有元素都被移除。
- 呼叫被移除元素的解构式。
- 呼叫者必须确保 *beg* 和 *end* 形成一个有效序列，并且该序列是容器的一部分。
- 对于 vectors 和 deques，此操作可能导致「指向其他元素」的 iterators 和 references 失效。
- 对于 vectors 和 deques，只要元素复制动作（*copy* 建构式和 operator=）不丢掷异常，此函数就不丢掷异常。对于其他容器，此函数不丢掷异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

`void container::pop_front ()`

- 将容器的第一个元素移除。
- 相当于 *container*.erase(*container*.begin())。
- 注意：如果容器是空的，会导致未定义行为。因此呼叫者必须确保容器至少有一个元素，也就是 size()>0。
- 此函数不丢掷异常。
- deques 和 lists 都支援。

`void container::pop_back ()`

- 将容器的最后一个元素移除。
- 相当于 *container*.erase(--*container*.end())，前提是其中的算式有效。在 vector 中此算式不一定有效（参见 p258）。
- 注意，如果容器为空，会导致未定义行为。因此呼叫者必须确保容器至少包含一个元素，也就是 size()>0。
- 此函数不丢掷异常。
- vectors、deques 和 lists 都支援。

```
void container::resize (size_type num)
void container::resize (size_type num, T value)
```

- 两者都将容器大小改为 *num*。
- 如果 size()原本就是 *num*，则两者皆不生效用。
- 如果 *num* 大于 size()，则在容器尾端产生并附加额外元素。第一形式透过 *default* 建构式来建构新元素，第二形式则以 *value* 的副本作为新元素。
- 如果 *num* 小于 size()，则移除尾端元素，直到大小为 size()。每个被移除元素的解构式都会被呼叫。
- 对于 vectors和 deques，这些函数可能导致「指向其他元素」的 iterators和 references 失效。
- 对于 vectors 和 deques，只要元素复制动作（*copy* 建构式和 operator=）不丢掷异常，这些函数就不丢掷异常。对于 lists，函如果失败不会带来任何影响。
- vectors、deques、lists 和 strings 都支援。

```
void container::clear ()
```

- 移除所有元素（将容器清空）。
- 呼叫被移除元素的解构式。
- 此一容器的所有 iterators 和 references 都将失效。
- 对于 vectors 和 deques，只要元素复制动作（*copy* 建构式和 operator=）不丢掷异常，此函数就不丢掷异常。对于其他容器，此函数不丢掷异常。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

6.10.8 Lists 的特殊成员函数

```
void list::unique ()
void list::unique (BinaryPredicate op)
```

- 移除 lists 之内相邻而重复的元素，使每一个元素都不同于下一个元素。
- 第一形式会将所有「和前一元素相等」的元素移除。
- 第二形式的意义是：任何一个元素 *elem*，如果其前一元素是 *e*，而 *elem* 和 *e* 造成二元判断式 *op(elem, e)*获得 true 值，那么就移除 *elem* [34]。换言之，这个判断式并非拿元素和其目前的前一紧临元素比较，而是拿元素和其未被移除的前一元素比较。

[3] 第二版的 unique()仅在支持 member templates 的系统中可用（参见 p11）。
[4]

- 注意，*op* 不应在函数调用过程中改变状态，详见 p302, 8.1.4 节。
- 被移除元素的解构式会被唤起。
- 这是 p381 `unique()`算法的 lists 特别版本。
- 如果「元素比较动作」中不丢掷异常，则此函数亦不丢掷异常。

`void list::splice (iterator pos, list& source)`

- 将 *source* 的所有元素搬移到`*this`，并安插到迭代器 *pos* 所指位置。
- 呼叫之后，*source* 清空。
- 如果 *source* 和`*this` 相同，会导致未定义的行为。所以呼叫端必须确定 *source* 和`*this` 是不同的 lists。如果要移动同一个 lists 内的元素，应该使用稍后提及的其他 `splice()`形式。
- 呼叫者必须确定 *pos* 是`*this` 的一个有效位置；否则会导致未定义的行为。
- 本函数不丢掷异常。

`void list::splice (iterator pos, list& source, iterator sourcePos)`

- 从 *source* list 中，将位于 *sourcePos* 位置上的元素搬移至 `*this`，并安插于迭代器 *pos* 所指位置。
- *source* 和 `*this` 可以相同。这种情况下，元素将在 lists 内部被搬移。
- 如果 *source* 和 `*this` 不是同一个 list，在此操作之后，其元素个数少 1。
- 呼叫者必须确保 *pos* 是 `*this` 的一个有效位置、*sourcePos* 是 *source* 的一个有效迭代器，而且 *sourcePos* 不是 `source.end()`；否则会导致未定义行为。
- 此函数不丢掷异常。

```
void list::splice (iterator pos, list& source,
                   iterator sourceBeg, iterator sourceEnd)
```

- 从 *source* list 中，将位于`[`*sourceBeg*`,`*sourceEnd*`)`区间内的所有元素搬移到`*this`，并安插于迭代器 *pos* 所指位置。
- *source* 和 `*this` 可以相同。这种情况下，*pos* 不得为被移动序列的一部分，而元素将在 lists 内部移动。
- 如果 *source* 和 `*this` 不是同一个 list，在此操作之后，其元素个数将减少。
- 呼叫者必须确保 *pos* 是 `*this` 的一个有效位置、*sourceBeg* 和 *sourceEnd* 形成一个有效区间，该区间是 *source* 的一部分；否则会导致未定义的行为。
- 本函数不丢掷异常。

`void list::sort ()`
`void list::sort (CompFunc op)`

- 对 lists 内的所有元素进行排序。
- 第一型式以 `operator<` 对 lists 中的所有元素进行排序。

- 第二型式透过如下的 *op* 动作来比较两元素，进而对 lists 中的所有元素排序[35]：*op*(*elem1*,*elem2*)。
- 实值相同的元素，其顺序保持不变（除非有异常被丢出）。
- 这是 p397 所讨论的 sort() 和 stable_sort()算法的「list 特殊版本」。

```
void list::merge (list& source)
void list::merge (list& source, CompFunc op)
```

- 将 lists *source* 内的所有元素并入*this。
- 呼叫后 *source* 变成空容器。
- 如果*this 和 *source* 在排序准则 operator< 或 op 之下已序（*sorted*），则新产生的 lists 也是已序。严格地说，标准规格书要求两个 lists 必须已序，但实际上对无序的 lists 进行合并也是可能的，不过使用前最好先确认一下。
- 第一形式采用 operator< 作为排序准则。
- 第二形式采用以下的 op 动作作为可有可无的排序准则，以此比较两个元素的大小[36]：op(*elem*,*sourceElem*)
- 这是 p416 所讨论的 merge()算法的 list 特殊版本。
- 只要元素的比较动作不丢掷异常，此函数万一失败也不会造成任何影响。

```
void list::reverse ()
```

- 将 lists 中的元素颠倒次序。
- 这是 p386 所讨论的 reverse()算法的「list 特殊版本」。
- 本函数不丢掷异常。

6.10.9 对配置器（Allocator）支持

所有 STL 容器都能够与某个配置器对象（allocator object）所定义的某种特定记忆体模型（memory model）搭配合作（详见第 15 章）。本节讨论的是支持配置器的各个成员。标准容器要求：配置器（型别）的每一个实体都必须是可互换的（interchangeable），所以某一容器的空间，可透过另一同型容器释放之。因此，元素（及其储存空间）在同型的两个容器之间移动，并不会出现问题。

[35] sort()的第二形式仅在支持 member templates 的系统中可用（参见 p11）

[36] merge()的第二形式仅在支持 member templates 的系统中可用（参见 p11）

基本的配置器相关成员（Fundamental Allocator Members）

container::**allocator_type**

- 配置器型别。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

allocator_type *container*::**get_allocator** () const

- 传回容器的内存模型（memory model）。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

带有「可选择之配置器参数」的建构式

explicit *container*::*container* (const Allocator& *alloc*)

- 产生一个新的空白容器，使用 *alloc* 作为内存模型（memory model）。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

container::*container* (const CompFunc& *op*, const Allocator& *alloc*)

- 产生一个新的空白容器，使用 *alloc* 作为内存模型，并以 *op* 为排序准则。
- *op* 排序准则必须定义 **strict weak ordering**（参见 p176）。
- sets、multisets、maps 和 multimaps 都支援。

container::*container* (size_type *num*, const T& *value*,
const Allocator& *alloc*)

- 产生一个拥有 *num* 个元素的容器，使用 *alloc* 作为内存模型。
- 所生成的元素都是 *value* 的副本。
- T 是容器元素的型别。注意，对于 strings，*value* 采用 *by value* 的型式传递。
- vectors、deques、lists 和 strings 都支援。

container::*container* (InputIterator *beg*, InputIterator *end*,
const Allocator& *alloc*)

- 产生一个容器，以区间[*beg*,*end*)内的所有元素为初值，并使用 *alloc* 作为记忆体模型。
- 此函数是一个 member template（参见 p11）。所以只要源序列的元素能够转换为容器元素的型别，此函数就可执行。
- vectors、deques、lists、sets、multisets、maps、multimaps 和 strings 都支援。

container::*container* (InputIterator *beg*, InputIterator *end*,
const CompFunc& *op*, const Allocator& *alloc*)

- 产生一个以 *op* 为排序准则的容器，以区间[*beg*,*end*)中的所有元素为初值，并使用 *alloc* 作为内存模型。

- 本函数是一个 member template（参见 p11）。所以只要源序列的元素能够转换为容器元素的型别，本函数就可执行。
- 排序准则 *op* 必须定义 **strict weak ordering**（参见 p176）
- sets、multisets、map 和 multimap 都支援。

6.10.10 综观 STL 容器的异常处理

p139, 5.11.2 节曾指出，不同的容器在异常发生时，给予不同程度的保证。通常 C++ 标准链接库在异常发生时并不会泄漏资源或破坏容器的恒常特性（invariants）。有些操作提供更强的保证（前提是其自变量必须满足某些条件）：它们可以保证 **commit-or-rollback**（意思是「要嘛成功，要嘛不带来任何影响」），甚至可以保证绝不丢掷异常。表 6.35 列出所有支持更强烈保证的操作函数[37]。

对于 vectors、deques 和 lists 而言，`resize()`也提供特别保证。其行为或许相当于 `erase()`，或许相当于 `insert()`，或许相当于什么也没做。

```
void container::resize (size_type num, T value = T())
{
    if (num > size()) {
        insert (end(), num-size(), value);
    }
    else if (num < size()) {
        erase (begin()+num, end());
    }
}
```

因此，它所提供的保证就是「`erase()`和 `insert()`两者所提供的保证」的组合（参见 p244）。

[37] 感谢 Greg Colvin 和 Dave Abrahams 提供这个表格。

操作	页次	保证
vector::push_back()	241	要不成功，要不无任何影响。
vector::insert()	240	要不成功，要不无任何影响 — 前提是元素的复制/赋值动作不丢掷异常
	243	不丢掷异常
vector::pop_back()	242	不丢掷异常 — 前提是元素的复制/赋值动作不丢掷异常
	244	不丢掷异常 — 前提是元素的复制/赋值动作不丢掷异常
vector::erase()	237	不丢掷异常
vector::clear()	241	要不成功，要不无任何影响。
vector::swap()	241	要不成功，要不无任何影响。
deque::push_back()	240	要不成功，要不无任何影响 — 前提是元素的复制/赋值动作不丢掷异常
deque::push_front()	243	不丢掷异常
	243	不丢掷异常
deque::insert()	242	不丢掷异常 — 前提是元素的复制/赋值动作不丢掷异常
	244	不丢掷异常 — 前提是元素的复制/赋值动作不丢掷异常
deque::pop_back()	237	不丢掷异常
	241	要不成功，要不无任何影响。
deque::pop_front()	241	要不成功，要不无任何影响。
	240	要不成功，要不无任何影响。
deque::erase()	243	不丢掷异常
deque::clear()	243	不丢掷异常
deque::swap()	242	不丢掷异常
list::push_back()	244	不丢掷异常
	242	不丢掷异常 — 前提是元素的比较动作不丢掷异常
list::push_front()	242	不丢掷异常 — 前提是判断式 *predicate* 不丢掷异常
	244	不丢掷异常 — 前提是元素的比较动作不丢掷异常
list::insert()	245	不丢掷异常
list::pop_back()	246	要不成功，要不无任何影响 — 前提是元素的比较动作不丢掷异常
list::pop_front()	246	不丢掷异常
list::erase()	237	不丢掷异常
[multi]set::insert()	240	要不成功，要不无任何影响 — 对单个元素而言
[multi]set::erase()	242	不丢掷异常
list::remove_if()	244	不丢掷异常
[multi]set::clear()	237	不丢掷异常 — 前提是对「比较准则」执行复制/赋值动作时不丢掷异常
list::unique()		
[multi]map::insert()	240	要不成功，要不无任何影响 — 对单个元素而言
[multi]map::erase()	242	不丢掷异常
list::merge()	244	不丢掷异常
list::swap()	237	不丢掷异常 — 前提是对「比较准则」执行复制/赋值动作时不丢掷异常
[multi]map::swap()		

表 6.35 「异常发生时带有特殊保证」的各个容器操作函数