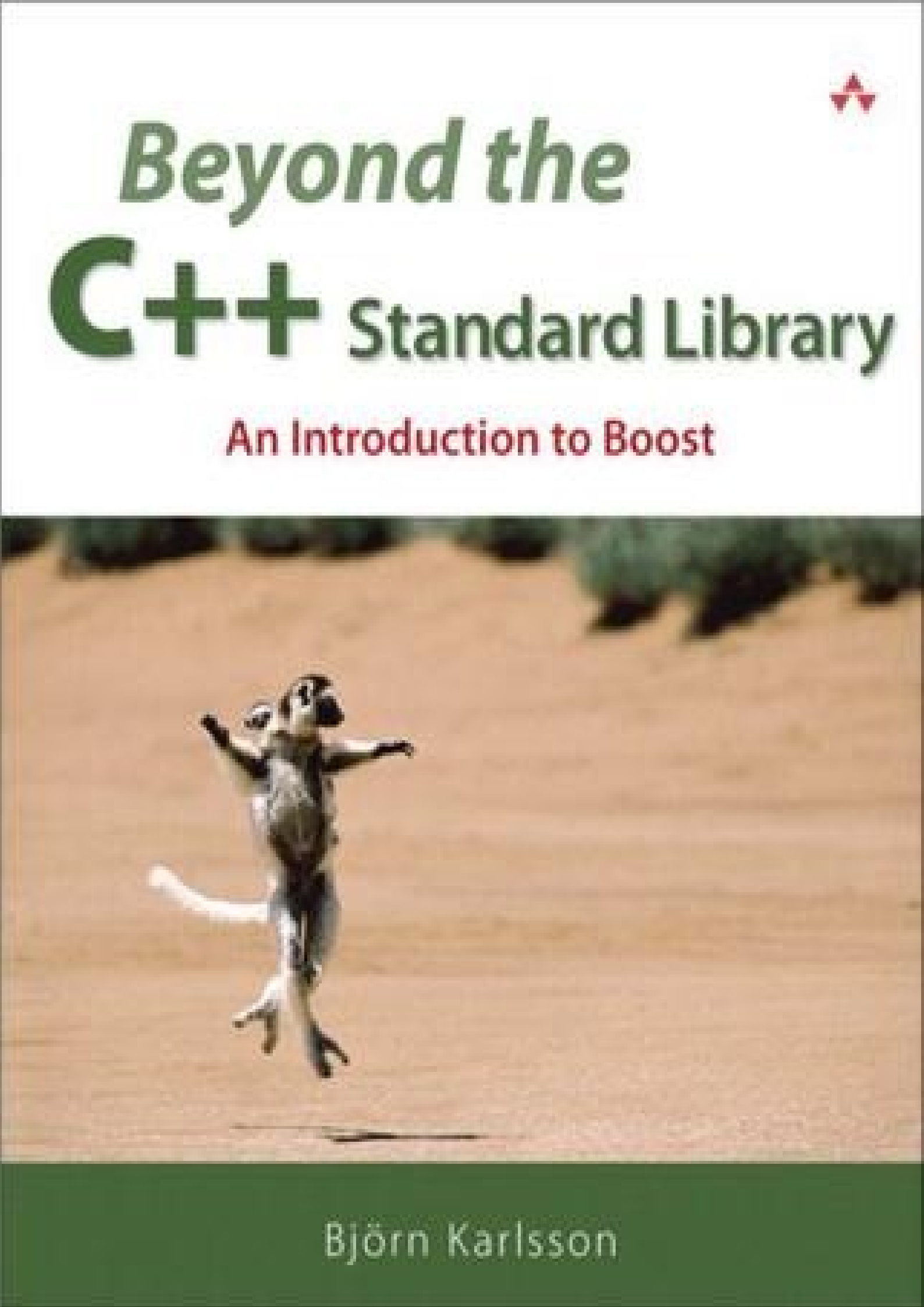
Beyond the
C++ Standard Library
An Introduction to Boost
Björn Karlsson

目錄

Beyond the C++ Standard Library 中文版

来源：boost-doc-zh

序

C++社区正在发生着好事。C++依旧是最广泛使用的编程语言，它正在变 得更强大和更易用。你不信？听我细说。

标准C++的当前版本发布于1998，它为传统的面向过程编程、面向对象 编程和泛型编程提供了坚实的 支持。正如旧C++ (1998之前的) 独力承担了把面向对象普及到日常的软件开发中一样，C++98在为泛型编程做着同样的事情。九十年代中期标准模板库(STL)与标准C++的集成已经引起 了另一次编程范式的转变，就象八十年代的时候Bjarne Stroustrup把类引入到C一样。现在大多数的C++开发者都熟悉STL的概念，这再次提升了整体的水平。

C++能力的应用仍旧被不断发现。今天许多的C++库，包括特殊的数学 库，都大量利用了模板元编译的技术，它是设计C++模板的时候没有预测到的幸运结果。随着C++社区里的高级工具和技术不断涌现，开发复杂应用软件正变得 更简单、更令人愉快。

很难描述Boost对于C++世界的重要性。自从C++98发布后，除了 ISO的标准C++委员会， 没有一个团体对于C++的发展方向有比Boost更大的影响(许多Boost的成员本身就是WG21的重要成员，包括它的创始人，我的朋友Beman Dawes)。成千上万个杰出的Boost志愿者无私地，以对等审查方式开发了许多C++98没有提供的很有用的库。这些库中的十个已被接受将加入到即将 到来的C++0x的库中，更多的库正被考虑接受。Where a library approach has been shown to be wanting, the wisdom gained from the cross-pollination of Boost and WG21 has suggested a few modest language enhancements, which are now being entertained.

你不太可能没有听说过Boost，我来问一下你... 你需要在文本和数字之间进行转换，或 在任意的可流处理的类之间进行转换？没有问题，用Boost.lexical_cast。噢，你有更多复杂的文本处理要求？那么你可以用 Boost.Tokenizer 或 Boost.Regex, 或Boost.Spirit, 如果你需要完整的语法分析。Boost.Bind 的函数反射和组合能力会让你吃惊。要用函数对象来编程，有 Boost.Lambda。静态断言？用MPL。如果你是用数学库，记住：你有 Boost.Math, Graph, Quaternion, Octonion, MultiArray, Random, 和 Rational。如果你刚好喜欢Python，有了Boost.Python的帮助，你可以和C++一起用它了。并且你可以从以上所有东西中挑选出你要的。

Björn Karlsson是一名Boost 狂热者以及C++社区的热心的支持者。你曾经在C/C++ Users Journal上出版了很多有用的好文章，最近还为C++ Source, 一个新的C++社区的在线杂志(见 www.artima.com/cppsource) 写文章。他在本书中讲述了关键的Boost组件并给出示例，展示了如何使用它们扩充C++标准库。本书不仅是Boost的深入教程，也是标准C++的未来 版本的预示。希望你喜欢！

Chuck Allison, Editor, The C++ Source

前言

亲爱的读者，

欢迎来到 Beyond the C++ Standard Library: An Introduction to Boost.

如果你对泛型编程、库设计以及C++标准库感兴趣，那么这本书正适合你。因为本书的目标读者是中级至高级的C++程序员，也覆盖了一点点C++的基本概念。正如题目所说的，本书的重点是在Boost库的普通使用、最佳实践、实现技术及设计原理。

几乎从我发现Boost的那一天起，它背后的人们，以及它里面的非凡的库，我都写进这本书里。令人惊 奇的是，象C++这样一门成熟的语言还能够在高级抽象及技术细节方面提供如此大的探索空间，而没有任何对语言的修改要求。当然，这也是C++与其它编程语 言最不同的地方：它是专门为扩展性而设计的，语言为泛型构造提供的便利极为强大。本书的探索是关于Boost库的核心以及Boost社区本身。Boost 使得C++编程更为优雅、更有活力，也更高效。正如已经看到的，C++社区正面临一个巨大的挑战是，与其它人分享知识。在相互孤立的时候，这些东西的价值 是非常有限的，但当它被大量观众接受时，整个工业都将有所发展。

本书展示了如何使用Boost库中的一些最有用的组件，教会你它们的最优使用方法，并到幕后看看它们是怎样工作的。Boost库的许可证允许你可以为任何目的(商业或非商业)拷贝、使用和修改这些软件，因此你需要做的就是访问 www.boost.org ，并下载最新的版本。

所有C++标准库的爱好者都知道有一个新的标准库修订正在进行。从标准化的观点出发，C++标准库有三个主要的地方会修改：

- 修复有问题的库
- 增加现有库中没有的特性
- 增加新的库，提供标准库中没有提供的功能

Boost库定位于所有三个方面。本书提到的12个库当中，有6个已经被接受进入即将到来的标准库的 技术报告，这意味着它们很有可能成为下一版本的标准库的成员。因此，学习这些库有长期价值。我希望你会发现本书可以帮助你使用、弄明白并扩展Boost 库。当你做到这些时，你就可以把这些库以及它们背后的知识组合进你的设计和实现之中。这就我所说的重用。

感谢你的阅读。

Björn Karlsson

Acknowledgments

A number of people have made all the difference for this book, and for my ability to write it. First of all, I wish to thank the Boost community for these astonishing libraries. Theythe libraries and the Boostersmake a very real difference for the C++ community and the whole software industry. Then, there are people who have very actively supported this effort of mine, and I wish to thank them personally. It's inevitable that I will fail to mention some: To all of you, please accept my sincere apologies. Beman Dawes, thank you for creating Boost in the first place, and for hooking me up with Addison-Wesley. Bjarne Stroustrup, thank you for providing guidance and pointing out important omissions from the nearly finished manuscript. Robert Stewart, thank you for the careful technical and general editing of this book. Rob has made this book much more consistent, more readable, and more accurateand all of this on his free time! The technical errors that remain are mine, not his. Rob has also been instrumental in finding ways to help the reader stay on track even when the author strays. Chuck Allison, thank you for your continuous encouragement and support for my authoring goals. David Abrahams, thank you for supporting this effort and for helping out with reviewing. Matthew Wilson, thank you for reviewing parts of this book and for being a good friend. Gary Powell, thank you for the excellent reviews and for your outstanding enthusiasm for this endeavor. All of the authors of Boost libraries have created online documentation for them: Without this great source of information, it nearly would have been impossible to write this book. Thanks to all of you. Many Boosters have helped out in different ways, and special thanks go to those who have reviewed various chapters of this book. Without their help, important points would not have been made and errors would have prevailed. Aleksey Gurtovoy, David Brownell, Douglas Gregor, Duane Murphy, Eric Friedman, Eric Niebler, Fernando Cacciola, Frank Griswold, Jaakko Järvi, James Curran, Jeremy Siek, John Maddock, Kevlin Henney, Michiel Salters, Paul Grenyer, Peter Dimov, Ronald Garcia, Phil Boyd, Thorsten Ottosen, Tommy Svensson, and Vladimir Prusthank you all so much!

Special thanks go to Microsoft Corporation and Comeau Computing for providing me with their excellent compilers.

I have also had the pleasure of working with two excellent editors from Addison-Wesley. Deborah Lafferty helped me with all of the initial work, such as creating the proposal for the book, and basically made sure that I came to grips with many of the authoring details that I was previously oblivious to. Peter Gordon, skillfully assisted by Kim Boedigheimer, took over the editing of the book and led it through to publishing. Further assistance was given by Lori Lyons, project editor, and Kelli Brooks, copy editor. I wish to thank them allfor making the book possible and for seeing it through to completion.

Friends and family have supported my obsession with C++ for many years now; thank you so much for being there, always.

And finally, many thanks to my wife Jeanette and our son SimonI am forever grateful for your love and support. I will always do my best to deserve it.

关于作者

Björn Karlsson 是ReadSoft公司的资深软件工程师，他的主要时间用于C++设计及编程。他曾在C/C++ Users Journal,Overload, 以及在线杂志 The C++ Source上发表过大量关于C++和Boost文章。

Karlsson是The C++ Source 顾问组的成员，同时也是C/C++ Users Journal编辑组的成员，同时还是专家论坛的专栏作家之一。他参与 了Boost新闻组，并且是Boost与用户的协调人之一。

译者：alai04

本书的组织结构

本书分为三个主要部分，每部分包含关于一个特定领域的库，不过肯定也有一些重叠的地方。这种分类可以让你更容易地找到与你的任务相关的信息，也使得阅读本书时可以更方便地找到相关的主题。大多数情况下，每章讨论一个单独的库，但也有时会一章里讨论一小组的库。

排版及编码的风格尽量保持简单。在这方面有很多好的方法，我只是挑选了一种我认为大多数人会习惯的方式，这样可以更容易传递所要的信息。另外，本书的代码风格会通过避免把大括号独立写一行来尽量节省垂直空间。

虽然很多书的例子都大量使用了声明和指示符，这里不会这样。我会尽力让名字清楚明白。这样做有另一个好处，可以展示类型和函数从何而来。如果是从标准库来的，它会有前缀 `std::` 。如果是从Boost来的，它会有前缀 `boost::` 。

本书介绍的一些库非常广泛，不可能详细解释这些库的所有各个方面。这种情况下，会有一个关于如何获得更多信息的注释，引用在线文档、相关文献等。同时，我会试图关注最常用的部分，和与C++标准库关系最密切的部分。

本书的第一部分是关于general libraries, 这些库非常有用，但不那么有吸引力。第二部分讨论重要的 data structures 和 containers。 第三部分讨论 higher-order programming。并不要求你必须按顺序来阅读这些库，但从最开始起按顺序进行肯定是无害的。

在深入到Boost库之前，会有一个对于目前可用的Boost库的概括介绍，向你介绍一下Boost库，并交待一下我在本书剩余部分要讨论的问题的背景。它对这个世界级的C++库集合的多功能性给出了一个有趣的介绍。

Boost的介绍

因为你正在读这本书，我希望你至少对Boost库有一点熟悉，或者你至少听说过Boost。Boost里有很多库，只有很少一些是你不感兴趣的。可以肯定你会在里面找到马上就要用的库。Boost库覆盖了广泛的领域，从数学库到智能指针，从模板 元编程库到预处理器库，从线程到lambda表达式，等等。所有Boost库都具有宽松的许可证，确保库可以被自由使用于商用软件。支持通过新闻组实现， 那是Boost社区最具活力的地方，而且至少有一家公司专门提供与Boost相关的咨询服务。对于Boost社区的在线介绍，我强烈建议你访问Boost 网站 www.boost.org.

在写本书之时，Boost的最新版本为1.32.0. 里面包括58个独立的库。后面将分类介绍这58个库，并给出关于每个库的简短描述。对于本书未详细讨论的库，可以看一下www.boost.org提供的文档，你也可以从那里下载Boost库。

字符串及文本处理

Boost.Regex

正则表达式是解决大量模式匹配问题的基础。它们常用于处理大的字符串，子串模糊查找，按某种格式tokenize字符串，或者是基于某种规则修改字符串。由于C++没有提供正则表达式支持，使得有些用户被迫转向其它支持正则表达式的语言，如Perl, awk, 和 sed。Regex提供了高效和强大的正则表达式支持，基于与STL同样的前提而设计，这使得它很容易使用。Regex已被即将发布的Library Technical Report接受。更多的信息，请见"Library 5: Regex."

Regex 的作者是 Dr. John Maddock.

Boost.Spirit

Spirit库是一个多用途的、递归的语法分析器生成框架。有了它，你可以创建命令行分析器，甚至是语言预处理器[1]。它允许程序员直接在C++代码里使用(近似于)EBNF的语法来指定语法规则。分析器非常难写，对于一个特定的问题，它们很快就变得难于维护和看懂。而Spirit解决了这些问题，而且达到了与手工制作的分析器一样或几乎一样的性能。

> [1] Wave库使用Spirit实现了一个与C++高度一致的预处理器，就证明了这一点。

Spirit 的作者是 Joel de Guzman, 以及一组熟练的程序员。

Boost.String_algo

这是一组与字符串相关的算法。包括很多有用的算法，用于大小写转换，空格清除，字符串分割，查找及替换，等等。这组算法是目前C++标准库里已有功能的扩展。

String_algo 的作者是 Pavol Droba.

Boost.Tokenizer

这个库提供了把字符序列分割成记号(token)的方法。通用的语法分析任务包括了在已分割的文本流里查找数据。如果可以把字符序列视为多个元素的容器将很有帮助，容器中的元素被执照用户定义的规则所分割。语法分析就成为了在这些元素上进行操作的单个任务，Tokenizer正好提供了这种功能。用户可以决定字符序列如何被分割，在用户请求新的元素时，库将找出相应的记号。

Tokenizer 的作者是 John Bandela.

数 据结构, 容器, 迭代器, 和算法

Boost.Any

Any库支持类型安全地存储和获取任意类型的值。当你需要一个可变的类型时，有三种可能的解决方案：

- 无限制的类型，如 `void*`. 这种方法不可能是类型安全的，应该象逃避灾难一样避免它。
- 可变的类型，即支持多种类型的存储和获取的类型。
- 支持转换的类型，如字符串类型与整数类型之间的转换。

Any实现了第二种方案，一个基于值的可变化的类型，无限可能的类型。这个库通常用于把不同类型的东西存储到标准库的容器中。更多的说明请见 "Library 6: Any."

Any 的作者是 Kevlin Henney.

Boost.Array

这个库包装了普通的C风格数组，给它们增加了一些来自于标准库容器的函数和 `typedef` 。其结果就是可以把普通的数组视为标准库的容器。这非常有用，因为它增加了类型安全性而没有降低效率，而且它使得标准库容器和普通数组拥有统一的语法。后 一点意味着可以把普通数组用于大多数的要求容器类来操作的函数。当要求软件要达到普通数组的性能时，可以用Array来替代 `std::vector` .

Array 的作者是 Nicolai Josuttis, 它在Matt Austern 和 Bjarne Stroustrup早期提出的思想之上建立了这个库。

Boost.Compressed_pair

这个库包括一个参数化的类型, `compressed_pair` , 它非常象标准库中的 `std::pair` .
与 `std::pair` 不同之处在于， `boost::compressed_pair` 对模板参数进行评估，看其中有没有空的参数，如果有，使用空类优化技术来压缩pair的大小。

Boost.Compressed_pair 常用于存放一对对象，其中之一或两个都可能是空的。

Compressed_pair 的作者是 Steve Cleary, Beman Dawes, Howard Hinnant, 和 John Maddock.

Boost.Dynamic_bitset

Dynamic_bitset库非常象 `std::bitset`, 除了 `std::bitset` 是用参数来指定位数(即容器的大小), 而 `boost::dynamic_bitset` 则支持在运行期指定大小。 `dynamic_bitset` 支持与 `std::bitset` 一样的接口，还增加了支持运行期特定功能的函数和一些 `std::bitset` 中没有的功能。在bitset的大小无法在编译期确定或在程序运行时可能变化的情况下，这个库通常用于替换 `std::bitset` 。

Dynamic_bitset 的作者是 Jeremy Siek 和 Chuck Allison.

Boost.Graph

Graph是一个处理图结构的库，它的设计受到STL的重要影响。它是泛型的，高度可配置，并且包括 多个不同的数据结构：邻接链表, 邻接矩阵, 和边列表。Graph还提供了大量的图算法，如Dijsktra最短路径算法，Kruskal最小生成树算法，拓朴逻辑排序，等等。

Graph 的作者是 Jeremy Siek, Lie-Quan Lee, 和 Andrew Lumsdaine.

Boost.Iterator

这个库提供一个创建新的迭代器类型的框架，还提供了许多有用的迭代器适配器，比C++标准中定义的更多。创建遵循标准的新迭代器类型是一件困难且乏味的工作。Iterator通过自动完成大多数细节，如提供所需的 `typedef` ，简化了这件工作。Iterator还可以改编已有的迭代器类型以赋于它新的行为。例如，间接迭代器适配器增加了一个额外的解引用操作，可以把一个包含某种对象的指针(或智能指针)的容器变成象一个包含该对象的容器。

Iterator 的作者是 Jeremy Siek, David Abrahams, 和 Thomas Witt.

Boost.MultiArray

MultiArray提供了一个多维容器，它很象标准库的容器，但比向量的向量更有效、更高效，更直接。容器的维数在声明时指定，但它支持限制(slicing)和映身(projecting)不同的视图(view)，也可以在运行期改变维数。

MultiArray 的作者是 Ronald Garcia.

Boost.Multi-index

Multi-index为底层的容器提供多个索引。这意味着一个底层的容器可以有不同的排序方法和不同的访问语义。当 `std::set` 和 `std::map` 不够用时，就可以用Boost.Multi-index，通常是在需要为查找元素而维护多个索引时。

Multi-index 的作者是 Joaquín M López Muñoz.

Boost.Range

这个库是一组关于范围的概念和工具。比起在算法中使用一对迭代器来指定范围，使用ranges更简单，并提升了用户代码的抽象水平。

Range 的作者是 Thorsten Ottosen.

Boost.Tuple

在标准C++中有Pairs(类模板 `std::pair`), 但它不支持n-tuples。用Tuple`.`不象用`struct`s或`class`es 来定义n-tuples, 这个类模板支持直接声明和使用，如函数返回类型或参数，并提供一个泛型的方法来访问tuple的元素。关于这个库的详细信息，请见"Library 8: Tuple 8"。Tuple已经被即将发布的Library Technical Report所接受。

Tuple 的作者是 Jaakko Järvi.

Boost.Variant

Variant库包含一个不同于union的泛型类，用于在存储和操作来自于不同类型的对象。这个库的一个特点是支持类型安全的访问，减少了不同数据类型的类型转换代码的共同问题。

Variant 的作者是 Eric Friedman 和 Itay Maman.

函数对象及高级编程

Boost.Bind

Bind是对标准库的绑定器 `bind1st` 和 `bind2nd` 的泛化。这个库支持使用统一的语法将参数绑定到任何类似于函数行为的东西，如函数指针、函数对象，以及成员函数指针。它还可以通过嵌套绑定器实现函数组合。这个库不要求那些对标准库绑定器的强制约束，最显著的就是不要求你的类提供 `typedef` s `result_type` , `first_argument_type` , 和 `second_argument_type` 等。这个库也使得我们不再需要用 `ptr_fun` , `mem_fun` , 和 `mem_fun_ref` 等适配器。Bind库的说明在"Library 9: Bind 9."。它是对C++标准库的一个重要且很有用的扩充。Bind可以被标准库的算法使用，也经常用于Boost的函数，它提供了一个强大的工具，用于存放后续调用的函数和函数对象。Bind 已被即将发布的Library Technical Report所接受。

Bind 的作者是 Peter Dimov.

Boost.Function

Function库实现了一个泛型的回调机制。它提供了函数指针、函数对象和成员函数指针的存储和后续的调用。当然，它与binder库，如Boost.Bind 和 Boost.Lambda一起工作，大大提高了回调(包括带态度的回调函数)的使用机会。这个库的详细介绍请见"Library 11: Function 11."。Function常用于需要把函数指针用于回调的地方。例如：信号/接收者的实现，GUI与业务逻辑的分离，以及在标准库容器中存储不同的类函数类型。Function已被即将发布的Library Technical Report所接受。

Function 的作者是 Douglas Gregor.

Boost.Functional

Functional库提供C++标准库的适配器的加强版。主要的优势是它有助于解决引用到引用(这是非法的)的问题，这个问题是由对常有一个或多个引用参数的函数使用标准库的绑定器所引起的。Functional同时消除了在标准库算法中使用函数指针时必须用 `ptr_fun` 的问题。

Functional 的作者是 Mark Rodgers.

Boost.Lambda

Lambda为C++提供lambda表达式及无名函数。在使用标准库算法时特别好用，Lambda允许函数在呼叫点创建，避免了创建多个小的函数对象。使用lambdas意味着更少的代码，在哪需要就在哪写，这比分散在代码各处的函数对象更清晰、更好维护。"Library 10: Lambda 10" 详细讨论了这个库。

Lambda 的作者是 Jaakko Järvi 和 Gary Powell.

Boost.Ref

许多函数模板，包括大量标准C++库里的函数模板，它们的参数采用传值的方式传递，有时候会有问题。 复制一个对象可能很昂贵或者甚至不可能，或者状态可能取决于特写的实例，因此这时复制是不希望的。在这些情况下，可用的办法是用引用传递取代值传递。 Ref包装了一个对象的引用，并把它放入一个对象以便被复制。这就允许了通过引用去调用那些采用传值参数的函数。R`ef` 已被即将发布的Library Technical Report所接受。

Ref 的作者是 Jaakko Järvi, Peter Dimov, Douglas Gregor, 和 David Abrahams.

Boost.Signals

信号和接收系统，基于称为publisher-subscriber 和 observer的模式，它是在一个最小相关性系统中管理事件的重要工具。很少有大型应用软件不采用这种强大设计模式的某种变形，尽管他们有各自的实现方式。Signals提供了一个已验证的、高效的手段，将信号(events/subjects)的发生和这些信号要通知的接收者(subscribers/observers)进行了分离。

Signals 的作者是 Douglas Gregor.

泛 型编程与模板元编程

Boost.Call_traits

这个库提供了传递参数给函数的最好方法的自动演绎，依据参数的类型。例如，当传递的是如 `int` 和 `double` 这样的内建类型，最高效的方式是传值。对于用户自定义类型，则传送 `const` 引用通常更好。Call_traits为你自动选择正确的参数类型。这个库还有助于声明参数为引用，而不用冒引用到引用的风险(在C++这是非法的)。Call_traits常用于要求以最高效方式传递参数而又不知道参数类型的泛型函数，并避免引用到引用的问题。

Call_traits 的作者是 Steve Cleary, Beman Dawes, Howard Hinnant, 和 John Maddock.

Boost.Concept_check

Concept_check提供一些类模板，用于测试特定的概念(需求的集合)。泛型(参数化的)代码要求实例化时的类型必须符合某些抽象概念，如LessThanComparable. 这个库提供了一些方法来明确地声明模板的参数化类型的特定需求。代码的用户可以获益，由于需求的文档化以及编译器可以产生错误信息以明确指出类型不符合这 些概念的地方。Boost.Concept_check提供了超过30个可用于泛型代码的概念，其中一些原型可用于校验包括所有相关概念的组件的实现。它 用于在泛型代码中声明和证明概念的需求。

Concept_check 的作者是 Jeremy Siek, 他从Alexander Stepanov and Matt Austern的前期工作中得到灵感。

Boost.Enable_if

Enable_if允许函数模板或类模板的特化体包括/排除在一组匹配的函数或特化体之中/之外。主要的用例是包括/排除基于某些特性的特化体。例如，仅当采用一个整数类型实例化时使能一个函数模板。这个库还为SFINAE(substitution failure is not an error)提供了一个非常有用的研究机会。

Enable_if 的作者是 Jaakko Järvi, Jeremiah Willcock, 和 Andrew Lumsdaine.

Boost.In_place_factory

In_place_factory库是一个直接构造所含对象的框架，包括用于初始化的可变参数列表。它可以消除对所含类型必须是CopyConstructible的要求，并减少了创建不必要的临时对象的需要，该临时对象仅用于提供复制所需的源对象。这个库有助于减少传送用于对象初始化的参数所需的工作量。

In_place_factory 的作者是 Fernando Cacciola.

Boost.Mpl

Mpl是一个模板元编程库。它包含了与C++标准库十分相象的数据结构和算法，但它们是在编译期使用的。甚至有编译期的lambda表达式支持！提供编译期的操作，如产生类型或操作类型序列，在现代C++中越来越普遍，而提供这些功能的库是非常重要的工具。就我所知，还没有其它象Mpl这样的库。它填充了C++元编程世界的空白。我可以告诉你在你读本书时有一本关于Boost.Mpl的书正在创作，它就快要面世了，它就是Aleksey Gurtovoy 和 David Abrahams所著的C++ Template Metaprogramming。你应该尽快获得一本。

Mpl 的作者是 Aleksey Gurtovoy, 并有许多其它人的重要贡献。

Boost.Property_map

Property_map是一个概念库而不是一个真正的实现。它引入了 `property_map` 概念以及 `property_map` 类型的一组要求，从而给出了对一个key和一个value的映射的语法和语义要求。这在需要声明必须支持的类型的泛型代码中很有用。C++数组是一个 `property_map` 的例子。这个库包含了Boost.Concept_check可以测试的概念的定义。

Property_map 的作者是 Jeremy Siek.

Boost.Static_assert

进行编译期编程的一个公共的需求是提供静态断言，即编译期断言。另外，获得一致的错误提示不是必然的，由于静态断言必须会产生失败断言的信号，跨不同的编译器。Static_assert 提供对名字空间、类、函数作用域的静态断言的支持。详细信息见"Library 3: Utility."

Static_assert 的作者是 Dr. John Maddock.

Boost.Type_traits

成功的泛型编程通常需要根据参数化类型进行决策或调整这些类型的属性(如cv-qualification[2])。Type_traits提供关于类型的编译期信息，如某个类型是否指针或引用，以及增加或去除类型基本属性。Type_traits已被加入即将发布的Library Technical Report。

> [2] 一个类型可以是cv-unqualified (非 `const` 或 `volatile`), const-qualified (`const`), volatile-qualified (声明为 `volatile`), or volatile-const-qualified (既 `const` 并 `volatile`); 类型的这些版本都是独特的。

Type_traits 的作者是 Steve Cleary, Beman Dawes, Aleksey Gurtovoy, Howard Hinnant, Jesse Jones, Mat Marcus, John Maddock, 和 Jeremy Siek, 以及其它许多人的贡献。

数学及数字处理

Boost.Integer

这个库提供了对整数类型的有用功能，如编译期的最小、最大值常数[3]， 基于给定位长的合适大小的类型，静态二进制对数计算等等。还包括从1999年C标准头文件 `<stdint.h>` 中的 `typedef` 。

> [3] `std::numeric_limits` 仅能以函数方式提供这些值。

Integer 的作者是 Beman Dawes 和 Daryle Walker.

Boost.Interval

Interval库帮助你使用数学区间。它提供类模板 `interval` 及相关算子。区间的常见用法(除了明显的进行区间计算的情况)是提供模糊结果的计算；区间的使用可以量化舍入误差的传播情况。

Interval 的作者是 Guillaume Melquiond, Sylvain Pion, 和 Hervé Brönniman, 该库从 Jens Maurer的前期工作获得灵感。

Boost.Math

Math是一组数学模板： `quaternion` s 和 `octonion` s (复数的特化)；数学函数如 `acosh` , `asinh` , 和 `sinhc` ；计算最大公约数(GCD)和最小公倍数(LCM)的函数等等。

Math 的作者是 Hubert Holin, Daryle Walker, 和 Eric Ford.

Boost.Minmax

Minmax可以同时计算最小和最大值，而使用 `std::min` 和 `std::max` 则要两次比较。对于 `n` 个元素的情况，只要 `3n/2+1` 次比较，而使用 `std::min_element` 和 `std::max_element` 则需要 `2n` 次比较。

Minmax 的作者是 Hervé Brönniman.

Boost.Numeric Conversion

Numeric Conversion库是一组用于在不同数字类型的值之间进行安全及可预言的转换的工具。例如，有一个名为 `numeric_cast` (最早来自于Boost.Conversion)的工具，提供了范围检测的转换以确定数值可被目标类型所表示，否则它会抛出异常。

Numeric Conversion 的作者是 Fernando Cacciola.

Boost.Operators

Operators库提供了相关操作符及概念(LessThanComparable, Arithmetic,等等)的实现。定义一个类型的操作符时，保证所有操作符都有定义是一件乏味并容易出错的工作。例如，你提供了 `operator<` (LessThanComparable)，通常都要同时提供 `operator<=`，`operator>`, 和 `operator>=` 。Operators可以根据给定类型的最小的用户自定义操作符集合，自动声明并定义其它所有的相关操作符。详细讨论见"Library 4: Operators 4."

Operators 的作者是 David Abrahams, Jeremy Siek, Aleksey Gurtovoy, Beman Dawes, 和 Daryle Walker.

Boost.Random

这是一个对随机数的专业使用的库，包括大量的生成器和分配器，可适用于多个不同的领域，如仿真和加密。Random已被收入即将发布的Library Technical Report.

Random 的作者是 Jens Maurer.

Boost.Rational

整数类型和浮点数类型都内建成于C++语言，复数类型也是C++标准库的一部分，但有理数类型呢？有 理数可以避免浮点数的精度损失问题，因此它们常被用于计算金钱等。Rational提供的有理数类型可以基于任意整数类型，包括用户自定义的整数类型(具 有无限精度的类型显然是很有用的).

Rational 的作者是 Paul Moore.

Boost.uBLAS

uBLAS库使用数学符号提供对向量和矩阵的基本线性代数操作，采用操作符重载，它可以生成紧凑的代码(使用表达式模板)。

uBLAS 的作者是 Joerg Walter 和 Mathias Koch.

输入/输出

Boost.Assign

Assign帮助你把一系列的值赋给容器。它通过对 `operator,` (逗号操作符) and `operator()()` (函数调用操作符)的重载，带给用户一种数据赋值的很容易的方法。除了对原型风格的代码特别有用，这个库的功能在其它时候也很有用，使用这个库有助于提高代码的可读性。使用本库中的 `list_of` 还可以就地生成无名数组。

Assign 的作者是 Thorsten Ottosen.

Boost.Filesystem

Filesystem库提供对路径、目录和文件操作的可移植性。这种高级抽象使C++程序员可以写出类似于其它编程语言脚本的代码。它提供了便于操作目录和文件的算法。编写要在不同文件系统平台间移植代码的困难工作由于这个库的帮助变得容易了。

Filesystem 的作者是 Beman Dawes.

Boost.Format

这个library加入了按格式化串进行格式化的功能，类似于 `printf`, 但增加了类型安全性。相反使用具有相同便利性的 `printf` 的最主要问题是参数类型的危险；它不保证格式化串中指定的类型与实际的参数类型是匹配的。除了消除了这种不匹配性的危险以外，Format还可以用于格式化用户自定义的类型。[4]

> [4] 格式化函数用省略号表示可变数量的参数是不可以的。

Format 的作者是 Samuel Krempp.

Boost.Io_state_savers

Io_state_savers库允许保存IOStream对象的状态，用于以后的恢复，以取消可能发生的任何状态的变化。许多操纵器会永久改变它们操作的流的状态，这可能是你不想要的，而手工重置状态又容易出错。这个状态保存器可以保存控制标志、精度、宽度、异常掩码、流的locale等等。

Io_state_savers 的作者是 Daryle Walker.

Boost.Serialization

这个库允许任意的C++数据结构存进来，再取出去，以及存档。例如，存档可以是文本文件或XML文件。Boost.Serialization是高度可移植的，并提供了非常成熟的特性，如类的版本、C++标准库中的通用类的序列化、共享数据的序列化，等等。

Serialization 的作者是 Robert Ramey.

杂项

Boost.Conversion

Conversion库包含有一些函数，它们是现有的强制类型转换操作符(`static_cast` , `const_cast` , 和 `dynamic_cast`)的增强。Conversion为安全的多态转换增加了 `polymorphic_cast` 和 `polymorphic_downcast` ，为安全的数字类型转换增加了 `numeric_cast` ，为文本转换(如 `string` 和 `double` 间的转换)增加 `lexical_cast` 。你可为了你自己的类型更好地工作而定制这些类型转换，可能这些类型并不可以使用语言本身所提供的类型转换。这个库的详细讨论在"Library 2: Conversion."

Conversion 的作者是 Dave Abrahams 和 Kevlin Henney.

Boost.Crc

Crc库提供了循环冗余码(CRC)的计算，常有于校验和类型。CRC被加到一个数据流中(它就是从这些数据中计算得来的)，用来对这些数据进行校验，例如PKZip就使用了CRC32。这个库包含了四个CRC类型： `crc_16_type` , `crc_ccitt_type` , `crc_xmodem_type` , 和 `crc_32_type5.`

Crc 的作者是 Daryle Walker.

Boost.Date_time

Date_time库提供了对日期和时间类型及对它们的操作的广泛支持。如果没有对日期和时间的支 持，程序开发任务会变得复杂并容易出错。使用Date_time，你想要的所有自然概念都被支持：日、周、月、持续时间(及时间间隔)、加、减等等。这个 库还提供了其它日期/时间库所忽略的东西，如闰秒处理以及高精度时间源的支持。这个库的设计是可扩展的，允许客户化定制行为或添加功能。

Date_time 的作者是 Jeff Garland.

Boost.Optional

要求函数可以指出它的返回值无效是一个很普通的要求，但通常返回类型并不存在某个状态来表示其无效。Optional提供了类模板 `optional` , 它是一个在语义上有额外状态的类型，它可以有效地表明 `optional` 的实例是否包含被封装对象实例。

Optional 的作者是 Fernando Cacciola.

Boost.Pool

Pool库提供了一个内存池分配器，它是一个工具，用于管理在一个独立的、大的分配空间里的动态内存。当你需要分配和回收许多不的对象或需要更高效的内存控制时，使用内存池是一个好的解决方案。

Pool 的作者是 Steve Cleary.

Boost.Preprocessor

当你要表示象循环这样的结构时，很难使用预处理器，它没有容器，不提供迭代器，等等。然而预处理器仍 是一个强大的可移植的工具。Preprocessor库提供了在预处理器之上的抽象。它包括lists, tuples, 和 arrays, 还有操作这些类型的algorithms。这个库有助于减少重复的代码，减轻你的负担，也使得代码更易读、更清晰、更具可维护性。

Preprocessor 的作者是 Vesa Karvonen 和 Paul Mensonides.

Boost.Program_options

Program_options库提供了程序选项配置(名字/值对), 程序选项通常是通过命令行参数或配置文件提供。这个库减轻了程序员手工分析这些数据的负担。

Program_options 的作者是 Vladimir Prus.

Boost.Python

Python库提供了C++与Python[6]的互操作性。它用于将C++类及函数提供给Python，同样把Python对象给C++。它是非插入式的，也就是说已有代码无需修改即可用于Python。

> [6] 一种你应该知道的非常流行的编程语言。

Python 的作者是 David Abrahams, 并得到Joel de Guzman 和 Ralf W. Grosse-Kunstleve的重要贡献。

Boost.Smart_ptr

智能指针是任何一个程序员工具包中的重要部分。它们用于防止资源泄漏、共享资源、对象生存期管理。有 很多好的智能指针库可用，有些是免费的，而有些是商业软件包的组成部分。Smart_ptr是其中的佼佼者，已被成千上万的用户所证实，并被该领域的专家 所推荐。Smart_ptr包括了非插入的智能指针用于限制范围(`scoped_ptr` 和 `scoped_array`)，用于共享资源(`shared_ptr` 和 `shared_array`), 一个配合 `shared_ptr` 使用的智能指针(`weak_ptr`), 还有一个插入式的智能指针类(`intrusive_ptr`). Smart_ptr的 `shared_ptr` (包括它的助手 `enable_shared_from_this`) 以及 `weak_ptr` 已被收入即将发布的Library Technical Report。关于智能指针更详细的说明请见"Library 1: Smart_ptr 1."

Smart_ptr 的作者是 Greg Colvin, Beman Dawes, Peter Dimov, 和 Darin Adler.

Boost.Test

Test库提供了一整组用于编写测试程序的组件，可以把测试组织成简单的测试用例及测试套装，并控制它们的执行。作为这个库的一个组件，程序执行监视器在某些生产(非测试)环境下也很有用。

Test 的作者是 Gennadiy Rozental (基于Beman Dawes早期的工作).

Boost.Thread

可移植的线程是很难处理的业务，也无法从C++本身获取帮助，因为语言本身不包括线程支持。当然，我们有POSIX, 它在许多平台上可用，但POSIX使用的是C API。Thread是一个提供可移植线程的库，它包含大量线程的原始概念和高度抽象。

Thread 的作者是 William Kempf.

Boost.Timer

Timer库包含计时所需的特性，它的目标是尽可能做到跨平台的一致性。虽然每个平台都有特定的 API可以让程序员用于计时，但对于高精度计时还没有可移植的方案。Boost.Timer通过提供最大可能的精度并同时保留可移植性解决了这个问题，从 而可以让你自由地确定精度。

Timer 的作者是 Beman Dawes.

Boost.Tribool

这个库包含一个 `tribool` 库，它实现了三状态布尔逻辑。三状态布尔类型除了true 和 false 以外还有一个额外的状态：indeterminate (这个状态也被称为maybe; 这个名字是可配置的).

Tribool 的作者是 Douglas Gregor.

Boost.Utility

一些本不应在一个库里出现的有用的东西，只是因为它们每个都不太复杂和广泛，不足够形成一个单独的库。但不是说它们没有什么用外；事实上小的工具通常都有最广泛的用处。在 Boost, 这些小工具被集中起来，形成一个称为Utility的库。你可以在这找到 `checked_delete`，一个函数，用于确认在删除点的类型是完整的；还有类 `noncopyable`，用于确保类不能被复制；还有 `enable_if`，用于对函数重载的完全控制。还有其它很多工具，详细请见"Library 3: Utility"。

Utility 的作者是 David Abrahams, Daryle Walker, Douglas Gregor, 和其它人。

Boost.Value_initialized

Value_initialized库帮助你用泛型的方法构造和初始化对象。在C++里，一个新构造的对象可以是零初始化的、缺省构造的，或是不确定的，这依赖于对象的类型。有了Boost.Value_initialized, 这种不一致的问题就没有了。

Value_initialized 的作者是 Fernando Cacciola.

Part I: 通用库

要给本书的这一部分起一个合适的名字并不容易。本书的结构是围绕各个不同领域(如容器和高级编程)，那些名字都好取；除了这一部分，它包括一些我们经常用到的东西：智能指针、类型转换工具等等。

总不能一开始第一部分就叫Miscellaneous, 或者 Ubiquitous, 或者 Frequently Used Libraries. 虽然它们的确就是这些东西，但这些名字并不能真正表达它们的重要性。因此，我决定命名为General Libraries, 希望可以表示出它们的无所不在。

一件经常困扰我的事情就是我们关注那些"简单"工具的方式，你应该同意它们是很有用的。在很多书 和文章中，它们都得到了很大的关注，但令人奇怪的是，在为产品代码选择工具(或创建工具)时，它们又往往被低估了。这是因为我们认为这些小组件太简单了吗？我们是否从根本上就忽略了类似组件的灵活性可以很容易地实现，而是为适应每个问题而手工去重做？如果这些是真的，我们这样做就错了。如果程序中有两百 万个智能指针的实现，会使得智能指针在效率和可靠性方面都很危险。一个程序中有二十个不同的通用类型转换的实现同样也会花掉不少的代码时间，但更重要的是 这样的代码会很难维护。系统应该由多层的抽象组成，底层通常由数据结构、算法和工具组成。如果你同意这一点，想一想这些小的、无关重要的、被忽视的工具发 生变更时的影响，或者是程序缺陷，或者是没有保证的坚固性。这些小工具是船，承载着我们程序的纹理进行交换。它们是我们的逻辑引擎中的油，是我们的隔板间 的胶水。够了，我们应该给予它们应用的信任，不是吗？我们将在这里讨论多个通用库，包括智能指针, 转换 (包括类型转换和文字的转换), 正则表达式, 操作符, 静态断言等等。

Library 1. Smart_ptr

- Smart_ptr库如何改进你的程序？
- 何时我们需要智能指针？
- Smart_ptr如何适应标准库？
- scoped_ptr
- scoped_array
- shared_ptr
- shared_array
- intrusive_ptr
- weak_ptr
- Smart_ptr总结

Smart_ptr库如何改进你的程序？

- 使用 `shared_ptr` 进行对象的生存期自动管理，使得分享资源所有权变得有效且安全。
- 使用 `weak_ptr` 可以安全地观测共享资源，避免了悬挂的指针。
- 使用 `scoped_ptr` 和 `scoped_array` 限制资源的使用范围，使得代码更易于编写和维护，并有助于写出异常安全的代码。

智能指针解决了资源生存期管理的问题(尤其是动态分配的对象[1]). 智能指针有各种不同的风格。多数都有一种共同的关键特性：自动资源管理。这种特性可能以不同的方式出现：如动态分配对象的生存期控制，和获取及释放资源 (文件, 网络连接)。Boost的智能指针主要针对第一种情况，它们保存指向动态分配对象的指针，并在正确的时候删除这些对象。你可能觉得奇怪为什么这些智能指针 不多做一点工作。它们不可以很容易就覆盖所有资源管理的不同情况吗？是的，它们可以(在一定范围内它们可以)，但不是没有代价的。通用的解决方案意味着更 高的复杂性，而对于Boost的智能指针，可用性比灵活性具有更高的优先级。但是，通过对可定制删除器的支持，Boost的最智能的智能指针(`boost::shared_ptr`)可以支持那些不是使用 `delete` 进行析构的资源。 `Boost.Smart_ptr` 的五个智能指针类型是专门特制的，适用于每天的编程中最常见的需求。

> [1] 因为泛型智能指针可以处理任何类型的资源。

何时我们需要智能指针？

有三种典型的情况适合使用智能指针：

- 资源所有权的共享
- 要编写异常安全的代码时
- 避免常见的错误，如资源泄漏

共享所有权是指两个或多个对象 需要同时使用第三个对象的情况。这第三个对象应该如何(或者说何时)被释放？为了确保释放的时机是正确的，每个使用这个共享资源的对象必须互相知道对方， 才能准确掌握资源的释放时间。从设计或维护的观点来看，这种耦合是不可行的。更好的方法是让这些资源所有者将资源的生存期管理责任委派给一个智能指针。当 没有共享者存在时，智能指针就可以安全地释放这个资源了。

异常安全，简单地说就是在异常 抛出时没有资源泄漏并保证程序状态的一致性。如果一个对象是动态分配的，当异常抛出时它不会自动被删除。由于栈展开以及指针离开作用域，资源可以会泄漏直 至程序结束(即使是程序结束时的资源回收也不是由语言所保证的)。不仅可能程序会由于内存泄漏而耗尽资源，程序的状态也可能变得混乱。智能指针可以自动地 为你释放这些资源，即使是在异常发生的情况下。

避免常见的错误。忘记调用 `delete` 是书本中最古老的错误(至少在这本书中)。一个智能指针不关心程序中的控制路径；它只关心在它所指向的对象的生存期结束时删除它。使用智能指针，你不再需要知道何时删除对象。并且，智能指针隐藏了释放资源的细节，因此使用者不需要知道是否要调用 `delete`, 有些特殊的清除函数并不总是删除资源的。

安全和高效的智能指针是程序员的军火库中重要的武器。虽然C++标准库中提供了 `std::auto_ptr`, 但是它不能完全满足我们对智能指针的需求。例如， `auto_ptr` 不能用作STL容器的元素。Boost的智能指针类填充了标准所留下来的缺口。

本章主要关注 scoped_ptr, shared_ptr, intrusive_ptr, 和 weak_ptr. 虽然剩下的 `scoped_array` 和 `shared_array` 有时候也很有用，但它们用的不是很多，而且它们与已讨论的非常相近，这里就不重复讨论它们了。

Smart_ptr如何适应标准库？

Smart_ptr库已被提议包含进标准库中，主要有以下三个原因：

- 标准库现在只提供了一个 `auto_ptr`，它仅是一类智能指针，仅仅覆盖了智能指针族谱中的一个部分。 `shared_ptr` 提供了不同的，也是更重要的功能。
- Boost的智能指针专门为了与标准库良好合作而设计，并可作为标准库的自然扩充。例如，在 `shared_ptr` 之前，还没有一个标准的智能指针可用作容器的元素。
- 长久以来，现实世界中的程序员已经在他们的程序中大量使用这些智能指针类，它们已经得到了充分的验证。

以上原因使得Smart_ptr库成为了C++标准库的一个非常有用的扩充。Boost.Smart_ptr的 `shared_ptr` (以及随同的助手 `enable_shared_from_this`) 和 `weak_ptr` 已被收入即将发布的Library Technical Report。

scoped_ptr

头文件: `"boost/scoped_ptr.hpp"`

`boost::scoped_ptr` 用于确保动态分配的对象能够被正确地删除。`scoped_ptr` 有着与 `std::auto_ptr` 类似的特性，而最大的区别在于它不能转让所有权而 `auto_ptr` 可以。事实上，`scoped_ptr` 永远不能被复制或被赋值！`scoped_ptr` 拥有它所指向的资源的所有权，并永远不会放弃这个所有权。`scoped_ptr` 的这种特性提升了我们的代码的表现，我们可以根据需要选择最合适的智能指针(`scoped_ptr` 或 `auto_ptr`)。

要决定使用 `std::auto_ptr` 还是 `boost::scoped_ptr` , 就要考虑转移所有权是不是你想要的智能指针的一个特性。如果不是，就用 `scoped_ptr` . 它是一种轻量级的智能指针；使用它不会使你的程序变大或变慢。它只会让你的代码更安全，更好维护。

下面是 `scoped_ptr` 的摘要，以及其成员的简要描述：

```
namespace boost {

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

    void reset(T* p = 0);

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

    void swap(scoped_ptr& b);
  };

  template<typename T>
    void swap(scoped_ptr<T> & a, scoped_ptr<T> & b);
}
```

成员函数

```
explicit scoped_ptr(T* p=0)
```

构造函数，存储 `p` 的一份拷贝。注意， `p` 必须是用 `operator new` 分配的，或者是null. 在构造的时候，不要求 `T` 必须是一个完整的类型。当指针 `p` 是调用某个分配函数的结果而不是直接调用 `new` 得到的时候很有用：因为这个类型不必是完整的，只需要类型 `T` 的一个前向声明就可以了。这个构造函数不会抛出异常。

```
~scoped_ptr()
```

删除被指物。类型 `T` 在被销毁时必须是一个完整的类型。如果 `scoped_ptr` 在它被析构时并没有保存资源，它就什么都不做。这个析构函数不会抛出异常。

```
void reset(T* p=0);
```

重置一个 `scoped_ptr` 就是删除它已保存的指针，如果它有的话，并重新保存 `p`. 通常，资源的生存期管理应该完全由 `scoped_ptr` 自己处理，但是在极少数时候，资源需要在 `scoped_ptr` 的析构之前释放，或者 `scoped_ptr` 要处理它原有资源之外的另外一个资源。这时，就可以用 `reset`，但一定要尽量少用它。(过多地使用它通常表示有设计方面的问题) 这个函数不会抛出异常。

```
T& operator*() const;
```

返回一个到被保存指针指向的对象的引用。由于不允许空的引用，所以解引用一个拥有空指针的 `scoped_ptr` 将导致未定义行为。如果不能肯定所含指针是否有效，就用函数 `get` 替代解引用。这个函数不会抛出异常。

```
T* operator->() const;
```

返回保存的指针。如果保存的指针为空，则调用这个函数会导致未定义行为。如果不能肯定指针是否空的，最好使用函数 `get`。这个函数不会抛出异常。

```
T* get() const;
```

返回保存的指针。应该小心地使用 `get`，因为它可以直接操作裸指针。但是， `get` 使得你可以测试保存的指针是否为空。这个函数不会抛出异常。 `get` 通常在调用那些需要裸指针的函数时使用。

```
operator unspecified_bool_type() const
```

返回 `scoped_ptr` 是否为非空。返回值的类型是未指明的，但这个类型可被用于Boolean的上下文中。在if语句中最好使用这个类型转换函数，而不要用 `get` 去测试 `scoped_ptr` 的有效性

```
void swap(scoped_ptr& b)
```

交换两个 `scoped_ptr` 的内容。这个函数不会抛出异常。

普通函数

```
template<typename T> void swap(scoped_ptr<T>& a,scoped_ptr<T>& b)
```

这个函数提供了交换两个scoped pointer的内容的更好的方法。之所以说它更好，是因为 `swap(scoped1,scoped2)` 可以更广泛地用于很多指针类型，包括裸指针和第三方的智能指针。[2] `scoped1.swap(scoped2)` 则只能用于它的定义所在的智能指针，而不能用于裸指针。

> [2] 你可为那些不够智能，没有提供它们自己的交换函数的智能指针创建你的普通swap函数。

用法

`scoped_ptr` 的用法与普通的指针没什么区别；最大的差别在于你不必再记得在指针上调用 `delete` ，还有复制是不允许的。典型的指针操作(`operator*` 和 `operator->`)都被重载了，并提供了和裸指针一样的语法。用 `scoped_ptr` 和用裸指针一样快，也没有大小上的增加，因此它们可以广泛使用。使用 `boost::scoped_ptr` 时，包含头文件 `"boost/scoped_ptr.hpp"` . 在声明一个 `scoped_ptr` 时，用被指物的类型来指定类模板的参数。例如，以下是一个包含 `std::string` 指针的 `scoped_ptr` ：

```
boost::scoped_ptr<std::string> p(new std::string("Hello"));
```

当 `scoped_ptr` 被销毁时，它对它所拥有的指针调用 `delete` 。

不需要手工删除

让我们看一个程序，它使用 `scoped_ptr` 来管理 `std::string` 指针。注意这里没有对 `delete` 的调用，因为 `scoped_ptr` 是一个自动变量，它会在离开作用域时被销毁。

```
#include "boost/scoped_ptr.hpp"
#include <string>
#include <iostream>

int main() {
  {
  boost::scoped_ptr<std::string>
  p(new std::string("Use scoped_ptr often."));

  // 打印字符串的值
  if (p)
    std::cout << *p << '\n';

  // 获取字符串的大小
  size_t i=p->size();

  // 给字符串赋新值
  *p="Acts just like a pointer";

  } // 这里p被销毁，并删除std::string
}
```

这段代码中有几个地方值得注明一下。首先， `scoped_ptr` 可以测试其有效性，就象一个普通指针那样，因为它提供了隐式转换到一个可用于布尔表达式的类型的方法。其次，可以象使用裸指针那样调用被指物的成员函数，因为重载了 `operator->` . 第三，也可以和裸指针一

样解引用 `scoped_ptr`，这归功于 `operator*` 的重载。这些特性正是 `scoped_ptr` 和其它智能指针的用处所在，因为它们和裸指针的不同之处在于对生存期管理的语义上，而不在于语法上。

和auto_ptr几乎一样

`scoped_ptr` 与 `auto_ptr` 间的区别主要在于对拥有权的处理。`auto_ptr` 在复制时会从源 `auto_ptr` 自动交出拥有权，而 `scoped_ptr` 则不允许被复制。看看下面这段程序，它把 `scoped_ptr` 和 `auto_ptr` 放在一起，你可以清楚地看到它们有什么不同。

```
void scoped_vs_auto() {

  using boost::scoped_ptr;
  using std::auto_ptr;

  scoped_ptr<std::string> p_scoped(new std::string("Hello"));
  auto_ptr<std::string> p_auto(new std::string("Hello"));

  p_scoped->size();
  p_auto->size();

  scoped_ptr<std::string> p_another_scoped=p_scoped;
  auto_ptr<std::string> p_another_auto=p_auto;

  p_another_auto->size();
  (*p_auto).size();
}
```

这个例子不能通过编译，因为 `scoped_ptr` 不能被复制构造或被赋值。`auto_ptr` 既可以复制构造也可以赋值，但这们同时也意味着它把所有权从 `p_auto` 转移给了 `p_another_auto`，在赋值后 `p_auto` 将只剩下一个空指针。这可能会导致令人不快的惊讶，就象你试图把 `auto_ptr` 放入容器内时所发生的那样。[3] 如果我们删掉对 `p_another_scoped` 的赋值，程序就可以编译了，但它的运行结果是不可预测的，因为它解引用了 `p_auto` 里的空指针 `(*p_auto)`.

> [3] 永远不要把 `auto_ptr` 放入标准库的容器里。如果你试一下，通常你会得到一个编译错误；如果你没有得到错误，你就麻烦了。

由于 `scoped_ptr::get` 会返回一个裸指针，所以就有可能对 `scoped_ptr` 做一些有害的事情，其中有两件是你尤其要避免的。第一，不要删除这个裸指针。因为它会在 `scoped_ptr` 被销毁时再一次被删除。第二，不要把这个裸指针保存到另一个 `scoped_ptr` (或其它任何的智能指针)里。因为这样也会两次删除这个指针，每个 `scoped_ptr` 一次。简单地说，尽量少用 `get`，除非你要使用那些要求你传送裸指针的遗留代码！

scoped_ptr 和Pimpl用法

`scoped_ptr` 可以很好地用于许多以前使用裸指针或 `auto_ptr` 的地方，如在实现pimpl用法时。[4]pimpl 用法背后的思想是把客户与所有关于类的私有部分的知识分隔开。由于客户是依赖于类的头文件的，头文件中的任何变化都会影响客户，即使仅是对私有节或保护节 的修

改。pimpl用法隐藏了这些细节，方法是将私有数据和函数放入一个单独的类中，并保存在一个实现文件中，然后在头文件中对这个类进行前向声明并保存 一个指向该实现类的指针。类的构造函数分配这个pimpl类，而析构函数则释放它。这样可以消除头文件与实现细节的相关性。我们来构造一个实现pimpl 用法的类，然后用智能指针让它更为安全。

> [4] 这也被称为Cheshire Cat 用法. 关于pimpl用法更多的说明请见 www.gotw.ca/gotw/024.htm 和 Exceptional C++ 。

```
// pimpl_sample.hpp

#if !defined (PIMPL_SAMPLE)
#define PIMPL_SAMPLE

class pimpl_sample {
  struct impl;  // 译者注：原文中这句在class之外，与下文的实现代码有矛盾
  impl* pimpl_;
public:
  pimpl_sample();
  ~pimpl_sample();
  void do_something();
};

#endif
```

这是 `pimpl_sample` 类的接口。 `struct impl` 是一个前向声明，它把所有私有成员和函数放在另一个实现文件中。这样做的效果是使客户与 `pimpl_sample` 类的内部细节完全隔离开来。

```
// pimpl_sample.cpp

#include "pimpl_sample.hpp"
#include <string>
#include <iostream>

struct pimpl_sample::impl {
  void do_something_() {
    std::cout << s_ << "\n";
  }

  std::string s_;
};

pimpl_sample::pimpl_sample()
  : pimpl_(new impl) {
  pimpl_->s_ = "This is the pimpl idiom";
}

pimpl_sample::~pimpl_sample() {
  delete pimpl_;
}

void pimpl_sample::do_something() {
  pimpl_->do_something_();
}
```

看起来很完美，但并不是的。这个实现不是异常安全的！原因是 `pimpl_sample` 的构造函数有可能在 `pimpl` 被构造后抛出一个异常。在构造函数中抛出异常意味着已构造的对象并不存在，因此在栈展开时将不会调用它的析构函数。这样就意味着分配给 `pimpl_` 指针的内存将泄漏。然而，有一样简单的解决方法：用 `scoped_ptr` 来解救！

```
class pimpl_sample {
  struct impl;
  boost::scoped_ptr<impl> pimpl_;
  ...
};
```

让 scoped_ptr 来处理隐藏类 impl 的生存期管理，并从析构函数中去掉对 impl 的删除(它不再需要，这要感谢 scoped_ptr)，这样就做完了。但是，你必须记住要手工定义析构函数；原因是在编译器生成隐式析构函数时，类 impl 还是不完整的，所以它的析构函数不能被调用。如果你用 auto_ptr 来保存 impl , 你可以编译，但也还是有这个问题，但如果用 scoped_ptr , 你将收到一个错误提示。

要注意的是，如果你使用 scoped_ptr 作为一个类的成员，你就必须手工定义这个类的复制构造函数和赋值操作符。原因是 scoped_ptr 是不能复制的，因此聚集了它的类也变得不能复制了。

最后一点值得注意的是，如果pimpl实例可以安全地被多个封装类(在这里是 pimpl_sample)的实例所共享，那么用 boost::shared_ptr 来管理pimpl的生存期才是正确的选择。
用 shared_ptr 比用 scoped_ptr 的优势在于，不需要手工去定义复制构造函数和赋值操作符，而且可以定义空的析构函数， shared_ptr 被设计为可以正确地用于未完成的类。

scoped_ptr 不同于 const auto_ptr

留心的读者可能已经注意到 auto_ptr 可以几乎象 scoped_ptr 一样地工作，只要把 auto_ptr 声明为 const ：

```
const auto_ptr<A> no_transfer_of_ownership(new A);
```

它们很接近，但不是一样。最大的区别在于 scoped_ptr 可以被 reset , 在需要时可以删除并替换被指物。而对于 const auto_ptr 这是不可能的。另一个小一点的区别是，它们的名字不同：尽管 const auto_ptr 意思上和 scoped_ptr 一样，但它更冗长，也更不明显。当你的词典里有了 scoped_ptr ，你就应该使用它，因为它可以更清楚地表明你的意图。如果你想说一个资源是要被限制在作用域里的，并且不应该有办法可以放弃它的所有权，你就应该用 boost::scoped_ptr .

总结

使用裸指针来写异常安全和无错误的代码是很复杂的。使用智能指针来自动地把动态分配对象的生存期限制在一个明确的范围之内，是解决这种问题的一个有效方法，并且提高了代码的可读性、可维护性和质量。 scoped_ptr 明确地表示被指物不能被共享和转移。正如你所看到的， std::auto_ptr 可以从另一个 auto_ptr 那里窃取被指物，那怕是无意的，这被认为是 auto_ptr 的最大缺点。正是这个缺点使得 scoped_ptr 成为 auto_ptr 最好的补充。当一个动态分配的对象被传送给 scoped_ptr , 它就成为了这个对象的唯一的拥有者。因

为 `scoped_ptr` 几乎总是以自动变量或数据成员来分配的，因此它可以在离开作用域时正确地销毁对象，从而在执行流由于返回语句或异常抛出而离开作用域时，也总能释放它所管理的内存。

在以下情况时使用 `scoped_ptr` ：

- 在可能有异常抛出的作用域里使用指针
- 函数里有几条控制路径
- 动态分配对象的生存期应被限制于特定的作用域内
- 异常安全非常重要时(总应如此!)

scoped_array

头文件: `"boost/scoped_array.hpp"`

需要动态分配数组时，通常最好用 `std::vector` 来实现，但是有两种情形看起来用数组更适合： 一种是为了优化，用 `vector` 多少有一些额外的内存和速度开销；另一种是为了某种原因，要求数组的大小必须是固定的。[5] 动态分配的数组会遇到与普通指针一样的危险，并且还多了一个(也是最常见的一个)，那就是错误调用 `delete` 操作符而不是 `delete[]` 操作符来释放数组。我曾经在你想象不到的地方见到过这个错误，那也是它常被用到的地方，就是在你自己实现的容器类里！ `scoped_array` 为数组做了 `scoped_ptr` 为单个对象指针所做的事情：它负责释放内存。区别只在于 `scoped_array` 是用 `delete[]` 操作符来做这件事的。

> [5] 如果没有非常清晰的优点，最好还是用 `std::vector` ，除非性能测试表明 `scoped_array` 的好处是有保证的。

`scoped_array` 是一个单独的类而不是 `scoped_ptr` 的一个特化，其原因是，因为不可能用元编程技术来区分指向单个对象的指针和指向数组的指针。不管如何努力，也没有人能发现一种可靠的方法，因为数组太容易退化为指针了，这使得没有类型信息可以表示它们是指向数组的。结果，只能由你来负责，使用 `scoped_array` 而不是 `scoped_ptr` ，就如你必须用 `delete[]` 操作符而不是用 `delete` 操作符一样。这样的好处是 `scoped_array` 负责为你处理释放内存的事情，而你则告诉 `scoped_array` 我们要处理的是数组，而不是裸指针。

`scoped_array` 与 `scoped_ptr` 非常相似，不同的是它提供了 `operator[]` 来模仿一个裸数组。

`scoped_array` 是比普通的动态分配数组更好用。它处理了动态分配数组的生存期管理问题，就如 `scoped_ptr` 管理对象指针的生存期一样。但是记住，多数情况下应该使用 `std::vector` ，它更灵活、更强大。只有当你需要确保数组的大小是固定的时候，才使用 `scoped_array` 来替代 `std::vector` .

shared_ptr

头文件: `"boost/shared_ptr.hpp"`

几乎所有稍微复杂点的程序都需要某种形式的引用计数智能指针。这些智能指针让我们不再需要为了管理被两个或多个对象共享的对象的生存期而编写复杂的逻辑。当引用计数降为零，没有对象再需要这个共享的对象时，这个对象就自动被销毁了。引用计数智能指针可以分为插入式(intrusive)和非插入式(non-intrusive)两类。前者要求它所管理的类提供明确的函数或数据成员用于管理引用计数。这意味着在类的设计时就必须预见到它将与一个插入式的引用计数智能指针一起工作，或者重新设计它。非插入式的引用计数智能指针对它所管理的类没有任何要求。引用计数智能指针拥有与它所存指针有关的内存的所有权。没有智能指针的帮助，对象的共享会存在问题，必须有人负负责删除共享的内存。谁负责？什么时候删除？没有智能指针，你必须在管理的内存之外增加生存期的管理，这意味着在各个拥有者之间存在更强的依赖关系。换言之，没有了重用性并增加了复杂性。

被管理的类可能拥有一些特性使得它更应该与引用计数智能指针一起使用。例如，它的复制操作很昂贵，或者它所代表的有些东西必须被多个实例共享，这些特性都值得去共享所有权。还有一种情形是共享的资源没有一个明确的拥有者。使用引用计数智能指针可以在需要访问共享资源的对象之间共享资源的所有权。引用计数智能指针还让你可以把对象指针存入标准库的容器中而不会有泄漏的风险，特别是在面对异常或要从容器中删除元素的时候。如果你把指针放入容器，你就可以获得多态的好处，可以提高性能(如果复制的代价很高的话)，还可以通过把相同的对象放入多个辅助容器来进行特定的查找。

在你决定使用引用计数智能指针后，你应该选择插入式的还是非插入式的？非插入式智能指针几乎总是更好的选择，由于它们的通用性、不需要修改已有代码，以及灵活性。你可以对你不能或不想修改的类使用非插入式的引用计数智能指针。而把一个类修改为使用插入式引用计数智能指针的常见方法是从一个引用计数基类派生。这种修改可能比你想象的更昂贵。至少，它增加了相关性并降低了重用性。[6] 它还增加了对象的大小，这在一些特定环境中可能会限制其可用性。[7]

> [6] 考虑一下对同一个类型使用两个以上引用计数智能指针的需要。如果两个都是插入式的，两个不同的基类可能会不兼容，而且也很浪费。如果其中一个是插入式的，那么使用非插入式的智能指针可以使基类的额外负担为零。
>
> [7] 另一方面，非插入式智能指针要求额外的存储用于智能指针本身。

`shared_ptr` 可以从一个裸指针、另一个 `shared_ptr`、一个 `std::auto_ptr`、或者一个 `boost::weak_ptr` 构造。还可以传递第二个参数给 `shared_ptr` 的构造函数，它被称为删除器(deleter)。删除器稍后会被调用，来处理共享资源的释放。这对于管理那些不是用 `new` 分

配也不是用 `delete` 释放的资源时非常有用(稍后将看到创建客户化删除器的例子)。 `shared_ptr` 被创建后，它就可象普通指针一样使用了，除了一点，它不能被显式地删除。

以下是shared_ptr的部分摘要；最重要的成员和相关普通函数被列出，随后是简单的讨论。

```
namespace boost {

  template<typename T> class shared_ptr {
  public:
    template <class Y> explicit shared_ptr(Y* p);
    template <class Y,class D> shared_ptr(Y* p,D d);

    ~shared_ptr();

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

    shared_ptr& operator=(const shared_ptr& r);

    void reset();

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

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

    operator unspecified_bool_type() const;  //译注：原文是unspecified-bool-type()，有误

    void swap(shared_ptr<T>& b);
  };

  template <class T,class U>
    shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);
}
```

成员函数

```
template <class Y> explicit shared_ptr(Y* p);
```

这个构造函数获得给定指针 `p` 的所有权。参数 `p` 必须是指向 `Y` 的有效指针。构造后引用计数设为1。唯一从这个构造函数抛出的异常是 `std::bad_alloc` (仅在一种很罕见的情况下发生，即不能获得引用计数器所需的自由空间)。

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

这个构造函数带有两个参数。第一个是 `shared_ptr` 将要获得所有权的那个资源，第二个是 `shared_ptr` 被销毁时负责释放资源的一个对象，被保存的资源将以 `d(p)` 的形式传给那个对象。因此 `p` 的值是否有效取决于 `d` 。如果引用计数器不能分配成功， `shared_ptr` 抛出一个类型为 `std::bad_alloc` 的异常。

```
shared_ptr(const shared_ptr& r);
```

`r` 中保存的资源被新构造的 `shared_ptr` 所共享，引用计数加一。这个构造函数不会抛出异常。

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

从一个weak_ptr (本章稍后会介绍)构造shared_ptr。这使得 `weak_ptr` 的使用具有线程安全性，因为指向 `weak_ptr` 参数的共享资源的引用计数将会自增(`weak_ptr` 不影响共享资源的引用计数)。如果 `weak_ptr` 为空 (`r.use_count()==0`), `shared_ptr` 抛出一个类型为 `bad_weak_ptr` 的异常。

```
template <typename Y> shared_ptr(std::auto_ptr<Y>& r);
```

这个构造函数从一个 `auto_ptr` 获取 `r` 中保存的指针的所有权，方法是保存指针的一份拷贝并对 `auto_ptr` 调用 `release` 。构造后的引用计数为1。而 `r` 当然就变为空的。如果引用计数器不能分配成功，则抛出 `std::bad_alloc` 。

```
~shared_ptr();
```

`shared_ptr` 析构函数对引用计数减一。如果计数为零，则保存的指针被删除。删除指针的方法是调用 `operator delete` 或者，如果给定了一个执行删除操作的客户化删除器对象，就把保存的指针作为唯一参数调用这个对象。析构函数不会抛出异常。

```
shared_ptr& operator=(const shared_ptr& r);
```

赋值操作共享 `r` 中的资源，并停止对原有资源的共享。赋值操作不会抛出异常。

```
void reset();
```

`reset` 函数用于停止对保存指针的所有权的共享。共享资源的引用计数减一。

```
T& operator*() const;
```

这个操作符返回对已存指针所指向的对象的一个引用。如果指针为空，调用 `operator*` 会导致未定义行为。这个操作符不会抛出异常。

```
T* operator->() const;
```

这个操作符返回保存的指针。这个操作符与 `operator*` 一起使得智能指针看起来象普通指针。这个操作符不会抛出异常。

```
T* get() const;
```

`get` 函数是当保存的指针有可能为空时(这时 `operator*` 和 `operator->` 都会导致未定义行为)获取它的最好办法。注意，你也可以使用隐式布尔类型转换来测试 `shared_ptr` 是否包含有效指针。这个函数不会抛出异常。

```
bool unique() const;
```

这个函数在 `shared_ptr` 是它所保存指针的唯一拥有者时返回 `true` ；否则返回 `false` 。`unique` 不会抛出异常。

```
long use_count() const;
```

`use_count` 函数返回指针的引用计数。它在调试的时候特别有用，因为它可以在程序执行的关键点获得引用计数的快照。小心地使用它，因为在某些可能的 `shared_ptr` 实现中，计算引用计数可能是昂贵的，甚至是不行的。这个函数不会抛出异常。

```
operator unspecified-bool-type() const;
```

这是个到 `unspecified-bool-type` 类型的隐式转换函数，它可以在Boolean上下文中测试一个智能指针。如果 `shared_ptr` 保存着一个有效的指针，返回值为 `True` ；否则为 `false` 。注意，转换函数返回的类型是不确定的。把返回类型当成 `bool` 用会导致一些荒谬的操作，所以典型的实现采用了safe bool idiom,[8] 它很好地确保了只有可适用的Boolean测试可以使用。这个函数不会抛出异常。

> [8] 由Peter Dimov发明的。

```
void swap(shared_ptr<T>& b);
```

这可以很方便地交换两个 `shared_ptr` 。 `swap` 函数交换保存的指针(以及它们的引用计数)。这个函数不会抛出异常。

普通函数

```
template <typename T,typename U>
  shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);
```

要对保存在 `shared_ptr` 里的指针执行 `static_cast`，我们可以取出指针然后强制转换它，但我们不能把它存到另一个 `shared_ptr` 里；新的 `shared_ptr` 会认为它是第一个管理这些资源的。解决的方法是用 `static_pointer_cast`. 使用这个函数可以确保被指物的引用计数保持正确。`static_pointer_cast` 不会抛出异常。

用法

使用 `shared_ptr` 解决的主要问题是知道删除一个被多个客户共享的资源的正确时机。下面是一个简单易懂的例子，有两个类 `A` 和 `B`, 它们共享一个 `int` 实例。使用 `boost::shared_ptr`, 你需要必须包含 `"boost/shared_ptr.hpp"`.

```
#include "boost/shared_ptr.hpp"
#include <cassert>

class A {
  boost::shared_ptr<int> no_;
public:
  A(boost::shared_ptr<int> no) : no_(no) {}
  void value(int i) {
    *no_=i;
  }
};

class B {
  boost::shared_ptr<int> no_;
public:
  B(boost::shared_ptr<int> no) : no_(no) {}
  int value() const {
    return *no_;
  }
};

int main() {
    boost::shared_ptr<int> temp(new int(14));
    A a(temp);
    B b(temp);
    a.value(28);
    assert(b.value()==28);
}
```

类 `A` 和 `B` 都保存了一个 `shared_ptr&lt;int&gt;`. 在创建 `A` 和 `B` 的实例时，`shared_ptr temp` 被传送到它们的构造函数。这意味着共有三个 `shared_ptr`：`a`，`b`，和 `temp`，它们都引向同一个 `int` 实例。如果我们用指针来实现对一个的共享，`A` 和 `B` 必须能够在某个时间指出这个 `int` 要被删除。在这个例子中，直到 `main` 的结束，引用计数为3，当所有 `shared_ptr` 离开了作用域，计数将达到0，而最后一个智能指针将负责删除共享的 `int`.

回顾**Pimpl**用法

前一节展示了使用 `scoped_ptr` 的pimpl 用法，如果使用这种用法的类是不允许复制的，那么 `scoped_ptr` 在保存pimpl的动态分配实例时它工作得很好。但是这并不适合于所有想从pimpl用法中获益的类型(注意，你还可以用 `scoped_ptr`，但必须手工实现复制构造函数和赋

值操作符)。对于那些可以处理共享的实现细节的类，应该用 `shared_ptr` 。当pimpl的所有权被传递给一个 `shared_ptr` ，复制和赋值操作都是免费的。你可以回忆起，当使用 `scoped_ptr` 去处理pimpl类的生存期时，对封装类的复制是不允许的，因为 `scoped_ptr` 是不可复制的。这意味着要使这些类支持复制和赋值，你必须手工定义复制构造函数和赋值操作符。当使用 `shared_ptr` 去处理pimpl类的生存期时，就不再需要用户自己定义复制构造函数了。注意，这时pimpl实例是被该类的多个对象所共享，因此如果规则是每个pimpl实例只能被类的一个实例使用，你还是要手工编写复制构造函数。解决的方法和我们在 `scoped_ptr` 那看到的很相似，只是把 `scoped_ptr` 换成了 `shared_ptr` 。

shared_ptr 与标准库容器

把对象直接存入容器中有时会有些麻烦。以值的方式保存对象意味着使用者将获得容器中的元素的拷贝，对于那些复制是一种昂贵的操作的类型来说可能会有性能的问题。此外，有些容器，特别是 `std::vector` ，当你加入元素时可能会复制所有元素，这更加重了性能的问题。最后，传值的语义意味着没有多态的行为。如果你需要在容器中存放多态的对象而且你不想切割它们，你必须用指针。如果你用裸指针，维护元素的完整性会非常复杂。从容器中删除元素时，你必须知道容器的使用者是否还在引用那些要删除的元素，不用担心多 个使用者使用同一个元素。这些问题都可以用 `shared_ptr` 来解决。

下面是如何把共享指针存入标准库容器的例子。

```
#include "boost/shared_ptr.hpp"
#include <vector>
#include <iostream>

class A {
public:
  virtual void sing()=0;
protected:
  virtual ~A() {};
};

class B : public A {
public:
  virtual void sing() {
    std::cout << "Do re mi fa so la";
  }
};

boost::shared_ptr<A> createA() {
  boost::shared_ptr<A> p(new B());
  return p;
}

int main() {
  typedef std::vector<boost::shared_ptr<A> > container_type;
  typedef container_type::iterator iterator;

  container_type container;
  for (int i=0;i<10;++i) {
    container.push_back(createA());
  }

  std::cout << "The choir is gathered: \n";
  iterator end=container.end();
  for (iterator it=container.begin();it!=end;++it) {
    (*it)->sing();
  }
}
```

这里有两个类，`A` 和 `B`，各有一个虚拟成员函数 `sing`。`B` 从 `A` 公有继承而来，并且如你所见，工厂函数 `createA` 返回一个动态分配的 `B` 的实例，包装在 `shared_ptr<A>` 里。在 `main` 里，一个包含 `shared_ptr<A>` 的 `std::vector` 被放入10个元素，最后对每个元素调用 `sing`。如果我们用裸指针作为元素，那些对象需要被手工删除。而在这个例子里，删除是自动的，因为在 `vector` 的生存期中，每个 `shared_ptr` 的引用计数都保持为1；当 `vector` 被销毁，所有引用计数器都将变为零，所有对象都被删除。有趣的是，即使 `A` 的析构函数没有声明为 `virtual`，`shared_ptr` 也会正确调用 `B` 的析构函数！

上面的例子示范了一个强有力的技术，它涉及 `A` 里面的protected析构函数。因为函数 `createA` 返回的是 `shared_ptr<A>`，因此不可能对 `shared_ptr::get` 返回的指针调用 `delete`。这意味着如果为了向某个需要裸指针的函数传送裸指针而从 `shared_ptr` 中取出裸指针的话，它不会由于意外地被删除而导致灾难。那么，又是如何允许 `shared_ptr` 删除它的对象的呢？这是因为指针指向的真正类型是 `B`；而 `B` 的析构函数不是protected的。这是非常有用的方法，用于给 `shared_ptr` 中的对象增加额外的安全性。

shared_ptr 与其它资源

有时你会发现你要把 `shared_ptr` 用于某个特别的类型，它需要其它清除操作而不是简单的 `delete`. `shared_ptr` 可以通过客户化删除器来支持这种需要。那些处理象 `FILE*` 这样的操作系统句柄的资源通常要使用象 `fclose` 这样的操作来释放。要在 `shared_ptr` 里使用 `FILE*`，我们要定义一个类来负责释放相应的资源。

```
class FileCloser {
public:
   void operator()(FILE* file) {
    std::cout << "The FileCloser has been called with a FILE*, "
      "which will now be closed.\n";
    if (file!=0)
      fclose(file);
  }
};
```

这是一个函数对象，我们用它来确保在资源要释放时调用 `fclose` 。下面是使用 `FileCloser` 类的示例程序。

```
int main() {
  std::cout <<
    "shared_ptr example with a custom deallocator.\n";
  {
    FILE* f=fopen("test.txt","r");
    if (f==0) {
      std::cout << "Unable to open file\n";
      throw "Unable to open file";
    }

    boost::shared_ptr<FILE>
      my_shared_file(f, FileCloser());

    // 定位文件指针
    fseek(my_shared_file.get(),42,SEEK_SET);
  }
  std::cout << "By now, the FILE has been closed!\n";
}
```

注意，在访问资源时，我们需要对 `shared_ptr` 使用 `&*` 用法, `get` , 或 `get_pointer` 。(请注意最好使用 `&*` . 另两个选择不太清晰) 这个例子还可以更简单，如果我们在释放资源时只需要调用一个单参数函数的话，就根本不需要创建一个客户化删除器类型。上面的例子可以重写如下：

```
{
  FILE* f=fopen("test.txt","r");
  if (f==0) {
    std::cout << "Unable to open file\n";
    throw file_exception();
  }

  boost::shared_ptr<FILE> my_shared_file(f,&fclose);

  // 定位文件指针
  fseek(&*my_shared_file,42,SEEK_SET);
}
std::cout << "By now, the FILE* has been closed!\n";
```

定制删除器在处理需要特殊释放程序的资源时非常有用。由于删除器不是 `shared_ptr` 类型的一部分，所以使用者不需要知道关于智能指针所拥有的资源的任何信息(当然除了如何使用它！)。例如，你可以使用对象池，定制删除器只需简单地把对象返还到池中。或者，一个 singleton 对象应该使用一个什么都不做的删除器。

使用定制删除器的安全性

我们已经看到对基类使用 protected 析构函数有助于增加使用 `shared_ptr` 的 类的安全性。另一个达到同样安全级别的方法是，声明析构函数为 protected (或 private) 并使用一个定制删除器来负责销毁对象。这个定制删除器必须是它要删除的类的友元，这样它才可以工作。封装这个删除器的好方法是把它实现为私有的嵌套类，如 下例所示：

```
#include "boost/shared_ptr.hpp"
#include <iostream>

class A {
  class deleter {
    public:
      void operator()(A* p) {
        delete p;
      }
  };
  friend class deleter;
public:

  virtual void sing() {
    std::cout << "Lalalalalalalalalalala";
  }

  static boost::shared_ptr<A> createA() {
    boost::shared_ptr<A> p(new A(),A::deleter());
    return p;
  }

protected:
  virtual ~A() {};
};

int main() {
  boost::shared_ptr<A> p=A::createA();
}
```

注意，我们在这里不能使用普通函数来作为 `shared_ptr<A>` 的工厂函数，因为嵌套的删除器是 `A` 私有的。使用这个方法，用户不可能在栈上创建 `A` 的对象，也不可能对 `A` 的指针调用 `delete` 。

从this创建shared_ptr

有时候，需要从 `this` 获得 `shared_ptr` ，即是说，你希望你的类被 `shared_ptr` 所管理，你需要把"自身"转换为 `shared_ptr` 的方法。看起来不可能？好的，解决方案来自于我们即将讨论的另一个智能指针 `boost::weak_ptr` 。`weak_ptr` 是 `shared_ptr` 的一个观察者；它只是安静地坐着并看着它们，但不会影响引用计数。通过存储一个指向 `this` 的 `weak_ptr` 作为类的成员，就可以在需要的时候获得一个指向 `this` 的 `shared_ptr` 。为了你可以不必编写代码来保

存一个指向 this 的 weak_ptr，接着又从 weak_ptr 获 shared_ptr 得，Boost.Smart_ptr 为这个任务提供了一个助手类，称为 enable_shared_from_this. 只要简单地让你的类公有地派生自 enable_shared_from_this，然后在需要访问管理 this 的 shared_ptr 时，使用函数 shared_from_this 就行了。下面的例子示范了如何使用 enable_shared_from_this ：

```
#include "boost/shared_ptr.hpp"
#include "boost/enable_shared_from_this.hpp"

class A;

void do_stuff(boost::shared_ptr<A> p) {
  ...
}

class A : public boost::enable_shared_from_this<A> {
public:
  void call_do_stuff() {
    do_stuff(shared_from_this());
  }
};

int main() {
  boost::shared_ptr<A> p(new A());
  p->call_do_stuff();
}
```

这个例子还示范了你要用 shared_ptr 管理 this 的情形。类 A 有一个成员函数 call_do_stuff 需要调用一个普通函数 do_stuff, 这个普通函数需要一个类型为 boost:: shared_ptr<A> 的参数。现在，在 A::call_do_stuff 里, this 不过是一个 A 指针, 但由于 A 派生自 enable_shared_from_this, 调用 shared_from_this 将返回我们所要的 shared_ptr 。在 enable_shared_from_this 的成员函数 shared_from_this 里，内部存储的 weak_ptr 被转换为 shared_ptr, 从而增加了相应的引用计数，以确保相应的对象不会被删除。

总结

引用计数智能指针是非常重要的工具。Boost的 shared_ptr 提供了坚固而灵活的解决方案，它已被广泛用于多种环境下。需要在使用者之间共享对象是常见的，而且通常没有办法通知使用者何时删除对象是安全的。 shared_ptr 让使用者无需知道也在使用共享对象的其它对象，并让它们 无需担心在没有对象引用时的资源释放。这对于Boost的智能指针类而言是最重要的。你会看到Boost.Smart_ptr中还有其它的智能指针，但这 一个肯定是你最想要的。通过使用定制删除器，几乎所有资源类型都可以存入 shared_ptr 。这使得 shared_ptr 成为处理资源管理的通用类，而不仅仅是处理动态分配对象。与裸指针相比， shared_ptr 会有一点点额外的空间代价。我还没有发现由于这些代价太大而需要另外寻找一个解决方案的情形。不要去创建你自己的引用计数智能指针类。没有比使用 shared_ptr 智能指针更好的了。

在以下情况时使用 shared_ptr ：

- 当有多个使用者使用同一个对象，而没有一个明显的拥有者时

- 当要把指针存入标准库容器时
- 当要传送对象到库或从库获取对象，而没有明确的所有权时
- 当管理一些需要特殊清除方式的资源时[9]

  > [9] 通过定制删除器的帮助。

shared_array

头文件: `"boost/shared_array.hpp"`

`shared_array` 用于共享数组所有权的智能指针。它与 `shared_ptr` 的关系就如 `scoped_array` 与 `scoped_ptr` 的关系。`shared_array` 与 `shared_ptr` 的不同之处主要在于它是用于数组的而不是用于单个对象的。在我们讨论 scoped_array时，我提到过通常 `std::vector` 是一个更好的选择。但 `shared_array` 比 `vector` 更有价值，因为它提供了对数组所有权的共享。`shared_array` 的接口与 `shared_ptr` 非常相似，差别仅在于增加了一个下标操作符，以及不支持定制删除器。

由于一个指向 `std::vector` 的 `shared_ptr` 提供了比shared_array更多的灵活性，所以我们就不对shared_array的用法进行讨论了。如果你发现自己需要 `boost::shared_array`，可以参考一下在线文档。

intrusive_ptr

头文件: `"boost/intrusive_ptr.hpp"`

`intrusive_ptr` 是 `shared_ptr` 的插入式版本。有时我们必须使用插入式的引用计数智能指针。典型的情况是对于那些已经写好了内部引用计数器的代码，而我们又没有时间去重写它(或者已经不能获得那些代码了)。另一种情况是要求智能指针的大小必须与裸指针大小严格相等，或者 `shared_ptr` 的引用计数器分配严重影响了程序的性能(我可以肯定这是非常罕见的情况！)。从功能的观点来看，唯一需要插入式智能指针的情况是，被指类的某个成员函数需要返回 `this`，以便它可以用于另一个智能指针(事实上，也有办法使用非插入式智能指针来解决这个问题，正如我们在本章前面看到的)。 `intrusive_ptr` 不同于其它智能指针，因为它要求你来提供它所要的引用计数器。

当 `intrusive_ptr` 递增或递减一个非空指针上的引用计数时，它是通过分别调用函数 `intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 来完成的。这两个函数负责确保引用计数的正确性，并且负责在引用计数降为零时删除指针。因此，你必须为你的类重载这两个函数，正如我们后面将看到的。

以下是 `intrusive_ptr` 的部分摘要，只列出了最重要的函数。

```
namespace boost {

  template<class T> class intrusive_ptr {
  public:
    intrusive_ptr(T* p,bool add_ref=true);

    intrusive_ptr(const intrusive_ptr& r);

    ~intrusive_ptr();

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

    operator unspecified-bool-type() const;
  };

  template <class T> T* get_pointer(const intrusive_ptr<T>& p);

  template <class T,class U> intrusive_ptr<T>
    static_pointer_cast(const intrusive_ptr<U>& r);
}
```

成员函数

```
intrusive_ptr(T* p,bool add_ref=true);
```

这个构造函数将指针 `p` 保存到 `*this` 中。如果 `p` 非空，并且 `add_ref` 为 `true`，构造函数将调用 `intrusive_ptr_add_ref(p)`. 如果 `add_ref` 为 `false`，构造函数则不调用 `intrusive_ptr_add_ref`. 如果 `intrusive_ptr_add_ref` 会抛出异常，则构造函数也会。

```
intrusive_ptr(const intrusive_ptr& r);
```

该复制构造函数保存一份 `r.get()` 的拷贝，并且如果指空非空则用它调用 `intrusive_ptr_add_ref` 。这个构造函数不会抛出异常。

```
~intrusive_ptr();
```

如果保存的指针为非空，则 `intrusive_ptr` 的析构函数会以保存的指针为参数调用函数 `intrusive_ptr_release`。 `intrusive_ptr_release` 负责递减引用计数并在计数为零时删除指针。这个函数不会抛出异常。

```
T& operator*() const;
```

解引用操作符返回所存指针的解引用。如果所存指针为空则会导致未定义行为。你应该确认 `intrusive_ptr` 有一个非空的指针，这可以用函数 `get` 实现，或者在Boolean上下文中测试 `intrusive_ptr` 。解引用操作符不会抛出异常。

```
T* operator->() const;
```

这个操作符返回保存的指针。在引用的指针为空时调用这个操作符会有未定义行为。这个操作符不会抛出异常。

```
T* get() const;
```

这个成员函数返回保存的指针。它可用于你需要一个裸指针的时候，即使保存的指针为空也可以调用。这个函数不会抛出异常。

```
operator unspecified-bool-type() const;
```

这个类型转换函数返回一个可用于布尔表达式的类型，而它绝对不是 `operator bool`，因为那样会允许一些必须要禁止的操作。这个转换允许 `intrusive_ptr` 在一个布尔上下文中被测试，例如， `if (p)`，`p` 是一个 `intrusive_ptr`. 这个转换函数当 `intrusive_ptr` 引向一个非空指针时返回 `True`；否则返回 `false`. 这个转换函数不会抛出异常。

普通函数

```
template <class T> T* get_pointer(const intrusive_ptr<T>& p);
```

这个函数返回 `p.get()`，它主要用于支持泛型编程。[10] 它也可以用作替代成员函数 `get`，因为它可以重载为可以与裸指针或第三方智能指针类一起工作。有些人宁愿用普通函数而不用成员函数。[11] 这个函数不会抛出异常。

> [10] 这种函数被称为 shims. 见参考书目的 [12]。
>
> [11]这种想法是出于以下原因，使用智能指针的成员函数时，很难分清它是操作智能指针还是操作它所指向的对象。例如，`p.get()` 和 `p->get()` 有完全不同的意思，不认真看还很难区别，而 `get_pointer(p)` 和 `p->get()` 则一看就知道不一样。对于你来说这是不是问题，主要取决于你的感觉和经验。

```
template <class T,class U>
  intrusive_ptr<T> static_pointer_cast(const intrusive_ptr<U>& r);
```

这个函数返回 `intrusive_ptr<T>(static_cast<T*>(r.get()))`. 和 `shared_ptr` 不一样，你可以对保存在 `intrusive_ptr` 中的对象指针安全地使用 `static_cast`。但是你可能出于对智能指针类型转换的用法一致性而想使用这个函数。`static_pointer_cast` 不会抛出异常。

用法

使用 `intrusive_ptr` 与使用 `shared_ptr` 相比，有两个主要的不同之处。第一个是你需要提供引用计数的机制。第二个是把 `this` 当成智能指针是合法的[12]，正如我们即将看到的，有时候这样很方便。注意，在多数情况下，应该使用非插入式的 `shared_ptr`.

> [12] 你不能用 `shared_ptr` 来做到这一点，如果没有进行特殊处理的话，如 `enable_shared_from_this`.

要使用 `boost::intrusive_ptr`，要包含 `"boost/intrusive_ptr.hpp"` 并定义两个普通函数 `intrusive_ptr_add_ref` 和 `intrusive_ptr_release`. 它们都要接受一个参数，即指向你要使用 `intrusive_ptr` 的类型的指针。这两个函数的返回值被忽略。通常的做法是，泛化这两个函数，简单地调用被管理类型的成员函数去完成工作(例如，调用 `add_ref` 和 `release`)。如果引用计数降为零，`intrusive_ptr_release` 应该负责释放资源。以下是你应该如何实现这两个泛型函数的示范：

```
template <typename T> void intrusive_ptr_add_ref(T* t) {
  t->add_ref();
}

template <typename T> void intrusive_ptr_release(T* t) {
  if (t->release()<=0)
    delete t;
}
```

注意，这两个函数应该定义在它们的参数类型所在的作用域内。这意味着如果这个函数接受的参数类型来自 于一个名字空间，则函数也必须定义在那里。这样做的原因是，函数的调用是非受限的，即允许采用参数相关查找，而如果有多个版本的函数被提供，那么全部名字 空间肯定不是放置它们的好地方。我们稍后将看到一个关于如何放置它们的例子，但首先，我们需要提供某类的引用计数器。

提供一个引用计数器

现在管理用的函数已经定义了，我们必须要提供一个内部的引用计数器了。在本例中，引用计数是一个初始化为零的私有数据成员，我们将公开 `add_ref` 和 `release` 成员函数来操作它。 `add_ref` 递增引用计数而 `release` 递减它[13]。 我们可以增加一个返回引用计数当前值的成员函数，但 `release` 也可以做到这一点。下面的基类， `reference_counter` , 提供了一个计数器以及 `add_ref` 和 `release` 成员函数，我们可以简单地用继承来为一个类增加引用计数了。

> [13] 注意，在多线程环境下，对保持引用计数的变量的任何操作都必须同步化。

```
class reference_counter {
  int ref_count_;
  public:
    reference_counter() : ref_count_(0) {}

    virtual ~reference_counter() {}

    void add_ref() {
      ++ref_count_;
    }

    int release() {
      return --ref_count_;
    }

  protected:
    reference_counter& operator=(const reference_counter&) {
    // 无操作
      return *this;
    }
  private:
    // 禁止复制构造函数
    reference_counter(const reference_counter&);
};
```

把 `reference_counter` 的析构函数声明为虚拟的原因是这个类将被公开继承，有可能会使用一个 `reference_counter` 指针来 `delete` 派生类。我们希望删除操作能够正确地调用派生类的析构函数。实现非常简单： `add_ref` 递增引用计数， `release` 递减引用计数并返回它。要使用这个引用计数，要做的就是公共地继承它。以下是一个类 `some_ class` ， 包含一个内部引用计数，并使用 `intrusive_ptr` 。

```
#include <iostream>
#include "boost/intrusive_ptr.hpp"

class some_class : public reference_counter {
public:
  some_class() {
    std::cout << "some_class::some_class()\n";
  }

  some_class(const some_class& other) {
    std::cout << "some_class(const some_class& other)\n";
  }

  ~some_class() {
    std::cout << "some_class::~some_class()\n";
  }
};

int main() {
  std::cout << "Before start of scope\n";
  {
    boost::intrusive_ptr<some_class> p1(new some_class());
    boost::intrusive_ptr<some_class> p2(p1);
  }
  std::cout << "After end of scope \n";
}
```

为了显示 `intrusive_ptr` 以及函数 `intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 都正确无误，以下是这个程序的运行输出：

```
Before start of scope
some_class::some_class()
some_class::~some_class()
After end of scope
```

`intrusive_ptr` 为我们打点一切。当第一个 `intrusive_ptr p1` 创建时，它传送了一个 `some_class` 的新实例。 `intrusive_ptr` 构造函数实际上有两个参数，第二个是一个 `bool` ，表示是否要调用 `intrusive_ptr_add_ref` 。由于这个参数的缺省值是 `True` , 所以在构造 `p1` 时， `some_class` 实例的引用计数变为1。然后，第二个 `intrusive_ptr` , `p2` 初构造。它是从 `p1` 复制构造的，当 `p2` 看到 `p1` 是引向一个非空指针时，它调用 `intrusive_ptr_add_ref` . 引用计数变为2。然后，两个 `intrusive_ptr` 都离开作用域了。首先， `p2` 被销毁，析构函数调用 `intrusive_ptr_release` . 它把引用计数减为1。然后， `p1` 被销毁，析构函数再次调用 `intrusive_ptr_release` ，导致引用计数降为0；这使得我们的 `intrusive_ptr_release` 去 `delete` 该指针。你可能注意到 `reference_counter` 的实现不是线程安全的，因此不能用于多线程应用，除非加上同步化。

比起依赖于 `intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 的泛型实现，我们最好有一些直接操作基类(在这里是 `reference_counter`)的函数。这样做的优点在于，即使从 `reference_counter` 派生的类定义在其它的名字空间， `intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 也还可以通过ADL (参数相关查找法)找到它们。修改 `reference_counter` 的实现很简单。

```
class reference_counter {
  int ref_count_;
  public:
    reference_counter() : ref_count_(0) {}

    virtual ~reference_counter() {}

     friend void intrusive_ptr_add_ref(reference_counter* p) {
       ++p->ref_count_;
     }

     friend void intrusive_ptr_release(reference_counter* p) {
       if (--p->ref_count_==0)
         delete p;
     }

  protected:
    reference_counter& operator=(const reference_counter&) {
    // 无操作
      return *this;
    }
  private:
    // 禁止复制构造函数
    reference_counter(const reference_counter&);
};
```

把 **this** 用作智能指针

总的来说，提出一定要用插入式引用计数智能指针的情形是不容易的。大多数情况下，但不是全部情况下，非插入式智能指针都可以解决问题。但是，有一种情形使用插入式引用计数会更容易：当你需要从一个成员函数返回 `this`，并把它存入另一个智能指针。当从一个被非插入式智能指针所拥有的类型返回 `this` 时，结果是有两个不同的智能指针认为它们拥有同一个对象，这意味着它们会在某个时候一起试图删除同一个指针。这导致了两次删除，结果可能使你的应用程序崩溃。必须有什么办法可以通知另一个智能指针，这个资源已经被一个智能指针所引用，这正好是内部引用计数器(暗地里)可以做到的。由于 intrusive_ptr 的逻辑不直接对它们所引向的对象的内部引用计数进行操作，这就不会违反所有权或引用计数的完整性。引用计数只是被简单地递增。

让我们先看一下一个依赖于 `boost::shared_ptr` 来共享资源所有权的实现中潜在的问题。它基于本章前面讨论 `enable_shared_from_this` 时的例子。

```
#include "boost/shared_ptr.hpp"

class A;

void do_stuff(boost::shared_ptr<A> p) {
  // ...
}

class A {
public:
  call_do_stuff() {
   shared_ptr<A> p(???);
    do_stuff(p);
  }
};

int main() {
  boost::shared_ptr<A> p(new A());
  p->call_do_stuff();
}
```

类 `A` 要调用函数 `do_stuff`，但问题是 `do_stuff` 要一个 `shared_ptr<A>`，而不是一个普通的 `A` 指针。因此，在 `A::call_do_stuff` 里，应该如何创建 `shared_ptr` ？现在，让我们重写 `A` ，让它兼容于 `intrusive_ptr`，通过从 `reference_counter` 派生，然后我们再增加一个 `do_stuff` 的重载版本，接受一个 `intrusive_ptr<A>` 类型的参数。

```
#include "boost/intrusive_ptr.hpp"

class A;

void do_stuff(boost::intrusive_ptr<A> p) {
  // ...
}

void do_stuff(boost::shared_ptr<A> p) {
  // ...
}

class A : public reference_counter {
public:
  void call_do_stuff() {
    do_stuff(this);
  }
};

int main() {
  boost::intrusive_ptr<A> p(new A());
  p->call_do_stuff();
}
```

如你所见，在这个版本的 `A::call_do_stuff` 里，我们可以直接把 this 传给需要一个 `intrusive_ptr<A>` 的函数，这是由于 `intrusive_ptr` 的类型转换构造函数。

最后，这里有一个特别的地方：现在 `A` 可以支持 `intrusive_ptr` 了，我们也可以创建一个包装 `intrusive_ptr` 的 `shared_ptr`，这们我们就可以调用原来版本的 `do_stuff`，它需要一个 `shared_ptr<A>` 作为参数。假如你不能控制 `do_stuff` 的源码，这可能是你要解决的一个非常真实的问题。这次，还是用定制删除器的方法来解决，它需要调用 `intrusive_ptr_release`. 下面是一个新版本的 `A::call_do_stuff`.

```
void call_do_stuff() {
  intrusive_ptr_add_ref(this);
  boost::shared_ptr<A> p(this,&intrusive_ptr_release<A>);
  do_stuff(p);
}
```

真是一个漂亮的方法。当没有 `shared_ptr` 剩下时，定制的删除器被调用，它调用 `intrusive_ptr_release`，递减 `A` 的内部引用计数。注意，如果 `intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 被实现为直接操作 `reference_counter`，你就要这样来创建 `shared_ptr` ：

```
boost::shared_ptr<A> p(this,&intrusive_ptr_release);
```

支持不同的引用计数器

我们前面提过可以为不同的类型支持不同的引用计数。这在集成已有的采用不同引用计数机制的类时是有必要的(例如，第三方的类使用它们自己版本的引用计数器)。又或者对于资源的释放有不同的需求，如调用 `delete` 以外的另一个函数。如前所述，对 `intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 的调用是非受限的。这意味着在名字查找时要考虑参数(指针的类型)的作用域，从而这些函数应该与它们操作的类型定义在同一个作用域。如果你在全局名字空间里实现 `intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 的泛型版本，你就不能在其它名字空间中再创建泛型版本了。例如，如果一个名字空间需要为它的所有类型定义一个特殊的版本，特化版本或重载版本必须提供给每 一个类型。否则，全局名字空间中的函数就会引起歧义。因此在全局名字空间中提供泛型版本不是一个好主意，而在其它名字空间中提供则可以。

既然我们已经用基类 `reference_counter` 实现了引用计数器，那么在全局名字空间中提供一个接受 `reference_counter*` 类型的参数的普通函数应该是一个好主意。这还可以让我们在其它名字空间中提供泛型重载版本而不会引起歧义。例如，考虑 `my_namespace` 名字空间中的两个类 `another_class` 和 `derived_class` ：

```
namespace my_namespace {
  class another_class : public reference_counter {
  public:
    void call_before_destruction() const {
      std::cout <<
        "Yes, I'm ready before destruction\n";
    }
  };

  class derived_class : public another_class {};

   template <typename T> void intrusive_ptr_add_ref(T* t) {
     t->add_ref();
   }

  template <typename T> void intrusive_ptr_release(T* t) {
    if (t->release()<=0) {
      t->call_before_destruction();
      delete t;
    }
  }
}
```

这里，我们实现了 `intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 的泛型版本。因此我们必须删掉在全局名字空间中的泛型版本，把它们替换为以一个 `reference_counter` 指针为参数的非模板版本。或者，我们干脆从全局名字空间中删掉这些函数，也可以避免引起混乱。对于这两个类 `my_namespace::another_class` 和 `my_namespace::derived_class`，将调用这个特殊版本(那个调用了它的参数的成员函数 `call_before_destruction` 的版本)。其它类型或者在它们定义所在的名字空间中有相应的函数，或者使用全局名字空间中的版本，如果有的话。下面程序示范了这如何工作：

```
int main() {
  boost::intrusive_ptr<my_namespace::another_class>
    p1(new my_namespace::another_class());
  boost::intrusive_ptr<A>
    p2(new good_class());
  boost::intrusive_ptr<my_namespace::derived_class>
    p3(new my_namespace::derived_class());
}
```

首先，`intrusive_ptr p1` 被传入一个新的 `my_namespace::another_class` 实例。在解析对 `intrusive_ptr_add_ref` 的调用时，编译器会找到 `my_namespace` 里的版本，即 `my_namespace::another_class*` 参数所在名字空间。因而，为那个名字空间里的类型所提供的泛型函数会被正确地调用。在查找 `intrusive_ptr_release` 时也是同样。然后，`intrusive_ptr p2` 被创建并被传入一个类型 `A` (我们早前创建的那个类型)的指针。那个类型是在全局名字空间里的，所以当编译器试图去找到函数 `intrusive_ptr_add_ref` 的最佳匹配时，它只会找到一个版本，即接受 `reference_counter` 指针类型的那个版本(你应该记得我们已经从全局名字空间中删掉了泛型版本)。因为 `A` 公共继承自 `reference_counter`，通过隐式类型转换就可以进行正确的调用。最后，`my_namespace` 里的泛型版本被用于类 `my_namespace::derived_class`；这与 `another_class` 例子中的查找是一样的。

这里最重要的教训是，在实现函数 `intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 时，它们应该总是定义在它们操作的类型所在的名字空间里。从设计的角度来看，这也是完美的，把相关的东西放在一起，这有助于确保总是调用正确的版本，而不用担心是否有多个不同的实现可供选择。

总结

在多数情况下，你不应该使用 `boost::intrusive_ptr`，因为共享所有权的功能已在 `boost::shared_ptr` 中提供，而且非插入式智能指针比插入式智能指针更灵活。但是，有时候也会需要插入式的引用计数，可能是由于旧的代码，或者是为了与第三方的类进行集成。当有这种需要时，可以用 `intrusive_ptr`，它具有与其它Boost智能指针相同的语义。如果你使用过其它的Boost智能指针，你就会发现不论是否插入式的，所有智能指针都有一致的接口。使用 `intrusive_ptr` 的类必须可以提供引用计数。 `ntrusive_ptr` 通过调用两个函数，`intrusive_ptr_add_ref` 和 `intrusive_ptr_release` 来管理引用计数；这两个函数必须正确地操作插入式的引用计数，以保证 `intrusive_ptr` 正确工作。在使用 `intrusive_ptr` 的类中已经内置有引用计数的情况下，实现对 `intrusive_ptr` 的支持就是实现这两个函数。有些情况下，可以创建这两个函数的参数化版本，然后对所有带插入式引用计数的类型使用相同的实现。多数时候，声明这两个函数的最好的地方就是它们所支持的类型所在的名字空间。

在以下情况时使用 `intrusive_ptr` ：

- 你需要把 `this` 当作智能指针来使用。
- 已有代码使用或提供了插入式的引用计数。
- 智能指针的大小必须与裸指针的大小相等。

weak_ptr

头文件: `"boost/weak_ptr.hpp"`

`weak_ptr` 是 `shared_ptr` 的观察员。它不会干扰 `shared_ptr` 所共享的所有权。当一个被 `weak_ptr` 所观察的 `shared_ptr` 要释放它的资源时，它会把相关的 `weak_ptr` 的指针设为空。这防止了 `weak_ptr` 持有悬空的指针。你为什么会需要 `weak_ptr`？许多情况下，你需要旁观或使用一个共享资源，但不接受所有权，如为了防止递归的依赖关系，你就要旁观一个共享资源而不能拥有所有权，或者为了避免悬空指针。可以从一个 `weak_ptr` 构造一个 `shared_ptr`，从而取得对共享资源的访问权。

以下是 `weak_ptr` 的部分定义，列出并简要介绍了最重要的函数。

```
namespace boost {

  template<typename T> class weak_ptr {
  public:
    template <typename Y>
      weak_ptr(const shared_ptr<Y>& r);

    weak_ptr(const weak_ptr& r);

    ~weak_ptr();

    T* get() const;
    bool expired() const;
    shared_ptr<T> lock() const;
  };
}
```

成员函数

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

这个构造函数从一个 `shared_ptr` 创建 `weak_ptr`，要求可以从 `Y*` 隐式转换为 `T*`. 新的 `weak_ptr` 被配置为旁观 `r` 所引向的资源。`r` 的引用计数不会有所改变。这意味着 `r` 所引向的资源在被删除时不会理睬是否有 `weak_ptr` 引向它。这个构造函数不会抛出异常。

```
weak_ptr(const weak_ptr& r);
```

这个复制构造函数让新建的 `weak_ptr` 旁观与 `weak_ptr r` <small class="calibre23">相关的 shared_ptr</small>所引向的资源。<small class="calibre23">shared_ptr</small>的引用计数保持不变。这个构造函数不会抛出异常。

```
~weak_ptr();
```

`weak_ptr` 的析构函数，和构造函数一样，它不改变引用计数。如果需要，析构函数会把 `*this` 与共享资源脱离开。这个析构函数不会抛出异常。

```
bool expired() const;
```

如果所观察的资源已经"过期"，即资源已被释放，则返回 `True` 。如果保存的指针为非空， `expired` 返回 `false` . 这个函数不会抛出异常。

```
shared_ptr<T> lock() const
```

返回一个引向 `weak_ptr` 所观察的资源的 `shared_ptr` ，如果可以的话。如果没有这样指针(即 `weak_ptr` 引向的是空指针)， `shared_ptr` 也将引向空指针。否则， `shared_ptr` 所引向的资源的引用计数将正常地递增。这个函数不会抛出异常。

用法

我们从一个示范 `weak_ptr` 的基本用法的例子开始，尤其要看看它是如何不影响引用计数的。这个例子里也包含了 `shared_ptr` ，因为 `weak_ptr` 总是需要和 `shared_ptr` 一起使用的。使用 `weak_ptr` 要包含头文件 `"boost/weak_ptr.hpp"` .

```
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"
#include <iostream>
#include <cassert>

class A {};

int main() {

  boost::weak_ptr<A> w;
  assert(w.expired());
  {
    boost::shared_ptr<A> p(new A());
    assert(p.use_count()==1);
    w=p;
    assert(p.use_count()==w.use_count());
    assert(p.use_count()==1);

    // 从weak_ptr创建shared_ptr
    boost::shared_ptr<A> p2(w);
    assert(p2==p);
  }
  assert(w.expired());
  boost::shared_ptr<A> p3=w.lock();
  assert(!p3);
}
```

`weak_ptr w` 被缺省构造，意味着它初始时不旁观任何资源。要检测一个 `weak_ptr` 是否在旁观一个活的对象，你可以使用函数 `expired` . 要开始旁观， `weak_ptr` 必须要被赋值一个 `shared_ptr` . 本例中， `shared_ptr p` 被赋值给 `weak_ptr w` , 这等于说 `p` 和 `w` 的引用计数应该是相同的。然后，再从 `weak_ptr` 构造一个 `shared_ptr` ，这是一种从 `weak_ptr` 那里获得

对共享资源的访问权的方法。如果在构造 `shared_ptr` 时， `weak_ptr` 已经过期了，将从 `shared_ptr` 的构造函数里抛出一个 `boost::bad_weak_ptr` 类型的异常。再继续，当 `shared_ptr p` 离开作用域， `w` 就变成过期的了。当调用它的成员函数 `lock` 来获得一个 `shared_ptr` 时，这是另一种获得对共享资源访问权的方法，将返回一个空的 `shared_ptr` 。注意，从这个程序的开始到结束， `weak_ptr` 都没有影响到共享对象的引用计数的值。

与其它智能指针不同的是， `weak_ptr` 不对它所观察的指针提供重载的 `operator*` 和 `operator->` . 原因是对 `weak_ptr` 所观察的资源的任何操作都必须是明显的，这样才安全；由于不会影响它们所观察的共享资源的引用计数器，所以真的很容易就会不小心访问到一个无效的指针。这就是为什么你必须要传送 `weak_ptr` 给 `shared_ptr` 的构造函数，或者通过调用 `weak_ptr::lock` 来获得一个 `shared_ptr` 。这两种方法都会使引用计数增加，这样在 `shared_ptr` 从 `weak_ptr` 创建以后，它可以保证共享资源的生存，确保在我们要使用它的时候它不会被释放掉。

常见问题

由于在智能指针中保存的是指针的值而不是它们所指向的指针的值，因此在标准库容器中使用智能指针有一个常见的问题，就是如何在算法中使用智能指针；算法通常需要访问实际对象的值，而不是它们的地址。例如，你如何调用 `std::sort` 并正确地排序？实际上，这个问题与在容器中保存并操作普通指针是几乎一样的，但事实很容易被忽略(可能是由于我们总是避免在容器中保存裸指针)。当然我们 不能直接比较两个智能指针的值，但也很容易解决。只要用一个解引用智能指针的谓词就可以了，所以我们将创建一个可重用的谓词，使得可以在标准库的算法里使 用引向智能指针的迭代器，这里我们选用的智能指针是 `weak_ptr` 。

```
#include <functional>
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"

template <typename Func, typename T>
  struct weak_ptr_unary_t :
    public std::unary_function<boost::weak_ptr<T>,bool> {
  T t_;
  Func func_;

  weak_ptr_unary_t(const Func& func,const T& t)
    : t_(t),func_(func) {}

  bool operator()(boost::weak_ptr<T> arg) const {
    boost::shared_ptr<T> sp=arg.lock();
    if (!sp) {
      return false;
    }
    return func_(*sp,t_);
  }
};

template <typename Func, typename T> weak_ptr_unary_t<Func,T>
  weak_ptr_unary(const Func& func, const T& value) {
    return weak_ptr_unary_t<Func,T>(func,value);
}
```

`weak_ptr_unary_t` 函数对象对要调用的函数以及函数所用的参数类型进行了参数化。把要调用的函数保存在函数对象中使用使得这个函数对象很容易使用，很快我们就能看到这一点。为了使这个谓词兼容于标准库的适配器，`weak_ptr_unary_t` 要从 `std::unary_function` 派生，后者保证了所有需要的 `typedefs` 都能提供(这些要求是为了让标准库的适配器可以这些函数对象一起工作)。实际的工作在调用操作符函数中完成，从 `weak_ptr` 创建一个 `shared_ptr` 。必须要确保在函数调用时资源是可用的。然后才可以调用指定的函数(或函数对象)，传入本次调用的参数(要解引用以获得真正的资源) 和在对象中保存的值，这个值是在构造 `weak_ptr_unary_t` 时给定的。这个简单的函数对象现在可以用于任意可用的算法了。为方便起见，我们还定义了一个助手函数，`weak_ptr_unary`，它可以推出参数的类型并返回一个适当的函数对象[14]。我们来看看如何使用它。

> [14] 要使得这个类型更通用，还需要更多的设计。

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

#include <vector>
#include <algorithm>
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"

int main() {
  using std::string;
  using std::vector;
  using boost::shared_ptr;
  using boost::weak_ptr;

  vector<weak_ptr<string> > vec;

  shared_ptr<string> sp1(
    new string("An example"));
  shared_ptr<string> sp2(
    new string("of using"));
  shared_ptr<string> sp3(
    new string("smart pointers and predicates"));
  vec.push_back(weak_ptr<string>(sp1));
  vec.push_back(weak_ptr<string>(sp2));
  vec.push_back(weak_ptr<string>(sp3));

  vector<weak_ptr<string> >::iterator
    it=std::find_if(vec.begin(),vec.end(),
     weak_ptr_unary(std::equal_to<string>(),string("of using")));

  if (it!=vec.end()) {
    shared_ptr<string> sp(*++it);
    std::cout << *sp << '\n';
  }
}
```

本例中，创建了一个包含 `weak_ptr` 的 `vector` 。最有趣的一行代码(是的，它有点长)就是我们为使用 `find_if` 算法而创建 `weak_ptr_unary_t` 的那行。

```
vector<weak_ptr<string> >::iterator it=std::find_if(
  vec.begin(),
  vec.end(),
  weak_ptr_unary(
    std::equal_to<string>(),string("of using")));
```

通过把另一个函数对象，`std::equal_to`，和一个用于匹配的`string`一起传给助手函数`weak_ptr_unary`，创建了一个新的函数对象。由于`weak_ptr_unary_t`完全兼容于各种适配器(由于它是从`std::unary_function`派生而来的)，我们可以再从它组合出各种各样的函数对象。例如，我们也可以查找第一个不匹配`"of using"`的串：

```
vector<weak_ptr<string> >::iterator it=std::find_if(
  vec.begin(),
  vec.end(),
std::not1(
    weak_ptr_unary(
      std::equal_to<string>(),string("of using"))));
```

Boost智能指针是专门为了与标准库配合工作而设计的。我们可以创建有用的组件来帮助我们可以更简单地使用这些强大的智能指针。象`weak_ptr_unary`这样的工具并不是经常要用到的；有一个库提供了比`weak_ptr_unary`更好用的泛型绑定器[15]。弄懂这些智能指针的语义，可以让我们更清楚地使用它们。

> <a>[15] 指Boost.Bind库。

两种从**weak_ptr**创建**shared_ptr**的惯用法

如你所见，如果你有一个旁观某种资源的`weak_ptr`，你最终还是会想要访问这个资源。为此，`weak_ptr`必须被转换为`shared_ptr`，因为`weak_ptr`是不允许访问资源的。有两种方法可以从`weak_ptr`创建`shared_ptr`：把`weak_ptr`传递给`shared_ptr`的构造函数，或者调用`weak_ptr`的成员函数`lock`，它返回`shared_ptr`。选择哪一个取决于你认为一个空的`weak_ptr`是错误的抑或不是。`shared_ptr`构造函数在接受一个空的`weak_ptr`参数时会抛出一个`bad_weak_ptr`类型的异常。因此应该在你认为空的`weak_ptr`是一种错误时使用它。如果使用`weak_ptr`的函数`lock`，它会在`weak_ptr`为空时返回一个空的`shared_ptr`。这在你想测试一个资源是否有效时是正确的，一个空的`weak_ptr`是预期中的。此外，如果使用`lock`，那么使用资源的正确方法应该是初始化并同时测试它，如下：

```
#include <iostream>
#include <string>
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"

int main() {
  boost::shared_ptr<std::string>
    sp(new std::string("Some resource"));
  boost::weak_ptr<std::string> wp(sp);
  // ...
  if (boost::shared_ptr<std::string> p=wp.lock())
    std::cout << "Got it: " << *p << '\n';
  else
    std::cout << "Nah, the shared_ptr is empty\n";
}
```

如你所见，shared_ptr `p` 被 `weak_ptr wp` 的 `lock` 函数的结果初始化。然后 `p` 被测试，只有当它非空时资源才能被访问。由于 `shared_ptr` 仅在这个作用域中有效，所以在这个作用域之外不会有机会让你不小心用到它。另一种情形是当 `weak_ptr` 逻辑上必须非空的时候。那种情形下，不需要测试 `shared_ptr` 是否为空，因为 `shared_ptr` 的构造函数会在接受一个空 `weak_ptr` 时抛出异常，如下：

```
#include <iostream>
#include <string>
#include "boost/shared_ptr.hpp"
#include "boost/weak_ptr.hpp"

void access_the_resource(boost::weak_ptr<std::string> wp) {
  boost::shared_ptr<std::string> sp(wp);
  std::cout << *sp << '\n';
}

int main() {
  boost::shared_ptr<std::string>
    sp(new std::string("Some resource"));
  boost::weak_ptr<std::string> wp(sp);
  // ...
  access_the_resource(wp);
}
```

在这个例子中，函数 `access_the_resource` 从一个 `weak_ptr` 构造 `shared_ptr sp` 。这时不需要测试 `shared_ptr` 是否为空，因为如果 `weak_ptr` 为空，将会抛出一个 `bad_weak_ptr` 类型的异常，因此函数会立即结束；错误会在适当的时候被捕获和处理。这样做比显式地测试 `shared_ptr` 是否为空然后返回要更好。这就是从 `weak_ptr` 获得 `shared_ptr` 的两种方法。

总结

`weak_ptr` 是Boost智能指针拼图的最后一块。`weak_ptr` 概念是 `shared_ptr` 的一个重要伙伴。它允许我们打破递归的依赖关系。它还处理了关于悬空指针的一个常见问题。在共享一个资源时，它常用于那些不参与生存期管理的资源用户。这种情况不能使用裸指针，因为在最后一个 `shared_ptr` 被销毁时，它会释放掉共享的资源。如果使用裸指针来引用资源，将无法知道资源是否仍然存在。如果资源已经不存在，访问它将会引起灾难。通过使用 `weak_ptr`，关于共享资源已被销毁的信息会传播给所有旁观的 `weak_ptr` s，这意味着不会发生无意间访问到无效指针的情形。这就象是观察员模式(Observer pattern)的一个特例；当资源被销毁，所有表示对此感兴趣的都会被通知到。

对于以下情形使用 `weak_ptr` ：

- 要打破递归的依赖关系
- 使用一个共享的资源而不需要共享所有权
- 避免悬空的指针

Smart_ptr总结

本章介绍了Boost的智能指针，它们是对C++社区的贡献，无论怎样评价都不过份。对于一个成功的 智能指针库，它必须考虑到并正确地处理大量的细节因素。我可以肯定你曾经见过很多种智能指针，你也可能曾经参与过编写它们，因此你应该知道做好这件事所要 花费的努力。没有其它的智能指针可以和它们一样智能，因此Boost.Smart_ptr库具有很高的价值。

作为软件工程中的重要组成部分，Boost的智能指针明显受到了广泛的关注和彻底的审查。因此很难列出所有的贡献者。很多人给出了有价值的意见和对当前的智能指针库进行了修正。这里列出一些突出的人员及其贡献：

- Greg Colvin, `auto_ptr` 之父, 还提出了 `counted_ptr` , 最后成为现在的 `shared_ptr` .
- Beman Dawes 重新激活了对智能指针的讨论，并提议了Greg Colvin原先建议的语义。
- Peter Dimov 重新设计了智能指针类，增加线程安全， `intrusive_ptr` , 以及 `weak_ptr` .

如此著名的概念不断地在发展，这是很吸引人的。毫无疑问，智能指针或者说智能资源的领域还会有更进一步的发展，但就今天而言，重要的是智能指针的质量。适 者生存，这就是为什么人们在使用Smart_ptr的原因。Boost 智能指针是一块精美的、精心挑选的、美味的软件巧克力，我经常吃它们(你也应该这样)。我们很快就会看到它们中的某些将成为C++标准库的一部分，因为它 们已经被收入Library Technical Report。

Library 2. Conversion

Conversion 库如何改进你的程序？

- 可理解、可维护，以及一致的多态类型转换
- 静态向下转型使用比 `static_cast` 更 安全的结构
- 进行范围判断的数字转换确保正确的值逻辑以及更少的调试时间
- 正确且可重用的文字转换导致更少的编码时间

C++的多功能性是它获得成功的主要原因之一，但有时也是麻烦的来源，因 为语言各部分的复杂性。例如，数字转换规则以及类型提升规则都很复杂。其它转换虽然简单，但也很乏味；多少次我们需要写一个安全的函数[1]来进 行 `string` s 和 `int` s, `double` s 和 `string` s 之间的转换？在你写的每个库和程序里，类型转换都可能是有问题的，这就是 Conversion 库可以帮助你的地方。它提供了防止危险转换及可复用的类型转换工具。

> [1] 避免使用 `sprintf` 及其相关函数。

Conversion 库由四个转换函数组成，分别提供了更好的类型安全性(`polymorphic_cast`), 更高效的类型安全防护(`polymorphic_downcast`), 范围检查的数字转换(`numeric_cast`), 以及文字转换(`lexical_cast`)。这些类cast函数共享C++转型操作符的语义。与C++的转型 操作符一样，这些函数具有一个重要的品质， 类型安全性，这是它们与C风格转型的区别：它们明确无误地表达了程序员的意图[2]。我们所写的代码的重要性不仅在于它可以 正确执行。更重要的是代码可否清晰地表达我们的意图。这个库使得我们可以更容易地扩展我们的C++词汇表。

> [2] 它们也可以被重载，以使得它们比C++转型操作符更高级。

polymorphic_cast

头文件: `"boost/cast.hpp"`

C++中的多态转型是用 `dynamic_cast` 来实现的。 `dynamic_cast` 有一个有时会导致错误代码的特性，那就是它对于所使用的不同类型会有不同的行为。在用于一个引用类型时，如果转型失败， `dynamic_cast` 会抛出一个 `std::bad_cast` 异常。这样做的原因很简单，因为C++里不允许有空的引用，所以要么转型成功，要么转型失败而你获得一个异常。当然，在 `dynamic_cast` 用于一个指针类型时，失败时将返回空指针。

`dynamic_cast` 的这种对指针和引用类型的不同行为以前被认为是一个有用 的特性，因为它允许程序员表达他们的意图。典型地，如果转型失败不是一种逻辑错误，就使用指针转型，如果它确是一种错误，就使用引用转型。不幸的是，两种 方法之间的区别仅在于一个*号和一个&号，这种细微的差别是不自然的。如果想把指针转型失败作为错误处理，该怎么办？为了通过自动抛出异常来清楚 地表达这一点，也为了让代码更一致，Boost提供了 `polymorphic_cast` . 它在转型失败时总是抛出一个 `std::bad_cast` 异常。

在《The C++ Programming Language 3rd Edition》中，Stroustrup对于指针类型的 `dynamic_cast` 说了以下一段话，事实是它可以返回空指针：

"偶尔可能会不小心忘了测试指针是否为空。如果这困扰了你，你可以写一转型函数在转型失败时抛出异常。"

`polymorphic_cast` 正是这样一个转型函数。

用法

`polymorphic_cast` 的用法类似于 `dynamic_cast` , 除了 (正是它的意图) 在转型失败时总是抛出一个 `std::bad_cast` 异常。 `polymorphic_cast` 的另一个特点是它是一个函数，必要时可以被重载。作为对我们的C++词汇表的一个自然扩展，它使得代码更清晰，类型转换也更少错误。要使用它，就要包含头文件 `"boost/cast.hpp"` . 这个函数泛化了要转换的类型，并接受一个要进行转型的参数。

```
template <class Target, class Source>
  polymorphic_cast(Source* p);
```

要注意的是， `polymorphic_cast` 没有针对引用类型的版本。原因是那是 `dynamic_cast` 已经实现了的，没有必须让 `polymorphic_cast` 重复C++语言中已有的功能。以下例子示范了与 `dynamic_cast` 类似的语法。

向下转型和交叉转型

使用 `dynamic_cast` 或 polymorphic_cast可能有两种典型的情况：从基类向派生类的向下转型，或者交叉转型，即从一个基类到另一个基类。以下例子示范了使用 `polymorphic_cast` 的两类转型。这里有两个基类， `base1` 和 `base2` ，以及一个从两个基类公有派生而来的类 `derived` 。

```
#include <iostream>
#include <string>
#include "boost/cast.hpp"

class base1 {
public:
  virtual void print() {
    std::cout << "base1::print()\n";
  }

  virtual ~base1() {}
};

class base2 {
public:

  void only_base2() {
    std::cout << "only_base2()\n";
  }

  virtual ~base2() {}
};

class derived : public base1, public base2 {
public:

  void print() {
    std::cout << "derived::print()\n";
  }

  void only_here() {
    std::cout << "derived::only_here()\n";
  }
  void only_base2() {
    std::cout << "Oops, here too!\n";
  }
};

int main() {
  base1* p1=new derived;

 p1->print();

  try {
    derived* pD=boost::polymorphic_cast<derived*>(p1);
    pD->only_here();
    pD->only_base2();

    base2* pB=boost::polymorphic_cast<base2*>(p1);
    pB->only_base2();

  }
  catch(std::bad_cast& e) {
    std::cout << e.what() << '\n';
  }

  delete p1;
}
```

我们来看看 `polymorphic_cast` 是如何工作的，首先我们创建一个 `derived` 的实例，然后通过不同的基类指针以及派生类指针来操作它。对 `p1` 使用的第一个函数是 `print`，它是 `base1` 和 `derived` 的一个虚拟函数。我们还使用了向下转型，以便可以调用 `only_here`，它仅在 `derived` 中可用：

```
derived* pD=boost::polymorphic_cast<derived*>(p1);
pD->only_here();
```

注意，如果 `polymorphic_cast` 失败了，将抛出一个 `std::bad_cast` 异常，因此这段代码被保护在一个 `try` / `catch` 块中。这种做法与使用引用类型的 `dynamic_cast` 正好是一样的。指针 `pD` 随后被用来调用函数 `only_base2`. 这个函数是 `base2` 中的非虚拟函数，但是在 `derived` 中也提供了，因此隐藏了 `base2` 中的版本。因而我们需要执行一个交叉转型来获得一个 `base2` 指针，才可以调用到 `base2::only_base2` 而不是 `derived::only_base2`.

```
base2* pB=boost::polymorphic_cast<base2*>(p1);
pB->only_base2();
```

再一次，如果转型失败，将会抛出异常。这个例子示范了如果转型失败被认为是错误的话，使用 `polymorphic_cast` 可以多容易地进行错误处理。不需要测试空指针，也不会把错误传播到函数以外。正如我们即将看到的，`dynamic_cast` 有时会为这类代码增加不必要的复杂性；它还可能导致未定义行为。

dynamic_cast 对 polymorphic_cast

为了看一下这两种转型方法之间的不同，[3] 我们把它们放在一起来比较一下复杂性。我们将重用前面例子中的类 `base1`，`base2`，和 `derived`。你会发现在对指针类型使用 `dynamic_cast` 时，测试指针的有效性是一种既乏味又反复的事情，这使得测试很容易被紧张的程序员所忽略掉。

> [3] 技术上，`dynamic_cast` 是转型操作符，而 `polymorphic_cast` 是函数模板。

```
void polymorphic_cast_example(base1* p) {
  derived* pD=boost::polymorphic_cast<derived*>(p);
  pD->print();

  base2* pB=boost::polymorphic_cast<base2*>(p);
  pB->only_base2();
}

void dynamic_cast_example(base1* p) {
  derived* pD=dynamic_cast<derived*>(p);
  if (!pD)
    throw std::bad_cast();
  pD->print();

  base2* pB=dynamic_cast<base2*>(p);
  if (!pB)
    throw std::bad_cast();

  pB->only_base2();

}

int main() {
  base1* p=new derived;
  try {
    polymorphic_cast_example(p);
    dynamic_cast_example(p);
  }
  catch(std::bad_cast& e) {
    std::cout << e.what() << '\n';
  }
  delete p;
}
```

这两个函数，`polymorphic_cast_example` 和 `dynamic_cast_example`，使用不同的方法完成相同的工作。差别在于无论何时对指针使用 `dynamic_cast`，我们都要记住测试返回的指针是否为空。在我们的例子里，这种情况被认为是错误的，因此要抛出一个类型为 `bad_cast` 的异常。[4] 如果使用 `polymorphic_cast`，错误的处理被局限在 `std::bad_cast` 的异常处理例程中，这意味着我们不需要为测试转型的返回值而操心。在这个简单的例子中，不难记住要测试返回指针的有效性，但还是要比使用 `polymorphic_cast` 做更多的工作。如果是几百行的代码，再加上两三个程序员来维护这个函数的话，忘记测试或者抛出了错误的异常的风险就会大大增加。

> [4] 当然，返回指针无论如何都必须被检查，除非你绝对肯定转型不会失败。

polymorphic_cast 不总是正确的选择

如果说失败的指针转型不应被视为错误，你就应该使用 `dynamic_cast` 而不是 `polymorphic_cast`. 例如，一种常见的情形是使用 `dynamic_cast` 来进行类型确定测试。使用异常处理来进行几种类型的转换测试是低效的，代码也很难看。这种情形下 `dynamic_cast` 就很有用了。当我们同时使用 `polymorphic_cast` 和 `dynamic_cast` 时，你应该非常清楚你自己的意图。即使没有 `polymorphic_cast`，如果人们知道使用 `dynamic_cast` 的方法，他仍然可以达到相同的安全性，如下例所示。

```
void failure_is_error(base1* p) {

  try {
    some_other_class& soc=dynamic_cast<some_other_class&>(*p);
    // 使用 soc
   }
  catch(std::bad_cast& e) {
    std::cout << e.what() << '\n';
  }
}

void failure_is_ok(base1* p) {
  if (some_other_class* psoc=
    dynamic_cast<some_other_class*>(p)) {
    // 使用 psoc
  }
}
```

在这个例子中，指针 `p` 被解引用[5] 并被转型为 `some_other_class` 的引用。这调用了 `dynamic_cast` 的异常抛出版本。例子中的第二部分使用了不会抛出异常的版本来转型到指针类型。你是否认为这是清晰、简明的代码，答案取决于你的经验。经验丰富的C++程序员会非常明白这段程序。是不是所有看到这段代码的人都十分熟悉 `dynamic_cast` 呢，或者他们不知道 `dynamic_cast` 的 行为要取决于进行转型的是指针还是引用呢？你或者一个维护程序员是否总能记得对空指针进行测试？维护代码的程序员是否知道要对指针进行解引用才可以在转型 失败时获得异常？你真的想在每次你需要这样的行为时都写相同的逻辑吗？抱歉说了这么多，这只是想表明，如果转型失败应该要抛出异常，那么 `polymorphic_cast` 要比 `dynamic_cast` 更坚固也更清晰。它要么成功，产生一个有效的指针，要么失败，抛出一个异常。简单的规则总是更容易被记住。

> [5] 如果指针 `p` 为空，该例将导致未定义行为，因为它解引用了一个空指针。

我们还没有看到如何通过重载 `polymorphic_cast` 来解决一些不常见的转型需求，但你应该知道这是可能的。何时你会想改变多态转型的缺省行为呢？有一种情形是句柄/实体类 (handle/body-classes), 向下转型的规则可能会与缺省的不同，或者是根本不允许。

总结

必须记住，其它人将要维护我们写的代码。这意味着我们必须确保代码以及它的意图是清晰并且易懂的。这一点可以通过注释部分地解决，但对于任何人，更容易的方法是不需加以说明的代码。当(指针)转型失败被认为是异常时， `polymorphic_cast` 比 `dynamic_cast` 更能清晰地表明代码的意图，它也导致更短的代码。如果转型失败不应被认为是错误，则应该使用 `dynamic_cast` ，这使得 `dynamic_cast` 的使用更为清楚。仅仅使用 `dynamic_cast` 来表明两种不同的意图很容易出错，而不够清楚。抛出异常与不抛出异常这两个不同的版本对于大多数程序员而言太微妙了。

何时使用 `polymorphic_cast` 和 `dynamic_cast`：

- 当一个多态转型的失败是预期的时候，使用 `dynamic_cast&lt;T*&gt;` . 它清楚地表明转型失败不是一种错误。

- 当一个多态转型必须成功以确保逻辑的正确性时，使用 `polymorphic_cast<T*>`. 它清楚地表明转型失败是一种错误。
- 对引用类型执行多态转型时，使用 `dynamic_cast`.

polymorphic_downcast

头文件: `"boost/cast.hpp"`

有时 `dynamic_cast` 被认为太过低效(的确如此)。执行 `dynamic_cast` 需要额外的运行时间。为了避免这些代价，常常会诱使你使用 `static_cast`，它没有这些性能代价。`static_cast` 用于向下转型可能在危险的，并会导致错误，但它的确比 `dynamic_cast` 要快。如果这些加速是需要的，那我们就要确保向下转型的安全性。`dynamic_cast` 会测试向下转型的结果，并在失败时返回空指针或抛出异常，而 `static_cast` 则仅仅执行需要的指针运算，并将保证转型有效的责任留给了程序员。为了确保用 `static_cast` 进行向下转型是安全的，你必须确保对每次要执行的转型进行测试。`polymorphic_downcast` 用 `dynamic_cast` 进行了转型的测试，但仅是在调试模式下；然后它就使用 `static_cast` 去执行转型。在发布模式下，只执行 `static_cast`。这样的转型方法意味着你知道它不可能失败，所以没有错误处理，也没有异常抛出。那么如果在非调试模式下 `polymorphic_downcast` 失败了，会发生什么呢？未定义的行为。你的计算机可能崩溃。地球可以停止自转。你可能飞到云上。你唯一可以肯定的是你的程序可能会发生不好的事情。如果 `polymorphic_downcast` 是在调试模式下失败的，它对 `dynamic_cast` 产生的空指针执行断言(并退出)。

在讨论用 `polymorphic_downcast` 更换 `dynamic_cast` 可以如何加速你的程序之前，你应该先检查一下设计。转型的优化几乎就代表着设计的问题。如果向下转型真的是必须的，并且被证实是性能的瓶颈，`polymorphic_downcast` 就是你需要的。你可以在测试时发现错误的转型，而不是在产品中(发布模式构建)，如果你曾经听到过从电话另一端传来的用户的尖叫，你就该知道在测试时找出错误是多么的重要，它使生活更轻松。很有可能你就是用户，而且知道发现并报告别人的错误是多么的讨厌。因此，在真正需要的时候才用 `polymorphic_downcast`，而且要小心。

用法

`polymorphic_downcast` 用于那些你应该用而又不想用 `dynamic_cast` 的情形，原因是你确认将要发生的转型肯定会成功，而且你需要提升它带来的性能。注意：一定要确保使用的 `polymorphic_downcast` 所有可能的类型及转换组合都经过测试。否则，不要使用 `polymorphic_downcast`；用 `dynamic_cast` 代替它。当你决定继续使用 `polymorphic_downcast`，包含头文件 `"boost/cast.hpp"`.

```
#include <iostream>
#include "boost/cast.hpp"

struct base {
  virtual ~base() {};
};

struct derived1 : public base {
  void foo() {
    std::cout << "derived1::foo()\n";
  }
};

struct derived2 : public base {
  void foo() {
    std::cout << "derived2::foo()\n";
  }
};

void older(base* p) {
  // Logic that suggests that p points to derived1 omitted
  derived1* pd=static_cast<derived1*>(p);
  pd->foo(); // <-- What will happen here?
}

void newer(base* p) {
  // Logic that suggests that p points to derived1 omitted
  derived1* pd=boost::polymorphic_downcast<derived1*>(p);
  // ^-- The above cast will cause an assertion in debug builds
  pd->foo();
}

int main() {
       derived2* p=new derived2;
       older(p); // <-- Undefined
       newer(p); // <-- Well defined in debug build
}
```

函数 `older` 中的 `static_cast` 会编译成功，[6] 但它会带来坏运气，成员函数 `foo` 的存在使得错误(可能有，但不保证)被错过，直到有人拿着一份错误报告，用调试器在别的地方查找奇怪的行为。当使用 `static_cast` 将指针向下转型为 `derived1*`，编译器没有选择，只能相信程序员，转型是有效的。但事实上，传送给 `older` 的指针是指向一个 `derived2` 实例的。因此，`older` 里的指针 `pd` 指向了一个完全不同的类型，这意味着什么都可能发生。这就是使用 `static_cast` 进行向下转型的风险。转型总是"成功"的，但指针可能是无效的。

> [6] 至少它会被编译。

在对函数 `newer` 的调用里，"更好的 `static_cast`," `polymorphic_downcast` 不仅捕捉到了错误，并且使用断言指出了发生错误的地方。当然，这仅在调试模式下是真的，使用 `dynamic_cast` 来测试转型是否成功。把一个无效的转型留在发布版本中会导致不幸。换言之，就算你在调试模式下获得了额外的安全性，但这并不足以代表你已经试过了所有可能的转换。

总结

使用 `static_cast` 进行向下转换通常是危险的。你不应该这样做，但如果一定要，使用 `polymorphic_downcast` 可以增加一点安全性。它在调试模式下增加了测试，可以帮助你发现转型的错误，但你必须测试所有可能的转型以确保它的安全使用。

- 如果你正在使用向下转型并需要在发布版本中获得 `static_cast` 的速度，就用 `polymorphic_downcast`；至少在测试时你可以在出错时得到断言的帮助。
- 如果不能测试所有可能的转型，就不要使用 `polymorphic_downcast`.

记住这是一种优化方法，你应该在确定需要它们时才使用。

numeric_cast

头文件: `"boost/cast.hpp"`

整数类型间的转换经常会产生意外的结果。例如，`long` 可以拥有比 `short` 更大范围的值，那么当从 `long` 赋值到 `short` 并且 `long` 的数值超出了 `short` 的 范围时会发生什么？答案是结果是由实现定义的(比"你不可能明确知道"好听一点的说法)。相同大小整数间的有符号数到无符号数的转换是好的，只要有符号数 的数值是正的，但如果有符号数的数值是负的呢？它将被转换为一个大的无符号数，如果这不是你的真实意图，那么就真的是一个问题了。`numeric_cast` 通过测试范围是否合理来确保转换的有效性，当范围超出时它会抛出异常。

在我们全面认识 `numeric_cast` 之前，我们必须弄清楚支配整数类型的转换及提升的规则。规则有很多并有时很微妙，即使是经验丰富的程序员也会被它们欺骗。与其写出所有这些规则[7]并展开它们，我更愿意给出一些有关转换的例子，它们会引起未定义或令人惊讶的行为，然后再解释所使用的转换规则。

```
[7]. C++标准在§4.5-4.9中讨论数字类型的提升及转换。
```

当从一种数字类型赋值给另一种数字类型的变量时，就会发生类型转换。在目标类型可以保存源类型的所有数值的情况下，这种转换是完全安全的，否则就是不安全的。例如，`char` 通常不能保存 `int` 的最大值，所以当从 `int` 到 `char` 的赋值发生时，很大可能 `int` 的值不能被表示为 `char` . 当类型可以表示的数值范围不同时，我们必须确认用于转换的实际数值在目标类型的有效范围之内。否则，我们就会进入实现定义行为的范畴；那就是在把一个超出数字类型可能的数值范围的值赋给这个数字类型时会发生的事情。[8] 实现定义行为意味着具体实现可以自由地做任何它想做的；不同的系统可能有完全不同的行为。`numeric_cast` 可以确保转换是有效的、合法的，否则就不允许转换。

> [8] 无符号数也算，尽管它的行为是有定义的。

用法

`numeric_cast` 是一个看起来象C++的转型操作符的函数模板，它泛化了目标类型及源类型。源类型可以从函数的参数隐式推导得到。使用 `numeric_cast` , 要包含头文件 `"boost/cast.hpp"` 。以下两个转换使用 `numeric_cast` 安全地将 `int` 转换为 `char` , 以及将 `double` 转换为 `float` .

```
char c=boost::numeric_cast<char>(12);
float f=boost::numeric_cast<float>(3.001);
```

一个最常见的数字转换问题是将来自一个更宽范围的值赋给范围较窄的类型。我们来看看numeric_cast如何帮忙。

从较大的类型到较小类型的赋值

从较大的类型(例如 `long`)向较小的类型(例如 `short`)赋值，有可能数值过大或过小而不能被目标类型所表示。如果这发生了，结果是(是的，正如你猜到的)实现所定义的。我们稍后将讨论无符号类型的潜在问题；我们先从有符号类型开始。C++中有四个内建的有符号类型：

- `signed char`
- `short int (short)`
- `int`
- `long int (long)`

没有人可以绝对肯定哪个类型比其它的大[9]，但典型地，上面的列表是按大小递增的，除了 `int` 和 `long` 通常具有相同的值范围。但它们都是独立的类型，即使是有相同的大小。想查看你的系统上的类型大小，可以使用 `sizeof(T)` 或 `std::numeric_limits<T>::max()` 和 `std::numeric_limits<T>::min()` .

> [9] 当然，有符号类型与无符号类型的范围是不同的，即使它们有相同的大小。

当把一个有符号整数类型赋给另一个时，C++标准说：

> "若目标类型为有符号类型，在数值可以被目标类型表示时，值不改变；否则，值为实现定义。"[10]
>
> > [10] 见C++标准 §4.7.3

以下代码段示范了看起来象是正确的赋值是如何导致实现定义的数值，最后看看如何通过 `numeric_cast` 的帮助避免它们。

```
#include <iostream>
#include "boost/cast.hpp"
#include "boost/limits.hpp"

int main() {
  std::cout << "larger_to_smaller example\n";

  // 没有使用numeric_cast的转换
  long l=std::numeric_limits<short>::max();

  short s=l;
  std::cout << "s is: " << s << '\n';
  s=++l;
  std::cout << "s is: " << s << "\n\n";

  // 使用numeric_cast的转换
  try {
    l=std::numeric_limits<short>::max();
    s=boost::numeric_cast<short>(l);
    std::cout << "s is: " << s << '\n';
    s=boost::numeric_cast<short>(++l);
    std::cout << "s is: " << s << '\n';
  }
  catch(boost::bad_numeric_cast& e) {
    std::cout << e.what() << '\n';
  }
}
```

通过使用 `std::numeric_limits`，`long l` 被初始化 `short` 可以表示的最大值。该值被赋给 `short s` 并输出。然后，`l` 被加一，这意味着它的值不能再被 `short` 所表示；它超出了 `short` 所能表示的范围。把 `l` 的新值赋给 `s`，`s` 再次被输出。你可能要问输出的值是什么？好的，因为赋值的结果属于实现定义的行为，这取决于你使用的平台。在我的系统中，使用我的编译器，它变成了一个大的负值，即它被回绕了。必须运行前面的代码才知道在你的系统中会有什么结果[11]。接着，再次执行相同的操作，但这次用了 `numeric_cast`. 第一个转型成功了，因为数值在范围之内。而第二个转型却会失败，结果是抛出一个 `bad_numeric_cast` 异常。程序的输出如下。

> [11] 这种行为和结果在32位平台上十分常见。

```
larger_to_smaller example
s is: 32767
s is: -32768

s is: 32767
bad numeric cast: loss of range in numeric_cast
```

比避开实现定义行为更为重要的是，`numeric_cast` 帮助我们避免了错误，否则会很难捕捉到这些错误。那个奇怪的数值可能被传送到应用程序的其它部分，程序可能会继续工作，但几乎可以肯定将产生错误的结果。 当然，这仅对于特定的数值会发生这样的情况，如果这些数值很少出现，那么错误将很难被发现。这种错误非常阴险，因为它们仅仅对某些特定值会发生，而不是总会发生。

精宽或取值范围的损失并不常见，如果你不确定一个值对于目标类型是否过大或过小，`numeric_cast` 就是你可以使用的工具。你甚至可以在不需要的时候使用 `numeric_cast`；维护的程序员可能没有象你一样的洞察力。注意，虽然我们在这里只讨论了有符号类型，但同样的原理可应用于于无符号类型。

特殊情况：目标类型为无符号整数

无符号整数类型有一个非常有趣的特性，任何数值都有可以合法地赋给它们！对于无符号类型而言，无所谓正或负的溢出。数值被简单地对目标类型最大值加一取模。什么意思？看看以下例子会更清楚一些。

```
#include <iostream>
#include "boost/limits.hpp"

int main() {
  unsigned char c;
  long l=std::numeric_limits<unsigned char>::max()+14;

  c=l;
  std::cout << "c is:       " << (int)c << '\n';
  long reduced=l%(std::numeric_limits<unsigned char>::max()+1);
  std::cout << "reduced is: " << reduced << '\n';
}
```

运行这个程序的输出如下：

```
c is:       13
reduced is: 13
```

这个例子把一个明显超出 `unsigned char` 可以表示的数值赋给它，然后再计算得到同样的数值。赋值的动作可以用这一行代码来示范：

```
long reduced=l%(std::numeric_limits<unsigned char>::max()+1);
```

这种行为通常被称为数值回绕(value wrapping)。如果你想用这个特性，就没有必要在这种情况下使用 `numeric_cast`。此外，`numeric_cast` 也不接受它。`numeric_cast` 的意图是捕捉错误，而错误应该是因为用户的误解而引起的。如果目标类型不能表示赋给它的数值，就抛出一个 `bad_numeric_cast` 异常。因为无符号整数的算法是明确定义的，不会引起程序员的重大错误[12]。对于 `numeric_cast`，重要的是确保获得实际的数值。

> [12] 观点是：如果你真的想要数值回绕，就不要使用 `numeric_cast`.

有符号和无符号整数类型的混用

混用有符号和无符号类型可能很有趣[13]， 特别是执行算术操作时。普通的赋值也会产生微妙的问题。最常见的问题是将一个负值赋给无符号类型。结果几乎可以肯定不是你原来的意图。另一种情形是从无符 号类型到同样大小的有称号类型的赋值。不知什么原因，人们总是会很容易忘记无符号类型可以持有比同样大小的有符号类型更大的值。特别是在表达式或函数调用 中更容易忘记。以下例子示范了如何通过 `numeric_cast` 来捕捉这种常见的错误。

> [13] 当然这是一个高度主观的问题，你的观点可能不同。

```
#include <iostream>
#include "boost/limits.hpp"
#include "boost/cast.hpp"

int main() {
  unsigned int ui=std::numeric_limits<unsigned int>::max();
  int i;

  try {
    std::cout << "Assignment from unsigned int to signed int\n";
    i=boost::numeric_cast<int>(ui);
  }
  catch(boost::bad_numeric_cast& e) {
    std::cout << e.what() << "\n\n";
  }

  try {
    std::cout << "Assignment from signed int to unsigned int\n";
    i=-12;
    ui=boost::numeric_cast<unsigned int>(i);
  }
  catch(boost::bad_numeric_cast& e) {
    std::cout << e.what() << "\n\n";
  }
}
```

输出清晰地表明了预期的错误。

```
Assignment from unsigned int to signed int
bad numeric cast: loss of range in numeric_cast
Assignment from signed int to unsigned int
bad numeric cast: loss of range in numeric_cast
```

基本的规则很简单：无论何时在不同的类型间执行类型转换，都应该使用 `numeric_cast` 来保证转换的安全。

浮点数类型

`numeric_cast` 不能帮助我们在浮点数间的转换中避免精度的损失。原因是 `float`，`double`，和 `long double` 间的转换不象整数类型间的隐式转换那样敏感。记住这点很重要，因为你可能会认为以下代码应该抛出异常。

```
double d=0.123456789123456;
float f=0.123456;

try {
  f=boost::numeric_cast<float>(d);
}
  catch(boost::bad_numeric_cast& e) {
    std::cout << e.what();
}
```

运行这段代码不会有异常抛出。在许多实现中，从 `double` 到 `float` 的转换都会导致精度的损失，虽然C++标准没有保证会这样。我们所能知道的就是， `double` 至少具有 `float` 的精度。

从浮点数类型转为整数类型又会怎样呢？当一个浮点数类型被转换为一个整数类型，它会被截断；小数部分会被扔掉。 `numeric_cast` 对截断后的数值与目标类型进行相同的检查，就象在两个整数类型间的检查一样。

```
double d=127.123456789123456;
char c;
std::cout << "char type maximum: ";
std::cout << (int)std::numeric_limits<char>::max() << "\n\n";

c=d;
std::cout << "Assignment from double to char: \n";
std::cout << "double: " << d << "\n";
std::cout << "char:   " << (int)c << "\n";

std::cout << "Trying the same thing with numeric_cast:\n";

try {
  c=boost::numeric_cast<char>(d);
  std::cout << "double: " << d;
  std::cout << "char:   " << (int)c;
}
  catch(boost::bad_numeric_cast& e) {
    std::cout << e.what();
}
```

象前面的代码那样进行范围检查以确保有效的赋值是一件令人畏缩的工作。虽然规则看起来很简单，但是有很多组合要被考虑。例如，测试从浮点数到整数的代码看起来就象这样：

```
template <typename INT, typename FLOAT>
  bool is_valid_assignment(FLOAT f) {
    return std::numeric_limits<INT>::max() >=
      static_cast<INT>(f);
  }
```

尽管我已经提起过在一个浮点数类型被转换时，小数部分会被丢弃，在这个实现中还得很容易忽略这个错误。这对于算术类型的转换和提升是自然的。去掉 `static_cast` 就可以正确地测试，因为这样 `numeric_limits<INT>::max` 的结果会被转换为浮点数类型[14]。如果是浮点数类型转为整数类型，它会被截断；换句话说，这个函数的问题在于丢失了小数部分。

> [14] 这是正常的算术转换结果。

总结

`numeric_cast` 提供了算术类型间高效的范围检查转换。在目标类型可以持有所有源类型的值时，使用 `numeric_cast` 没有额外的效率代价。它只在目标类型仅能表示源类型的值的子集时有影响。当转换失败时， `numeric_cast` 通过抛出一个 `bad_numeric_cast` 异常来表示失败。对于数值类型间的转换有很多复杂的规则，确保转换的正确性是很重要的。

以下情况时使用 `numeric_cast`：

- 在无符号与有符号类型间进行赋值或比较时
- 在不同大小的整数类型间进行赋值或比较时
- 从一个函数返回类型向一个数值变量赋值，为了预防该函数未来的变化

在这里注意到一个模式了吗？模仿已有的语言和库的名字及行为是简化学习及使用的好方法，但也需要仔细地考虑。增加内建的C++转型就象沿着狭窄的小路行走；一旦迷路会带来很高的代价。遵循语言的语法及语义规则才是负责任的。事实上，对于初学者，内建的转型操作符与看起来象转型操作符的函数可能并没有不同，所以如果行为错误将会导致灾难。 `numeric_cast` 有着与 `static_cast`，`dynamic_cast`，和 `reinterpret_cast` 类似的语法和语义。如果它看起来和用起来象转型操作，它就是转型操作，是对转型操作的一个良好的扩展。

lexical_cast

头文件: `"boost/lexical_cast.hpp"`

所有应用都会使用字面转换。我们把字符串转为数值，反之亦然。许多用户定义的类型可以转换为字符串或者由字符串转换而来。你常常是在需要这些转换时才编写代码，而更好的方法是提供一个可重用的实现。这就是 `lexical_cast` 的用途所在。你可以把 `lexical_cast` 想象为使用一个 `std::stringstream` 作为字符串与数值的表示之间的翻译器。这意味着它可以与任何用 `operator<<` 进行输出的源以及任何用 `operator<<` 进行输入的目标一起工作。这个要求对于所有内建类型与多数用户自定义类型(UDTs)都可以做到。

用法

`lexical_cast` 在类型之间进行转换，就象其它的类型转换操作一样。当然，使它得以工作的必须是一个转换函数，但从概念上说，你可以把它视为转型操作符。比起调用一堆的转换子程序，或者是编写自己的转换代码， `lexical_cast` 可以更好地为任何满足它的要求的类型服务。它的要求就是，源类型必须是可流输出的(OutputStreamable)，而目标类型必须是可流输入的 (InputStreamable)。另外，两种类型都必须是可复制构造的(CopyConstructible)，并且目标类型还要是可缺省构造的 (DefaultConstructible)和可赋值的(Assignable)。可流输出(OutputStreamable)意味着存在一个为该类 型定义的 `operator<<` ，可流输入(InputStreamable)则要求有一个 `operator>>` . 对于许多类型，包括所有内建类型和标准库中的字符串类型，这个条件都满足。要使用 `lexical_cast` , 就要包含头文件 `"boost/lexical_cast.hpp"` .

让 lexical_cast 工作

我不想通过跟你示范手工编写转换用的代码来说明 `lexical_cast` 如何节省了你的时间，因为我可以很肯定你一定写过这样的转换代码，并且很可能不只一次。相反，只用一个例子来示范如何使用 `lexical_cast` 来进行通用的(字面上的)类型转换。

```
#include <iostream>
#include <string>
#include "boost/lexical_cast.hpp"

int main() {
  // string to int
  std::string s="42";
  int i=boost::lexical_cast<int>(s);

  // float to string
  float f=3.14151;
  s=boost::lexical_cast<std::string>(f);

  // literal to double
  double d=boost::lexical_cast<double>("2.52");

  // 失败的转换
  s="Not an int";
  try {
    i=boost::lexical_cast<int>(s);
  }
  catch(boost::bad_lexical_cast& e) {
    // 以上lexical_cast将会失败，我们将进入这里
  }
}
```

这个例子仅仅示范了多种字面转换情形中的几种，我想你应该同意为了完成这些工作，通常你需要更多的代码。无论何时你不确定转换是否有效，都应该用一个 `try/catch` 块来保护 `lexical_cast` ，就象你在这个例子看到的那样。你可能注意到了没有办法控制这些转换的格式；如果你需要这种级别的控制，你要用 `std::stringstream` ！

如果你曾经手工进行过类型间的转换，你应该知道，对于不同的类型，需要使用不同的办法来处理转换以及可能出现的转换失败。这不仅是有点不便而已，它还妨碍了用泛型代码执行转换的努力。稍后我们将看到 lexical_cast 如何帮助你实现这一点。

这个例子中的转换用手工来实现也非常简单，但可能会失去转型操作的美观和优雅。而 `lexical_cast` 做起来更简单，并且更美观。再考虑一下 `lexical_cast` 对与之一起工作的类型所需的简单要求。考虑到对所有符合该要求的类型的转换可以在一行代码内完成的事实。再结合该实现依赖于标准库的 `stringstream` 这一事实[15]，你可以看到 lexical_cast 不仅是执行字面转换的便利方法，它更是C++编译艺术的一个示范。

> [15] 事实上，对于某些转换，有一些优化的方法可以避免使用 `std::stringstream` 带来的额外开销。当然，你可以在需要的时候对你自己的类型定制它的行为。

用 lexical_cast 进行泛型编程

作为使用 `lexical_cast` 进行泛型编程的简单例子，来看一下如何用它创建一个 `to_string` 函数。这个函数接受任何类型的参数(当然它要符合要求)并返回一个表示该值的 `string` 。标准库的用法当然也可以在 `std::stringstream` 的帮助下用几行代码完成这个任务。在这里，我们使用 `lexical_cast` 来实现，只需要一个前转换函数调用及一些错误处理。

```
#include <iostream>
#include <string>
#include "boost/lexical_cast.hpp"

template <typename T> std::string to_string(const T& arg) {
  try {
    return boost::lexical_cast<std::string>(arg);
  }
  catch(boost::bad_lexical_cast& e) {
    return "";
  }
}

int main() {
  std::string s=to_string(412);
  s=to_string(2.357);
}
```

这个小程序不仅易于实现，它还因为 `lexical_cast` 而增加了价值。

使类可以用于 lexical_cast

因为 `lexical_cast` 仅要求它所操作的类型提供适当的 `operator<<` 和 `operator>>` ，所以很容易为用户自定义类型增加字面转换的支持。一个可以同时作为 `lexical_cast` 的目标和源的简单UDT看起来就象这样：

```
class lexical_castable {
public:
  lexical_castable() {};
  lexical_castable(const std::string s) : s_(s) {};

  friend std::ostream operator<<
    (std::ostream& o, const lexical_castable& le);
  friend std::istream operator>>
    (std::istream& i, lexical_castable& le);

private:
  virtual void print_(std::ostream& o) const {
    o << s_ <<"\n";
  }

  virtual void read_(std::istream& i) const {
    i >> s_;
  }

  std::string s_;
};

std::ostream operator<<(std::ostream& o,
  const lexical_castable& le) {
  le.print_(o);
  return o;
}

std::istream operator>>(std::istream& i, lexical_castable& le) {
  le.read_(i);
  return i;
}
```

`lexical_castable` 类现在可以这样用了：

```
int main(int argc, char* argv[]) {
  lexical_castable le;
  std::cin >> le;

  try {
    int i = boost::lexical_cast<int>(le);
  }
  catch(boost::bad_lexical_cast&) {
     std::cout << "You were supposed to enter a number!\n";
  }
}
```

当然，输入和输出操作符最好可以允许这个类于于其它流。如果你使用标准库的IOStreams，或者其它使用 `operator<<` 和 `operator>>` 的库，你可能已经有很多可以用于 `lexical_cast` 的类。它们不需要进行修改。直接对它们进行字面转换就行了！

总结

`lexical_cast` 是用于字符串与其它类型之间的字面转换的一个可重用及高效的工具。它是功能性和优雅性的结合，是杰出程序员的伟大杰作[16]。 不要在需要时实现小的转换函数，更不要在其它函数中直接插入相关逻辑，应该使用象 `lexical_cast` 这样的泛型工具。它有助于使代码更清晰，并让程序员专注于解决手上的问题。

> [16] 我知道，我总是很傲慢的，我们这些程序员，工作中常常需要数学、物理学、工程学、建筑学，和其它一些艺术和学科。这会使人畏缩，但也有无穷的回报。

以下情况时使用 `lexical_cast`：

- 从字符串类型到数值类型的转换
- 从数值类型到字符串类型的转换
- 你的自定义类型所支持的所有字面转换

Conversion 总结

在这一章里，你学习了 Boost.Conversion 库，从 `polymorphic_cast` 开始。 `polymorphic_cast` 的基本原理是代码的清晰性和安全性，它使我们在代码中更灵活地表达我们的意图，还有安全性，与它的竞争者 `dynamic_cast&lt;T*&gt;` 相比它更为安全，因为对结果指针的测试很容易忘记。

接着，你看到了安全的优化，使用 `polymorphic_downcast`，它在调试模式下增加了类似于 `dynamic_cast` 的安全性，但却是使用 `static_cast` 来进行转换。这样比单独使用 `static_cast` 更安全。

`numeric_cast` 帮助你避免数值转换中的某些困难。还有，代码的清晰性也得到提高，从而避免了未定义的行为以及实现定义的行为。

最后一个是 `lexical_cast`. 没有重复的转换函数。这就是为什么它被提议纳入下一个版本的C++标准库的原因。它是一个非常小巧的、用于转换不同的可流数据类型的工具。

如果你曾经看到过这些转型的实现，你会同意它们之间没有一个是复杂的。还有，它具有它们所需的洞察力、远见和知识，并正确地、可移植地、高效地实现了它们。不是所有人都认识到使用 `dynamic_cast` 时会发生某些错误。不是很多人都知道整数类型转换和提升的复杂规则。Boost提供的转换操作包含了所有这些知识，并具有良好的设计和测试；它们是你所要的最好的选择。

Library 3. Utility

Utility 库如何改进你的程序？

- 编译期断言 `BOOST_STATIC_ASSERT`
- 安全的析构 `checked_delete` 和 `checked_array_delete`
- 禁止复制 `noncopyable`
- `operator&` 被重载时用 `addressof` 取得对象地址
- 用 `enable_if` 和 `disable_if` 控制重载与特化

有些工具还不够组成它们自己的库，因此它们与其它实体被集合到一起。这就形成了Boost.Utility，收集了一些没有更合适地方存放的、有用的工具。它们很有用，应该被加入到Boost，但它们又太小，不足以形成自己的库。本 章介绍Boost.Utility中最基本的以及最广泛使用的工具。

我们将从 `BOOST_STATIC_ASSERT` 开始，它是一个在编译期判断整型常量表达式的工具。然后，我们看看当你通过一个指向不完整类型的指针 `delete` 对象时，即当被删除的对象的内存布局未知时，会发生什么。 `checked_delete` 使得这个讨论更为有趣。我们还会看到 `noncopyable` 如何防止一个类被复制，这也是本章最重要的主题。然后我们将看到 `addressof`，它用于阻止那些重载了 `operator&` 的险恶的程序员[1]的病态行为。最后，我们将测试 `enable_if`，它非常有用，可用于在名字查找时控制函数重载与模板特化是否被考虑。

> [1] 如果你认为我说的不对，请把你认为最合理的重载了 `operator&` 的用例发给我。

BOOST_STATIC_ASSERT

头文件: `"boost/static_assert.hpp"`

在运行期执行断言可能是你经常用到的，也是非常合理的。它是测试前置条件、后置条件以及不变式的好方 法。执行运行期断言有很多不同的方法，但是在编译期你如何进行断言呢？当然，唯一的方法就是让编译器产生一个错误，这是很平常的事情(我在无意中都做过几 千次了)，但如何从错误信息中获得有意义的信息却不是那么明显的。而且，即使你在一个编译器上找到了办法，也很难把它移植到其它编译器上。这就是使用 `BOOST_STATIC_ASSERT` 的原因。它可以在不同的平台上使用，正如我们即将看到的。

用法

要开始使用静态断言，就要包含头文件 `"boost/static_assert.hpp"`. 该头文件定义了宏[2] `BOOST_STATIC_ASSERT`. 作为它的第一个使用范例，我们来看看如何在类作用域中使用它。考虑一个泛化的类，它要求实例化时所用的类型是一个整数类型。我们不想为所有类型提供特化， 因此我们需要在编译期进行测试，以确保我们的类的确是用一个整数类型进行实例化的。现在，我们先提前一点使用另一个Boost库来进行测试，它就是 Boost.Type_traits. 我们使用一个称为 `is_integral` 的断言，它对它的参数执行一个编译期求值，正如你从它的名字可以猜到的一样，求值的结果是表明该类型是否一个整数类型。

> [2] 是的，它是一个宏。你知道，宏也可以很有用的。

```
#include <iostream>

#include "boost/type_traits.hpp"
#include "boost/static_assert.hpp"

template <typename T> class only_compatible_with_integral_types {
  BOOST_STATIC_ASSERT(boost::is_integral<T>::value);
};
```

有了这个断言，在实例化类 `only_compatible_with_integral_types` 时如果试图使用一个非整型的类型，就会导致一个编译期的失败。输出信息取决于编译器，但在多数编译器下输出信息会惊人地一致。

假设我们试图这样实例化：

```
only_compatible_with_integral_types<double> test2;
```

编译器将会有类似下面的输出：

```
Error: use of undefined type
  'boost::STATIC_ASSERTION_FAILURE<false>'
```

在类的作用域里，你可以明确类的要求：象在前面这样的模板中明确参数的类型就是一个明显的例子。你也可以使用断言来明确类所要求的其它前提条件，如类型的大小等等。

函数作用域中的BOOST_STATIC_ASSERT

`BOOST_STATIC_ASSERT` 也可以用在函数作用域中。例如，考虑一个泛化的函数，它带有一个非类型模板参数，并且该参数只接受1至10的值。与其在运行期执行断言，我们不如在编译器使用静态断言。

```
template <int i> void accepts_values_between_1_and_10() {
  BOOST_STATIC_ASSERT(i>=1 && i<=10);
}
```

该函数的用户不能使用超出允许范围的数值来实例化这个函数。当然，断言中的表达式必须是一个纯粹的编译期表达式，也就是说，表达式中的参数和操作符都必须被编译器所认识。 `BOOST_STATIC_ASSERT` 当然并不是只能用于泛型函数；我们可以在任何函数中很方便地测试条件。例如，一个函数需要一个与平台相关的前提条件，就常常需要一个断言。

```
void expects_ints_to_be_4_bytes() {
  BOOST_STATIC_ASSERT(sizeof(int)==4);
}
```

总结

你所看到的这种静态断言在C++中正变得象运行期断言 `assert` 那样常用。这应该至少部分地归功于"元编程革命"，它使得一个程序中更多的计算量在编译期执行。表达编译期断言的唯一方法就是让编译器产生一个错误。为了让断言可用，错误提示必须可以传达有用的信息，但这很难做到可移植(事实上，根本不可能做到)。这正是 `BOOST_STATIC_ASSERT` 所要做的，它在大多数的编译器下提供了编译期断言的一致输出。它可用于名字空间、类、函数以及作用域。

以下情形下使用 `BOOST_STATIC_ASSERT` ：

- 当条件可以在编译期进行求值
- 对类型的要求可以在编译期表示
- 你需要对两个或以上的整型常量间的关系进行断言

checked_delete

头文件: `"boost/checked_delete.hpp"`

通过指针来删除一个对象时，执行的结果取决于执行删除时被删除的类型是否可知。对一个指向不完整类型的指针执行 `delete` 几乎不可能有编译器警告，这会导致各种各样的麻烦，由于析构函数可以没有被执行。换句话说，即进行清除的代码没有被执行。 `checked_delete` 在对象析构时执行一个静态断言，测试类是否可知，以确保析构函数被执行。

用法

`checked_delete` 是一个 `boost` 名字空间中的模板函数。它用于删除动态分配的对象，对于动态分配的数组，同样有一个称为 `checked_array_delete` 的模板函数。这些函数接受一个参数：要删除的指针，或是要删除的数组。这两个函数都要求在销毁对象时(即对象被传给函数时)，这些被删除的类型必须是可知的。使用这些函数，要包含头文件 `"boost/checked_delete.hpp"` . 使用这些函数时，你只需象调用 `delete` 那样简单地调用它们。以下程序前向声明了一个类 `some_class` , 而没有定义它。有些编译器允许对一个指向 `some_class` 的指针被删除(稍后再讨论这个)，但使用 `checked_delete` 后，就不能通过编译了，除非有一个 `some_class` 的定义。

```
#include "boost/checked_delete.hpp"

class some_class;

some_class* create() {
  return (some_class*)0;
}

int main() {
  some_class* p=create();
  boost::checked_delete(p2);
}
```

如果你试图编译这段代码，对函数 `checked_delete&lt;some_class&gt;` 的实例化将失败，因为 `some_class` 是一个不完整的类型。你的编译器会输出类似下面的信息：

```
checked_delete.hpp: In function 'void
boost::checked_delete(T*) [with T = some_class]':
checked_sample.cpp:11:   instantiated from here
boost/checked_delete.hpp:34: error: invalid application of 'sizeof' to an incomplete type
boost/checked_delete.hpp:34: error: creating array with
size zero ('-1')
boost/checked_delete.hpp:35: error: invalid application of
'sizeof' to an incomplete type
boost/checked_delete.hpp:35: error: creating array with
size zero ('-1')
boost/checked_delete.hpp:32: warning: 'x' has incomplete type
```

错误信息的前面部分清楚地说明了问题：`checked_delete` 遇到了一个不完整的类型。但我们的代码中哪里存在不完整的类型呢？接下来的章节我们来讨论它。

究竟是什么问题？

在我们深入了解 `checked_delete` 的好处之前，让我们先来彻底弄清楚问题所在。如果你试图删除一个指针，而该指针指向的是一个带有非平凡析构函数[4]的不完整类型[3]，结果将是未定义的行为。这是如何发生的呢？让我们来看一个例子。

> [3] 不完整的类型是指已声明但未定义的类型。
>
> [4] 标准说法是，类的一个或多个直接基类，或者一个或多个非静态数据成员，具有用户定义的析构函数。

```
// deleter.h
class to_be_deleted;

class deleter {
public:
  void delete_it(to_be_deleted* p);
};

// deleter.cpp
#include "deleter.h"

void deleter::delete_it(to_be_deleted* p) {
  delete p;
}

// to_be_deleted.h
#include <iostream>
class to_be_deleted
{
public:
  ~to_be_deleted() {
    std::cout <<
      "I'd like to say important things here, please.";
  }
};

// Test application
#include "deleter.h"
#include "to_be_deleted.h"

int main() {
  to_be_deleted* p=new to_be_deleted;

  deleter d;
  d.delete_it(p);
}
```

以上代码试图 `delete` 一个指向不完整类型 `to_be_deleted` 的指针，这会导致未定义行为。注意，`to_be_deleted` 在 `deleter.h` 中是前向声明的；`deleter.cpp` 包含了 `deleter.h` 而没有包含 `to_be_deleted.h`：而 `to_be_deleted.h` 中为 `to_be_deleted` 定义了一个非平凡析构函数。这种麻烦很容易出现，尤其是在使用智能指针的时候。我们要做的就是在调用 `delete` 时确认类型是完整的，这正是 `checked_delete` 所做的。

checked_delete 来解决问题

前面的例子说明了删除不完整类型时不进行确认很可能会引起麻烦，而且不是所有编译器会对此给出警告。编写泛型代码时，避免这种情况是非常必要的。使用 `checked_delete` 重写这个例子，你只需要把 `delete p` 改为 `checked_delete(p)`.

```
void deleter::do_it(to_be_deleted* p) {
  boost::checked_delete(p);
}
```

`checked_delete` 基本上就是一个判断类是否完整的断言，它的实现如下：

```
template< typename T > inline void checked_delete(T * x) {
  typedef char type_must_be_complete[sizeof(T)];
  delete x;
}
```

这里的想法是创建一个 `char` 的数组，数组的元素数量为 `T` 的大小。如果 `checked_delete` 被一个不完整的类型 `T` 所实例化，编译将会失败，因为 `sizeof(T)` 会返回 0, 而创建一个0个元素的(自动)数组是非法的。你也可以用 `BOOST_STATIC_ASSERT` 来执行这个断言。

```
BOOST_STATIC_ASSERT(sizeof(T));
```

在编写要求使用完整类型进行实例化的模板时，这个工具非常方便。对于数组，也有一个相应的"checked deleter"，称为 `checked_array_delete`, 它的用法类似于 `checked_delete`.

```
to_be_deleted* p=new to_be_deleted[10];
boost::checked_array_delete(p);
```

总结

删除一个动态分配的对象时，必须调用它的析构函数。如果这个类型是不完整的，即只有声明没有定义，那么析构函数可能会没被调用。这是一种潜在的危险状态，所以应该避免它。对于类模板及函数模板，风险会更大，因为无法预先知道会使用什么类型。使用 `checked_delete` 和 `checked_array_delete`, 可以解决这个删除不完整类型的问题。它没有运行期的额外开销，只是直接调用 `delete`, 因此说 `checked_delete` 带来的安全性实际上是免费的。

如果你需要在调用 `delete` 时确保类型是完整的，就使用 `checked_delete` 。

noncopyable

头文件: `"boost/utility.hpp"`

通常编译器都是程序员的好朋友，但并不总是。它的好处之一在于它会自动为我们提供复制构造函数和赋值操作符，如果我们决定不自己动手去 做的话。这也可能会导致一些不愉快的惊讶，如果这个类本身就不想被复制(或被赋值)。如果真是这样，我们就需要明确地告诉这个类的使用者复制构造以及赋值 是被禁止的。我不是说在代码中进行注释说明，而是说要禁止对复制构造函数以及赋值操作符的访问。幸运的是，当类 带有不能复制或不能赋值的基类或成员函数 时，编译器生成的复制构造函数及赋值操作符就不能使用。`boost::noncopyable` 的工作原理就是禁止访问它的复制构造函数和赋值操作符，然后使用它作为基类。

用法

要使用 `boost::noncopyable`，你要从它私有地派生出不可复制类。虽然公有继承也可以，但这是一个坏习惯。公有继承对于阅读类声明的人而言，意味着IS-A (表示派生类IS-A 基类)关系，但表明一个类IS-A `noncopyable` 看起来有点不太对。要从 `noncopyable` 派生，就要包含 `"boost/utility.hpp"` 。

```
#include "boost/utility.hpp"

class please_dont_make_copies : boost::noncopyable {};

int main() {
  please_dont_make_copies d1;
  please_dont_make_copies d2(d1);
  please_dont_make_copies d3;
  d3=d1;
  }
```

这个例子不能通过编译。由于 `noncopyable` 的复制构造函数是私有的，因此对 `d2` 进行复制构造的尝试会失败。同样，由于 `noncopyable` 的赋值操作符也是私有的，因此将 `d1` 赋值给 `d3` 的尝试也会失败。编译器会给出类似下面的输出：

```
noncopyable.hpp: In copy constructor
' please_dont_make_copies::please_dont_make_copies (const please_dont_make_copies&)':
boost/noncopyable.hpp:27: error: '
  boost::noncopyable::noncopyable(const boost::noncopyable&)' is
private
noncopyable.cpp:8: error: within this context
boost/noncopyable.hpp: In member function 'please_dont_make_copies&
  please_dont_make_copies::operator=(const please_dont_make_copies&)':
boost/noncopyable.hpp:28: error: 'const boost::noncopyable&
  boost::noncopyable::operator=(const boost::noncopyable&)' is private
noncopyable.cpp:10: error: within this context
```

下一节我们将测试这是如何工作的。很清楚从 `noncopyable` 派生将禁止复制和赋值。这也可以通过把复制构造函数和赋值操作符定义为私有的来实现。 我们来看一下怎么样做。

使类不能复制

再看一下类 `please_dont_make_copies`，为了某些原因，它不能被复制。

```
class please_dont_make_copies {
public:
  void do_stuff() {
    std::cout <<
      "Dear client, would you please refrain from copying me?";
  }
};
```

由于编译器生成了复制构造函数和赋值操作符，所以现在不能禁止类的复制和赋值。

```
please_dont_make_copies p1;
please_dont_make_copies p2(p1);
please_dont_make_copies p3;
p3=p2;
```

解决的方法是把复制构造函数和赋值操作符声明为私有的或是保护的，并增加一个缺省构造函数(因为编译器不再自动生成它了)。

```
class please_dont_make_copies {
public:
  please_dont_make_copies() {}

  void do_stuff() {
    std::cout <<
      "Dear client, would you please refrain from copying me?";
  }
private:
  please_dont_make_copies(const please_dont_make_copies&);
  please_dont_make_copies& operator=
    (const please_dont_make_copies&);
};
```

这可以很好地工作，但它不能马上清晰地告诉 `please_dont_make_copies` 的使用者它是不能复制的。下面看一下换成 `noncopyable` 后，如何使得类更清楚地表明不能复制，并且也可以打更少的字。

用 **noncopyable**

类 `boost::noncopyable` 被规定为作为私有基类来使用，它可以有效地关闭复制构造和赋值操作。用前面的例子来看看使用 `noncopyable` 后代码是什么样子的：

```
#include "boost/utility.hpp"

class please_dont_make_copies : boost::noncopyable {
public:
  void do_stuff() {
    std::cout << "Dear client, you just cannot copy me!";
 }
};
```

不再需要声明复制构造函数或赋值操作符。由于我们是从 `noncopyable` 派生而来的，编译器不会再生成它们了，这样就禁止了复制和赋值。简洁可以带来清晰，尤其是象这样的基本且清楚的概念。对于阅读这段代码的使用者来说，马上就清楚地知道这个类是不能复制和赋值的，因为 `boost::noncopyable` 在类定义的一开始就出现了。最后要提醒的一点是：你还记得类的缺省访问控制是私有的吗？这意味着缺省上继承也是私有的。你也可以象这样写，来更加明确这个事实：

```
class please_dont_make_copies : private boost::noncopyable {
```

这完全取决于观众；有些程序员认为这种多余的信息是令人讨厌并且会分散注意力，而另一些程序员则认同这种清晰性。由你来决定哪一种方法适合你的类和你的程序员。无论哪一种方法，使用 `noncopyable` 都要比"忘记"复制构造函数和赋值操作符的方法更加明确，也比私有地声明它们更为清晰。

记住 the Big Three

正如我们看到的那样，`noncopyable` 为禁止类的复制和赋值提供了一个方便的办法。但何时我们需要这样做呢？什么情况下我们需要自定义复制构造函数或赋值操作符？这个问题有一个通用的答案，一 个几乎总是正确的答案：无论何时你需要定义析构函数、复制构造函数、或赋值操作符三个中的任意一个，你也需要定义另外两个[5]。它们三者间的互动性非常重要，其中一个存在，其它的通常也都必须要有。我们假设你的一个类有一个成员是指针。你定义了一个析构函数用于正确地释放空间，但你没有定义复制构造函数和赋值操作符。这意味着你的代码中至少存在两个潜在的危险，它们很容易被触发。

> [5] 这个定律的名字叫the Big Three，来自于C++ FAQs (详情请见参考书目[2])。

```
class full_of_errors {
  int* value_;
public:
  full_of_errors() {
    value_=new int(13);
  }

 ~full_of_errors() {
    delete value_;
  }
};
```

使用这个类时，如果你忽视了编译器为这个类生成的复制构造函数和赋值操作符，那么至少有三种情况会产生错误。

```
full_of_errors f1;
full_of_errors f2(f1);
full_of_errors f3=f2;
full_of_errors f4;
f4=f3;
```

注意，第二行和第三行是调用复制构造函数的两个等价的方法。它们都会调用生成的复制构造函数，虽然语法有所不同。最后一个错误在最后一行，赋值操作符使得同一个指针被至少两个 `full_of_errors` 实例所删除。正确的方法是，我们需要自己的复制构造函数和赋值操作符，因为我们定义了我们自己的析构函数。以下是正确的方法：

```
  class not_full_of_errors {
    int* value_;
  public:
    not_full_of_errors() {
      value_=new int(13);
    }

    not_full_of_errors(const not_full_of_errors& other) :
      value_(new int(*other.value_)) {}

    not_full_of_errors& operator=
      (const not_full_of_errors& other) {
      *value_=*other.value_;
      return *this;
    }

    ~not_full_of_errors() {
      delete value_;
    }
};
```

所以，无论何时，一个类的the big three：复制构造函数、(虚拟)析构函数、和赋值操作符，中的任何一个被手工定义，在决定不需要定义其余两个之前必须认真仔细地考虑清楚。还有，如果你不想要复制，记得使用 `boost::noncopyable` ！

总结

有很多类型需要禁止复制和赋值。但是，我们经常忽略了把这些类型的复制构造函数和赋值操作符声明为私 有的，而把责任转嫁给了类的使用者。即使你使用了私有的复制构造函数和赋值操作符来确保它们不被复制或赋值，但是对于使用者而言这还不够清楚。当然，编译 器会友好地提醒试图这们做的人，但错误来自何处也不是清晰的。最好我们可以清晰地做到这一点，而从 `noncopyable` 派生就是一个清晰的声明。当你看一眼类型的声明就可以马上知道了。编译的时候，错误信息总会包含名字 noncopyable. 而且它也节省了一些打字，这对于某些人而言是关键的因素。

以下情形下使用 `noncopyable` ：

- 类型的复制和赋值都不被允许
- 复制和赋值的禁止应该尽可能明显

addressof

头文件: `"boost/utility.hpp"`

要取得一个变量的地址，我们要依赖于返回的值是否真的是这个变量的地址。但是，技术上重载 `operator&` 是有可能的，这意味着存有恶意的人可以破坏你的地址相关的代码。 `boost::addressof` 被用于获得变量的地址，不管取址操作符是否被误用。通过使用一些灵巧的内部机制，模板函数 `addressof` 确保可以获得真实的对象及其地址。

用法

为确保获得一个对象的真实地址，你要使用 `boost::addressof` . 它定义在 `"boost/utility.hpp"` . 它常用于原本要使用 `operator&` 的地方，它接受一个参数，该参数为要获得地址的那个对象的引用。

```
#include "boost/utility.hpp"

class some_class {};

int main() {
  some_class s;
  some_class* p=boost::addressof(s);
}
```

在进一步学习如何使用 `addressof` 的细节前，了解一下 `operator&` 为何以及如何不一定会返回对象的地址是非常有用的。

快速了解一下存有恶意的人

如果你真的，真的，真的需要重载 `operator&` , 或者只是想试验一下操作符重载可能的用法，这的确很容易。当你重载 `operator&` 时，它的语义肯定会与多数用户(以及函数！)所期望的不同，所以千万不要为了好玩而做这件事；除非有非常好的理由，否则不要去做它。以下有一段code-breaker代码：

```
class codebreaker {
public:
  int operator&() const {
    return 13;
  }
};
```

对于这个类，任何人想获取一个 `codebreaker` 实例的地址都会得到一个不可思议的数字13.

```
template <typename T> void print_address(const T& t) {
  std::cout << "Address: " << (&t) << '\n';
}

int main() {
  codebreaker c;
  print_address(c);
}
```

这不难做到，但是在实际的代码中这样做有没有好的理由？也许没有，因为除非是用在局部的类上，否则它是不安全的。原因是，虽然获取一个不完整类型的地址是合法的，但如果是要获取一个带有用户自定义 `operator&` 的不完整类型的地址则是未定义的行为。因为我们不能保证这不会发生，所以我们最好不要重载 `operator&` .

迅速的解决方法

即使一个类的 `operator&` 被重载了，也还是有办法获得这个类的实例的真实地址。 `addressof` 使用了一些幕后的巧妙方法[6]来获得真实的地址，而不会受任何 `operator&` 的欺骗。如果你把函数(`print_address`)改为使用 `addressof` , 你就可以得到以下代码：

> [6] 非常出名的一种ingenious hack.

```
template <typename T> void print_address(const T& t) {
  std::cout << "&t: " << (&t) << '\n';
  std::cout << "addressof(t): " << boost::addressof(t) << '\n';
}
```

执行时，该函数将给出如下输出(或类似于以下的输出，因为准确的地址值取决于你的系统).

```
&t: 13
addressof(t): 0012FECB13
```

差不多就是这样了！如果有什么情况让你知道或怀疑一个类的 `operator&` 被重载了，而你又需要确保得到真实的地址(由于 `operator&` 被重载而变得不可信了), 你就应该使用 `addressof` .

总结

没有多少有力的论点支持重载 `operator&` ,[7] 但由于这是可能的，总有些人会这样做。当你编写一些需要依赖于获得对象真实地址的代码时， `addressof` 可以帮助你确保得到真实的地址。在编写泛型代码时，没有办法知道将会操作什么类型，因此如果需要获取参数化类型的地址的话，就使用 `addressof` .

> [7] 即使是定制的硬件设备驱动程序

当你需要获得一个对象的真实地址时，使用 `addressof` ，不必管 `operator&` 的语义。

enable_if

头文件: `"boost/utility/enable_if.hpp"`

有时候，我们希望控制某个函数或类模板的特化是否可以加入到重载决议时使用的重载或特化的集合中。例如，考虑一个重载的函数，它有一个版本是带一个 `int` 参数的普通函数，另一个版本是一个函数模板，它要求参数类型 `T` 具有一个名为 `type` 的嵌套类型。它们看起来可能象这样：

```
void some_func(int i) {
  std::cout << "void some_func(" << i << ")\n";
}

template <typename T> void some_func(T t) {
  typename T::type variable_of_nested_type;
  std::cout <<
    "template <typename T> void some_func(" << t << ")\n";
}
```

现在，想象一下当你在代码中调用 `some_func` 将发生什么。如果参数的类型为 `int`，第一个版本将被调用。如果参数的类型是 `int` 以外的其它类型，则第二个(模板)版本将被调用。

这没问题，只要这个类型有一个名为 `type` 的嵌套类型，但如果它没有，这段代码就不能通过编译。这会是一个问题吗？好的，考虑一下如果你用其它整数类型来调用，如 `short`，或 `char`，或 `unsigned long`，那么又会发生什么。

```
#include <iostream>

void some_func(int i) {
  std::cout << "void some_func(" << i << ")\n";
}

template <typename T> void some_func(T t) {
  typename T::type variable_of_nested_type;
  std::cout <<
    "template <typename T> void some_func(" << t << ")\n";
}

int main() {
  int i=12;
  short s=12;

  some_func(i);
  some_func(s);
}
```

编译这段程序时，你将从失败的编译器中得到类似以下的输出：

```
enable_if_sample1.cpp: In function 'void some_func(T)
  [with T = short int]':
enable_if_sample1.cpp:17:   instantiated from here
enable_if_sample1.cpp:8: error:
  'short int' is not a class, struct, or union type

Compilation exited abnormally with code 1 at Sat Mar 06 14:30:08
```

就是这样。`some_func` 的模板版本被选为最佳的重载，但这个版本中的代码对于类型 `short` 而言是无效的。我们怎样才能避免它呢？好的，我们希望仅对含有名为type的嵌套类型的类使用模板版本的 `some_func` ，而对于其它没有这个嵌套类型的类则忽略它。我们能够做到。最简单的方法，但不一定是实际中总能使用的方法，是把模板版本的返回类型改为如下：

```
template <typename T> typename T::type* some_func(T t) {
  typename T::type variable_of_nested_type;
  std::cout <<
    "template <typename T> void some_func(" << t << ")\n";
  return 0;
}
```

如果你没有学过 SFINAE (匹配失败不是错误),[8] 很可能现在你的脸上会有困惑的表情。编译修改过的代码，我们的例子会通过编译。`short` 被提升为 `int` , 并且第一个版本被调用。这种令人惊奇的行为的原因是模板版本的 `some_func` 不再包含在重载决议的集合内了。它被排除在内是因为，编译器看到了这个函数的返回类型要求模板类型 `T` 要有一个嵌套类型 `type` ，而它知道 `short` 不满足这个要求，所以它把这个函数模板从重载决议集合中删掉了。这就是 Daveed Vandevorde 和 Nicolai Josuttis 教给我们的 SFINAE, 它意味着宁可对有问题的类型不考虑函数的重载，也不要产生一个编译器错误。如果类型有一个符合条件的嵌套类型，那么它就是重载决议集合的一部分。

> [8] 见参考书目[3]。

```
class some_class {
public:
  typedef int type;
};

int main() {
  int i=12;
  short s=12;

  some_func(i);
  some_func(s);
  some_func(some_class());
}
```

运行该程序的输出如下：

```
void some_func(12)
void some_func(12)
template <typename T> void some_func(T t)
```

这种办法可以用，但它不太好看。在这种情形下，我们可以不管原来的 `void` 返回类型，我们可以用其它类型替换它。但如果不是这种情形，我们就要给函数增加一个参数并给它指定一个缺省值。

```
template <typename T>
  void some_func(T t,typename T::type* p=0) {
  typename T::type variable_of_nested_type;
  std::cout << "template <typename T> void some_func(T t)\n";
}
```

这个版本也是使用 SFINAE 来让自己不会被无效类型所使用。这两种解决方案的问题都在于它们有点难看，我们把它们弄成了公开接口的一部分，并且它们只能在某些情形下使用。Boost 提供了一个更干净的解决方法，这种方法不仅在语法上更好看，而且提供了比前面的解决方法更多的功能。

用法

要使用 `enable_if` 和 `disable_if`, 就要包含头文件 `"boost/utility/enable_if.hpp"`. 在第一个例子中，我们将禁止第二个版本的 `some_func` ，如果参数的类型是整型的话。象一个类型是否整型这样的类型信息可以用另一个Boost库 `Boost.Type_traits` 来取得。 `enable_if` 和 `disable_if` 模板都通过接受一个谓词来控制是否启用或禁止一个函数。

```
#include <iostream>
#include "boost/utility/enable_if.hpp"
#include "boost/type_traits.hpp"

void some_func(int i) {
  std::cout << "void some_func(" << i << ")\n";
}

template <typename T> void some_func(
  T t,typename boost::disable_if<
    boost::is_integral<T> >::type* p=0) {
    typename T::type variable_of_nested_type;
    std::cout << "template <typename T> void some_func(T t)\n";
}
```

虽然这看起来与我们前面所做的差不多，但它表达了一些我们使用直接的方法所不能表达的东西，而且它在函数的声明中表达了关于这个函数的重要信息。看到这些，我们可以清楚的知道这个函数要求类型 `T` 不能是一个整数类型。如果我们希望仅对含有嵌套类型 `type` 的类型启用这个函数，它也可以做得更好，而且我们还可以用另一个库Boost.Mpl[9] 来做。如下：

> [9] Boost.Mpl 超出了本书的范围。访问 http://www.boost.org 获得更多关于Mpl的信息。另外，也可以看一下David Abrahams 和 Aleksey Gurtovoy 的书, C++ Template Metaprogramming!

```
#include <iostream>
#include "boost/utility/enable_if.hpp"
#include "boost/type_traits.hpp"
#include "boost/mpl/has_xxx.hpp"

BOOST_MPL_HAS_XXX_TRAIT_DEF(type)

void some_func(int i) {
  std::cout << "void some_func(" << i << ")\n";
}

template <typename T> void some_func(T t,
  typename boost::enable_if<has_type<T> >::type* p=0) {
    typename T::type variable_of_nested_type;
    std::cout << "template <typename T> void some_func(T t)\n";
}
```

这真的很酷！我们现在可以对没有嵌套类型 `type` 的 `T` 禁用 `some_func` 的模板版本了，而且我们清晰地表达了这个函数的要求。这里的窍门在于使用了Boost.Mpl的一个非常漂亮的特性，它可以测试任意类型 `T` 是否内嵌有某个指定类型。通过使用宏 `BOOST_MPL_HAS_XXX_TRAIT_DEF(type)`，我们定义了一个名为 `has_type` 的新的trait，我们可以在函数 `some_func` 中使用它作为 `enable_if` 的谓词。如果谓词为 `True`，这个函数就是重载决议集合中的一员；如果谓词为 `false`，这个函数就将被排除。

也可以包装返回类型，而不用增加一个额外的(缺省)参数。我们最后一个也是最好的一个 `some_func`，在它的返回类型中使用 `enable_if`，如下：

```
template <typename T> typename
boost::enable_if<has_type<T>,void>::type
  some_func(T t) {
    typename T::type variable_of_nested_type;
    std::cout << "template <typename T> void some_func(T t)\n";
}
```

如果你需要返回你想启用或禁用的类型，那么在返回类型中使用 `enable_if` 和 `disable_if` 会比增加一个缺省参数更合适。另外，有可能有的人真的为缺省参数指定一个值，那样就会破坏这段代码。有时，类模板的特化也需要被允许或被禁止，这时也可以使用 `enable_if` / `disable_if`。不同的是，对于类模板，我们需要对主模板进行一些特别的处理：增加一个模板参数。考虑一个带有返回一个 `int` 的成员函数 `max` 的类模板：

```
template <typename T> class some_class {
public:
  int max() const {
    std::cout << "some_class::max() for the primary template\n";
    return std::numeric_limits<int>::max();
  }
};
```

假设我们决定对于所有算术类型(整数类型及浮点数类型)，给出一个特化版本的定义，`max` 返回的是该算术类型可以表示的最大值。那么我们需要对模板类型 `T` 使用 `std::numeric_limits`，而对其它类型我们还是使用主模板。要做到这样，我们必须给主模

板加一个模板参数，该参数的缺省类型为 `void` (这意味着用户不需要显式地给出该参数)。结果主模板的定义如下：

```
template <typename T,typename Enable=void> class some_class {
public:
  int max() const {
    std::cout << "some_class::max() for the primary template\n";
    return std::numeric_limits<int>::max();
  }
};
```

现在我们已经为提供特化版本作好了准备，该特化版本为算术类型所启用。该特性可通过 Boost.Type_traits 库获得。以下是特化版本：

```
template <typename T> class some_class<T,
  typename boost::enable_if<boost::is_arithmetic<T> >::type> {
public:
  T max() const {
   std::cout << "some_class::max() with an arithmetic type\n";
   return std::numeric_limits<T>::max();
  }
};
```

该版本只有当实例化所用的类型为算术类型时才会启用，这时特性 `is_arithmetic` 为 `true`. 它可以正常工作是因为 `boost::enable_if&lt;false&gt;::type` 是 `void`, 会匹配到主模板。以下程序用不同的类型测试这个模板：

```
#include <iostream>
#include <string>
#include <limits>
#include "boost/utility/enable_if.hpp"
#include "boost/type_traits.hpp"

// Definition of the template some_class omitted

int main() {
  std::cout << "Max for std::string: " <<
    some_class<std::string>().max() << '\n';
  std::cout << "Max for void: " <<
    some_class<void>().max() << '\n';
  std::cout << "Max for short: " <<
    some_class<short>().max() << '\n';
  std::cout << "Max for int: " <<
    some_class<int>().max() << '\n';
  std::cout << "Max for long: " <<
    some_class<long>().max() << '\n';
  std::cout << "Max for double: " <<
    some_class<double>().max() << '\n';
}
```

我们预期前两个 `some_class` 会实例化主模板，剩下的将会实例化算术类型的特化版本。运行该程序可以看到的确如此。

```
some_class::max() for the primary template
Max for std::string: 2147483647
some_class::max() for the primary template
Max for void: 2147483647
some_class::max() with an arithmetic type
Max for short: 32767
some_class::max() with an arithmetic type
Max for int: 2147483647
some_class::max() with an arithmetic type
Max for long: 2147483647
some_class::max() with an arithmetic type
Max for double: 1.79769e+308
```

一切正常！以前，要允许或禁止重载函数和模板特化需要一些编程的技巧，多数看到代码的人都不能完全明白。通过使用 `enable_if` 和 `disable_if`，代码变得更容易写也更容易读了，并且可以从声明中自动获得正确的类型要求。在前面的例子中，我们使用了模板 `enable_if`，它要求其中的条件要有一个名为 `value` 的嵌套定义。对于多数可用于元编程的类型而言这都是成立的，但对于整型常量表达式则不然。如果没有名为 `value` 的嵌套类型，就要使用 `enable_if_c` 来代替，它接受一个整型常量表达式。使用 `is_arithmetic` 并直接取出它的值，我们可以这样重写 `some_class` 的启用条件：

```
template <typename T> class some_class<T,
  typename boost::enable_if_c<
    boost::is_arithmetic<T>::value>::type> {
public:
  T max() const {
   std::cout << "some_class::max() with an arithmetic type\n";
   return std::numeric_limits<T>::max();
  }
};
```

`enable_if` 和 `enable_if_c` 原则上并没有不同。它们的区别仅在于是否要求有嵌套类型 value。

总结

被称为SFINAE的C++语言特性是很重要的。没有它，很多新的代码会破坏已有的代码，并且某些类型的函数重载(以及模板特化)将会无法实现。直接使用SFINAE来控制特定的函数或类型，使之被允许或被禁止用于重载决议，会很复杂。这样也会产生难以阅读的代码。使用 `boost::enable_if` 是更好的办法，它可以规定重载仅对某些特定类型有效。如果相同的参数用于 `disable_if`，则规定重载对于符合条件的类型无效。虽然使用SFINAE也可以实现，但该库可以更好地表达相关意图。本章忽略了 `enable_if` 和 `disable_if` 的lazy版本(名为 `lazy_enable_if` 和 `lazy_disable_if`)，不过我在这里简单地提及一下。lazy版本被用于避免实例化类型可能无效的情形(取决于条件的取值).

以下情形时使用 `enable_if` ：

- 你需要在把一个符合某些条件的函数加入到或排除出重载决议集合中。
- 你需要根据某个条件将一个类模板的特化版本加入到或排除出特化集合中。

Utility 总结

本章介绍了几种工具类，它们可以大大简化我们的日常工作。`BOOST_STATIC_ASSERT` 提供编译期断言，它有助于我们测试前提条件或强制某些要求。对于泛型编程，`checked_delete` 在检查错误用法时非常有用，它可以节省我们大量的阅读可怕的错误信息和研究代码的时间。我们还讨论了 `addressof`，它是一个获得对象真实地址的小工具，不用管 `operator&` 有否被重载。我们还看到了 `enable_if` 和 `disable_if` 如何控制某些函数参与重载决议，并学习了 SFINAE 有何意义！

我们也讨论了基类 `noncopyable`. 它既提供了好的习惯用法，也清楚地向任何看到这段代码的人表达了正确的意图，它值得你经常使用。总是在类中定义冗长的复制构造函数和赋值操作符，而不管它们是否需要定制，或是需要禁用，这种情况在很多代码中经常出现，它们浪费了太多的时间和金钱。

这是本书中最短的一章，我猜你一定很快就读完了。它很快就会给予你回报的，如果你马上开始使用这些工具的话。在Boost.Utility中还有一些其它的工具，我没在这里讨论它们。你可以访问Boost的网站，看看在线文档，找一下其它适合你当前工作的小工具。

Library 4. Operators

- Operators库如何改进你程序？
- Operators
- 用法
- Operators 总结

Operators库如何改进你的程序？

- 提供一组完整的比较操作符
- 提供一组完整的算术操作符
- 提供一组完整的迭代器操作符

C++定义的操作符可以分成几组。当你在一个类中碰到某组操作符中的一个，通常你还会碰到同组中的其它操作符。例如，如果一个类提供了 `operator==`，你通常还会看到 `operator!=`，或许还有 `operator<`，`operator<=`，`operator>`，和 `operator>=`. 有时，一个类仅提供了 `operator<` 以定义一个次序，这个类的对象就可以用于关联容器，但通常忘了类的使用者想要更多的操作符。同样地，一个具有值语义的类可以只提供了 `operator+` 而没有 `operator+=` 或 `operator-` 会限制它的使用。当你为你的类定义了一组操作符中的一个时，你应该也提供该组中其余的操作符，以避免令人惊讶。不幸的是，为一个类增加多个操作符以支持比 较或算术运算是很麻烦并且容易出错的，还有，迭代器类必须根据它所依照的迭代器种类来提供一组特定的操作符以确保其功能正确。

定义多个所需的操作符除了沉闷以外，还必须确保它们的语义符合用户的期望。否则，这个类将没有任何实际的用途。但我们可以无须全部依靠手工来实现它们。如你所知，某些操作符是根据其它操作符实现的，如 `operator+` 就可以参照 `operator+=` 实现，这意味着可以自动实现部分工作。事实上，这正是Boost.Operators的目的。它允许你只定义所需的比较或算术操作符的一个子集，然后基于你提供的操作符自动定义其它的操作符，Boost.Operators保证了正确的操作符语义，并减少了你犯错的机会。

Operators库的另一个好处在于为不同操作符给出了明确的概念命名，例如支持 `operator+` 和 `operator+=` 的类称为addable，支持 `operator<<` 和 `operator>>` 的类称为shiftable，等等。这很重要，有两个原因：一个统一的命名方法更为易懂；而且这些概念以及其后命名的类，可以是类接口的一部分，清晰地表明了重要的行为。

Operators如何配合标准库？

在使用标准库容器和算法时，通常至少要支持一些关系操作符(最常见的是 `operator<`) 以提供排序，从而可以在关联容器中按顺序存储对象。常见的惯例是仅定义所需操作符的最小集合，其副作用是类的定义不够完整，也更难理解。另一方面，如果定 义全部完整的操作符，就会有语义错误的风险。这种情况下，Operators库帮助我们确保类的行为是正确的，并同时满足标准库和用户的要求。最后，对于 定义了算术操作符的类型，也有一些操作符是依照其它操作符而实现的，所以Boost.Operators在这里也很有帮助。

Operators

头文件: `"boost/operators.hpp"`

Operators库由多个基类组成。每一个类生成与其名字概念相关的操作符。你可以用继承的方式来 使用它们，如果你需要一个以上的功能，则需要使用多重继承。幸运的是，Operators中定义了一些复合的概念，在大多数情况下可以无须使用多重继承。 下面将介绍最常用的一些Operator类，包括它们所表示的概念，以及它们对派生类的要求。某些情况下，使用Operators时，对真实概念的要求会 不同于对该概念基类的要求。例如，概念 addable 要求有一个操作符 `T operator+(const T& lhs,const T& rhs)` 的定义，而Operators的基类 `addable` 却要求有一个成员函数， `T operator+=(const T& other)` . 使用这个成员函数，基类 `addable` 为派生类自动增加了 `operator+` . 在以下章节中，都是首先给出概念，然后再给出对派生自该概念的类的要求。我没有重复本库中的所有概念，仅是挑选了最重要的一些；你可以在 www.boost.org 上找到完整的参考文档。

less_than_comparable

`less_than_comparable` 要求类型 `T` 具有以下语义。

```
bool operator<(const T&,const T&);
bool operator>(const T&,const T&);
bool operator<=(const T&,const T&);
bool operator>=(const T&,const T&);
```

要派生自 `boost::less_than_comparable` , 派生类(`T`)必须提供：

```
bool operator<(const T&, const T&);
```

注意，返回值的类型不必是真正的 `bool` , 但必须可以隐式转换为 `bool` . C++标准中的概念 LessThanComparable 要求提供 `operator<` ， 所以从 `less_than_comparable` 派生的类必须符合该要求。作为回报， `less_than_comparable` 将依照 `operator<` 实现其余的三个操作符。

equality_comparable

`equality_comparable` 要求类型 `T` 具有以下语义。

```
bool operator==(const T&,const T&);
bool operator!=(const T&,const T&);
```

要派生自 `boost::equality_comparable` , 派生类(`T`)必须提供：

```
bool operator==(const T&,const T&);
```

同样，返回值的类型不必是 `bool`，但必须可以隐式转换为 `bool`。C++标准中的概念 EqualityComparable 要求必须提供 `operator==`，因此从 `equality_comparable` 派生的类必须符合该要求。`equality_comparable` 类为 `T` 提供 `bool operator!=(const T&,const T&)`。

addable

`addable` 概念要求类型 `T` 具有以下语义。

```
T operator+(const T&,const T&);
T operator+=(const T&);
```

要派生自 `boost::addable`，派生类(`T`)必须提供：

```
T operator+=(const T&);
```

返回值的类型必须可以隐式转换为 `T`。类 `addable` 为 `T` 实现 `T operator+(const T&,const T&)`。

subtractable

`subtractable` 概念要求类型 `T` 具有以下语义。

```
T operator-(const T&,const T&);
T operator-=(const T&);  // 译注：原文为 T operator+=(const T&); 有误
```

要派生自 `boost::subtractable`，派生类(`T`)必须提供：

```
T operator-=(const T&,const T&);
```

返回值的类型必须可以隐式转换为 `T`。类 `addable` 为 `T` 实现 `T operator-(const T&,const T&)`。

orable

`orable` 概念要求类型 `T` 具有以下语义。

```
T operator|(const T&,const T&);
T operator|=(const T&,const T&);
```

要派生自 `boost::orable`，派生类(`T`)必须提供：

```
T operator|=(const T&,const T&);
```

返回值的类型必须可以隐式转换为 `T`。类 `addable` 为 `T` 实现 `T operator|(const T&,const T&)`。

andable

`andable` 概念要求类型 `T` 具有以下语义。

```
T operator&(const T&,const T&);
T operator&=(const T&,const T&);
```

要派生自 `boost::andable`，派生类(`T`)必须提供：

```
T operator&=(const T&,const T&);
```

返回值的类型必须可以隐式转换为 `T`。类 `addable` 为 `T` 实现 `T operator&(const T&,const T&)`。

incrementable

`incrementable` 概念要求类型 `T` 具有以下语义。

```
T& operator++(T&);
T operator++(T&,int);
```

要派生自 `boost::incrementable`，派生类(`T`)必须提供：

```
T& operator++(T&);
```

返回值的类型必须可以隐式转换为 `T`。类 `addable` 为 `T` 实现 `T operator++(T&,int)`。

decrementable

`decrementable` 概念要求类型 `T` 具有以下语义。

```
T& operator--(T&);
T operator--(T&,int);
```

要派生自 `boost::decrementable`，派生类(`T`)必须提供：

```
T& operator--(T&);
```

返回值的类型必须可以隐式转换为 `T`。类 `addable` 为 `T` 实现 `T operator--(T&,int)`。

equivalent

`equivalent` 概念要求类型 `T` 具有以下语义。

```
bool operator<(const T&,const T&);
bool operator==(const T&,const T&);
```

要派生自 `boost::equivalent`，派生类(`T`)必须提供：

```
bool operator<(const T&,const T&);
```

返回值的类型必须可以隐式转换为 `bool`。类 `equivalent` 为 `T` 实现 `T operator==(const T&,const T&)`。注意，等价(equivalence)和相等(equality)准确的说是不一样的；两个等价(equivalent)的对象并不一定是相等的(equal)。但对于这里的 `equivalent` 概念而言，它们是一样的。

解引用操作符

对于迭代器，有两个概念特别有用， `dereferenceable` 和 `indexable`，分别表示了解引用的两种情况：一个是 `*t`，`t` 是一个支持解引用的迭代器(显然所有迭代器都支持)，另一个是 indexing, `t[x]`，`t` 是一个支持下标操作符寻址的类型，而 `x` 通常是一个整数类型。在更高的抽象级别，它们两个通常一起使用，合称迭代器操作符，包括这两个解引用操作符和一些简单的算术操作符。

dereferenceable

`dereferenceable` 概念要求类型 `T` 具有以下语义，假设 `T` 是操作数， `R` 是引用类型，而 `P` 是指针类型(例如， `T` 是一个迭代器类型， `R` 是该迭代器的 `value_type` 的引用，而 `P` 则是该迭代器的 `value_type` 的指针)。

```
P operator->() const;
R operator*() const;
```

要派生自 `boost::dereferenceable`，派生类(`T`)必须提供：

```
R operator*() const;
```

另外， `R` 的一元 `operator&` 必须可以被隐式转换为 `P`。这意味着 `R` 不必一定要是引用类型，它可以是一个代理类(proxy class)。类 `dereferenceable` 为 `T` 实现 `P operator->() const`.

indexable

`indexable` 概念要求类型 `T` 具有以下语义，假设 `T` 是操作数， `R` 是引用类型， `P` 是指针类型，而 `D` 是 `difference_type` (例如， `T` 是一个迭代器类型， `R` 是该迭代器的 `value_type` 的引用， `P` 是该迭代器的 `value_type` 的指针，而 `D` 则是 `difference_type`)。

```
R operator[](D) const;
R operator+(const T&,D);
```

要派生自 `boost::indexable` , 派生类(`T`)必须提供：

```
R operator+(const T&,D);
```

类 `indexable` 为 `T` 实现 `R operator[](D) const` .

复合算术操作符

到目前为止我们看到的概念都只代表了最简单的功能。但是，还有一些高级的，或是复合的概念，它们由几个简单概念组合而成，或是在复合概念之上再增加简单的概念而成。例如，一个类是 `totally_ordered` 的，如果它同时是 `less_than_comparable` 的和 `equality_comparable` 的。这些组合很有用，因为它们减少了代码的数量，同时还表明了重要且常用的概念。由于它们只是表示了已有概念的组合，所以这些概念很容易用一个表格来表示它们所包含的简单概念。例如，如果一个类派生自 `totally_ordered` , 它必须实现 `less_than_comparable` 所要求的操作符(`bool operator&lt;(const T&,const T&)`) 和 `equality_comparable` 所要求的操作符(`bool operator==(const T&,const T&)`)。

组合概念	由以下概念组成
totally_ordered	less_than_comparableequality_comparable
additive	addablesubtractable
multiplicative	multipliabledividable
integer_multiplicative	multiplicativemodable
arithmetic	additivemultiplicative
integer_arithmetic	additiveinteger_multiplicative
bitwise	andableorablexorable
unit_steppable	incrementabledecrementable
shiftable	left_shiftableright_shiftable
ring_operators	additivemultipliable
ordered_ring_operators	ring_operatorstotally_ordered
field_operators	ring_operatorsdividable
ordered_field_operators	field_operatorstotally_ordered
euclidian_ring_operators	ring_operatorsdividablemodable
ordered_ euclidian_ring_operators	euclidean_ring_operatorstotally_ordered

用法

为了开始使用Operators库，为你的类实现适用的操作符，就要包含头文件 `"boost/operators.hpp"`，并从一个或多个Operator基类(它们的名字与它们所表示的概念一样)进行派生，它们都定义在名字空间 `boost` 中。注意，继承不一定要是公有的，私有继承也可以。在这一节，我们将看到几个使用不同概念的例子，并关注一下在C++里以及在概念上，算术操作符和关系操作符是如何工作的。作为第一个例子，我们定义一个类， `some_class`，带有一个 `operator<`. 我们决定把 `operator<` 所暗指的等价关系定义为 `operator==`. 这个工作可以通过从 `boost::equivalent` 继承而完成。

```
#include <iostream>
#include "boost/operators.hpp"

class some_class : boost::equivalent<some_class> {
  int value_;
public:
  some_class(int value) : value_(value) {}

  bool less_than(const some_class& other) const {
    return value_<other.value_;
  }
};

bool operator<(const some_class& lhs, const some_class& rhs) {
  return lhs.less_than(rhs);
}

int main() {
  some_class s1(12);
  some_class s2(11);

  if (s1==s2)
    std::cout << "s1==s2\n";
  else
    std::cout << "s1!=s2\n";
}
```

`operator<` 依照成员函数 `less_than` 实现。 `equivalent` 基类的要求就是派生的类必须提供 `operator<` 。从 `equivalent` 派生时，我们要把派生类 `some_class` 作为模板参数传送。在 `main` 里，使用了Operators库为我们生成的 `operator==` 。接下来，我们再看一来 `operator<` ，看看其它的关系操作符如何依照 less than 实现。

对比较操作符的支持

我们最常实现的关系操作符就是less than，也就是 `operator<`. 为了要把对象存入关联容器，或者是为了要排序，我们都要提供它。然而，通常我们仅支持这一个操作符，这样会把类的使用者弄糊涂。例如，多数人知道对 `operator<` 的结果取反就相当于 `operator>=`. [1] Less than 也可以用来计算 greater than, 等等。所以，一个支持less than关系的类的使用

者有充足的理由相信，支持(至少隐式地支持)其它的比较操作符也应该是类的接口的一部分。唉，如果我们仅仅支持了 `operator<` 而忽略了其它的，这个类就不是它可以的或者它应该的那样有用了。这里有一个类，它已经可以用于标准库容器的排序程序。

> [1] 尽管也有很多人以为是 `operator>`！

```
class thing {
  std::string name_;
public:
  thing() {}
  explicit thing(const std::string& name):name_(name) {}

  friend bool operator<(const thing& lhs, const thing& rhs) {
    return lhs.name_<rhs.name_;
  }
};
```

这个类支持排序，也可以被存入关联容器中，但它可能还不能满足用户的期望！例如，如果一个用户需要知道 `thing a` 是否大于 `thing b`，他就要这样写：

```
// is a greater than b?
if (b<a) {}
```

虽然这段代码是正确的，但是它未能清晰地表达作者的意图，而这对于代码的正确性是很重要的。如果用户想知道 `a` 是否小于或等于 `b`，他不得不这样写：

```
// is a less than, or equal to, b?
if (!(b<a)) {}
```

同样，这段代码是正确的，但它会让人糊涂；对于多数不留意的读者来说，代码的意图真的很不清晰。如果要引入等价的概念，代码将变得更令人糊涂，而我们是支持等价关系的(否则我们的类不能存入关联容器中)。

```
// is a equivalent to b?
if (!(a<b) && !(b<a)) {}
```

请注意，等价和相等是不一样的，后面的章节将展开讨论这个主题。在C++中，前面所述的所有关系特性都有不同的表示方式，它们是通过不同的操作符来明确地进行测试的。前面的例子应该是象这样(等价关系可能是个例外，但我们在这先不管它)：

```
if (a>b) {}
if (a<=b) {}
if (a==b) {}
```

现在，注释是多余的了，因为代码已经讲清楚了一切。照这个样子，代码是不能编译的，因为 `thing` 类不支持 `operator>`，`operator<=`，或 `operator==`. 但是，对于具有 `less_than_comparable` 概念的类型，这些操作符(除了 `operator==`)都能被表达，Operators

库可以帮助我们。我们要做的全部工作就是让 `thing` 派生自 `boost::less_than_comparable`，如下：

```
class thing : boost::less_than_comparable<thing> {
```

仅仅通过指定一个基类，就可以依照 `operator<` 实现所有的操作符，`thing` 类现在可以按你所期望的那样工作了。如你所见，要从Operators库中的类派生出 `thing`，我们必须把 `thing` 作为模板参数传递给基类。这种技术将在后面的章节里讨论。注意，`operator==` 对于支持 `less_than_comparable` 的类并无定义，但我们还有一个概念可用，即 `equivalent`. 从 `boost::equivalent` 派生就可以增加 `operator==`, 但是要注意，这里的 `operator==` 是定义为等价关系，而不是相等关系。等价意味着严格的弱序关系[2]。我们最后版本的类 `thing` 看起来应该是这样的：

> [2] 如果你对严格弱序感到奇怪，可以跳到下一节，但是不要忘了稍后回到这里！

```
class thing :
  boost::less_than_comparable<thing>,
  boost::equivalent<thing> {

  std::string name_;
public:
  thing() {}
  explicit thing(const std::string& name):name_(name) {}

  friend bool operator<(const thing& lhs,const thing& rhs) {
    return lhs.name_<rhs.name_;
  }
};
```

这个版本在 `thing` 的定义中仅给出了一个操作符，保持了定义的简洁，依靠派生自 `less_than_comparable` 和 `equivalent`, 它提供了一整组有用的操作符。

```
bool operator<(const thing&,const thing&);
bool operator>(const thing&,const thing&);
bool operator<=(const thing&,const thing&);
bool operator>=(const thing&,const thing&);
bool operator==(const thing&,const thing&);
```

你肯定见过很多提供了多个操作符的类。这些类的定义很难阅读，由于有太多的操作符函数的声明/实现。通过从 `operators` 中的概念类派生，你提供了相同的接口，但做得更清楚也更少代码。在类的定义中提及这些概念，可以使熟悉 `less_than_comparable` 和 `equivalent` 的读者清楚地知道这个类支持这些关系操作符。

Barton-Nackman技巧

在前面两个例子中，我们看到从operator基类继承的方法，一个看起来怪怪的地方是，把派生类传给基类作为模板参数。这是一种著名的技巧，被称为 Barton-Nackmann 技巧[3] 或 Curiously Recurring Template Pattern[4]。这种技巧所解决的问题是循环的依赖性。考虑一下

实现一个泛型类，它为另一个定义了 operator< 的类提供 operator== 。顺便说一下，这就是这个库(当然还有mathematics库)中称为 equivalent 的概念。很明显，任何类要利用提供了这种服务的具体实现，它都要了解提供服务的这个类，我们以这个类所实现的概念来命名它，称之为 equivalent 类。然而，我们刚刚还在说 equivalent 要了解那个它要为之定义 operator== 的类！这是一种循环的依赖性，乍一看，好象没有办法可以解决。但是，如果我们把 equivalent 实现为类模板，然后指定那个要定义 operator== 的类为模板的参数，这样我们就已经有效地把相关类型，也即是那个派生类，加入到 equivalent 的作用域中了。以下例子示范了如何使用这个技巧。

> [3] 由John Barton 和 Lee Nackmann "发明"。
>
> [4] 由James Coplien"发明"。

```
#include <iostream>

template <typename Derived> class equivalent {
public:
  friend bool operator==(const Derived& lhs,const Derived& rhs) {
    return !(lhs<rhs) && !(rhs<lhs);
  }
};

class some_class : equivalent<some_class> {
  int value_;
public:
  some_class(int value) : value_(value) {}
  friend bool operator<(const some_class& lhs,
    const some_class& rhs) {
    return lhs.value_<rhs.value_;
  }
};

int main() {
  some_class s1(4);
  some_class s2(4);

  if (s1==s2)
    std::cout << "s1==s2\n";
}
```

基类 equivalent 接受一个要为之定义 operator== 的类型为模板参数。它通过使用 operator< 为该参数化类型实现泛型风格的 operator== 。然后，类 some_class 想要利用 equivalent 的服务，就从它派生并把自己作为模板参数传递给 equivalent 。因此，结果就是为类型 some_class 定义了 operator== ，是依照 some_class 的 operator< 实现的。这就是Barton-Nackmann技巧的全部内容。这是一种简单且非常有用的模式，相当优美。

严格弱序(Strick Weak Ordering)

在本书中，我已经两次提到了严格弱序(strict weak orderings)，如果你不熟悉它，本节将离开主题一会，给你解释一下。严格弱序是两个对象间的一种关系。首先我们来一点理论的讲解，然后再具体地讨论。一个函数 f(a,b) 如果实现了一种严格弱序关系，这里的 a 和 b 是同一类型的两个对象，我们说， a 和 b 是等价的，如果 f(a,b) 是 false 并且 f(b,a)

也是 false。这意味着 `a` 不在 `b` 之前，而且 `b` 也不在 `a` 之前。因此我们可以认为它们是等价的。此外，`f(a,a)` 必须总是 `false` [5]，而且如果 `f(a,b)` 为 `true`，则 `f(b,a)` 必须为 `false`.[6] 还有，如果 `f(a,b)` 与 `f(b,c)` 均为 `true`，则有 `f(a,c)`.[7] 最后，如果 `f(a,b)` 为 `false` 且 `f(b,a)` 也为 `false`，并且如果 `f(b,c)` 为 `false` 且 `f(c,b)` 也为 `false`，则 `f(a,c)` 为 `false` 且 `f(c,a)` 为 `false`.[8]

> [5] 即自反性。
>
> [6] 即反称性。
>
> [7] 即传递性。
>
> [8] 即等价关系的传递性。

我们前面的例子(类 `thing`)可以有助于澄清这个理论。`thing` 的小于比较是依照 `std::string` 的小于比较实现的。也就是说，是一种字面的比较。因此，给出一个包含字符串"First"的 `thing a`，和一个包含字符串"Second"的 `thing b`，还有一个包含字符串"Third"的 `thing c`，我们可以 `assert` 前面给出的定义和公理。

```
#include <cassert>
#include <string>
#include "boost/operators.hpp"

// Definition of class thing omitted

int main() {
  thing a("First");
  thing b("Second");
  thing c("Third");

  // assert that a<b<c
  assert(a<b && a<c && !(b<a) && b<c && !(c<a) && !(c<b));

  // 等价关系
  thing x=a;
  assert(!(x<a) && !(a<x));

  // 自反性
  assert(!(a<a));

  // 反对称性
  assert((a<b)==!(b<a));

  // 传递性
  assert(a<b && b<c && a<c);

  // 等价关系的传递性
  thing y=x;
  assert( (!(x<a) && !(a<x)) &&
    (!(y<x) && !(x<y)) &&
    (!(y<a) && !(a<y)));
}
```

现在，所有这些 `assert`s 都成立，因为 `std::string` 实现了严格弱序[9]。就象 `operator<` 可以定义一个严格弱序，`operator>` 也可以。稍后，我们将看到一个非常具体的例子，看看如果我们未能区分等价(它是一个严格弱序所要求的)与相等(它不是严格弱序所要求的)之间的不同，将会发生什么。

> [9] 事实上，`std::string` 定义了一个全序，全序即是严格弱序外加一个要求：等价即为相等。

避免对象膨胀

在前面的例子中，我们的类派生自两个基类：`less_than_comparable<thing>` 和 `equivalent<thing>`. 根据你所使用的编译器，你需要为这个多重继承付出一定的代价；`thing` 可能要比它所需的更大。标准允许编译器使用空类优化来创建一个没有数据成员、没有虚拟函数、也没有重复基类的基类，这样在派生类的对象中只会占用零空间， 而多数现代的编译器都会执行这种优化。不幸的是，使用Operators库常常会导致从多个基类进行继承，这种情况下只有很少编译器会使用空类优化。为了 避免潜在的对象大小膨胀，Operators支持一种称为基类链(base class chaining) 的技术。每个操作符类接受一个可选的额外的模板参数，该参数来自于它的派生类。采用以下方法：一个概念类派生自另一个，后者又派生自另一个，后者又派生自 另一个...(你应该明白了吧)，这样就不再需要多重继承了。这种方法很容易用。不要再从几个基类进行继承了，只要简单地把几个类链在一起就行了，如下所示：

```
// Before
boost::less_than_comparable<thing>,boost::equivalent<thing>
// After
boost::less_than_comparable<thing,boost::equivalent<thing> >
```

这种方法避免了从多个空基类进行继承，而从多个基类继承可能会阻碍你的编译器进行空类优化，使用从一 个空基类链进行继承的方法，增加了编译器进行空类优化的机会，也减少了派生类的大小。你可以用你的编译器做一下试验，看看你可以从这个技术中获得多少好 处。注意，基类链长度是有限制的，这取决于编译器。对于程序员可以接受的基类链长度也是很有限的！这意味着对那些需要从很多operator类进行继承的类来说，我们需要把它们组合起来。更好的方法是，使用Operators库已提供的复合概念。

以我的测试来看，在某个对多重继承不执行空类优化的流行编译器上[10]， 使用基类链和使用多重继承所得到的类的大小有很大的差别。使用基类链确实可以避免类型增大的负作用，而使用多重继承则会有类型的增大，对于一个普通类型， 大小将增加8个字节(无可否认，8个额外的字节对于多数应用来说并不是问题)。如果被包装的类型的大小非常小，那么多重继承带来的额外开销就不是可以接受 的了。由于基类链很容易使用，我们应该在所有情况下都使用它！

> [10] 我这样说一方面是因为没有必要讲出它的名字，另一方面也因为每个人都知道我说的是 Microsoft的老编译器 (他们的新编译器可能不是)。

Operators 与不同的类型

有时候，一个操作符要包括一个以上的类型。例如，考虑一个字符串类，它支持通过 `operator+` 和 `operator+=` 从字符数组进行字符串连接。这种情况下，Operators库也可以帮忙，使用双参数版本的操作符模板。这个字符串类可能拥有一个接受 `char*` 的转换构造函数，但正如我们将看到的，这不能解决这个类的所有问题。以下是我们要用的字符串类。

```
class simple_string {
public:
  simple_string();

  explicit simple_string(const char* s);
  simple_string(const simple_string& s);

  ~simple_string();

  simple_string& operator=(const simple_string& s);

  simple_string& operator+=(const simple_string& s);
  simple_string& operator+=(const char* s);

  friend std::ostream&
    operator<<(std::ostream& os,const simple_string& s);
};
```

如你所见，我们为 `simple_string` 增加两个版本的 `operator+=` 。一个接受 `const simple_string&` ，另一个接受 `const char*` . 这样，我们的类支持如下用法：

```
simple_string s1("Hello there");
simple_string s2(", do you like the concatenation support?");
s1+=s2;
s1+=" This works, too";
```

虽然前面的工作符合要求，但我们还没有提供二元的 `operator+` ，这个疏忽肯定是类的使用者所不乐意的。注意，对于我们的 `simple_string` ，我们可以通过忽略显式的转换构造函数来允许字符串连接。但是，这样做会导致对字符缓冲的一次额外(不必要)的复制，而仅仅节省了一个操作符的定义。

```
// 以下不能编译
simple_string s3=s1+s2;
simple_string s4=s3+" Why does this class behave so strangely?";
```

现在让我们来用Operators库来为这个类提供漏掉的操作符。注意共有三个操作符没有提供。

```
simple_string operator+(const simple_string&,const simple_string&);
simple_string operator+(const simple_string& lhs, const char* rhs);
simple_string operator+(const char* lhs, const simple_string& rhs);
```

如果手工定义这些操作符，很容易就会忘记那个接受一个 `const simple_string&` 和一个 `const char*` 的重载。而使用Operators库，你就不会忘记了，因为库已经为你实现了这些漏掉的操作符！我们想为 `simple_string` 做的就是加一个 addable 概念，所以我们只要简单地从 `boost::addable<simple_string>` 派生 `simple_string` 就行了。

```
class simple_string : boost::addable<simple_string> {
```

但是，在这个例子中，我们还想要一个允许 `simple_string` 和 `const char*` 混合使用的操作符。为此，我们需要指定两个类型，结果类型是 `simple_string`，以及第二参数类型是 `const char*`. 我们可以利用基类链来避免增大类的大小。

```
class simple_string :
  boost::addable<simple_string,
    boost::addable2<simple_string,const char*> > {
```

这就是为了支持我们想要的全部操作符所要做的全部事情！如你所见，我们用了一个不同的operator类：`addable2`. 如果你用的编译器支持模板偏特化，你就不需要限定这个名字；你可以用 `addable` 代替 `addable2`. 为了对称性，还有一个版本的类，它带有后缀"1"。它可以增加可读性，它总是明确给出参数的数量，它带给我们以下对 `simple_string` 的派生写法：

```
class simple_string :
  boost::addable1<simple_string,
    boost::addable2<simple_string,const char*> > {
```

选择哪种写法，完全取决于你的品味，如果你的编译器支持模板偏特化，最简单的选择是忽略所有后缀。

```
class simple_string :
  boost::addable<simple_string,
    boost::addable<simple_string,const char*> > {
```

相等与等价的区别

为类定义关系操作符时，很重要的一点是分清楚相等和等价。要使用关联容器，就要求有等价关系，它通过概念LessThanComparable[11]定义了一个严格弱序。这个关系只需最小的假设，并且对于要用于标准库容器的类型来说，这是最低的要求。但是，相等与等价之间的区别有时会令人混淆，弄明白它们之间的差别是很重要的。如果一个类支持概念LessThanComparable, 通常它也就支持等价的概念。如果两个元素进行比较，没有一个比另一个小，我们称它们是等价的。但是，等价并不意味着相等。例如，有可能在一个less than关系中忽略某些特性，但并不意味着它们就是相等的[12]。为了举例说明这一点，我们来看一个类，`animal`, 它同时支持等价关系和相等关系。

> [11] 大写的概念，如 LessThanComparable 直接来自于C++标准。所有 Boost.Operators 中的概念使用小写名字。
>
> [12] 这意味着一个严格弱序，但不是一个全序。

```
class animal : boost::less_than_comparable<animal,
boost::equality_comparable<animal> > {
  std::string name_;
  int age_;
public:
  animal(const std::string& name,int age)
    :name_(name),age_(age) {}

  void print() const {
    std::cout << name_ << " with the age " << age_ << '\n';
  }

  friend bool operator<(const animal& lhs, const animal& rhs) {
    return lhs.name_<rhs.name_;
  }

  friend bool operator==(const animal& lhs, const animal& rhs) {
    return lhs.name_==rhs.name_ && lhs.age_==rhs.age_;
  }

};
```

请注意 `operator<` 和 `operator==` 的实现间的区别。在less than关系中仅使用了动物的名字，而在相等检查中则同时比较了名字和年龄。这种方法没有任何错误，但是它会导致有趣的结果。现在让我们把这个类的一些元素存入 `std::set`. 和其它关联容器一样， `set` 仅依赖于概念 LessThanComparable. 以下面例子中，我们创建四个不一样的动物，然后试图把它们插入一个 `set`, 完全假装我们不知道相等和等价之间的差别。

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include "boost/operators.hpp"
#include "boost/bind.hpp"

int main() {
  animal a1("Monkey", 3);
  animal a2("Bear", 8);
  animal a3("Turtle", 56);
  animal a4("Monkey", 5);

  std::set<animal> s;
  s.insert(a1);
  s.insert(a2);
  s.insert(a3);
  s.insert(a4);

  std::cout << "Number of animals: " << s.size() << '\n';
  std::for_each(s.begin(),s.end(),boost::bind(&animal::print,_1));
  std::cout << '\n';

  std::set<animal>::iterator it(s.find(animal("Monkey",200)));
  if (it!=s.end()) {
    std::cout << "Amazingly, there's a 200 year old monkey "
      "in this set!\n";
    it->print();
  }

  it=std::find(s.begin(),s.end(),animal("Monkey",200));
  if (it==s.end()) {
    std::cout << "Of course there's no 200 year old monkey "
      "in this set!\n";
  }
}
```

运行这个程序，会有以下完全荒谬的输出结果。

```
Number of animals: 3
Bear with the age 8
Monkey with the age 3
Turtle with the age 56

Amazingly, there's a 200 year old monkey in this set!
Monkey with the age 3
Of course there's no 200 year old monkey in this set!
```

问题不在于猴子的年龄——它的确不寻常——而在于没有了解这两种关系概念间的区别。首先，当四个 `animal` s (`a1` , `a2` , `a3` , `a4`)被插入到 `set` , 第二只猴子， `a4` , 其实并没有被插入，因为 `a1` 和 `a4` 是等价的。原因是， `std::set` 使用表达式 `!(a1<a4) && !(a4<a1)` 来决定是否已有一个匹配的元素。由于这个表达式的结果为 `true` (我们的 `operator<` 不比较年龄), 所以插入失败了[13]。然后，当我们询问这个set，使用 `find` 查找一个200岁的猴子时，它找到了这样一只怪物。同样，这是由于 `animal` 的等价关系，仅依赖于 `animal` 的 `operator<` ，因而还是不关心年龄。最后，我们再次用 `find` 在 `set` 中定位这只猴子(`a1`), 但这次我们调用 `operator==` 来判断它是否匹配，从而没有找到匹配的猴子。通过对这些猴子的讨论，不难理解相等与等价之间的差别，但你必须知道在给定的上下文中使用的是哪一个概念。

> [13] 一个set, 根据定义, 不存在重复的元素。

算术类型

定义算术类型时，Operators库尤其有用。对于一个算术类型，通常有很多操作符要定义，而手工 去做这些工作是一项令人畏缩和沉闷的任务，并很可能发生错误或疏忽。Operators库中定义的概念使这项工作变得容易，你只需为类定义最少的必须的操 作符，剩下的操作符就可以自动实现。考虑一个支持加法和减法的类。假设这个类使用一个内建类型作为实现。现在要增加适当的操作符，并确保它们不仅可以用于 这个类的实例，还可以用于可转换为这个类的内建类型。你将要提供12个不同的加法和减法操作符。当然，更容易(也是更安全)的方法是，使用 `addable` 和 `subtractable` 类的双参数形式。现在假设你还需要增加一组关系操作符。你可能要自己增加10个操作符，但现在你知道了最容易的方法是使用 `less_than_comparable` 和 `equality_comparable` . 这样做之后，你拥有了22个操作符而只定义了6个。然而，你可能也注意到了这些概念对于数值类型来说是很常见的。的确如此，作为这四个类的替换，你可以仅使用 `additive` 和 `totally_ordered` .

我们先从四个概念类进行派生开始： `addable` , `subtractable` , `less_than_comparable` , 和 `equality_comparable` . 类 `limited_type` , 仅仅包装了一个内建类型并将所有操作符前转给那个类型。它限制可用操作的数量，仅提供了关系操作符和加减法。

```
#include "boost/operators.hpp"

template <typename T> class limited_type :
  boost::addable<limited_type<T>,
    boost::addable<limited_type<T>,T,
      boost::subtractable<limited_type<T>,
        boost::subtractable<limited_type<T>,T,
          boost::less_than_comparable<limited_type<T>,
            boost::less_than_comparable<limited_type<T>,T,
              boost::equality_comparable<limited_type<T>,
                boost::equality_comparable<limited_type<T>,T >
> > > > > > > {

  T t_;
public:
  limited_type():t_() {}
  limited_type(T t):t_(t) {}

  T get() {
    return t_;
  }

  // 为less_than_comparable提供
  friend bool operator<(
      const limited_type<T>& lhs,
      const limited_type<T>& rhs) {
    return lhs.t_<rhs.t_;
  }

  // 为equality_comparable提供
  friend bool operator==(
      const limited_type<T>& lhs,
      const limited_type<T>& rhs) {
    return lhs.t_==rhs.t_;
  }

  // 为addable提供
  limited_type<T>& operator+=(const limited_type<T>& other) {
    t_+=other.t_;
    return *this;
  }

  // 为subtractable提供
  limited_type<T>& operator-=(const limited_type<T>& other) {
    t_-=other.t_;
    return *this;
  }
};
```

这是一个不错的例子，示范了使用Operators库后实现变得多么容易。仅需实现几个必须实现的操 作符，就很容易地获得了全组的操作符，而类也变得更易懂以及更易于维护。(即使实现这些操作符是困难的，你也可以把注意力集中于正确地实现它们)。这个类 唯一的潜在问题就是，它派生自八个不同的operator类，使用了基类链的方式，对于某些人而言，这可能不好阅读。我们可以通过使用复合概念来大大简化 我们的类。

```
template <typename T> class limited_type :
  boost::additive<limited_type<T>,
    boost::additive<limited_type<T>,T,
      boost::totally_ordered<limited_type<T>,
        boost::totally_ordered<limited_type<T>,T > > > > >  {
```

这更好看，而且也减少了击键的次数。

仅在应用使用Operators时使用它

很明显operators应该仅在适当的时候使用，但出于某些原因，operators的某些"很酷 的因素"常常诱使一些人在不清楚它们的语义时就使用它们。很多情形下都需要操作符，如在同一类型的实例间存在某种关系时，又或者在创建一个算术类型时。但 也有一些不太清晰的情形，你就需要考虑使用者的真正期望，如果用户的期望是模糊的，最好还是选择用成员函数。

多年以来，Operators已经被用于一些不平常的服务。增加操作符用于连接字符串，和使用位移操作符进行I/O，就是两个操作符不再具有数学意义而被用于其它语义用途的最常见的例子。也有人对于在 `std::map` 中使用下标操作符访问元素表示质疑(当 然其它人认为这是很自然的。他们是对的)。有时候，把操作符用于某种与内建类型规则不一致的用途是有意义的。而其它时候，它则是非常错误的，会引起混乱和 歧义。当你决定将一个操作符重载为与内建类型不一致的意义时，你必须很小心地去做。你必须确保它的意义是明显的，并且优先级是正确的。这也是在 IOStream库中使用位移操作符进行I/O的原因。位移操作符清晰地表明了将某物移向某个方向，并且位移操作符的优先级比多数操作符都低。如果你创建 一个表示汽车的类，可能发现 `operator-=` 很方便。但是，对于使用者这个操作符意味着什么？有些人可能认为它被用来表示在驾驶中使用的汽油。其它人可能认为它被用来表示汽车价值的贬值(当然会计师 会这样想)。增加这个操作符是错误的，因为它没有一个清晰的意图，而一个成员函数可以更清楚地为这些操作命名。不要仅仅为了它可以写出"酷"的代码而增加 操作符。要因为它们真的有用而增加它们，确认增加所有适用的操作符，并且确认使用Boost.Operators 库！

弄明白它是如何工作的

我们现在来看一看这个库是如何工作的，以进一步加深你对于如何正确使用它的理解。对于Boost.Operators, 这并不难。我们来看看如何实现对 `less_than_comparable` 的支持。你需要了解你要增加支持的那个类，并且你要为这个类增加操作符，这个操作符将用于实现该类的其它相关操作符。 `less_than_comparable` 要求我们提供 `operator<` , `operator>` , `operator<=` , 和 `operator>=` . 现在，你已经知道如何依照 `operator<` 来实现 `operator>` , `operator<=` , and `operator>=` 。下面是一种可能的实现方法。

```
template <class T>
class less_than1
{
public:
  friend bool operator>(const T& lhs,const T& rhs)  {
    return rhs<lhs;
  }

  friend bool operator<=(const T& lhs,const T& rhs) {
    return !(rhs<lhs);
  }

  friend bool operator>=(const T& lhs,const T& rhs) {
    return !(lhs<rhs);
  }
};
```

对于 `operator>` , 你只需要交换两个参数的顺序。对于 `operator<=` , 注意到 `a<=b` 即意味着 `b` 不小于 `a` . 因此，实现的方法就是以相反的参数顺序调用 `operator<` 并对结果取反。对于 `operator>=` , 同样由于 `a>=b` 意味着 `a` 不小于 `b` . 因此，实现方法就是对调用 `operator<` 的结果取反。这是一个可以工作的例子：你可以直接使用它并且它将完成正确的工作。然而，如果可以提供一个支持 `T` 与兼容类型之间进行比较的版本就更好了，这只要简单地增加几个重载就可以了。出于对称性的考虑，你应该允许兼容类型出现在操作符的左边(这一点在手工增加操作符时很容易忘记；人们通常仅留意到操作符的右边要接受其它类型。当然，你的双类型版本 `less_than` 不会犯如此愚蠢的错误，对吗？)

```
template <class T,class U>
class less_than2
{
public:
  friend bool operator<=(const T& lhs,const U& rhs) {
    return !(lhs>rhs);
  }

  friend bool operator>=(const T& lhs,const U& rhs) {
    return !(lhs<rhs);
  }

  friend bool operator>(const U& lhs,const T& rhs) {
    return rhs<lhs;
  }

  friend bool operator<(const U& lhs,const T& rhs)  {
    return rhs>lhs;
  }

  friend bool operator<=(const U& lhs,const T& rhs) {
    return !(rhs<lhs);
  }

  friend bool operator>=(const U& lhs,const T& rhs) {
    return !(rhs>lhs);
  }
};
```

这就是了！两个功能完整的 `less_than` 类。当然，要与Operators库中的 `less_than_comparable` 具有同样的功能，我们必须去掉表示使用几个类型的后缀。我们真正想要的是一个版本，或者说至少是一个名字。如果你使用的编译器支持模板偏特化，你就是

幸运 的，因为基本上只要几行代码就可以实现了。但是，还有很多程序员没有这么幸运，所以我们还是要用健壮的方法来实现它，以完全避开偏特化。首先，我们知道我 们需要某个东西用来调用 `less_than`，它是一个接受一个或两个类型参数的模板。我们也知道第二个类型是可选的，我们可以给它加一个缺省类型，我们知道用户不会传递这样一个类型给这个模板。

```
struct dummy {};
template <typename T,typename U=dummy> class less_than {};
```

我们需要某种机制来选择正确版本的 `less_than`（`less_than1` 或 `less_than2`）；我们可以无需借助模板偏特化，而通过使用一个辅助类来做到，这个辅助类有一个类型参数，并内嵌一个接受另一个类的嵌套模板 `struct`。然后，使用全特化，我们可以确保类型 `U` 是 `dummy` 时，`less_than1` 将被选中。

```
template <typename T> struct selector {
  template <typename U> struct type {
    typedef less_than_2<U,T> value;
  };
};
```

前面这个版本创建了一个名为 `value` 的类型定义，这个类型正是我们已经创建的 模板的一个正确的实例化。

```
template<> struct selector<dummy> {
  template <typename U> struct type {
    typedef less_than1<U> value;
  };
};
```

全特化的 `selector` 创建了另一个版本 `less_than1` 的 `typedef`。为了让编译器更容易做，我们将创建另一个辅助类，专门负责收集正确的类型，并把它存入适当的typedef `type`.

```
template <typename T,typename U> struct select_implementation {
  typedef typename selector<U>::template type<T>::value type;
};
```

这种语法看上去不讨人喜欢，因为 `selector` 类中的嵌套模板 `struct`，但类的使用者并不需要看到这段代码，所以这不是什么大问题。现在我们有了所有的因素，我们需要从中选择一个正确的实现，我们最终从 `select_implementation&lt;T,U&gt;::type` 派生 `less_than`，前者将会是 `less_than1` 或 `less_than2`，这取决于用户给出了一个还是两个类型。

```
template <typename T,typename U=dummy> class less_than :
  select_implementation<T,U>::type {};
```

就是它了！我们现在有了一个完全可用的 `less_than`，由于我们付出的额外努力，增加了一种检测并选择正确的实现版本的机制，用户现在可以以最容易的方式来使用它。我们还正确地了解了 `operator&lt;` 如何用于创建一个 `less_than_comparable` 类所用的其它操作符。对其它

操作符完成同样的任务只需要小心行事，并弄清楚不同的操作符是如何共同组成新的概念的就行了。

剩下的事情

我们还没有讨论Operators库中的剩余部分，迭代器助手类。我不想给出示例了，因为你主要是在 定义迭代器类型时会用到它们，这需要额外的解释，这超出了本章甚至是本书的范围。我在这里之所以提及它们，是因为如果你正在定义迭代器类型而不借助于 Boost.Iterators的话，你肯定会想用它些助手类的。解引用操作符帮助你定义正确的操作符而无须顾及你是否在需要一个代理类。在定义智能指针 是它们也很有用，智能指针通常要求定义 `operator->` 和 `operator*`. 迭代器助手类把不同类型的迭代器所需的概念组合在了一起。例如，一个随机访问迭代器必须是 `bidirectional_iterable`，`totally_ordered`，`additive`，和 `indexable` 的。定义新的迭代器类型时，更适当的做法是借助于Boost.Iterator库，不过Operators库也可以帮忙。

Operators 总结

为用户自定义类型提供一组正确的关系操作符和算术操作符是非常重要的，而且正确地实现它也是一个重大 的挑战。通过使用Operators库，这个任务大大地简化了，正确性和对称性也随之而来。除此之外，这个库还提供了一组完整的操作符定义，这些类所支持 的概念被适当地命名和定义，可以在定义你的类时明确这些概念(也是通过Operators库！)。在本章中，我们已经看了几个例子，关于如何使用这个库来 改进带有操作符的程序，使程序更为简单，正确性也更有保证。一个可悲的事实是，为用户自定义类型提供重要的关系操作符和算术操作符常常被忽略掉，部分的原 因是由于为了正确获得它们需要做大量的工作。现在这种情形不会再出现了，因为有了Boost.Operators。

在提供关系操作符和算术操作符时要重点考虑的一点是，首先要确认提供它们是有必要的。当类型间存在顺 序关系时，或者在创建数值类型时，总是需要提供操作符的，但对于其它类型，操作符可能就不能清晰地传递设计的意图。操作符几乎总是提供语法上的好处，这种 语法上的好处不应被低估。不幸的是，操作符又是诱人的。明智地使用它们，它们就会发挥巨大的威力。当你决定为一个类增加操作符， Boost.Operators库可以为你的工作提高质量和效率。结论是，你应该在深思熟虑之后再决定是否给你的类增加操作符，当你决定要增加时，就使用 Operators库。

Operators库是多个人的贡献的结果。它从David Abrahams开始，并接受了Jeremy Siek, Aleksey Gurtovoy, Beman Dawes, 和 Daryle Walker等人的有价值的补充。正如多数Boost库一样，无数其它人的贡献才形成了今天这个库。

Library 5. Regex

Regex库如何改进你的程序？

- 为C++带来了对正则表达式的支持
- 改进有效输入的健壮性

在文本处理中常常会用到正则表达式。例如，有很多验证有效性的工作适合使用正则表达式。考虑一个应用 程序，它要求输入只由数字组成。而另一个程序可能要求一种特殊的格式，如三个数字，后跟一个字母，再后跟两个数字。你可能要验证邮政编码、信用卡号码、社 会保险号码，或者其它东西；使用正则表达式来做这些验证是很简单的。另外一个可以使用正则表达式的地方是文本替换，即用某些文本替换掉另一些文本。假如你 需要在很多文档中将单词colour 转换为 color。正则表达式提供了最好的方法来做这个工作，包括同时记住替换 Colour 和 COLOUR, 以及复数形式的 colours, 动词 colourize, 等等。还有一个可以使用正则表达式的地方就是，格式化文本。

许多流行的编程语言，Perl是其中一例，都内建了对正则表达式的支持，但在C++里没有。C++标 准在提到正则表达式时也保持了沉默。Boost.Regex是一个非常完善并有效的库，它将正则表达式并入了C++程序，并且它包含了Perl, grep和Emacs等常见工具所使用的几种不同语法。它是最著名的C++正则表达式库之一，既容易使用又异常强大。

Regex 如何适用于标准库？

目前的C++标准库是不支持正则表达式的。这是令人遗憾的，有那么多对正则表达式的需要，有时用户为了编写需要支持正则表达式的程序而不得不放弃使用C++。Boost.Regex填补了标准在这方面的空白，并且它已经被提议加入到下一个版本的C++标准中。Boost.Regex已经被即将发布的Library Technical Report所接受。

Regex

头文件: `"boost/regex.hpp"`

正则表达式被封装为一个类型 `basic_regex` 的对象。我们将在下一节更深入地讨论正则表达式如何被编译和分析，这里我们首先粗略地看看 `basic_regex` ，以及这个库中三个最重要的算法。

```
namespace boost {
  template <class charT,
            class traits=regex_traits<charT> >
  class basic_regex {
  public:
    explicit basic_regex(
      const charT* p,
      flag_type f=regex_constants::normal);

    bool empty() const;

    unsigned mark_count() const;

    flag_type flags() const;
  };

  typedef basic_regex<char> regex;
  typedef basic_regex<wchar_t> wregex;
}
```

成员函数

```
explicit basic_regex (
  const charT* p,
  flag_type f=regex_constants::normal);
```

这个构造函数接受一个包含正则表达式的字符序列，还有一个参数用于指定使用正则表达式时的选项，例如是否忽略大小写。如果 `p` 中的正则表达式无效，则抛出一个 `bad_expression` 或 `regex_error` 的异常。注意这两个异常其实是同一个东西；在写这本书之时，尚未改变当前使用的名字 `bad_expression` ，但下一个版本的Boost.Regex将会使用 `regex_error` .

```
bool empty() const;
```

这个成员函数是一个谓词，当 `basic_regex` 实例没有包含一个有效的正则表达式时返回 `true` ，即它被赋予一个空的字符序列时。

```
unsigned mark_count() const;
```

`mark_count` 返回 `regex` 中带标记子表达式的数量。带标记子表达式是指正则表达式中用圆括号括起来的部分。匹配这个子表达式的文本可以通过调用某个正则表达式算法而获得。

```
flag_type flags() const;
```

返回一个位掩码，其中包含这个 `basic_regex` 所设置的选项标志。例如标志 `icase`，表示正则表达式忽略大小写，标志 `JavaScript`，表示regex使用JavaScript的语法。

```
typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;
```

不要使用类型 `basic_regex` 来定义变量，你应该使用这两个 `typedef` 中的一个。这两个类型，`regex` 和 `wregex`，是两种字符类型的缩写，就如 `string` 和 `wstring` 是 `basic_string<char>` 和 `basic_string<wchar_t>` 的缩写一样。这种相似性是不一样的，某种程度上，`regex` 是一个特定类型的字符串的容器。

普通函数

```
template <class charT,class Allocator,class traits >
  bool regex_match(
    const charT* str,
    match_results<const charT*,Allocator>& m,
    const basic_regex<charT,traits >& e,
    match_flag_type flags = match_default);
```

`regex_match` 判断一个正则表达式(参数 `e`)是否匹配整个字符序列 `str`. 它主要用于验证文本。注意，这个正则表达式必须匹配被分析串的全部，否则函数返回 `false`. 如果整个序列被成功匹配，`regex_match` 返回 `True`.

```
template <class charT,class Allocator, class traits>
  bool regex_search(
    const charT* str,
    match_results<const charT*,Allocator>& m,
    const basic_regex<charT,traits >& e,
    match_flag_type flags = match_default);
```

`regex_search` 类似于 `regex_match`，但它不要求整个字符序列完全匹配。你可以用 `regex_search` 来查找输入中的一个子序列，该子序列匹配正则表达式 `e`.

```
template <class traits,class charT>
  basic_string<charT> regex_replace(
    const basic_string<charT>& s,
    const basic_regex<charT,traits >& e,
    const basic_string<charT>& fmt,
    match_flag_type flags = match_default);
```

`regex_replace` 在整个字符序列中查找正则表达式 `e` 的所有匹配。这个算法每次成功匹配后，就根据参数 `fmt` 对匹配字符串进行格式化。缺省情况下，不匹配的文本不会被修改，即文本会被输出但没有改变。

这三个算法都有几个不同的重载形式：一个接受 `const charT*` (`charT` 为字符类型), 另一个接受 `const basic_string<charT>&` , 还有一个重载接受两个双向迭代器作为输入参数。

用法

要使用Boost.Regex, 你需要包含头文件 `"boost/regex.hpp"` . Regex是本书中两个需要独立编译的库之一(另一个是Boost.Signals)。你会很高兴获知如果你已经构建了Boost— —那只需在命令提示符下打一行命令——就可以自动链接了(对于Windows下的编译器)，所以你不需要为指出那些 库文件要用而费心。

你要做的第一件事就是声明一个类型 `basic_regex` 的变量。这是该库的核心类之一，也是存放正则表达式的地方。创建这样一个变量很简单；只要将一个含有你要用的正则表达式的字符串传递给构造函数就行了。

```
boost::regex reg("(A.*)");
```

这个正则表达式具有三个有趣的特性。第一个是，用圆括号把一个子表达式括起来，这样可以稍后在同一个正则表达式中引用它，或者取出匹配它的文本。我们稍后会详细讨论它，所以如果你还不知道它有什么用也不必担心。第二个是，通配符(wildcard)字符，点。这个通配符在正则表达式中有非常特殊的意义；这可以匹配任意字符。最后一个是，这个表达式用到了一个重复符， `*` , 称为Kleene star, 表示它前面的表达式可以被匹配零次或多次。这个正则表达式已可以用于某个算法了，如下：

```
bool b=boost::regex_match(
  "This expression could match from A and beyond.",
  reg);
```

如你所见，你把正则表达式和要分析的字符串传递给算法 `regex_match` . 如果的确存在与正则表达式的匹配，则该函数调用返回结果 `true` ；否则，返回 `false` . 在这个例子中，结果是 `false` , 因为 `regex_match` 仅当整个输入数据被正则表达式成功匹配时才返回 `true` 。你知道为什么是这样吗？再看一下那个正则表达式。第一个字符是大写的 `A` , 很明显能够匹配这个表达式的第一个字符在哪。所以，输入的一部分 `"A and beyond."` 可以匹配这个表达式，但这不是整个输入。让我们试一下另一个输入字符串。

```
bool b=boost::regex_match(
  "As this string starts with A, does it match? ",
  reg);
```

这一次， `regex_match` 返回 `true` . 当正则表达式引擎匹配了 `A` , 它接着看后续有什么。在我们的regex变量中， `A` 后跟一个通配符和一个Kleene star, 这意味着任意字符可以被匹配任意次。因而，分析过程开始扔掉输入字符串的剩余部分，即匹配了输入的所有部分。

接下来，我们看看如何使用regexes 和 `regex_match` 来进行数据验证。

验证输入

正则表达式常用于对输入数据的格式进行验证。应用软件通常要求输入符合某种结构。考虑一个应用软件，它要求输入一定要符合如下格式，"3个数字, 一个单词, 任意字符, 2个数字或字符串"N/A," 一个空格, 然后重复第一个单词." 手工编写代码来验证这个输入既沉闷又容易出错，而且这些格式还很可能会改变；在你弄明白之前，可能就需要支持其它的格式，你精心编写的分析器可能就需要修 改并重新调试。让我们写出一个可以验证这个输入的正则表达式。首先，我们需要一个匹配3个数字的表达式。对于数字，我们应该使用一个特别的缩写，`\d`。要表示它被重复3次，需要一个称为bounds operator的特定重复，它用花括号括起来。把这两个合起来，就是我们的正则表达式的开始部分了。

```
boost::regex reg("\\d{3}");
```

注意，我们需要在转义字符()之前加一个转义字符，即在我们的字符串中，缩写 `\d` 变成了 `\\d` 。这是因为编译器会把第一个\当成转义字符扔掉；我们需要对\进行转义，这样\才可以出现在我们的正则表达式中。

接下来，我们需要定义一个单词的方法，即定义一个字符序列，该序列结束于一个非字母字符。有不只一种方法可以实现它，我们将使用字符类别(也称为字符集)和范围这两个正则表达式的特性来做。字符类别即一个用方括号括起来的表达式。例如，一个匹配字符 `a`, `b`, 和 `c` 中任一个的字符类别表示为：`[abc]`. 如果用范围来表示同样的东西，我们要写：`[a-c]`. 要写一个包含所有字母的字符类型，我们可能会有点发疯，如果要把它写成：`[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]`, 但不用这样；我们可以用范围来表示：`[a-zA-Z]`. 要注意的是，象这样使用范围要依赖于当前所用的locale，如果正则表达式的 `basic_regex::collate` 标志被打开。使用以上工具以及重复符 `+`, 它表示前面的表达式可以重复，但至少重复一次，我们现在可以表示一个单词了。

```
boost::regex reg("[a-zA-Z]+");
```

以上正则表达式可以工作，但由于经常要表示一个单词，所以有一个更简单的方法：`\w`. 这个符号匹配所有单词，不仅是ASCII的单词，因此它不仅更短，而且也更适用于国际化的环境。接下来的字符是一个任意字符，我们已经知道要用点来表示。

```
boost::regex reg(".");
```

再接下来是 2个数字或字符串 "N/A." 为了匹配它，我们需要用到一个称为选择的特性。选择即是匹配两个或更多子表达式中的任意一个，每种选择之间用 `|` 分隔开。就象这样：

```
boost::regex reg("(\\d{2}|N/A)");
```

注意，这个表达式被圆括号括了起来，以确保整个表达式被看作为两个选择。在正则表达式中增加一个空格是很简单的；用缩写 `\s` . 把以上每一样东西合并起来，就得到了以下表达式：

```
boost::regex reg("\\d{3}[a-zA-Z]+.(\\d{2}|N/A)\\s");
```

现在事情变得有点复杂了。我们需要某种方法，来验证接下来的输入数据中的单词是否匹配第一个单词(即那个我们用表达式 `[a-zA-Z]+` 所捕获的单词)。关键是要使用后向引用(back reference)，即对前面的子表达式的引用。为了可以引用表达式 `[a-zA-Z]+` , 我们必须先把它用圆括号括起来。这使得表达式 `([a-zA-Z]+)` 成为我们的正则表达式中的第一个子表达式，我们就可以用索引1来建立一个后向引用了。

这样，我们就得到了整个正则表达式，用于表示"3个数字, 一个单词, 任意字符, 2个数字或字符串"N/A," 一个空格, 然后重复第一个单词." :

```
boost::regex reg("\\d{3}([a-zA-Z]+).(\\d{2}|N/A)\\s\\1");
```

干的好！下面是一个简单的程序，把这个表达式用于算法 `regex_match` , 验证两个输入字符串。

```
#include <iostream>
#include <cassert>
#include <string>
#include "boost/regex.hpp"

int main() {
  // 3 digits, a word, any character, 2 digits or "N/A",
  // a space, then the first word again
  boost::regex reg("\\d{3}([a-zA-Z]+).(\\d{2}|N/A)\\s\\1");

  std::string correct="123Hello N/A Hello";
  std::string incorrect="123Hello 12 hello";

  assert(boost::regex_match(correct,reg)==true);
  assert(boost::regex_match(incorrect,reg)==false);
}
```

第一个字符串， `123Hello N/A Hello` , 是正确的； `123` 是3个数字， `Hello` 是一个后跟任意字符(一个空格)的单词， 然后是N/A和另一个空格，最后重复单词 `Hello` 。第二个字符串是不正确的，因为单词 `Hello` 没有被严格重复。缺省情况下，正则表达式是大小写敏感的，因而反向引用不能匹配。

写出正则表达式的一个关键是成功地分解问题。看一下你刚才建立的最终的那个表达式，对于未经过训练的人来说它的确很难懂。但是，如果把这个表达式分解成小的部分，它就不太复杂了。

查找

现在我们来看一下另一个Boost.Regex算法, `regex_search`. 与 `regex_match` 不同的是， `regex_search` 不要求整个输入数据完全匹配，则仅要求部分数据可以匹配。作为说明，考虑一个程序员的问题，他可能在他的程序中有一至两次忘记了调用 `delete` 。虽然他知道这个简单的测试可能没什么意义，他还是决定计算一下new 和 delete出现的次数，看看数字是否符合。这个正则表达式很简单；我们有两个选择，new 和 delete.

```
boost::regex reg("(new)|(delete)");
```

有两个原因我们要把子表达式用括号括起来：一个是为了表明我们的选择是两个组。另一个原因是我们想在调用 `regex_search` 时引用这些子表达式，这样我们就可以判断是哪一个选择被匹配了。我们使用 `regex_search` 的一个重载，它接受一个 `match_results` 类型的参数。当 `regex_search` 执行匹配时，它通过一个 `match_results` 类型的对象报告匹配的子表达式。类模板 `match_results` 使用一个输入序列所用的迭代器类型来参数化。

```
template <class Iterator,
  class Allocator=std::allocator<sub_match<Iterator> >
    class match_results;

typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t> wcmatch;
typedef match_results<std::string::const_iterator> smatch;
typedef match_results<std::wstring::const_iterator> wsmatch;
```

我们将使用 `std::string`, 所以要留意 `typedef smatch`, 它是 `match_results&lt;std::string::const_iterator&gt;` 的缩写。如果 `regex_search` 返回 `true`, 传递给该函数的 `match_results` 引用将包含匹配的子表达式结果。在 `match_results` 里，用已索引的 `sub_match` 来表示正则表达式中的每个子表达式。我们来看一下我们如何帮助这位困惑的程序员来计算对 `new` 和 `delete` 的调用。

```
boost::regex reg("(new)|(delete)");
boost::smatch m;
std::string s=
  "Calls to new must be followed by delete. \
  Calling simply new results in a leak!";

if (boost::regex_search(s,m,reg)) {
  // Did new match?
  if (m[1].matched)
    std::cout << "The expression (new) matched!\n";
  if (m[2].matched)
    std::cout << "The expression (delete) matched!\n";
}
```

以上程序在输入字符串中查找 `new` 或 `delete`, 并报告哪一个先被找到。通过传递一个类型 `smatch` 的对象给 `regex_search`, 我们可以得知算法如何执行成功的细节。我们的表达式中有两个子表达式，因此我们可以通过 `match_results` 的索引1得到子表达式 `new` . 这样我们得到一个 `sub_match` 实例，它有一个Boolean成员， `matched`, 告诉我们这个子表达式是否参与了匹配。因此，对于上例的输入，运行结果将输出"The expression (new) matched!\n". 现在，你还有一些工作要做。你需要继续把正则表达式应用于输入的剩余部分，为此，你要使

用另外一个 `regex_search` 的重载，它接受两个迭代器，指示出要查找的字符序列。因为 `std::string` 是一个容器，它提供了迭代器。现在，在每一次匹配时，你必须把指示范围起始点的迭代器更新为上一次匹配的结束点。最后，增加两个变量来记录 `new` 和 `delete` 的次数。以下是完整的程序：

```
#include <iostream>
#include <string>
#include "boost/regex.hpp"

int main() {
  // "new" and "delete" 出现的次数是否一样？
  boost::regex reg("(new)|(delete)");
  boost::smatch m;
  std::string s=
    "Calls to new must be followed by delete. \
     Calling simply new results in a leak!";
  int new_counter=0;
  int delete_counter=0;
  std::string::const_iterator it=s.begin();
  std::string::const_iterator end=s.end();

  while (boost::regex_search(it,end,m,reg)) {
    // 是 new 还是 delete?
    m[1].matched ? ++new_counter : ++delete_counter;
    it=m[0].second;
  }

  if (new_counter!=delete_counter)
    std::cout << "Leak detected!\n";
  else
    std::cout << "Seems ok...\n";
}
```

注意，这个程序总是把迭代器 `it` 设置为 `m[0].second`。 `match_results[0]` 返回对匹配整个正则表达式的子匹配的引用，因此我们可以确认这个匹配的结束点就是下次运行 `regex_search` 的起始点。运行这个程序将输出"Leak detected!", 因为这里有两次 `new`, 而只有一次 `delete`. 当然，一个变量也可能在两个地方删除，还有可能调用 `new[]` 和 `delete[]`, 等等。

现在，你应该已经对子表达式如何分组使用有了较好的了解。现在是时候进入到最后一个Boost.Regex算法，该算法用于执行替换工作。

替换

Regex算法家族中的第三个算法是 `regex_replace`. 顾名思义，它是用于执行文本替换的。它在整个输入数据中进行搜索，查找正则表达式的所有匹配。对于表达式的每一个匹配，该算法调用 `match_results::format` 并输入结果到一个传入函数的输出迭代器。

在本章的介绍部分，我给出了一个例子，将英式拼法的 colour 替换为美式拼法 color. 不使用正则表达式来进行这个拼写更改会非常乏味，也很容易出错。问题是可能存在不同的大小写，而且会有很多单词被影响，如colourize. 要正确地解决这个问题，我们需要把正则表达式分为三个子表达式。

```
boost::regex reg("(Colo)(u)(r)",
  boost::regex::icase|boost::regex::perl);
```

我们将要去掉的字母u独立开，为了在所有匹配中可以很容易地删掉它。另外，注意到这个正则表达式是大小写无关的，我们要把格式标志 `boost::regex::icase` 传给 `regex` 的构造函数。你还要传递你想要设置的其它标志。设置标志时一个常见的错误就是忽略了 `regex` 缺省打开的那些标志，如果你没有设置这些标志，它们不会打开，你必须设置所有你要打开的标志。

调用 `regex_replace` 时，我们要以参数方式提供一个格式化字符串。该格式化字符串决定如何进行替换。在这个格式化字符串中，你可以引用匹配的子表达式，这正是我们想要的。你想保留第一个和第三个匹配的子表达式，而去掉第二个 `(u)` 。表达式 `$N` 表示匹配的子表达式，`N` 为子表达式索引。因此我们的格式化串应该是 `"$1$3"`，表示替换文本为第一个和第三个子表达式。通过引用匹配的子表达式，我们可以保留匹配文本中的所有大小写，而如果我们用字符串来作替换文本则不能做到这一点。以下是解决这个问题的完整程序。

```
#include <iostream>
#include <string>
#include "boost/regex.hpp"

int main() {
  boost::regex reg("(Colo)(u)(r)",
    boost::regex::icase|boost::regex::perl);

  std::string s="Colour, colours, color, colourize";

  s=boost::regex_replace(s,reg,"$1$3");
  std::cout << s;
}
```

程序的输出是 `"Color, colors, color, colorize"`. `regex_replace` 对于这样的文本替换非常有用。

用户常见的误解

我所见到的与Boost.Regex相关的最常见的问题与 `regex_match` 的语义有关。人们很容易忘记必须使 `regex_match` 的所有输入匹配给定的正则表达式。因此，用户常以为以下代码会返回 `true`.

```
boost::regex reg("\\d*");
bool b=boost::regex_match("17 is prime",reg);
```

无疑这个调用永远不会得到成功的匹配。只有所有输入被 `regex_match` 匹配才可以返回 `true` ！几乎所有的用户都会问为什么 `regex_search` 不是这样而 `regex_match` 是。

```
boost::regex reg("\\d*");
bool b=boost::regex_search("17 is prime",reg);
```

这次肯定返回 `true`. 值得注意的是，你可以用一些特定的缓冲操作符来让 `regex_search` 象 `regex_match` 那样运行。 `\A` 匹配缓冲的起始点，而 `\Z` 匹配缓冲的结束点，因此如果你把 `\A` 放在正则表达式的开始，把 `\Z` 放在最后，你就可以让 `regex_search` 象 `regex_match` 那样使用，即必须匹配所有输入。以下正则表达式要求所有输入被匹配掉，而不管你使用的是 `regex_match` 或是 `regex_search`.

```
boost::regex reg("\\A\\d*\\Z");
```

请记住，这并不表示可以无需使用 `regex_match` ；相反，它可以清晰地表明我们刚才说到的语义，即必须匹配所有输入。

关于重复和贪婪

另一个容易混淆的地方是关于重复的贪婪。有些重复，如 `+` 和 `*`，是贪婪的。即是说，它们会消耗掉尽可能多的输入。以下正则表达式并不罕见，它用于在一个贪婪的重复后捕获两个数字。

```
boost::regex reg("(.*)(\\d{2})");
```

这个正则表达式是对的，但它可能不能匹配你想要的子表达式！表达式 `.*` 会吞掉所有东西而后续的子表达式将不能匹配。以下是示范这个行为的一个例子：

```
int main() {
  boost::regex reg("(.*)(\\d{2})");
  boost::cmatch m;
  const char* text = "Note that I'm 31 years old, not 32.";
  if(boost::regex_search(text,m, reg)) {
    if (m[1].matched)
      std::cout << "(.*) matched: " << m[1].str() << '\n';
    if (m[2].matched)
      std::cout << "Found the age: " << m[2] << '\n';
  }
}
```

在这个程序中，我们使用了 `match_results` 的另一个特化版本，即类型 `cmatch`. 它就是 `match_results<const char*>` 的 `typedef`, 之所以我们必须用它而不是用之前用过的 `smatch`，是因为我们现在是用一个字符序列调用 `regex_search` 而不是用类型 `std::string` 来调用。你期望这个程序的运行结果是什么？通常，一个刚开始使用正则表达式的用户会首先想到 `m[1].matched` 和 `m[2].matched` 都为 `true`, 且第二个子表达式的结果会是 " `31` ". 接着在认识到贪婪的重复所带来的效果后，即重复会尽可能消耗输入，用户会想到只有第一个子表达式是 `true`，即 `.*` 成功地吞掉了所有的输入。最后，新用户得到了下结论，两个子表达式都被匹配，但第二个表达式匹配的是最后一个可能的序列。即第一个子表达式匹配的是 " `Note that I'm 31 years old, not` " 而第二个匹配 " `32` ".

那么，如果你想使用重复并匹配另一个子表达式的第一次出现，该怎么办？要使用非贪婪的重复。在重复符后加一个 `?` ，重复就变为非贪婪的了。这意味着该表达式会尝试发现最短的匹配可能而不再阻止表达式的剩余部分进行匹配。因此，要让前面的正则表达式正确工作，我们需要把它改为这样。

```
boost::regex reg("(.*?)(\\d{2})");
```

如果我们用这个正则表达式来修改程序，那么 `m[1].matched` 和 `m[2].matched` 都会为 `true` . 表达式 `.*?` 只消耗最少可能的输入，即它将在第一个字符 `3` 处停止，因为这就是表达式要成功匹配所需要的。因此，第一个子表达式会匹配 " `Note that I'm` " 而第二个匹配 " `31` ".

看一下 **regex_iterator**

我们已经看过如何用几次 `regex_search` 调用来处理所有输入，但另一方面，更为优雅的方法是使用 `regex_iterator` . 这个迭代器类型用一个序列来列举正则表达式的所有匹配。解引用一个 `regex_iterator` 会产生对一个 `match_results` 实例的引用。构造一个 `regex_iterator` 时，你要把指示输入序列的迭代器传给它，并提供相应的正则表达式。我们来看一个例子，输入数据是一组由逗号分隔的整数。相应的正则表达式很简单。

```
boost::regex reg("(\\d+),?");
```

在正则表达式的最后加一个 `?` (匹配零次或一次) 确保最后一个数字可以被成功分析，即使输入序列不是以逗号结束。另外，我们还使用了另一个重复符 `+` . 这个重复符表示匹配一次或多次。现在，不需要多次调用 `regex_search` , 我们创建一个 `regex_iterator` , 并调用算法 `for_each` , 传给它一个函数对象，该函数对象以迭代器的解引用进行调用。下面是一个接受任意形式的 `match_results` 的函数对象，它有一个泛型的调用操作符。它所执行的就是把当前匹配的值加到一个总和中(在我们的正则表达式中，第一个子表达式是我们要用的)。

```
class regex_callback {
  int sum_;
public:
  regex_callback() : sum_(0) {}

  template <typename T> void operator()(const T& what) {
    sum_+=atoi(what[1].str().c_str());
  }

  int sum() const {
    return sum_;
  }
};
```

现在把这个函数对象的一个实例传递给 `std::for_each` , 结果是对每一个迭代器 `it` 的解引用调用该函数对象，即对每一次匹配的子表达式进行调用。

```
int main() {
  boost::regex reg("(\\d+),?");
  std::string s="1,1,2,3,5,8,13,21";

  boost::sregex_iterator it(s.begin(),s.end(),reg);
  boost::sregex_iterator end;

  regex_callback c;
  int sum=for_each(it,end,c).sum();
}
```

如你所见，传递给 `for_each` 的end迭代器是 `regex_iterator` 一个缺省构造实例。 `it` 和 `end` 的类型均为 `boost::sregex_iterator`，即为 `regex_iterator<std::string::const_iterator>` 的 `typedef`。这种使用 `regex_iterator` 的方法要比我们前面试过的多次匹配的方法更清晰，在多次匹配的方法中我们不得不在一个循环中让起始迭代器不断地前进并调用 `regex_search` 。

用 regex_token_iterator 分割字符串

另一个迭代器类型，或者说得更准确些，迭代器适配器，就是 `boost::regex_token_iterator`. 它与 `regex_iterator` 很类似，但却是用于列举不匹配某个正则表达式的每一个字符序列，这对于分割字符串很有用。它也可以用于选择对哪一个子表达式感兴趣，当解引用 `regex_token_iterator` 时，只有预订的那个子表达式被返回。考虑这样一个应用程序，它接受一些用斜线号分隔的数据项作为输入。两个斜线号之间的数据组成应用程序要处理的项。使用 `regex_token_iterator` 来分割这个字符串很容易。该正则表达式很简单。

```
boost::regex reg("/");
```

这个 regex 匹配各项间的分割符。要用它来分割输入，只需简单地把指定的索引 `1` 传递给 `regex_token_iterator` 的构造函数。以下是完整的程序：

```
int main() {
  boost::regex reg("/");
  std::string s="Split/Values/Separated/By/Slashes,";
  std::vector<std::string> vec;
  boost::sregex_token_iterator it(s.begin(),s.end(),reg,-1);
  boost::sregex_token_iterator end;
  while (it!=end)
    vec.push_back(*it++);

  assert(vec.size()==std::count(s.begin(),s.end(),'/')+1);
  assert(vec[0]=="Split");
}
```

就象 `regex_iterator` 一样， `regex_token_iterator` 是一个模板类，它使用所包装的序列的迭代器类型来进行特化。这里，我们用的是 `sregex_token_iterator`，它是 `regex_token_iterator<std::string::const_iterator>` 的 `typedef` 。每一次解引用这个

迭代器 `it`，它返回当前的 `sub_match`，当这个迭代器前进时，它尝试再次匹配该正则表达式。这两个迭代器类型，`regex_iterator` 和 `regex_token_iterator`，都非常有用；你应该明白，当你考虑反复调用 `regex_search` 时，就该用它们了。

更多的正则表达式

你已经看到了不少正则表达式的语法，但还有更多的要了解。这一节简单地示范一些你每天都会使用的正则表达式的其它功能。作为开始，我们先看一下一组完整的重复符；我们之前已经看到了 `*`，`+`，以及使用 `{}` 进行限定重复。还有一个重复符，即是 `?`. 你可能已经留意到它也可以用于声明非贪婪的重复，但对于它本身而言，它是表示一个表达式必须出现零次或一次。还有一点值得提及的是，限定重复符可以很灵活；下面是三种不同的用法：

```
boost::regex reg1("\\d{5}");
boost::regex reg2("\\d{2,4}");
boost::regex reg3("\\d{2,}");
```

第一个正则表达式匹配5个数字。第二个匹配 2个, 3个, 或者 4个数字。第三个匹配2个或更多个数字，没有上限。

另一种重要的正则表达式特性是使用元字符 `^` 表示非字符类别。用它来表示一个匹配任意不在给定字符类别中的字符；即你所列字符类别的补集。例如，看如下正则表达式。

```
boost::regex reg("[^13579]");
```

它包含一个非字符类别，匹配任意不是奇数数字的字符。看一下以下这个小程序，试着给出程序的输出。

```
int main() {
  boost::regex reg4("[^13579]");
  std::string s="0123456789";
  boost::sregex_iterator it(s.begin(),s.end(),reg4);
  boost::sregex_iterator end;

  while (it!=end)
    std::cout << *it++;
}
```

你给出答案了吗？输出是 "`02468`"，即所有偶数数字。注意，这个字符类别不仅匹配偶数数字，如果输入字符串是 "`AlfaBetaGamma`"，那么也会全部匹配。

我们看到的这个元字符，`^`，还有另一个意思。它可以用来表示一行的开始。而元字符 `$` 则表示一行的结束。

错的正则表达式

一个错的正则表达式就是一个不遵守规则的正则表达式。例如，你可能忘了一个右括号，这样正则表达式引擎将无法成功编译这个正则表达式。这时，将抛出一个 `bad_expression` 类型的异常。正如我前面提到的，这个异常的名字将会在下一版本的Boost.Regex中被修改，还有在即将加入Library Technical Report的版本中也是。异常类型 `bad_expression` 将被更名为 `regex_error`.

如果你的应用程序中的正则表达式全都是硬编码的，你可能不用处理错误表达式，但如果你是接受了用户的 输入来作为正则表达式，你就必须准备进行错误处理。这里有一个程序，提示用户输入一个正则表达式，接着输入一个用来对正则表达式进行匹配的字符串。由用户 进行输入时， 总是有可能会导致无效的输入。

```
int main() {
  std::cout << "Enter a regular expression:\n";
  std::string s;
  std::getline(std::cin, s);
  try {
    boost::regex reg(s);
    std::cout << "Enter a string to be matched:\n";

    std::getline(std::cin,s);

    if (boost::regex_match(s,reg))
      std::cout << "That's right!\n";
    else
      std::cout << "No, sorry, that doesn't match.\n";
  }
  catch(const boost::bad_expression& e) {
    std::cout <<
      "That's not a valid regular expression! (Error: " <<
      e.what() << ") Exiting...\n";
  }
}
```

为了保护应用程序和用户，一个 `try` / `catch` 块用于处理构造时抛出 `boost::regex` 的情形，这时会打印一个提示信息，而程序会温和地退出。用这个程序来测试，我们开始时输入一些合理的数据。

```
Enter a regular expression:
\d{5}
Enter a string to be matched:
12345
That's right!
```

现在，给一些错误的数据，试着输入一个错误的正则表达式。

```
Enter a regular expression:
(\w*))
That's not a valid regular expression! (Error: Unmatched ( or \() Exiting...
```

在 `regex reg` 构造时，就会抛出一个异常，因为这个正则表达式不能被编译。因此，进入到 `catch` 的处理例程中，程序将打印一个错误信息并退出。你只需知道有三个可能会发生异常的地方。一个是在构造一个正则表达式时，就象你刚刚看到的那样；另一个是使用成员函数

`assign` 把正则表达式赋给一个 `regex` 时。最后一个是，regex迭代器和算法也可能抛出异常，如果内存不够或者匹配的复杂度过快增长的话。

Regex 总结

无可争议，正则表达式是非常有用和重要的，而本库给C++带来了强大的正则表达式功能。传统上，用户除了使用POSIX C API来实现正则表达式功能以外，别无选择。对于文本处理的验证工作，正则表达式比手工编写分析代码要灵活和可靠得多。对于查找和替换，使用正则表达式可以优美地解决很多相关问题，而不用它们则根本无法解决。

Boost.Regex是一个强大的库，因此不可能在这一章中完全覆盖它所有的内容。同样，正则表达式的完美表现和广泛的应用范围意味着本章也不仅仅是简单地介绍一下它们。这个主题可以写成一本单独的书。要知道更多，可以学习Boost.Regex的在线文档，并且找一本关于正则表达式的书(考虑一下参考书目中的建议)。不论Boost.Regex有多强大，正则表达式有多广多深，初学者还是可以有效地使用本库中的正则表达式。对于那些由于C++不支持正则表达式而选择了其它语言的程序员，欢迎你们回家。

Boost.Regex并不是C++程序员唯一可以使用的正则表达式库，但它的确是最好的一个。它易于使用，并且在匹配你的正则表达式时快如闪电。你应该尽可能去用它。

Boost.Regex 的作者是 Dr. John Maddock.

Part II: 容器及数据结构

> 本部分讨论三个库：Boost.Any, Boost.Variant, 和 Boost.Tuple. 在某种意义上，它们都是容器，虽然它们与标准库的容器类毫无共同之处。它们都是非常有用的库，许多人和我一样，每天都在使用它们来解决编程上的问题。它们 所解决的问题是C++或C++标准库所未能覆盖的范畴，因此它们确实是我们的库工具箱的非常重要的扩展。想一下基础数据结构的有效性将影响我们编程及设计 的方法，这是多么有趣啊。如果没有这些数据结构，我们将需要自己来实现它，而我们通常只会考虑到在解决方案范畴内的实现，这样会大大限制我们的工作成果的 可重用性。当然对于所有编程风格，存在着共同的主题，而普遍性和基本性的折衷可以很好地完成这项工作。库的灵活性不仅可以让我们完成当前的任务，更可以让 我们在以后解决更多的类似问题。这些库在某种意义上也扩展了我们的C++词汇表，而且更多的用户使用这些库，就会有更大的社区使用同样的词汇。我确信本章 中的每一个库都应该在每个C++专业工具箱中占有一席之地。

Library 6. Any

Any 库如何改进你的程序？

- 任意类型的类型安全存储以及安全的取回
- 在标准库容器中存放不同类型的方法
- 可以在无须知道类型的情况下传送类型

Any库提供一个类型, `any`, 它允许存入任意类型且稍后取回，而不损失类型安全性。它有点象是可变类型的化合物：它可以持有任意类型，但你必须知道类型才能取回值。有很多次你想在同一 个容器中存入互不相关的类型。有很多次某些代码只想从一个指针向另一个指针传送数据，而不关心数据的类型。从表面看，这些事情很容易做。它们可以通过一个 无类的类型来实现，如 `void*`. 它们也可以通过使用一个含有不同类型的union来实现。有很多可变类型通过一些类型标识机制来实现。不幸的是，所有这些都缺乏类型安全性，而只有在最可 控的情形下我们才应该故意绕过类型系统。标准库的容器是要通过它们所包含的类型来特化的，这意味着不可能把不同类型的元素存入容器之内。幸运的是，解决的 方案不一定要 `void*`, 因为 Any 库允许你将存入不同的类型而稍后取回。没有办法在不知道实际类型的情况下取回存入的值， 类型安全从而得到保证。

在设计框架时，不可能预先知道哪些类型要和框架类一起使用。一个常见的方 法是，要求框架的使用者遵守某种接口，或者从框架所提供的某个基类进行派生。这是合理的，因为框架可能需要与不同的高级类进行通信才能使用。但是也存在这 样的情形，框架对于存入或接受的类型无须(或不能)知道任何相关信息。不要绕过类型系统去使用 `void*` 方法，框架可以使用 `any` 。

Any 如何适用于标准库？

Any的一个重要特性是，它提供了存储不同类型的对象到标准库容器中的能力。它也是一种可变数据类型，这正是C++标准库非常需要而又缺乏的。

Any

头文件: `"boost/any.hpp"`

类 `any` 允许对任意类型进行类型安全的存储和取回。不象无类类型， `any` 保存了类型信息，并且不会让你在不知道正确类型的情况下获得存入的值。当然，有办法可以让你询问关于类型的信息，也有测试保存的值的方法，但最终，调用者必须知道在 `any` 对象中的值的真实类型，否则不能访问 `any` 。可以把 `any` 看作为上锁的安全性。没有正确的钥匙，你不能进入其中。 `any` 对它所保存的类型有以下要求：

- CopyConstructible 它必须可以复制这个类型
- Non-throwing destructor 就象所有析构函数应该的那样！
- Assignable 为了保证强异常安全(不符合可赋值要求的类型也可以用于 `any` , 但没有强异常安全的保证)

以下是 `any` 的公有接口：

```
namespace boost {

  class any {
  public:
    any();
    any(const any&);

    template<typename ValueType>
     any(const ValueType&);

    ~any();

    any& swap(any &);
    any& operator=(const any&);

    template<typename ValueType>
     any& operator=(const ValueType&);

    bool empty() const;
    const std::type_info& type() const;
  };
}
```

成员函数

```
any();
```

缺省构造函数创建一个空的 `any` 实例，即一个不含有值的 `any` 。当然，你无法从一个空的 `any` 中取回值，因为没有值存在。

```
any(const any& other);
```

创建一个已有 `any` 对象的独立拷贝。 `other` 中含有的值被复制并存入 `this` .

```
template<typename ValueType> any(const ValueType&);
```

这个模板构造函数存入一个传入的 `ValueType` 类型参数的拷贝。参数是一个 `const` 引用，因此传入一个临时对象来存入 `any` 是合法的。注意，该构造函数不是 explicit 的，如果是的话， `any` 会难以使用，而且也不会增加安全性。

```
~any();
```

析构函数销毁含有的值，但注意，由于对一个裸指针的析构不会调用operator `delete` 或 operator `delete[]` ，所以在 `any` 中使用指针时，你应该把裸指针包装成一个象 `shared_ptr` (见 "Library 1: Smart_ptr 1") 那样的智能指针。

```
any& swap(any& other);
```

交换存在两个 `any` 对象中的值。

```
any& operator=(const any& other);
```

如果 `any` 实例非空，则丢弃所存放的值，并存入 `other` 值的拷贝。

```
template<typename ValueType>
  any& operator=(const ValueType& value);
```

如果 `any` 实例非空，则丢弃所存放的值，并存入 `value` 的一份拷贝， `value` 可以是任意符合 `any` 要求的类型。

```
bool empty() const;
```

给出 `any` 实例当前是否有值，不管是什么值。因而，当 `any` 持有一个指针时，即使该指针值为空，则 `empty` 也返回 `false` 。

```
const std::type_info& type() const;
```

给出所存值的类型。如果 `any` 为空，则类型为 `void` .

普通函数

```
template<typename ValueType>
  ValueType any_cast(const any& operand);
```

`any_cast` 让你访问 `any` 中存放的值。参数为需要取回值的 `any` 对象。如果类型 `ValueType` 与所存值不符， `any` 抛出一个 `bad_any_cast` 异常。请注意，这个语法有点象 `dynamic_cast` .

```
template<typename ValueType>
  const ValueType* any_cast(const any* operand);
```

`any_cast` 的一个重载，接受一个指向 `any` 的指针，并返回一个指向所存值的指针。如果 `any` 中的类型不是 `ValueType` , 则返回一个空指针。请再次注意，这个语法也有点象 `dynamic_cast` .

```
template<typename ValueType>
  ValueType* any_cast(any* operand);
```

`any_cast` 的另一个重载，与前一个版本相似，但前一个版本使用 `const` 指针的参数并返回 `const` 指针，这个版本则不是。

异常

```
bad_any_cast
```

当试图将一个 `any` 对象转换为该对象所存类型以外的其它类型，将抛出该异常。 `bad_any_cast` 派生自 `std::bad_cast` . 注意，使用指针参数调用 `any_cast` 时，将不抛出异常(类似于对指针使用 `dynamic_cast` 时返回空指针一样)，反之对引用类型使用 `dynamic_cast` 则会在失败时抛出异常。

用法

Any库定义在名字空间 `boost` 内。你要用类 `any` 来保存值，用模板函数 `any_cast` 来取回存放的值。为了使用 `any`，要包含头文件 `"boost/any.hpp"`。创建一个可以存放任意值的实例是很容易的。

```
boost::any a;
```

把任意类型的值赋给它也很容易。

```
a=std::string("A string");
a=42;
a=3.1415;
```

`any` 几乎可以接受任何东西！但是，为了真正能使用存放在 `any` 中的值，我们需要取回它，对吧？为此，我们需要知道这个值的类型。

```
std::string s=boost::any_cast<std::string>(a);
// 抛出 boost::bad_any_cast.
```

这显然不行；因为当前的 `a` 所持的是一个 `double`，`any_cast` 抛出一个 `bad_any_cast` 异常。以下这样则可以。

```
double d=boost::any_cast<double>(a);
```

`any` 只允许你在知道类型的前提下访问它的值，这是很明智的。对于这个库，典型情况下你只需记住两件事：类 `any`，用于存放值，还有模板函数 `any_cast`，用于取回值。

任意的东西！

考虑三个类，`A`，`B`，和 `C`，它们没有共同的基类，而我们想把它们存入一个 `std::vector`。如果它们没有共同基类，看起来我们不得不把它们当成 `void*` 来保存，对吗？唔，not any more (相关语，没有更多的了)，因为类型 `any` 没有改变对所存值的类型的依赖。以下代码示范了如何解决这个问题。

```
#include <iostream>
#include <string>
#include <utility>
#include <vector>
#include "boost/any.hpp"

class A {
public:
  void some_function() { std::cout << "A::some_function()\n"; }
};

class B {
public:
  void some_function() { std::cout << "B::some_function()\n"; }
};

class C {
public:
  void some_function() { std::cout << "C::some_function()\n"; }
};

int main() {
  std::cout << "Example of using any.\n\n";

  std::vector<boost::any> store_anything;

  store_anything.push_back(A());
  store_anything.push_back(B());
  store_anything.push_back(C());

  // 我们再来，再加一些别的东西
  store_anything.push_back(std::string("This is fantastic! "));
  store_anything.push_back(3);
  store_anything.push_back(std::make_pair(true, 7.92));

  void print_any(boost::any& a);
  // 稍后定义；打印a中的值

  std::for_each(
  store_anything.begin(),
  store_anything.end(),
  print_any);
}
```

运行以上例子，将有如下输出。

```
Example of using any.

A::some_function()
B::some_function()
C::some_function()
string: This is fantastic!
Oops!
Oops!
```

好的，我们可以保存任意我们想要的东西，但我们如何取回保存在 `vector` 的元素中的值呢？在前例中，我们用 `for_each` 来为 `vector` 中的每个元素调用 `print_any()` 。

```
void print_any(boost::any& a) {
  if (A* pA=boost::any_cast<A>(&a)) {
    pA->some_function();
  }
  else if (B* pB=boost::any_cast<B>(&a)) {
    pB->some_function();
  }
  else if (C* pC=boost::any_cast<C>(&a)) {
    pC->some_function();
  }
}
```

到目前为止，`print_any` 会试着取回一个指向 `A`，`B`，或 `C` 对象的指针。这可以使用普通函数 `any_cast` 来完成，使用要"转换"成的类型来特化该函数。看清楚些这个转换，我们试着解开这个 `any a`，对它说我们认为 `a` 包含一个类型 `A` 的值。请注意，我们以指针方式来传递我们的 `any` 给 `any_cast` 函数。因此，返回值将会是一个指向 `A`，`B`，或 `C` 的指针。如果 `any` 没有包含我们要转换成的类型，将返回空指针。在本例中，如果转型成功，我们就使用返回的指针来调用 `some_function` 成员函数。但 `any_cast` 也可以作一些小的调整。

```
  else {
    try {
      std::cout << boost::any_cast<std::string>(a) << '\n';
    }
    catch(boost::bad_any_cast&) {
      std::cout << "Oops!\n";
    }
  }
}
```

现在有点不同了。我们还是执行一个用我们要取回的类型来特化的 `any_cast`，但不是用指针的方式来传递 `any` 实例，而是用 `const` 引用来传递。这改变了 `any_cast` 的行为；这种情况下，失败——即请求一个错误的类型——将会抛出一个 `bad_any_cast` 类型的异常。因此，如果我们不能绝对肯定 `any` 中包含的是什么类型时，我们必须用一个 `try` / `catch` 块来保护执行 `any_cast` 的代码。这种行为上的差异(类似于 `dynamic_cast`)给了你更大的自由度。在转型失败不是一种错误时，使用指针来传递 `any`，但如果转型失败是一种错误，则应该使用 `const` 引用来传递，这样可以让 `any_cast` 在失败时抛出异常。

使用 `any` 让你能够在原来不能使用标准库容器和算法的地方下使用它们，从而让你可以写出更具有可维护性和更易懂的代码。

属性类

现在我们想定义一个可用于容器的属性类。我们将以字符串方式来保存属性的名字，而属性值则可以为任意 类型。虽然我们可以要求所有属性值从一个共同的基类派生而来，但这样常常不可行。例如，我们可能无法访问所有那些我们需要用作属性值的类的源代码，也有可 能有些属性值是内建类型，不可能是派生类(此外，那样也不能做出一个好的 `any` 示例)。通过把属性值存入一个 `any` 实例，我们可以让使用者去处理那些他们知道类型且感兴趣的属性值。

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include "boost/any.hpp"

class property {
  boost::any value_;
  std::string name_;

public:
  property(
    const std::string& name,
    const boost::any& value)
    : name_(name),value_(value) {}

  std::string name() const { return name_; }
  boost::any& value() { return value_; }
  friend bool operator<
    (const property& lhs, const property& rhs) {
    return lhs.name_<rhs.name_;
  }
};
```

这个例子中的 `property` 类有一个保存为 `std::string` 的名字来标识，并用一个 `any` 来保存属性值。`any` 为该实现带来的灵活性在于，我们可以使用内建类型或用户自定义类型而无须修改这个属性类。不管它简单或是复杂，一个 `any` 实例总是可以保存任意的东西。当然，使用 `any` 同时也意味着我们预先不知道可以对一个 `property` 中保存的属性值进行哪些操作，我们需要首先把属性值取出来。这暗示了如果对一个属性类预先知道有哪些类型适用，我们可能要考虑 `any` 以外的其它方法。这在进行框架设计时是罕见的，如果我们不要求一个确定的基类，我们所能够保证的就是，我们完全不知道会用到哪些类。如果你可以获得任意类型而不需要对它做任何动作，除了保有它一段时间并稍后把它还回去，那么你会发现 `any` 是最合适的。注意，这个属性类提供了 `operator&lt;` 以允许该类可以保存在标准库的关联容器中；即使没有这个操作符，`property` 仍然可以很好地用于序列容器。

以下程序使用了我们新的、灵活的 `property` 类，全亏了 `any`！`property` 类的实例被存入一个 `std::vector` 并以属性名排序(译注：原文是说存入一个 `std::map`，似有误)。

```
void print_names(const property& p) {
  std::cout << p.name() << "\n";
}

int main() {
  std::cout << "Example of using any for storing properties.\n";

  std::vector<property> properties;
  properties.push_back(
  property("B", 30));
  properties.push_back(
  property("A", std::string("Thirty something")));
  properties.push_back(property("C", 3.1415));

  std::sort(properties.begin(),properties.end());
  std::for_each(
  properties.begin(),
  properties.end(),
  print_names);

  std::cout << "\n";

  std::cout <<
  boost::any_cast<std::string>(properties[0].value()) << "\n";
  std::cout <<
  boost::any_cast<int>(properties[1].value()) << "\n";
  std::cout <<
  boost::any_cast<double>(properties[2].value()) << "\n";
}
```

注意，我们不必为 `property` 的构造函数显式创建 `any` 。这是因为 `any` 的类型转换构造函数不是 explicit 的。虽然构造函数通常应该接受一个显式声明的参数，但 `any` 是这个规则的一个例外。运行这段程序会得到以下输出。

```
Example of using any for storing properties.
A
B
C

Thirty something
30
3.1415
```

在这个例子中，由于容器被排序了，我们可以按索引取回属性值，而且我们预先知道它们各自的类型，所以我们不需要用 `try` / `catch` 块来处理取回操作。从一个 `any` 实例中取回值时，如果失败表示一个真正的错误，则应该用 `const` 引用来传递 `any` 。

```
std::string s=boost::any_cast<std::string>(a);
```

当失败不应是一个错误时，用指针来传递 `any` 。

```
std::string* ps=boost::any_cast<std::string>(&a);
```

这两种不同风格的取回操作不仅在语义上有所不同，而且返回的值也不同。如果你传递一个指针参数，你会得到一个指向保存值的指针；如果你传递一个 `const` 引用参数，你会得到一个保存值的拷贝。

如果值的类型在拷贝时代价很昂贵，就应该传递指针以避免值的拷贝。

关于 **any** 的更多

`any` 还提供了其它几个成员函数，如测试一个 `any` 实例是否为空，交换两个 `any` 实例的值。以下例子示范了如何使用它们。

```
#include <iostream>
#include <string>
#include "boost/any.hpp"

int main() {
  std::cout << "Example of using any member functions\n\n";

  boost::any a1(100);
  boost::any a2(std::string("200"));
  boost::any a3;

  std::cout << "a3 is ";
  if (!a3.empty()) {
    std::cout << "not ";
  }
  std::cout << "empty\n";

  a1.swap(a2);

  try {
    std::string s=boost::any_cast<std::string>(a1);
  std::cout << "a1 contains a string: " << s << "\n";
  }
  catch(boost::bad_any_cast& e) {
    std::cout << "I guess a1 doesn't contain a string!\n";
  }

  if (int* p=boost::any_cast<int>(&a2)) {
    std::cout << "a2 seems to have swapped contents with a1: "
      << *p << "\n";
  }
  else {
    std::cout << "Nope, no int in a2\n";
  }

  if (typeid(int)==a2.type()) {
    std::cout << "a2's type_info equals the type_info of int\n";
  }

}
```

以下是这段程序的输出结果。

```
Example of using any member functions

a3 is empty
a1 contains a string: 200
a2 seems to have swapped contents with a1: 100
a2's type_info equals the type_info of int
```

让我们来更进一步分析这段代码。为了测试一个 `any` 实例是否包含值，我们调用成员函数 `empty`. 我们这样来测试 `any a3` 。

```
std::cout << "a3 is ";
if (!a3.empty()) {
  std::cout << "not ";
}
std::cout << "empty\n";
```

因为我们是缺省构造 `a3` 的，因此 `a3.empty()` 返回 `true`. 下一件事情是交换 `a1` 和 `a2` 的内容。你可能会想为什么要交换它们的内容。一种可能的情形是当 `any` 实例的标识非常重要时(`swap` 仅交换其中包含的值)。另一个原因是在你不需要原来的值时避免拷贝。

```
a1.swap(a2);
```

最后，我们使用了成员函数 `type`, 它返回一个 `const std::type_ info&`, 用于测试所含的值是否类型 `int` 的值。

```
if (typeid(int)==a2.type()) {
```

注意，如果一个 `any` 保存了一个指针类型，则 `std::type_info` 表示的是相应的指针类型。

在any中保存指针

通常，测试 `empty` 足以知道对象是否真的包含有效的东西。但是，如果 `any` 持有的是一个指针，则要在解引用它之前额外小心地测试这个指针。仅仅简单地测试 `any` 是否为空是不够的，因为一个 `any` 在持有一个指针时会被认为是非空的，即使这个指针是空的。

```
boost::any a(static_cast<std::string*>(0));

if (!a.empty()) {
  try {
    std::string* p=boost::any_cast<std::string*>(a);
    if (p) {
      std::cout << *p;
    }
    else {
      std::cout << "The any contained a null pointer!\n";
    }
  }
  catch(boost::bad_any_cast&) {}
}
```

一个更好的办法——使用shared_ptr

在 `any` 中保存裸指针的另一个麻烦在于析构的语义。 `any` 类接受了它所存值的所有权，因为它保持一个该值的内部拷贝，并与 `any` 一起销毁它。但是，销毁一个裸指针并不会对它调用 `delete` 或 `delete[]` ！它仅仅要求归还属于指针的那点内存。这使得在 `any` 中保存指

针是有问题的，所以更好的办法是使用智能指针来代替。的确，使用智能指针(见 "Library 1: Smart_ptr 1")是在一个 `any` 中保存指针的好办法。它解决了要保证所存指针指向的内存被正确释放的问题。当智能指针被销毁时，它会正确地确保内存及其中的数据被正确销毁。作为对比，要注意 `std::auto_ptr` 不是合适的智能指针。这是因为 `auto_ptr` 没有通常的复制语义；访问一个 `any` 中的值会把内存及其中数据的所有权从 `any` 转移到返回的 `auto_ptr` 中。

考虑以下代码。

```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>
#include "boost/any.hpp"
#include "boost/shared_ptr.hpp"
```

首先，我们定义两个类， `A` 和 `B`，每个都有成员函数 `is_virtual`，它是虚拟的，还有一个成员函数 `not_virtual`，它不是虚拟的(如果它也是虚拟的，那么这个名字就真是糟透了)。我们想把这两类对象存入 `any` 。

```
class A {
public:
  virtual ~A() {
    std::cout << "A::~A()\n";
  }

  void not_virtual() {
    std::cout << "A::not_virtual()\n";
  }

  virtual void is_virtual () {
    std::cout << "A:: is_virtual ()\n";
  }
};

class B : public A {
public:

  void not_virtual() {
    std::cout << "B::not_virtual()\n";
  }

  virtual void is_virtual () {
    std::cout << "B:: is_virtual ()\n";
  }
};
```

我们现在来定义一个普通函数， `foo`，它接受一个 `any` 引用的参数，并使用 `any_cast` 来尝试将该 `any` 转为这个函数知道如何处理的类型。如果不能匹配，这个函数简单地忽略该 `any` 并返回。它分别对类型 `shared_ptr<A>` 和 `shared_ptr<B>` 进行测试，并对调用它们的 `is_virtual` (虚拟函数) 和 `not_virtual` 。

```
void foo(boost::any& a) {

  std::cout << "\n";

  // 试一下 boost::shared_ptr<A>
  try {
    boost::shared_ptr<A> ptr=
    boost::any_cast<boost::shared_ptr<A> >(a);

    std::cout << "This any contained a boost::shared_ptr<A>\n";
    ptr-> is_virtual ();
    ptr->not_virtual();
    return;
  }
  catch(boost::bad_any_cast& e) {}

  // 试一下 boost::shared_ptr<B>
  try {
    boost::shared_ptr<B> ptr=
    boost::any_cast<boost::shared_ptr<B> >(a);

    std::cout << "This any contained a boost::shared_ptr<B>\n";
    ptr-> is_virtual ();
    ptr->not_virtual();
    return;
  }
  catch(boost::bad_any_cast& e) {}

  // 如果是其它东西(如一个字符串), 则忽略它
  std::cout <<
    "The any didn't contain anything that \
     concerns this function!\n";
}
```

在 `main` 里面，我们创建两个 `any` 。然后我们引入一个作用域，再创建两个新的 `any` 。接着，我们把所有 `any` 存入 `vector` 并把其中所有元素传给函数 `foo` , 它测试它们的内容并操作它们。这时要注意我们违反了前面给出的建议，即在失败不代表错误时应该使用指针形式的 `any_cast` 。但是，因为我们在这儿用的是智能指针，这时使用异常抛出形式的 `any_cast` 的语法优势完全有理由忽略这个建议。

```
int main() {
  std::cout << "Example of any and shared_ptr\n";

  boost::any a1(boost::shared_ptr<A>(new A));
  boost::any a2(std::string("Just a string"));

  {
    boost::any b1(boost::shared_ptr<A>(new B));
    boost::any b2(boost::shared_ptr<B>(new B));
    std::vector<boost::any> vec;
    vec.push_back(a1);
    vec.push_back(a2);
    vec.push_back(b1);
    vec.push_back(b2);

    std::for_each(vec.begin(),vec.end(),foo);
    std::cout << "\n";
  }
1
  std::cout <<
    "any's b1 and b2 have been destroyed which means\n"
    "that the shared_ptrs' reference counts became zero\n";

}
```

程序运行时，将有如下输出。

```
Example of any and shared_ptr
This any contained a boost::shared_ptr<A>
A:: is_virtual ()
A::not_virtual()
The any didn't contain anything that concerns this function!
This any contained a boost::shared_ptr<A>
B:: is_virtual ()
A::not_virtual()
This any contained a boost::shared_ptr<B>
B:: is_virtual ()
B::not_virtual()
A::~A()
A::~A()
any's b1 and b2 have been destroyed which means
that the shared_ptrs' reference counts became zero
A::~A()
```

(译注：最后三行输出是原文没有的，但应该有)

首先，我们看到传给 `foo` 的 `any` 含有一个 `shared_ptr<A>`，它恰好拥有一个 `A` 的实例。输出正是我们所期望的。

接着是我们加到 `vector` 中的含有 `string` 的 `any` 。这显示了保存一些对稍后要被调用的函数而言是未知类型的类型到一个 `any`，是很有可能的，通常也是合理的；这些函数只需要处理它们需要操作的类型！

接下来的事情更有趣了，第三个元素含有一个指向 `B` 实例的 `shared_ptr<A>` 。这是一个例子，说明了 `any` 如何与其它类型一样实现多态性。当然，如果我们使用裸指针，就需要用 `static_cast` 来保存指针为我们想标识的类型。注意，函数 `A::not_virtual` 被调用而不是 `B::not_virtual`. 原原因是这个指针的静态类型是 `A*`, 而不是 `B*`.

最后一个元素含有一个 `shared_ptr<B>` ，它也正好指向一个 `B` 的实例。再一次，我们可以控制保存在 `any` 的类型，在后面一个 try 中设置相应的参数来打开它。

在里面的那个作用域结束时， `vector` 被销毁，它又销毁了内含的 `any` 实例，后者再销毁所有的 `shared_ptr`，正确地设置引用参数为零。接着，我们的指针被安全和不费力气地销毁！

这个例子显示了一些比如何与 `any` 一起使用智能指针更为重要的东西；它(又一次)显示了存入 `any` 的类型有是简单的或是复杂的都无关紧要。如果复制被存值的代价是高得惊人的，或者如果需要共享使用和控制生存期，就应该考虑使用智能指针，就象使用标准库的容器保存值一样。同样的推理也适用于 `any`, 通常这两个原则是一致的，正如在容器中使用 `any` 就意味着要保存不同的类型。

输入和输出操作符怎么啦？

`any` 用户的一个常见问题是，"为什么没有输入和输出操作符？" 这真的是有原因的。让我们从输入操作符开始。输入的语义应该是什么？它应该缺省为一个 `string` 类型吗？ `any` 当前持有的类型应该被用于从流中读取数据吗？如果是的话，那么首先为什么要用 `any` 呢？这些

问题没有好的答案，这正是为什么 `any` 没有输入操作符的原因。回答第二个问题并不容易，但差不多。让 `any` 支持一个输出操作符意味着 `any` 不再能够保存任意的类型，因为这个操作符对于保存在 `any` 中的类型增加了一个要求。如果我们不有意去使用 `operator<<`，这本无关紧要；但一个含有不支持输出操作符的类型的 `any` 实例仍是非法的，在编译时会导致一个错误。当然，只要我们提供一个模板版本的 `operator<<`，我们就可以使用 `any` 而无须要求被包含的类型支持流输出，但一旦这个操作符被实例化，这个要求还是会被打开。

看起来，这些就是没有这些操作符的原因了，对吗？如果我们给可以匹配任意东西的 `any` 提供一个有效的输出操作符，并把 `operator<<` 引入到只能从 `any` 类的实现细节进行访问的作用域，它会是什么样的呢？那样的话，我们可以在执行输出到一个流时选择抛出一个异常或返回一个错误代码(这个功能仅用于那些不支持 `operator<<` 的参数)，我们将要在运行期去做这些动作，而不影响其它代码的合法性。这种想法非常吸引我，我用几个手边的编译器上试了一下。结果不太好。我不想详细讨论它，但简而言之，这种方法需要一些很多编译器目前还不能处理的技术。但是，我们无需修改 `any` 类，我们可以创建一个利用 `any` 来保存任意类型的新类，并让这个新类支持 `operator<<`. 基本上，我们无论如何都需要做的是， `any` 要了解被含类型，知道如何进行输出，然后加到输出流中去。

增加对输出的支持——any_out

我们将定义一个类，它具有通过 `operator<<` 进行输出的功能。这增加了对被存类型的要求；作为可以保存在类 `any_out` 中的有效类型，必须要支持 `operator<<`.

```
#include <iostream>
#include <vector>
#include <string>
#include <ostream>

#include "boost/any.hpp"

class any_out {
```

该 `any_out` 类保存(任意的)值在一个 `boost::any` 类型中。总是应该选择重用，而不是重新发明！

```
boost::any o_;
```

接下来，我们声明一个抽象类 `streamer`, 它使用和 `any` 同样的设计。我们不能直接使用泛型类型，因为那样我们还不如泛化 `any_out` 算了，这样会使 `any_out` 的类型依赖于它所含值的类型，在需要存储不同类型的上下文中这样的类是没有用的。所含值的类型不能成为 `any_out` 类的标识的一部分。

```
struct streamer {
  virtual void print(std::ostream& o,boost::any& a)=0;
  virtual streamer* clone()=0;
  virtual ~streamer() {}
};
```

这里有一个窍门：我们增加了一个泛型类 `streamer_imp`，用所含类型来特化，并派生自 `streamer`. 因而，我们可以在 `any_out` 中保存一个 `streamer` 指针，并依靠多态性来完成剩余的工作(接下来，我们将为此增加一个虚拟成员函数)。

```
template <typename T> struct streamer_imp : public streamer {
```

现在，让我们实现一个虚拟函数 `print`，通过执行一个到用来特化 `streamer_imp` 的类型的 `any_cast` 来输出 `any` 中所含的值。因为我们将会用与存入 `any` 中的值相同的类型来实例化一个 `streamer_imp`，因此这个转型不会失败。

```
virtual void print(std::ostream& o,boost::any& a) {
  o << *boost::any_cast<T>(&a);
}
```

复制一个 `any_out` 时需要一个克隆函数，由于我们准备保存一个 `streamer` 指针，所以虚拟函数 `clone` 负责拷贝正确的 `streamer` 类型。

```
  virtual streamer* clone() {
    return new streamer_imp<T>();
  }
};

class any_out {
  streamer* streamer_;
  boost::any o_;
public:
```

缺省构造函数用于创建一个空的 `any_out`, 并设置 `streamer` 指针为零。

```
any_out() : streamer_(0) {}
```

`any_out` 中最有趣的函数是泛型构造函数。通过存入的值来推断出类型 `T`，并用于创建 `streamer`. 同时，值被存入 `any o_`.

```
template <typename T> any_out(const T& value) :
  streamer_(new streamer_imp<T>),o_(value) {}
```

复制构造函数很简单；我们所需做的只是确保源 `any_out a` 中的 `streamer` 非零。

```
any_out(const any_out& a)
  : streamer_(a.streamer_?a.streamer_->clone():0),o_(a.o_) {}<sup class="docfootnote">\[1

template<typename T> any_out& operator=(const T& r) {
  any_out(r).swap(*this);
  return *this;
}

any_out& operator=(const any_out& r) {
  any_out(r).swap(*this);
  return *this;
}

~any_out() {
  delete streamer_;
}
```

> [1] Rob Stewart 问我写这一行是否为了在让人困惑的比赛中拿冠军，或者只是想写 ():0) 。我不能肯定，但可以肯定看到这一行你会开心....

`swap` 函数用于更容易地实现异常安全的赋值。

```
any_out& swap(any_out& r) {
  std::swap(streamer_, r.streamer_);
  std::swap(o_,r.o_);
  return *this;
}
```

现在，我们来增加那个让我们到此的东西：输出操作符。它应该接受一个 `ostream` 引用和一个 `any_out` 引用。被保存在 `any_out` 中的 `any` 将被传递给 streamer 的虚拟函数 `print` 。

```
  friend std::ostream& operator<<(std::ostream& o,any_out& a) {
    if (a.streamer_) {
      a.streamer_->print(o,a.o_);
    }
    return o;
  }
};
```

这个类不仅提供了对包含在一个泛型类中的简单(未知)类型执行流输出的方法，它还示范了 `any` 是如何设计的。这种设计，以及这种用于安全地把一个类型包装在一个多态化的表面之后的技术，是通用的，被应用于其它很多地方。例如，它可以用于创建一个泛型的函数适配器。

让我们来测试一下我们的 `any_out` 类。

```
int main() {
  std::vector<any_out> vec;

  any_out a(std::string("I do have operator<<"));

  vec.push_back(a);
  vec.push_back(112);
  vec.push_back(65.535);

  // 打印vector vec中的所有东西
  std::cout << vec[0] << "\n";
  std::cout << vec[1] << "\n";
  std::cout << vec[2] << "\n";

  a=std::string("This is great!");
  std::cout << a;
}
```

如果类 `X` 不支持 `operator<<`，这段代码就不能编译。不幸的是，这与我们是否真的使用了 `operator<<` 无关，它就是不能工作。`any_out` 总是要求输出操作符可用。

```
  any_out nope(X());
  std::cout << nope;

}
```

很方便，你不这样认为吗？如果在某个特定上下文中，你计划使用的所有类型有某个共同的操作可用，你可以象我们前面为 `any_out` 类增加对 `operator<<` 的支持那样加上它。推广这种方法和泛化该操作并不难，这可以用来扩展 `any` 可重用性的接口。

谓词

在我们结束关于 `any` 用法的这一节之前，让我们看一下如何围绕 `any` 来建立一些功能，来简化使用和增加表现力。`any` 可用于在容器类中保存不同的类型，它可以很容易地保存这些值，但很难去操作它们。

首先，我们创建两个谓词，`is_int` 和 `is_string`，它们分别用于判断一个 `any` 是否包含一个 `int` 或一个 `string`。在我们想在一个存有不同类型对象的容器中查找特定类型时，或者是想测试一个 `any` 的类型以决定后面的动作时，这很有用。实现方法是用 `any` 的成员函数 `type` 来测试。

```
bool is_int(const boost::any& a) {
  return typeid(int)==a.type();
}

bool is_string(const boost::any& a) {
  return typeid(std::string)==a.type();
}
```

这种办法可以工作，但为每一种我们想测试的类型写一个谓词会很乏味。实现的方法是重复的，这很适合用模板的方法来解决，如下。

```
template <typename T> bool contains (const boost::any& a) {
  return typeid(T)==a.type();
}
```

函数 `contains` 让我们不必手工创建新的谓词了。这是一个示范模板如何用来最小化冗余代码的典型例子。

对非空值计数

对于某些应用，可能要对容器中所有元素进行迭代并测试每个 `any` 是否含有值。一个空的 `any` 可能意味着要被删除，也可能我们要为了更一步的处理而取出所有非空的 `any` 元素。在一个算法中这是很常用到的，我们创建一个函数对象，它的函数调用操作符接受一个 `any` 参数。该操作符只是测试 `any` 是否为空，如果不是则递增计数器。

```
class any_counter {
  int count_;
public:
  any_counter() : count_(0) {}

  int operator()(const boost::any& a) {
    return a.empty() ? count_ : ++count_;
  }

  int count() const { return count_; }
};
```

对于一个保存 `any` 的容器 `C` ，计算其中的非空值个数可以这样写。

```
int i=std::for_each(C.begin(),C.end(),any_counter()).count();
```

注意， `for_each` 算法返回的是函数对象，所以我们可以很容易地取到计数值。因为 `for_each` 是以值的方式接受参数的，所以以下代码完成的不是同一件事情。

```
any_counter counter;
std::for_each(C.begin(),C.end(),counter);
int i=counter.count();
```

第二个版本总是得到 0, 因为函数对象 counter 在调用 `for_each` 时被复制。第一个版本可以工作，因为返回的函数对象(`counter` 的一份拷贝)被用来取出计数值。

从容器中取出某种类型的元素

下面是另一个好东西：一个从容器中取出某种类型元素的提取器。在把异类容器中的一部分传递给一个同类 容器时，这是一个有用的工具。手工来做这件事是乏味且容易出错的，但一个简单的函数对象可以为我们照看好一切。我们泛化这个函数对象，用取出元素的输出迭 代器的类型，以及要从传递给该函数对象的 `any` 参数中取出的类型来参数化。

```
template <typename OutIt,typename Type> class extractor {
  OutIt it_;
public:
  extractor(OutIt it) : it_(it) {}

  void operator()(boost::any& a) {
    Type* t(boost::any_cast<Type>(&a));
    if (t) {
      *it_++ = *t;
    }
  }
};
```

为了更方便地创建一个取出器, 这里有一个函数，它可以推断出输出迭代器的类型，并返回一个相应的取出器.

```
template <typename Type, typename OutIt>
  extractor<OutIt,Type> make_extractor(OutIt it) {
    return extractor<OutIt,Type>(it);
  }
```

使用谓词和取出器

现在该用一个例程来测试一下我们新的 `any` 同伴了。

```
int main() {
  std::cout << "Example of using predicates and the "
   "function object any_counter\n";

  std::vector<boost::any> vec;
  vec.push_back(boost::any());
  for(int i=0;i<10;++i) {
    vec.push_back(i);
  }
  vec.push_back(boost::any());
```

我们把12个 `any` 对象加入到 `vec` , 现在我们想找出有多少个元素包含有值。为了计算含值元素的数量，我们使用前面创建的函数对象 `any_counter` 。

```
// 计算含有值的any实例的数量
int i=std::for_each(
  vec.begin(),
  vec.end(),
  any_counter()).count();
std::cout
  << "There are " << i << " non-empty any's in vec\n\n";
```

下面看操作一个 `any` 容器的取出器函数对象如何工作，它用来自源容器的特定类型的元素组成一个新的容器。

```
 // 从vec中取出所有int
std::list<int> lst;
std::for_each(vec.begin(),vec.end(),
  make_extractor<int>(std::back_inserter(lst)));
std::cout << "Found " << lst.size() << " ints in vec\n\n";
```

让我们清除容器 `vec` 中的内容，再加一些新的值。

```
vec.clear();

vec.push_back(std::string("This is a string"));
vec.push_back(42);
vec.push_back(3.14);
```

现在，我们试用一下已创建的谓词。首先，我们分别用两个谓词来显示 `any` 是否包含一个 `string` 或一个 `int` 。

```
if (is_string(vec[0])) {
  std::cout << "Found me a string!\n";
}

if (is_int(vec[1])) {
  std::cout << "Found me an int!\n";
}
```

正如我们前面指出的，为每一种我们要用到的类型定义一个谓词是乏味的，也是不必要的，我们只要简单地使用我们的语言优势。

```
  if (contains<double>(vec[2])) {
    std::cout <<
      "The generic tool is sweeter, found me a double!\n";
  }
}
```

运行这个例子，有如下输出。

```
Example of using predicates and the function object any_counter
There are 10 non-empty any's in vec

Found 10 ints in vec

Found me a string!
Found me an int!
The generic tool is sweeter, found me a double!
```

象以上这些小而简单的工具已经被证实是非常有用的。当然，不仅对 `any` 是这样；它是标准库容器和算法的设计中的一个特点。这些例子示范了如何与 `any` 一起使用组合函数。提供过滤、计数、操作特定类型等等，是隐藏实现细节的有效方法，并简单化了对 `any` 的使用。

遵守标准库适配器的要求

如果你觉得谓词 `contains` 很有用，你可能要注意它并不是到处都能用。它不能和标准库的适配器一起使用。下面的例子稍稍超出了本章的范围，但由于 `any` 是那么地适用于容器类，所以留下 `contains` 谓词的这点缺陷是不应该的。问题在于标准库的适配器(`bind1st` , `bind2nd` , `not1` , 和 `not2`)利用了它们所适配的谓词的一些必要条件。参数类型和结果类型必须用 `typedef` 暴露出来，这意味着我们需要的是函数对象而不是函数。

先来定义一个新的函数对象， `contains_t` . 它可以派生自辅助类 `std::unary_function` (它是C++标准库的组成部分，以便创建正确的 `typedef`)，自动地定义参数类型和结果类型，但为了让事情更清楚，我们自己来提供所需的 `typedef` 。参数类型由 `const boost::any&` 改为 `boost::any` , 以避免产生到引用的引用，那是非法的。实现和前面的一样，这里只给出函数调用操作符。

```
template <typename T> struct contains_t {
  typedef boost::any argument_type;
  typedef bool result_type;
  bool operator()(boost::any a) const {
    return typeid(T)==a.type();
  }
};
```

为了保留名字 `contains` 以用在稍后的辅助函数，我们用 `contains_t` 作这个函数对象的名字。这里有一个辅助函数用来创建并返回一个自动设置为相应类型的 `contains_t` 实例。原因是我们想重载 `contains` ，以便我们还可以提供我们原来创建的谓词。

```
template <typename T> contains_t<T> contains() {
  return contains_t<T>();
}
```

最后，旧的谓词被改为利用 `contains_t` 来实现。现在，如果我们为了某些原因要改变 `contains_t` 的实现， `contains` 可以反应出这些修改而无须更多的改进。

```
template <typename T> bool contains(const boost::any& a) {
  return contains_t<T>()(a);
}
```

下面这个例程示范了我们已经得到的东西，包括新的函数对象和前例中的谓词。

```
int main() {
  std::cout << "Example of using the improved is_type\n";

  std::vector<boost::any> vec;

  vec.push_back(std::string("This is a string"));
  vec.push_back(42);
  vec.push_back(3.14);
```

使用的谓词与前面没有什么不同。测试一个 `any` 是否某种类型仍然很容易。

```
if (contains<double>(vec[2])) {
  std::cout << "The generic tool has become sweeter! \n";
}

vec.push_back(2.52f);
vec.push_back(std::string("Another string"));
```

另一个使用 `contains` 的例子是，在一个容器中查找某种类型。这个例子查找第一个 `float`.

```
std::vector<boost::any>::iterator
  it=std::find_if(vec.begin(),vec.end(),contains<float>());
```

现在，从一个中取回所含值的两种方法都被示范出来。通过 `const` 引用传递 `any` 给 `any_cast` 的是异常抛出版本。传递 `any` 地址的版本则返回一个所存值的指针。

```
if (it!=vec.end()) {
  std::cout << "\nPrint the float twice!\n";
  std::cout << boost::any_cast<float>(*it) << "\n";
  std::cout << *boost::any_cast<float>(&*it) << "\n";
}
std::cout <<
  "There are " << vec.size() << " elements in vec\n";
```

我还没有给出一个好的例子来说明为什么 `contains` 应该是一个发育完全的函数对象。在很多情形下，原因可能无法预先知道，因为我们不能预见我们的实现将会面对的每一种情形。一个强烈的原因是遵守标准库的要求，更适用于我们已知的用例以外的情形。然而，我还是给你一个例子：任务是从一个容器 `vec` 中删除所有不含 `string` 的元素。当然，另写一个谓词来做与 `contains` 相反的事情是一种方法，但这样会很快导致维护的恶梦，因为类似作用的函数对象要不断增生。标准库提供给我们一个名为 `not1` 的适配器，它对一个函数对象的结果取反，它可以轻易地从我们的 `vector vec` 中清除所有非 `string` 元素。

```
  vec.erase(std::remove_if(vec.begin(),vec.end(),
    std::not1(contains<std::string>())),vec.end());

  std::cout << "Now, there are only " << vec.size()
    << " elements left in vec!\n";
}
```

本节的例子示范了如何有效地使用 `any`. 因为所存值的类型不是 `any` 的类型的组成部分，要在不对所存类型强加要求(包括从同一基类派生)的前提下提供存储， `any` 是一个基本工具。我们已经看到这个类型隐藏有某种价值。 `any` 不允许在对值的类型不了解的情况下访问所保存的值，限制了对所存值进行操作的机会。作为大的扩展，通过创建一些辅助类——谓词和函数对象——提供所需的逻辑来访问所存值，可以进行补偿。

Any 总结

这个类型可以包含不同类型的值，而且与无类类型(如 `void*`)有很大不同。我们总是严重地依赖C++中的类型安全，只有在极少数情形下我们会愿意没有它来干活。

这是有很好的原因的：类型安全防止我们犯错，并改善了我们代码的性能。因此，我们应该避免无类类型。还有，发现自己需要异类存储的情形很少见，或者为了将使用者隔离于类型的细节，或者为了在更低的层次获得极度的灵活性。 `any` 提供了这些功能，同时维护了类型安全，它是我们的工具箱的最好扩充！

在以下情形时使用 Any 库：

- 你需要在容器中存放不同类型的值
- 需要保存未知类型
- 类型被传递到无须知晓任何有关该类型信息的层次

Any 的设计同时也是一门很有价值的课程，关于如何封装一个类型而不影响到该类型的封套类。这种设计可以用于创建泛型函数对象、泛型迭代器等等。它是一个展示封装的威力以及与模板相关的多态性的例子。

在标准库中，有很好的工具来存放多个元素。当需要存储异类的元素时，我们想避免使用新的集合类型。 `any` 提供了一种方法，在大多数情况下它可以与已有容器一起使用。在某种程度上，模板类 `any` 扩展了标准库容器的能力，把不同的类型封入一个同类型的包装中，就可以把它们放入前述容器中了。

把 Boost.Any 加到已有代码中是很简单的。它不需要修改设计，并且立即就增加了灵活性。接口非常小，这使得它成为一个很容易理解的工具。

Any 库由 Kevlin Henney 创建，与所有 Boost 库一样，它由 Boost 社区复审、改进和强化。

Library 7. Variant

- Variant 库如何改进你的程序？
- Variant 如果适用于标准库？
- Variant
- 用法
- Variant 总结

Variant 库如何改进你的程序？

- 对用户指定的多种类型的进行类型安全的存储和取回
- 在标准库容器中存储不同类型的方法
- 变量访问的编译期检查
- 高效的、基于栈的变量存储

Variant 库关注的是对一组限定类型的类型安全存储及取回，即非无类的联合。Boost.Variant 库与 Boost.Any 有许多共同之外，但在功能上也有不同的考虑。在每天的编程中通常都会需要用到非无类的联合(不同的类型)。保持类型安全的一个典型方法是使用抽象基类，但这不总是可以做到的；即使可以做得，堆分配和虚拟函数[1]的代价也可能太高。你也可以尝试用不安全的无类类型，如 `void*` (它会导致不幸)，或者是类型安全得无限制的可变类型，如 Boost.Any. 这里我们将看到 Boost.Variant，它支持限定的可变类型，即元素来自于一组支持的类型。

> [1] 尽管虚拟函数在性能方面有非常合理的代价。

许多其它的编程语言支持可变类型，它们也再次被证实是值得的。在C++内建的对可变类型的支持非常有限，只有某种形式的联合(union)，而且主要是为了与C兼容而保留。Boost.Variant 通过一个类型模板 `variant` 补救了这种情形，并附随有安全的存储及取回值的工具。一个可变数据类型提供一个与当前值的类型无关的接口。如果你曾经用过别的可变类型，可能是仅能支持固定的一组类型。这个库不是这样的；你在使用 `variant` 时自己定义一组允许使用的类型，而一个程序中可以包含任意个不同的 `variant` 实例。为了取回保存在 `variant` 中的值，你要么知道当前值的真实类型，要么使用已提供的类型安全的访问者(visitor)机制。访问者机制使得 Variant 非常不同于其它可变类型的库，包括 Boost.Any (它可以持有任意类型的值)，从而为处理这些类型提供了一个安全而健壮的环境。C++ 的联合只对内建类型以及 POD 类型有用，但这个库提供的非无类联合可以支持所有类型。最后，效率方面也被考虑到了，这个库基于栈存储来保存它的值，从而避免了昂贵的堆分配。

Variant 如何适用于标准库？

Boost.Variant 允许在标准库容器中存储不同的类型。由于在C++或C++标准库中都没有对可变类型的真正支持，这使得 Variant 成为了标准库的一个杰出且有用的扩充。

Variant

头文件: `"boost/variant.hpp"`

通过单个头文件就包含了所有 Variant 库。

```
"boost/variant/variant_fwd.hpp"
```

包含了 `variant` 类模板的前向声明。

```
"boost/variant/variant.hpp"
```

包含了 `variant` 类模板的定义。

```
"boost/variant/apply_visitor.hpp"
```

包含了对 `variant` 应用访问者机制的功能。

```
"boost/variant/get.hpp"
```

包含了模板函数 `get`.

```
"boost/variant/bad_visit.hpp"
```

包含了异常类 `bad_visit` 的定义。

```
"boost/variant/static_visitor.hpp"
```

包含了 `visitor` 类模板的定义。

以下部分摘要包含了 `variant` 类模板中最重要的成员。其它功能，如访问者机制，类型安全的直接取回，还有更先进的特性，如通过类型列表创建类型组等等，在 "Usage" 节讨论。

```
namespace boost {
  template <typename T1,typename T2=unspecified, ...,
    typename TN=unspecified>
  class variant {
  public:

    variant();

    variant(const variant& other);

    template <typename T> variant(const T& operand);

    template <typename U1, typename U2, ..., typename UN>
      variant(const variant<U1, U2, ..., UN>& operand);

    ~variant();

    template <typename T> variant& operator=(const T& rhs);

    int which() const;
    bool empty() const;
    const std::type_info& type() const;

    bool operator==(const variant& rhs) const;
    bool operator<(const variant& rhs) const;
  };
}
```

成员函数

```
variant();
```

这个构造函数对 `variant` 的类型组中的第一个类型进行缺省构造。这意味着在声明 `variant` 类型时，第一个类型必须是可以被缺省构造的，或者 `variant` 类型本身不能被缺省构造。该构造函数传播任何从第一个类型的构造函数抛出的异常。

```
variant(const variant& other);
```

这个复制构造函数复制 `other` 的当前值，并传播任何从 `other` 的当前类型的复制构造函数抛出的异常。

```
template <typename T> variant(const T& operand);
```

这个构造函数从 `operand` 构造一个新的 `variant` 。operand 的类型 `T` , 必须可以转换为限定类型组中的某个类型。复制或转换 operand 时抛出的异常将被传播。

```
template <typename U1,typename U2,...,typename UN>
  variant(const variant<U1,U2,...,UN>& operand);
```

这个构造函数允许从另一个 `variant` 类型进行构造，后者的类型组为 `U1`, `U2` ... `UN`, 它们必须可以转换为 `T1`, `T2` ... `TN` (被构造的 variant 的类型组)。复制或转换 operand 时抛出的异常将被传播。

```
~variant();
```

销毁 variant, 并调用当前值的析构函数。注意，对于指针类型，不调用析构函数(销毁指针是无操作的)。析构函数不抛出异常。

```
template <typename T> variant& operator=(const T& rhs);
```

这个操作符放弃当前值，并赋予值 `rhs`. 类型 `T` 必须可以转换为 `variant` 的限定类型组中的某个类型。如果 `T` 正好是 `variant` 当前值的类型， `rhs` 被复制赋值给当前值；从 `T` 的赋值操作符抛出的异常将被传播。如果 `variant` 当前值的类型不是 `T`, 则当前值被替换为从类型 `T` 的(复制)构造函数所创建的值。从构造函数抛出的异常将被传播。这个函数还可能抛出 `bad_alloc`.

```
int which() const;
```

返回一个从零起计的索引，表示当前值类型在限定类型组中的位置。这个函数不会抛出异常。

```
bool empty() const;
```

这个函数永远返回 `false`, 因为一个 `variant` 永远不会为空。这个函数的存在是为了允许泛型代码把 `variant` 和 `boost::any` 视为同一种类型来处理。这个函数不会抛出异常。

```
const std::type_info& type() const;
```

返回当前值的 `type_info` 。这个函数不会抛出异常。

```
bool operator==(const variant& rhs) const;
```

如果 `*this` and `rhs` 相等则返回 `true` ，即 `which()==rhs.which()` 且 `*this` 的当前值与 `rhs` 根据当前值的类型的相等操作是相等的。这要求限定类型组中的所有类型都必须是可以进行等同性比较的(EqualityComparable)。当前值的类型的 `operator==` 抛出的任何异常将被传播。

```
bool operator<(const variant& rhs) const;
```

小于比较返回 `which()&lt;rhs.which()` 或者，如果该索引相等，则返回对 `*this` 的当前值与 `rhs` 调用 `operator&lt;` 所返回的结果。当前值的类型的 `operator&lt;` 抛出的任何异常将被传播。

用法

在你的程序中使用 `variant`，要包含头文件 `"boost/variant.hpp"`。这个头文件包含了整个库，所以你不必知道要使用哪些单独的特性；以后，如果你要降低相关性，可以只包含那些解决问题所要的头文件。声明一个 `variant` 类型时，我们必须定义一组它可以存储的类型。最常用的办法是使用模板参数。一个可以持有类型为 `int`，`std::string`，或 `double` 的值的 `variant` 声明如下。

```
boost::variant<int,std::string,double> my_first_variant;
```

当变量 `my_first_variant` 被创建时，它含有一个缺省构造的 `int`，因为 `int` 是这个 `variant` 可以持有的类型中的第一种类型。我们也可以传递一个可以转换为可用类型之一的值来初始化 `variant`.

```
boost::variant<int,std::string,double>
 my_first_variant("Hello world");
```

我们可以随时赋给新的值，只要这个新值有确定的类型并且可以转换为 `variant` 可以持有的类型中的某一种，它可以很好地工作。

```
my_first_variant=24;
my_first_variant=2.52;
my_first_variant="Fabulous!";
my_first_variant=0;
```

在第一个赋值后，所含值的类型为 `int`；第二个赋值后，类型为 `double`；第三个后，类型为 `std::string`；最后，又变回 `int`. 如果我们想看看，我们可以用函数 `boost::get` 取出这个值，如下：

```
assert(boost::get<int>(my_first_variant)==0);
```

注意，如果调用 `get` 失败(当 `my_first_variant` 所含值不是类型 `int` 时就会发生)，会抛出一个类型为 `boost::bad_get` 的异常。为了避免在失败时得到一个异常，我们可以传给 `get` 一个 `variant` 指针，这样 `get` 将返回一个指向它所含值的指针，或者如果给定类型与 `variant` 的值的类型不符则返回空指针。以下是它的用法：

```
int* val=boost::get<int>(&my_first_variant);
assert(val && (*val)==0);
```

函数 `get` 是访问所含值的一种直接方法，事实上它与 `boost::any` 的 `any_cast` 很相似。注意，类型必须完全符合，包括相同的 cv-限定符(`const` 和 `volatile`)。但是，可以使用限制更多的 cv-限定符。如果类型不匹配且传给 `get` 的是一个 `variant` 指针，将返回空指针。否则，抛出一个类型为 `bad_get` 的异常。

```
const int& i=boost::get<const int>(my_first_variant);
```

过分依赖于 `get` 的代码很容易变得脆弱；如果我们不知道所含值的类型，我们可能会想测试所有可能的组合，就如下面这个例子的做法。

```
#include <iostream>
#include <string>
#include "boost/variant.hpp"

template <typename V> void print(V& v) {
  if (int* pi=boost::get<int>(&v))
    std::cout << "It's an int: " << *pi << '\n';
  else if (std::string* ps=boost::get<std::string>(&v))
    std::cout << "It's a std::string: " << *ps << '\n';
  else if (double* pd=boost::get<double>(&v))
    std::cout << "It's a double: " << *pd << '\n';

  std::cout << "My work here is done!\n";
}

int main() {
  boost::variant<int,std::string,double>
    my_first_variant("Hello there!");
  print(my_first_variant);
  my_first_variant=12;
  print(my_first_variant);
  my_first_variant=1.1;
  print(my_first_variant);
}
```

函数 `print` 现在可以正确工作，但如果我们决定改变 `variant` 的类型组的话会怎样？我们将引入一个微妙的bug，而不能在编译期捉住它；函数 `print` 不能打印任何其它我们没有预先想到的类型的值。如果我们没有使用模板函数，而是要求一个明确的 `variant` 类型，我们就要为不同类型的 `variant` 重载多个相同功能的函数。下一节将讨论访问 `variant` 的概念，以及这种(类型安全的)访问机制解决的问题。

访问Variants

让我们从一个例子开始，它解释了为什么使用 `get` 并没有你想要的那么可靠。从前面父子的代码开始，我们来修改一下 `variant` 可以包含的类型，并对 `variant` 的一个 `char` 值来调用 `print` 。

```
int main() {
  boost::variant<int,std::string,double,char>
    my_first_variant("Hello there!");

  print(my_first_variant);
  my_first_variant=12;
  print(my_first_variant);
  my_first_variant=1.1;
  print(my_first_variant);
  my_first_variant='a';
  print(my_first_variant);
}
```

虽然我们给 `variant` 的类型组增加了 `char`，并且程序的最后两行设置了一个 `char` 值并调用 `print`，编译器也不会有意见 (注意，`print` 是以 `variant` 的类型来特化的，所以它可以很容易适应新的 `variant` 定义)。以下是这个程序的运行结果：

```
It's a std::string: Hello there!
My work here is done!
It's an int: 12
My work here is done!
It's a double: 1.1
My work here is done!
My work here is done!
```

这个输出显示了一个问题。最后一个"My work here is done!"之前没有值的报告。原因是很简单，`print` 不能输出除了它原来设计好的那些类型(`std::string`, `int`, 和 `double`)以外的任何值，但它可以干净地编译和运行。如果 `variant` 的当前类型不被 `print` 支持，它的值就会被简单地忽略掉。使用 `get` 还有更多潜在的问题，例如 if 语句的顺序要与类的层次相一致。注意，这并不是说你应该完全避免使用 `get`；它只是说有些时候它不是最好的方法。有一种更好的机制，可以允许我们规定哪些类型的值可以接受，并且这些规定是在编译期生效的。这是就 `variant` 访问机制的作用。通过把一个访问器应用到 `variant`，编译器可以保证它们完全兼容。Boost.Variant 中这些访问器是带有一些函数调用操作符的函数对象，这些函数调用操作符接受与它们所访问的 `variant` 可以包含的类型组相对应的参数。

现在我们用访问器来重写那个声名狼籍的函数 `print`，如下：

```
class print_visitor : public boost::static_visitor<void> {
public:
  void operator()(int i) const {
    std::cout << "It's an int: " << i << '\n';
  }

  void operator()(std::string s) const {
    std::cout << "It's a std::string: " << s << '\n';
  }

  void operator()(double d) const {
    std::cout << "It's a double: " << d << '\n';
  }

};
```

要让 `print_visitor` 成为 `variant` 的一个访问器，我们要让它派生自 `boost::static_visitor` 以获得正确的 `typedef` (`result_type`), 并明确地声明这个类是一个访问器类型。这个类实现了三个重载版本的函数调用操作符，分别接受一个 `int` , 一个 `std::string` , 和一个 `double` 。为了访问 `variant` , 你要用函数 `boost::apply_visitor` (visitor, variant). 如果我们用对 `apply_visitor` 的调用来替换前面的 `print` 调用，我们可以得到如下代码：

```
int main() {
  boost::variant<int,std::string,double,char>
    my_first_variant("Hello there!");

  print_visitor v;

  boost::apply_visitor(v,my_first_variant);
  my_first_variant=12;
  boost::apply_visitor(v,my_first_variant);
  my_first_variant=1.1;
  boost::apply_visitor(v,my_first_variant);
  my_first_variant='a';
  boost::apply_visitor(v,my_first_variant);
}
```

这里，我们创建了一个 `print_visitor` , 名为 `v` , 并把它应用于赋值后的 `my_first_ variant` 。因为我们没有一个函数调用操作符接受 `char` , 这段代码会编译失败，是吗？错！一个 `char` 可以转换为一个 `int` , 所以这个访问器可以兼容我们的 `variant` 类型。以下是程序运行的结果。

```
It's a std::string: Hello there!
It's an int: 12
It's a double: 1.1
It's an int: 97
```

这里我们可以学到两件事情：第一个是字母 `a` 的 ASCII 码值为 97, 更重要的是第二个，如果一个访问器以传值的方式传递参数，则传送的值可以应用隐式转换。如果我们想访问器只能使用精确的类型(同时也避免拷贝从 `variant` 得到的值)，我们必须修改访问器的调用操作符传递参数的方式。以下这个版本的 `print_visitor` 只能使用类型 `int` , `std::string` , 和 `double` ; 以及可以隐式转换到这些类型的引用的其它类型。

```
class print_visitor : public boost::static_visitor<void> {
public:
  void operator()(int& i) const {
    std::cout << "It's an int: " << i << '\n';
  }

  void operator()(std::string& s) const {
    std::cout << "It's a std::string: " << s << '\n';
  }

  void operator()(double& d) const {
    std::cout << "It's a double: " << d << '\n';
  }
};
```

如果再编译一下这个程序，编译器就不高兴了，它会输出如下信息：

```
c:/boost_cvs/boost/boost/variant/variant.hpp:
In member function `typename Visitor::result_type boost::detail:: variant::
invoke_visitor<Visitor>::internal_visit(T&, int)
[with T = char, Visitor = print_visitor]':

[Snipped lines of irrelevant information here]

c:/boost_cvs/boost/boost/variant/variant.hpp:807:
error: no match for call to `(print_visitor) (char&)'
variant_sample1.cpp:40: error: candidates are:
 void print_visitor::operator()(int&) const
variant_sample1.cpp:44: error:
 void print_visitor::operator()(std::string&) const
variant_sample1.cpp:48: error:
 void print_visitor::operator()(double&) const
```

这个错误指出了问题：没有一个候选的函数接受 `char` 参数！为什么说类型安全的编译期访问机制是一个强大的机制，这正是一个重要的原因。它使得访问机制强烈依赖于类型，避免了讨厌的类型变换。创建访问器与创建其它函数对象一样容易，因此学习曲线并不陡峭。当 `variant` 中的类型组可能会改变时(它们总是倾向于变化！)，创建访问器要比单单依赖 `get` 更可靠。虽然开始需要更高的代价，但绝对是值得的。

泛型访问器

通过使用访问器机制和泛型的调用操作符，可以创建能够接受任意类型的泛型访问器(无论是在语法上还是语义上，都可以实现泛型调用操作符)。这对于统一地处理不同的类型非常有用。C++的操作符就是"通用"性的一个典型例子，如算术和IO流的位移操作符。以下例子使用 `operator&lt;&lt;` 来输出 `variant` 的值到一个流。

```
#include <iostream>
#include <sstream>
#include <string>
#include <sstream>
#include "boost/variant.hpp"

class stream_output_visitor :
  public boost::static_visitor<void> {
  std::ostream& os_;
public:
  stream_output_visitor(std::ostream& os) : os_(os) {}

  template <typename T> void operator()(T& t) const {
    os_ << t << '\n';
  }
};

int main() {
  boost::variant<int,std::string> var;
  var=100;
  boost::apply_visitor(stream_output_visitor(std::cout),var);
  var="One hundred";
  boost::apply_visitor(stream_output_visitor(std::cout),var);
}
```

主要思想是 `stream_output_visitor` 中的调用操作符是一个成员函数模板，它在访问每一种类型(本例中是 `int` 和 `std::string`)时分别实例化。因为 `std::cout &lt;&lt; 100` 和 `std::cout &lt;&lt; std::string("One hundred")` 都已经有定义了，所以这段代码可以编译并工作良好。

当然，操作符仅是可以使用泛型访问器的一个例子；它们常常应用于更多的类型。在某些值上调用函数，或 者将它们作为参数传给其它的函数时，要求就是对于所有传给操作符的类型都要有相应的成员函数存在，并且对于被调用的函数要有合适的重载。这种泛型调用操作 符的另一个有趣的方面是，可以对某些类型特化其行为，但对于其余类型则仍允许泛型的实现。在某种意义上，这涉及到模板特化，即基于类型信息的行为特殊化。

二元访问器

我们前面看到的访问器都是一元的，即它们只接受一个 `variant` 作为唯一的参数。二元访问器接受两个(可能是不同的) `variant` . 这种概念对于实现两个 `variant` 间的关系很有用。作为例子，我们为 `variant` 类型将创建一个按字典顺序的排序。为此，我们使用一个来自于标准库的非常有用的组件： `std::ostringstream` . 它接受任意可流输出的东西，并且在需要时产生一个独立的 `std::string` 。我们从而可以按字典序比较完全不同的 `variant` 类型，只要假设所有限定的类型都支持流输出。和普通的访问器一样，二元访问器也派生自 `boost::static_visitor` , 并且用模板参数表示调用操作符的返回类型。因为我们是创建一个谓词，因此返回类型为 `bool` . 以下是一个我们即将用到的二元谓词。

```
class lexicographical_visitor :
  public boost::static_visitor<bool> {
public:
  template <typename LHS,typename RHS>
  bool operator()(const LHS& lhs,const RHS& rhs) const {
    return get_string(lhs)<get_string(rhs);
  }
private:
  template <typename T> static std::string
  get_string(const T& t) {
    std::ostringstream s;
    s << t;
    return s.str();
  }

  static const std::string& get_string(const std::string& s) {
    return s;
  }
};
```

这里的调用操作符泛化了它的两个参数，这意味着它接受任意两种类型的组合。对于 `variant` 的可用类型组的要求就是它们必须是可流输出(OutputStreamable)的。成员函数模板 `get_string` 使用一个 `std::ostringstream` 来把它的参数转换为字符串表示，所以要求参数必须是可流输出的(为了使用 `std::ostringstream` , 记得要包含头文件 `&lt;sstream&gt;`)。成员函数 `get_string` 针对类型为 `std::string` 的参数进行特化，由于类型已经符合要求，所以它跳过了 `std::ostringstream` 而直接返回它的参数。在两个参数都转为 `std::string` 以

后，剩下的就是使用 `operator<` 来比较它们了。现在我们把这个访问器放入测试代码，来对一个容器中的元素进行排序(我们还将重用我们在本章前面创建的 `stream_output_visitor`)。

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include "boost/variant.hpp"

int main() {
  boost::variant<int,std::string> var1="100";
  boost::variant<double> var2=99.99;

  std::cout << "var1<var2: " <<
    boost::apply_visitor(
      lexicographical_visitor(),var1,var2) << '\n';

  typedef std::vector<
    boost::variant<int,std::string,double> > vec_type;

  vec_type vec;
  vec.push_back("Hello");
  vec.push_back(12);
  vec.push_back(1.12);
  vec.push_back("0");

  stream_output_visitor sv(std::cout);
  std::for_each(vec.begin(),vec.end(),sv);

  lexicographical_visitor lv;
  std::sort(vec.begin(),vec.end(),boost::apply_visitor(lv));

  std::cout << '\n';
  std::for_each(vec.begin(),vec.end(),sv);
};
```

首先，我们将访问应用于两个 `variants` , `var1` 和 `var2` , 如下：

```
boost::apply_visitor(lexicographical_visitor(),var1,var2)
```

如你所见，与一元访问器不同的是，有两个 `variant` 被传递给函数 `apply_visitor` . 一个更为常见的用例是使用这个谓词来对元素进行排序，我们这样来做：

```
lexicographical_visitor lv;
std::sort(vec.begin(),vec.end(),boost::apply_visitor(lv));
```

当 `sort` 算法被执行时，它使用我们传入的谓词来比较它的元素，它是一个 `lexicographical_visitor` 实例。注意， `boost::variant` 已经定义了 `operator<` , 所以不使用谓词也可以对容器进行排序。

```
std::sort(vec.begin(),vec.end());
```

但是这种缺省的排序是首先使用 `which` 来检查当前值的索引，所以元素的排列顺序将是 12, 0, Hello, 1.12, 而我们想要的是按字典序来排序。因为 `variant` 类已经提供了 `operator<` 和 `operator==` ，所以 `variant` 可以用作所有标准库容器的元素类型。当缺省的关系比较不够用时，你需要用二元访问器来实现一个。

更多应该知道的事情

我们并没有涉及到 Boost.Variant 库的所有功能。其它更为先进的特性不如我们已经提到的那么常用。但是，我会简要地说一下，因此你将至少知道在需要时可以找到哪些可用的东西。宏 `BOOST_VARIANT_ENUM_PARAMS` , 可用于为 `variant` 类型重载/特化函数和类模板。这个宏用于列举 `variant` 可以包含的类型组。还有支持使用类型序列来创建 `variant` 类型，即通过 `make_variant_over` 编译期列表来表示 `variant` 的类型组。还有递归的 `variant` 类型，可用于创建它们自己类型的表达式，递归 `variant` 类型使用 `recursive_wrapper` , `make_recursive_variant` , 和 `make_recursive_variant_over` . 如果你需要这些额外的特性，在线文档可以很好地解释它们。

Variant 总结

类别联合(discriminated unions)在日常编程中非常有用，这个事实无须惊讶，Boost.Variant 库提供了高效且易用的 `variant` 类型，它正是基于类别联合的。因为C++的联合对于很多类型很难使用(它只支持内建类型和 POD 类型)，长期以来一直需要别的东西来取代它。许多创建类别联合的尝试都存在某些重要的缺点。例如，早期的尝试通常仅支持固定的一组类型，的确妨碍了维护性 和灵活性。Boost.Variant 通过模板避免了这些限制，理论上允许创建任意的 `variant` 类型。在处理类别联合时类型转换代码总会成为问题所在；在处理前需要测试当前值的类型，这导致了维护的麻烦。Boost.Variant 提供了简单的值取回操作以及类型安全的访问机制，这是解决问题的新颖方法。最后，效率也是早期的尝试所关心的，这个库也很好地照顾到了效率，它使用基于栈 的存储，而不是基于堆的。

Boost.Variant 是一个成熟的库，有非常多的特性，使用 `variant` 类型容易且高效。这是 Boost.Any 库的补充，同样应该成为你的专业C++工具箱中的一员。

Boost.Variant 的作者是 Eric Friedman 和 Itay Maman.

Library 8. Tuple

- Tuple 库如何改进你的程序？
- Tuple 库如何适用于标准库？
- Tuple
- 用法
- Tuple 总结

Tuple 库如何改进你的程序？

- 从函数返回多个返回值
- 相关类型的组合
- 将数值组合起来

与许多其它的编程语言一样，C++允许函数返回一个数值。但是，这一个数值可以是任意的类型，你可以用一个 `struct` 或 `class` 把多个数值组合起来作为结果。虽然可以，但是用这样的结构来组合相关的返回值通常都是很不方便的，因为这意味着要为对一种返回类型进行定义。为了避免在返回值中拷贝大量的对象，同时也为了避免创建一个特殊的类型用于从函数返回多个数值，我们常常使用非 `const` 引用参数或者指针参数，从而允许函数通过这些参数设置调用者的变量。在多数情况下这样做都工作良好，但也有人不愿意使用输出参数。还有，输出参数不能明确指出返回值就是返回值。有些时候， `std::pair` 可以满足要求，但在需要返回两个以上数值时，它就不能满足要求了。

为了提供多个返回值，我们需要一个 tuple 结构。一个 tuple 是一个固定大小的、多个指定类型的数值的聚集。相应的例子包括有：pairs, triples, quadruples, 等等。有些语言本身就内建有这样的 tuple 类型，但C++没有。借助C++本身的强大功能，这一缺点可以通过库来弥补，如你所想， Boost.Tuple 正是这样的一个库。

Tuple 库提供了 tuple 结构，它可以方便地用于返回多个数值，也可以组合任意的类型并以泛型代码来操作它们。

Tuple 库如何适用于标准库？

标准库提供了一个 tuple 的特例，一个 2-tuple, 名为 `std::pair`. 这个结构被用于标准库的容器，你可能在操作 `std::map` 的元素时已经留意到了。你也可以在容器类中存储 `pair` 。当然， `std::pair` 不仅是为了给容器类使用的，它还有它自己的用途，它附带有一个方便的函数 `std::make_pair`, 可以自动地进行类型推断，还有一组操作符用于 `pair` 的比较。一个 tuple 的通常解决方案，而不仅仅是 2-tuples，会更加有用。Tuple 库所提供的还不是完全通用的，它最多可以允许10个元素的 tuple (如果需要更多的，看起来不常见但也不是没有可能的，这个限制可以放松)。还有，这些 tuples 的效率与使用 `struct` 的手工解决方案同样高！

Tuple

头文件: `"boost/tuple/tuple.hpp"`

它包含了 `tuple` 类模板及库的核心部分。

```
Header: "boost/tuple/tuple_io.hpp"
```

包含了对 `tuple` 的输入输出操作符。

```
Header: "boost/tuple/tuple_comparison.hpp"
```

包含了 `tuple` 的关系操作符。

Tuple 库位于 `boost` 里的嵌套名字空间 `boost::tuples` 中。要使用 tuples, 需要包含 " `boost/tuple/tuple.hpp"` , 它包含了核心库。要进行输入输出操作，就包含 `"boost/tuple/tuple_io.hpp"` , 要支持 `tuple` 的比较，就包含 `"boost/tuple/tuple_comparison.hpp"` . 有些 Boost 库提供一个包含了所有相关库的头文件以方便使用；但 Boost.Tuple 没有。原因是把库分到各个不同的头文件中可以减少编译时间；如果你不使用关系操作符，你就无须为此付出时间和依赖性的代价。为了方便使用，Tuple 库中有些名字位于名字空间 `boost` ：如 `tuple` , `make_tuple` , `tie` , 和 `get` . 以下是 Boost.Tuple 的部分摘要，列出并简要讨论了最主要的一些函数。

```
namespace boost {

  template <class T1,class T2,...,class TM> class tuple {
  public:
    tuple();

    template <class P1,class P2...,class PM>
      tuple(class P1,class P2,...,PN);

    template <class U1,class U2,...,class UN>
    tuple(const tuple<U1,U2,...,UN>&);

    tuple& operator=(const tuple&);
  };

  template<class T1,class T2,...,class TN> tuple<V1,V2,...,VN>
    make_tuple(const T1& t1,const T2& t2,...,const TN& tn);

  template<class T1,class T2,...,class TN> tuple<T1&,T2&,...,TN>
    tie(T1& t1,T2& t2,...,TN& tn);

  template <int I,class T1,class T2,...,class TN>
    RI get(tuple<T1,T2,...,TN>& t);

  template <int I,class T1,class T2,...,class TN>
    PI get(const tuple<T1,T2,...,TN>& t);

  template <class T1,class T2,...,class TM,
            class U1,class U2,...,class UM>
    bool operator==(const tuple<T1,T2,...,TM>& t,
                    const tuple<U1,U2,...,UM>& u);

  template <class T1,class T2,...,class TM,
            class U1,class U2,...,class UM>
    bool operator!=(const tuple<T1,T2,...,TM>& t,
                    const tuple<U1,U2,...,UM>& u);

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

成员函数

```
tuple();
```

`tuple` 的缺省构造函数初始化所有元素，这意味着这些元素必须是可以缺省构造的，它们必须有一个公有的缺省构造函数。任何从这些所含元素的构造函数抛出的异常都会被传播。

```
template <class P1,class P2...,class PM>
  tuple(class P1,class P2,...,PN);
```

这个构造函数接受一些参数，用于初始化 `tuple` 相应元素。对于一些带有非缺省构造类型的 `tuple` ，就需要用这种构造方式；不能缺省构造一个 `tuple` 而不构造它的所有元素。例如，引用类型的元素必须在构造时初始化。注意，参数的数量不必与 tuple 类型中的元素数量一致。可以仅给出部分元素的值，而让剩余元素初始化为缺省值。任何从元素的构造函数抛出的异常都会被传播。

```
template <class U1,class U2,...,class UN>
  tuple(const tuple<U1,U2,...,UN>&);
```

这个构造函数用来自另一个 tuple 的元素来进行初始化，要求被构造的 tuple (`T1` , `T2` , ..., `TM`) 的每一个元素都必须可以从 (`U1` , `U2` ,..., `UN`) 构造。任何从元素的构造函数抛出的异常都会被传播。

```
TIndex & get<int Index>();
const TIndex & get<int Index>() const;
```

返回位于给定的 `Index` 处的元素的引用。 `Index` 必须是一个常量整型表达式；如果索引大于或等于 `tuple` 中的元素数量，将产生一个编译期错误。结果的类型通过相应的模板参数 `TIndex` 给出。

```
tuple& operator=(const tuple& other);
```

tuple 的赋值要求两个 tuples 具有相同的长度和元素类型。 `*this` 中的每一个元素被赋值为 `other` 的对应元素。元素赋值中的任何异常都会被传播。

普通函数

```
template<class T1,class T2,...,class TN> tuple<V1,V2,...,VN>
  make_tuple(const T1& t1,const T2& t2,...,const TN& tn);
```

函数模板 `make_tuple` 是 `tuple` 版本的 `std::make_pair` . 它使用函数模板参数推断来决定一个包含这些参数的 `tuple` 的元素类型。创建这个 `tuple` 的元素类型时不使用这些参数的高级 cv-限定符。要控制对引用类型的类型推断，可以使用 Boost.Ref 的工具 `ref` 和 `cref` 来包装这些参数，从而影响返回的 tuple 结果类型。(稍后我们将看到关于 `ref` 和 `cref` 的更多内容)

```
template<class T1,class T2,...,class TN> tuple<T1&,T2&,...,TN>
  tie(T1& t1,T2& t2,...,TN& tn);
```

函数模板 `tie` 类似于 `make_tuple` . 调用 `tie(t1,t2,...,tn)` 等同于调用 `make_tuple(ref(t1),ref(t2)... ref(tn))` ，即它创建一个由函数参数的引用组成的 `tuple` 。实际结果是把一个 `tuple` 赋值为由 `tie` 创建的对象，拷贝源 `tuple` 的元素到 `tie` 的参数。这样， `tie` 可以很容易地从一个由函数返回的 `tuple` 拷贝值到一个已有变量中。你也可以让 `tie` 从一个 `std::pair` 创建一个 2-tuple 。

```
template <int I,class T1,class T2,...,class TN>
  RI get(tuple<T1,T2,...,TN>& t);
```

这个函数 `get` 的重载版本用于取出 `tuple t` 的一个元素。索引 `I` 必须位于范围 [0..N), N 为 `tuple` 中的元素数量。如果 `TI` 是一个引用类型，则 `RI` 为 `TI`；否则，`RI` 为 `TI&`.

```
template <int I,class T1,class T2,...,class TN>
  RI get(const tuple<T1,T2,...,TN>& t);
```

这个函数 `get` 用于取出 `tuple t` 的一个元素。索引 `I` 必须位于范围 [0..N), N 为 `tuple` 中的元素数量。如果 `TI` 是一个引用类型，则 `RI` 为 `TI`；否则，`RI` 为 `const TI&`.

关系操作符

```
bool operator==(
  const tuple<T1,T2,...,TN>& lhs,
  const tuple<U1,U2,...,UN>& rhs);
```

如果对于所有位于范围[0..N)的 `i`，都有 `get&lt;i&gt;(lhs)==get&lt;i&gt;(rhs)`，N 为元素数量，则相等操作符返回 `true`。这两个 `tuple`s 必须具有相同数量的元素。对于 N=0 的空 `tuple`，总是返回 `true`。

```
bool operator!=(
  const tuple<T1,T2,...,TN>& lhs,
  const tuple<U1,U2...,...,>& rhs);
```

如果对于任意一个位于范围[0..N)的 `i`，有 `get&lt;i&gt;(lhs)!=get&lt;i&gt;(rhs)`，N 为元素数量，则不等操作符返回 `true`。这两个 `tuple`s 必须具有相同数量的元素。对于 N=0 的空 `tuple`，总是返回 `false`。

```
bool operator<(
  const tuple<T1,T2,...,TN>& lhs,
  const tuple<U1,U2,...,UN>& rhs);
```

如果对于任意一个位于范围[0..N)的 `i`，有 `get&lt;i&gt;(lhs)&lt;get&lt;i&gt;(rhs)`，N 为元素数量，则小于操作符返回 `true`；假如对每个比较都返回 `false`，则表达式 `!(get&lt;i&gt;(rhs)&lt;get&lt;i&gt;(lhs))` 为 `true`；否则表达式为 `false`。这两个 `tuple`s 必须具有相同数量的元素。对于 N=0 的空 `tuple`，总是返回 `true`。

值得注意的是，对于所有支持的关系操作符(operators `==`, `!=`, `&lt;`, `&gt;`, `&lt;=`, 和 `&gt;=`), 两个 `tuple`s 必须有相同的约束。首先，它们必须有相同的长度。其次，两个 `tuple` 间的每对元素(第一个对第一个，第二个对第二个，等等)必须支持同一个关系操作符。当这些约束被满足时，`tuple` 的操作符才可以实现，它按顺序比较每一对元素，即关系操作符是短路(short-circuited)的，一旦有了明确结果就马上返回。操作符 `&lt;`, `&gt;`, `&lt;=`, 和 `&gt;=` 执行字典序的比较，并要求元素对执行同样的操作。元素对的比较操作符产生的任何异常都会被传播，但 `tuple` 操作符本身不抛出异常。

用法

Tuples 位于名字空间 `tuples`，后者又位于名字空间 `boost`。使用这个库要包含头文件 `"boost/tuple/tuple.hpp"`。关系操作符的定义在头文件 `"boost/tuple/tuple_comparison.hpp"` 中。tuples 的输入输出定义在头文件 `"boost/tuple/tuple_io.hpp"` 中。`tuple` 的一些关键部件(`tie` 和 `make_tuple`)也可以直接在名字空间 `boost` 中使用。在本节中，我们将讨论如何在一些常见情形下使用 tuples，以及如何可以扩展这个库的功能以最好地符合我们的意图。我们将从构造 tuples 开始，并逐渐转移到其它主题，包括如何利用 tuples 的细节。

构造 Tuples

构造一个 `tuple` 包括声明各种类型，并可选地提供一组兼容类型的初始值。[1]

> [1] 在特化时 `tuple`，构造函数的参数不必与元素的类型精确相同，只要它们可以隐式地转换为元素的类型就可以了。

```
boost::tuple<int,double,std::string>
  triple(42,3.14,"My first tuple!");
```

类模板 `tuple` 模板参数指定了元素的类型。前面这个例子示范了一个带有三个类型的 `tuple` 的创建：一个 `int`，一个 `double`，和一个 `std::string`。并向构造函数提供了三个参数来初始化所有三个元素的值。也可以传递少于元素数量的参数，那样的话剩下的元素将被缺省初始化。

```
boost::tuple<short,int,long> another;
```

在这个例子中，`another` 有类型为 `short`，`int`，和 `long` 的元素，并且它们都被初始化为 0.[2] 不管你的 `tuple` 是什么类型，这就是它如何定义和构造的方式。所以，如果你的 `tuple` 有一个元素类型不能缺省构造，你就需要自己初始化它。与定义 `struct` 相比，`tuple` 更容易声明、定义和使用。还有一个便于使用的函数，`make_tuple`，它使得创建 `tuple`s 更加容易。它自动推断元素的类型，不用你来重复指定(这也会是出错的机会！)。

> [2] 在一个模板上下文中，`T()` 对于一个内建类型而言意味着初始化为零。

```
boost::tuples::tuple<int,double> get_values() {
  return boost::make_tuple(6,12.0);
}
```

函数 `make_tuple` 类似于 `std::make_pair`。缺省情况下，`make_tuple` 设置元素类型为非 `const`，非引用的，即是最简单的、根本的参数类型。例如，考虑以下变量：

```
int plain=42;
int& ref=plain;
const int& cref=ref;
```

这三个变量根据它们的cv限定符(常量性)以及是否引用来命名。通过调用以下 `make_tuple` 创建的 `tuple` 都带有一个 `int` 元素。

```
boost::make_tuple(plain);
boost::make_tuple(ref);
boost::make_tuple(cref);
```

这种行为不总是正确的，但通常是，这正是为什么它是缺省行为的原因。为了使一个 `tuple` 的元素设为引用类型，你要使用函数 `boost::ref`，它来自另一个名为 Boost.Ref 的 Boost 库。以下三行代码使用了我们前面定义的三个变量，但这次 `tuple` 带有一个 `int&` 元素，除了最后一个，它带的是一个 `const int&` 元素 (我们不能去掉 `cref` 的常量性)：

```
boost::make_tuple(boost::ref(plain));
boost::make_tuple(boost::ref(ref));
boost::make_tuple(boost::ref(cref));
```

如果元素需要是 `const` 引用的，就使用来自 Boost.Ref 的 `boost::cref`。下面三个 tuples 带有一个 `const int&` 元素：

```
boost::make_tuple(boost::cref(plain));
boost::make_tuple(boost::cref(ref));
boost::make_tuple(boost::cref(cref));
```

`ref` 和 `cref` 在其它地方也经常使用。事实上，它们原先是作为 Boost.Tuple 库的一部分而建立的，但后来因为它们的通用性而移出去成为一个独立的库。

访问 tuple 元素

一个 `tuple` 的元素可以通过 `tuple` 成员函数 `get` 或普通函数 `get` 来访问。它们都要求用一个常量整型表达式来指定要取出的元素的索引。

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

#include "boost/tuple/tuple.hpp"

int main() {
  boost::tuple<int,double,std::string>
    triple(42,3.14,"The amazing tuple!");

  int i=boost::tuples::get<0>(triple);
  double d=triple.get<1>();
  std::string s=boost::get<2>(triple);
}
```

这个例子中，一个三元素的 `tuple` 取名为 `triple` 。 `triple` 含有一个 `int` , 一个 `double` , 和一个 `string` , 它们可以用 `get` 函数取出。

```
int i=boost::tuples::get<0>(triple);
```

这里，你看到的是普通函数 `get` 。它把 `tuple` 作为一个参数。注意，给出一个无效的索引会导致一个编译期错误。这个函数的前提是索引值对于给定的 `tuple` 类型必须有效。

```
double d=triple.get<1>();
```

这段代码使用的是成员函数 `get` . 它也可以写成这样：

```
double& d=triple.get<1>();
```

这个绑定到一个引用的方式可以使用，因为 `get` 总是返回一个到元素的引用。如果 `tuple` , 或者其类型，是 `const` 的， 则返回一个 `const` 引用。这两个函数是等价的，但在某些编译器上，只有普通函数可以正确工作。普通函数有一个优点，它提供了与 `tuple` 之外的其它类型一致的提取元素的风格。通过索引来访问 `tuple` 的元素而不是通过名字来访问，这样做的一个优点是它可以支持泛型的解决方法，因为这样做不依赖于某个特定的名字，仅仅是一个索引值。稍后对此有更多介绍。

Tuple 赋值及复制构造

`tuple` s 可以被赋值和被复制构造，可以在两个 `tuple` 间进行，只要它们的元素类型可以相互转换。要赋值或复制 `tuple` s, 就是执行成员间的赋值或复制，因此这两个 `tuple` s 必须具有相同数量的元素。源 `tuple` 的元素必须可以转换为目标 `tuple` 的元素。以下例子示范了如何使用。

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

#include "boost/tuple/tuple.hpp"

class base {
public:
  virtual ~base() {};
  virtual void test() {
    std::cout << "base::test()\n";
  }
};

class derived : public base {
public:
  virtual void test() {
    std::cout << "derived::test()\n";
  }
};

int main() {
  boost::tuple<int,std::string,derived> tup1(-5,"Tuples");
  boost::tuple<unsigned int,std::string,base> tup2;
  tup2=tup1;

  tup2.get<2>().test();
  std::cout << "Interesting value: "
    << tup2.get<0>() << '\n';

  const boost::tuple<double,std::string,base> tup3(tup2);
  tup3.get<0>()=3.14;
}
```

这个例子开始时定义两个类，`base` 和 `derived`，它们被用作两个 `tuple` 类型的元素。第一个 `tuple` 有三个元素，类型为 `int`，`std::string`，和 `derived`。第二个 `tuple` 有三个兼容类型的元素，分别为 `int`，`std::string`，和 `base`。因此，这两个 `tuple`s 符合赋值的要求，这就是为什么 `tup2=tup1` 有效的原因。在这个赋值中，`tup1` 的第三个元素类型为 `derived`，被赋值给 `tup2` 的第三个元素，后者类型为 `base`。赋值可以成功，但 `derived` 对象被切割，因此这样会破坏多态性。

```
tup2.get<2>().test();
```

这一行取出一个 `base&`，但 `tup2` 中的对象类型为 `base`，因此它将调用 `base::test`。我们可以通过把 `tuple`s 修改为分别包含 `base` 和 `derived` 的引用或指针来获得多态行为。注意数值转换的危害(精度损失、正负溢出)也会出现在 `tuple`s 间的转换。这种危险的转换可以通过 Boost.Conversion 库的帮助来变得安全，请参见"Library 2: Conversion"。

下一行是复制构造一个新的 `tuple`，`tup3`，它的类型不同但还是兼容于 `tup2`。

```
const boost::tuple<double,std::string,base> tup3(tup2);
```

注意，`tup3` 被声明为 `const`。这意味着本例中有一个错误。看你能否找到它。我等一下你...，你看出来了吗？就是这里：

```
tup3.get<0>()=3.14;
```

因为 `tup3` 是 `const`，`get` 将返回一个 `const double&`. 这意味着这个赋值语句是非法的，这个例子不能编译。`tuple`s 间的赋值与复制构造是符合直觉的，因为它的语义与单个元素是相同的。通过下面这个例子，我们来看看如何在 `tuple`s 间的派生类至基类的赋值中获得多态行为。

```
derived d;
boost::tuple<int,std::string,derived*>
  tup4(-5,"Tuples",&d);
boost::tuple<unsigned int,std::string,base*> tup5;
tup5=tup4;

tup5.get<2>()->test();

boost::tuple<int,std::string,derived&>
  tup6(12,"Example",d);

boost::tuple<unsigned int,std::string,base&> tup7(tup6);

tup7.get<2>()->test();
```

在这两种情况下，都会调用 `derived::test`，这正是我们想要的。`tup6` 和 `tup7` 间不能赋值，因为你不能对一个引用进行赋值，这就是为什么 `tup7` 要从 `tup6` 复制构造，以及 `tup6` 要用 `d` 进行初始化的原因。因为 `tup4` 和 `tup5` 对它们的第三个元素使用的是指针，因此它们可以支持赋值。注意，通常在 tuple 中最好使用智能指针(与裸指针相比)，因为它们可以减轻对指针所指向的资源的生存期管理的压力。但是，正如 `tup4` 和 `tup5` 所示，在 `tuple`s 中，指针不总是指向需要进行内存管理的东西的。(参考 "Library 1: Smart_ptr 1"，可获得 Boost 强大的智能指针的更多信息)

比较 Tuples

要比较 `tuple`s, 你必须包含头文件 `"boost/tuple/tuple_comparison.hpp"`. `tuple` 的关系操作符有 `==`，`!=`，`<`，`>`，`<=` 和 `>=`，它们按顺序地对要比较的 `tuple`s 中的每一对元素调用相应的操作符。这些比较是短路的，即只比较到可以得到正确结果为止。只有具有相同数量元素的 `tuple`s 可以进行比较，并且显然两个 `tuple`s 的对应的元素必须是可比较的。如果两个 `tuple`s 的所有元素对都相等，则相等比较返回 `true`。如果任意一对元素的相等比较返回 `false`，`operator==` 也将返回 `false`。不等比较也是类似的，但返回的是相反的结果。其它关系操作符按字典序进行比较。

以下例程示范比较操作符的行为。

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

#include "boost/tuple/tuple.hpp"
#include "boost/tuple/tuple_comparison.hpp"

int main() {
  boost::tuple<int,std::string> tup1(11,"Match?");
  boost::tuple<short,std::string> tup2(12,"Match?");

  std::cout << std::boolalpha;

  std::cout << "Comparison: tup1 is less than tup2\n";

  std::cout << "tup1==tup2: " << (tup1==tup2) << '\n';
  std::cout << "tup1!=tup2: " << (tup1!=tup2) << '\n';
  std::cout << "tup1<tup2: " << (tup1<tup2) << '\n';
  std::cout << "tup1>tup2: " << (tup1>tup2) << '\n';
  std::cout << "tup1<=tup2: " << (tup1<=tup2) << '\n';
  std::cout << "tup1>=tup2: " << (tup1>=tup2) << '\n';

  tup2.get<0>()=boost::get<0>(tup1); //tup2=tup1 also works

  std::cout << "\nComparison: tup1 equals tup2\n";

  std::cout << "tup1==tup2: " << (tup1==tup2) << '\n';
  std::cout << "tup1!=tup2: " << (tup1!=tup2) << '\n';
  std::cout << "tup1<tup2: " << (tup1<tup2) << '\n';
  std::cout << "tup1>tup2: " << (tup1>tup2) << '\n';
  std::cout << "tup1<=tup2: " << (tup1<=tup2) << '\n';
  std::cout << "tup1>=tup2: " << (tup1>=tup2) << '\n';
}
```

如你所见，这两个 `tuple`s, `tup1` 和 `tup2`, 并不是严格相同的类型，但它们的类型是可比较的。在第一组比较中， `tuple`s 的第一个元素值不同，而在第二组中， `tuple`s 是相同的。以下是程序的运行输出。

```
Comparison: tup1 is less than tup2
tup1==tup2: false
tup1!=tup2: true
tup1<tup2: true
tup1>tup2: false
tup1<=tup2: true
tup1>=tup2: false

Comparison: tup1 equals tup2
tup1==tup2: true
tup1!=tup2: false
tup1<tup2: false
tup1>tup2: false
tup1<=tup2: true
tup1>=tup2: true
```

支持比较的一个重要方面是， `tuple`s 可以被排序，这意味着它们可以在关联容器中被排序。有些时候，我们需要按 `tuple` 中的某一个元素进行排序(建立一个弱序)，我们可以用一个简单的泛型方法来实现。

```
template <int Index> class element_less {
public:
  template <typename Tuple>
  bool operator()(const Tuple& lhs,const Tuple& rhs) const {
    return boost::get<Index>(lhs)<boost::get<Index>(rhs);
  }
};
```

这显示了使用索引而不是用名字来访问元素的优势；它可以很容易地创建泛型的构造来执行强大的操作。我们的 `element_less` 可以这样用：

```
#include <iostream>
#include <vector>
#include "boost/tuple/tuple.hpp"
#include "boost/tuple/tuple_comparison.hpp"

template <int Index> class element_less {
public:
  template <typename Tuple>
  bool operator()(const Tuple& lhs,const Tuple& rhs) const {
    return boost::get<Index>(lhs)<boost::get<Index>(rhs);
  }
};

int main() {
  typedef boost::tuple<short,int,long,float,double,long double>
    num_tuple;

  std::vector<num_tuple> vec;

  vec.push_back(num_tuple(6,2));
  vec.push_back(num_tuple(7,1));
  vec.push_back(num_tuple(5));

  std::sort(vec.begin(),vec.end(),element_less<1>());

  std::cout << "After sorting: " <<
    vec[0].get<0>() << '\n' <<
    vec[1].get<0>() << '\n' <<
    vec[2].get<0>() << '\n';
}
```

`vec` 由三个元素组成。使用从我们前面创建的模板所特化的 `element_less&lt;1&gt;` 函数对象，来执行基于 `tuple` s 的第二个元素的排序。这类函数对象还有更多的应用，如用于查找指定的 `tuple` 元素。

绑定 Tuple 元素到变量

Boost.Tuple 库的一个方便的特性是"绑定" `tuple` s 到变量。绑定者就是用重载函数模板 `boost::tie` 所创建的 `tuple` s，它的所有元素都是非 `const` 引用类型。因此， `tie` s 必须使用左值进行初始化，从而 `tie` 的参数也必须是非 `const` 引用类型。由于结果 `tuple` s 具有非 `const` 引用类型，对这样一个 `tuple` 的元素进行赋值，就会通过非 `const` 引用赋值给调用 `tie` 时的左值。这样就绑定了一个已有变量给 `tuple` , tie 的名字由此而来！

以下例子首先示范了一个通过返回 `tuple` 获得值的明显的方法。然后，它示范了通过一个 `tie`d `tuple` 直接赋值给变量以完成相同操作的方法。为了让这个例子更为有趣，我们开始时定义了一个返回两个数值的最大公约数和最小公倍数的函数。当然，这两个结果值被组合成一个 `tuple` 返回类型。你将发现计算最大公约数和最小公倍数的函数来自于另一个 Boost 库——Boost.Math.

```
#include <iostream>
#include "boost/tuple/tuple.hpp"
#include "boost/math/common_factor.hpp"

boost::tuple<int,int> gcd_lcm(int val1,int val2) {
  return boost::make_tuple(
    boost::math::gcd(val1,val2),
    boost::math::lcm(val1,val2));
}

int main() {
  //"老"方法
  boost::tuple<int,int> tup;
  tup=gcd_lcm(12,18);
  int gcd=tup.get<0>();  // 译注：原文为 int gcd=tup.get<0>(); 明显有误
  int lcm=tup.get<1>();  // 译注：原文为 int gcd=tup.get<1>(); 明显有误

  std::cout << "Greatest common divisor: " << gcd << '\n';
  std::cout << "Least common multiple: " << lcm << '\n';

  //"新"方法
  boost::tie(gcd,lcm)=gcd_lcm(15,20);

  std::cout << "Greatest common divisor: " << gcd << '\n';
  std::cout << "Least common multiple: " << lcm << '\n';
}
```

有时我们并不是对返回的 tuple 中所有的元素感兴趣，`tie` 也可以支持这种情况。有一个特殊的对象，`boost:: tuples::ignore`，它忽略一个 `tuple` 元素的值。如果前例中我们只对最大公约数感兴趣，我们可以这样写：

```
boost::tie(gcd,boost::tuples::ignore)=gcd_lcm(15,20);
```

另一种方法是创建一个变量，传递给 `tie`，然后在后面的处理中忽略它。这样做会令维护人员弄不清楚这个变量为什么存在。使用 `ignore` 可以清楚地表明代码将不使用 `tuple` 的那个值。

注意，`tie` 也支持 `std::pair`. 用法与从 `boost::tuple`s 绑定值一样。

```
std::pair<short,double> p(3,0.141592);
short s;
double d;

boost::tie(s,d)=p;
```

绑定 `tuple`s 不仅仅是方便使用；它有助于使代码更为清晰。

Tuples的流操作

在本章的每一个例子中，取出 `tuple`s 的元素都只是为了能够把它们输出到 `std::cout`. 可以象前面那样做，但还有更容易的方法。 `tuple` 库支持输入和输出流操作； `tuple` 重载了 `operator>>` 和 `operator<<` 。还有一些操纵器用于改变输入输出的缺省分隔符。对输入操作改变分隔符改变 `operator>>` 查找元素值的结果。我们用一个简单的读写 `tuple`s 的程序来测试一下这些情况。注意，要使用 `tuple` 的流操作，你必须包含头文件 `"boost/tuple/tuple_io.hpp"` .

```
#include <iostream>
#include "boost/tuple/tuple.hpp"
#include "boost/tuple/tuple_io.hpp"

int main() {
  boost::tuple<int,double> tup1;
  boost::tuple<long,long,long> tup2;

  std::cout << "Enter an int and a double as (1 2.3):\n";
  std::cin >> tup1;

  std::cout << "Enter three ints as |1.2.3|:\n";
  std::cin >> boost::tuples::set_open('|') >>
  boost::tuples::set_close('|') >>
  boost::tuples::set_delimiter('.') >> tup2;

  std::cout << "Here they are:\n"
    << tup1 << '\n'
    << boost::tuples::set_open('\"') <<
  boost::tuples::set_close('\"') <<
  boost::tuples::set_delimiter('-');

  std::cout << tup2 << '\n';
}
```

上面这个例子示范了如何对 `tuple`s 使用流操作符。 `tuple`s 的缺省分隔符是： `(` (左括号) 作为开始分隔符， `)` (右括号) 作为结束分隔符，空格用于分隔各个 `tuple` 元素值。这意味着我们的程序要正确运行的话，我们需要象这样输入： `(12 54.1)` 和 `|4.5.3|` . 以下是运行的例子。

```
Enter an int and a double as (1 2.3):
(12 54.1)
Enter three ints as |1.2.3|:
|4.5.3|
Here they are:
(12 54.1)
"4-5-3"
```

对流操作的支持是很方便的，通过对分隔符操纵器的支持，可以很容易让使用 `tuple` 的代码的流操作兼容于已有代码。

关于 Tuples 的更多

还有很多我们没有看到的工具可用于 `tuple`s。这些更为先进的特性对于创建使用 `tuple` 的泛型结构至为重要。例如，你可以获得一个 `tuple` 的长度(即元素的数量)，取出某个元素的类型，使用 `null_type tuple` 哨兵来终结递归模板实例化。

不可能用一个 `for` 循环来迭代一个 `tuple` 里的元素，因为 `get` 要求提供一个常量整型表达式。但是，使用模板元编程，我们可以打印一个 tuple 的所有元素。

```
#include <iostream>
#include <string>
#include "boost/tuple/tuple.hpp"

template <typename Tuple,int Index> struct print_helper {
  static void print(const Tuple& t) {
    std::cout << boost::tuples::get<Index>(t) << '\n';
    print_helper<Tuple,Index-1>::print(t);
  }
};

template<typename Tuple> struct print_helper<Tuple,0> {
  static void print(const Tuple& t) {
    std::cout << boost::tuples::get<0>(t) << '\n';
  }
};

template <typename Tuple> void print_all(const Tuple& t) {
  print_helper<
    Tuple,boost::tuples::length<Tuple>::value-1>::print(t);
}

int main() {
  boost::tuple<int,std::string,double>
    tup(42,"A four and a two",42.424242);

  print_all(tup);
}
```

在这个例子中有一个辅助类模板，`print_helper`，它是一个元程序，访问 `tuple` 的所有索引，并对每个索引打印出相应元素。偏特化版本用于结束模板递归。函数 `print_all` 用它的 `tuple` 参数的长度以及这个 `tuple` 来调用 `print_helper` 构造函数。tuple 的长度可以这样来取得：

```
boost::tuples::length<Tuple>::value
```

这是一个常量整型表达式，这意味着它可以作为第二个模板参数传递给 `print_helper`. 但是，我们的解决方案中有一个警告，我们看看这个程序的输出结果就清楚了。

```
42.4242
A four and a two
42
```

我们按反序来打印元素了！虽然有些情况下可能会需要这种用法(他狡猾地辩解说)，但在这里不是。问题在于 `print_helper` 先打印 `boost::tuples::length&lt;Tuple&gt;::value-1` 元素的值，然后才到前一个元素，直到偏特化版本打印第一个元素值为止。我们不应该使用第一个元素作为特化条件以及从最后一个元素开始，而是需要从第一个元素开始以及使用最后一个

元素作为特化条件。这怎么可能？在你明白到 `tuple` 是以一个特殊的类型 `boost::tuples:: null_type` 作为结束以后，解决的方法就很明显了。我们可以确保一个 `tuple` 中的最后一个类型是 `null_type`，这也意味着我们的解决方法应该是对 `null_type` 进行特化或函数重载。

剩下的问题就是取出第一个元素的值，然后继续，并在列表尾部结束。`tuple`s 提供了成员函数 `get_head` 和 `get_tail` 来访问其中的元素。顾名思义，`get_head` 返回数值序列的头，即第一个元素的值。`get_tail` 返回一个由该 `tuple` 中除了第一个值以外的其它值组成的 `tuple`. 这就引出了 如下 `print_all` 的解决方案。

```
void print_all(const boost::tuples::null_type&) {}

template <typename Tuple> void print_all(const Tuple& t) {
  std::cout << t.get_head() << '\n';
  print_all(t.get_tail());
}
```

这个解决方案比原先的更短，并且按正确的顺序打印元素的数值。每一次函数模板 `print_all` 执行时，它打印 `tuple` 的第一个元素，然后用一个由 `t` 中除第一个值以外的其它值所组成的 `tuple` 递归调用它自己。当 `tuple` 没有数值时，结尾是一个 `null_type`，将调用重载函数 `print_all` ，递归结束。

可以知道某个元素的类型有时是有用的，例如当你要在泛型代码中声明一个由 `tuple` 的元素初始化的变量时。考虑一个返回 `tuple` 中前两个元素的和的函数，它有一个额外的要求，即返回值的类型必须是两个中较大的那个类型(例如，考虑整数类型)。如果不清楚元素的类型，就不可能创建一个通用的解决方案。这正是辅助模板 `element&lt;N,Tuple&gt;::type` 要做的，如下例所示。我们所面对的问题不仅仅是计算哪个元素具有较大的类型，还有如何声明这个函数的返回值类型。这有点复杂，但我们可以通过增加一个间接层来解决。这个间接层以一个额外的辅助模板形式出现，它有一个责任：提供一个 `typedef` 以定义两种类型中的较大者。代码可能看起来有点多，但它的确可以完成任务。

```
#include <iostream>
#include "boost/tuple/tuple.hpp"
#include <cassert>
#include <typeinfo>  //译注：原文没有这行，不能通过编译

template <bool B,typename Tuple> struct largest_type_helper {
  typedef typename boost::tuples::element<1,Tuple>::type type;
};

template<typename Tuple> struct largest_type_helper<true,Tuple> {
  typedef typename boost::tuples::element<0,Tuple>::type type;
};

template<typename Tuple> struct largest_type {
  typedef typename largest_type_helper<
    (sizeof(boost::tuples::element<0,Tuple>)>
    sizeof(boost::tuples::element<1,Tuple>)),Tuple>::type type;
};

template <typename Tuple>
typename largest_type<Tuple>::type sum(const Tuple& t) {
  typename largest_type<Tuple>::type
    result=boost::tuples::get<0>(t)+
      boost::tuples::get<1>(t);

  return result;
}

int main() {
  typedef boost::tuple<short,int,long> my_tuple;

  boost::tuples::element<0,my_tuple>::type first=14;
  assert(typeid(first) == typeid(short));
  //译注：原文为assert(type_id(first) == typeid(short)); 明显有误
  boost::tuples::element<1,my_tuple>::type second=27;
  assert(typeid(second) == typeid(int));
  //译注：原文为assert(type_id(second) == typeid(int)); 明显有误
  boost::tuples::element<
    boost::tuples::length<my_tuple>::value-1,my_tuple>::type
    last;

  my_tuple t(first,second,last);

  std::cout << "Type is int? " <<
    (typeid(int)==typeid(largest_type<my_tuple>::type)) << '\n';

  int s=sum(t);
}
```

如果你不太清楚模板元编程的运用，不用担心，对于使用 Tuple 库这不是必需的。虽然这类代码有时会用到，它的思想其实也很简单。`largest_type` 从两个辅助类模板 `largest_type_helper` 中的一个获得 `typedef`，具体使用哪一个就要靠那个布尔参数来特化了。这个参数通过比较 `tuple` (第二个模板参数) 的头两个元素的大小来决定。这样的结果是，`typedef` 会表现为两个类型中较大的那一个。我们的函数 `sum` 使用这个类型来作为返回值的类型，剩下的部分就是简单地对两个元素进行相加了。

这个例子的剩余部分示范了如何使用函数 `sum`，还有如何从某个 `tuple` 元素的类型来声明变量。头两个对 `tuple` 中的索引使用了硬编码。

```
boost::tuples::element<0,my_tuple>::type first=14;
boost::tuples::element<1,my_tuple>::type second=27;
```

最后一个声明取出 `tuple` 的最后一个元素的索引，并用它作为辅助模板的输入来(通用地)声明这个类型。

```
boost::tuples::element<
  boost::tuples::length<my_tuple>::value-1,my_tuple>::type last;
```

Tuples 与 for_each

我们前面用于创建 `print_all` 函数的方法可以被推广至创建象 `std::for_each` 那样的更通用的机制。例如，如果我们不想打印元素，而是想对它们取和或是复制它们，又或者我们只想打印它们中的一部分，要怎么做呢？对 tuple 的元素进行顺序访问并不简单，正如我们在前面的例子所见的一样。创建一个通用性的解决方案来接受一个函数或函数对象参数来调用 tuple 的元素是很有意义的。这样就不仅可以实现(有点限制的) `print_all` 函数的功能，还可以执行任何函数，只要这个函数可以接受 `tuple` 中的元素的类型。下面的例子创建了一个名为 `for_each_element` 的函数模板来实现这个功能。这个例子用两个函数对象作为参数来示范如何使用 `for_each_element` 。

```
#include <iostream>
#include <string>
#include <functional>
#include "boost/tuple/tuple.hpp"

template <typename Function> void for_each_element(
  const boost::tuples::null_type&, Function) {}

template <typename Tuple, typename Function> void
for_each_element(Tuple& t, Function func) {
  func(t.get_head());
  for_each_element(t.get_tail(),func);
}

struct print {
  template <typename T> void operator()(const T& t) {
    std::cout << t << '\n';
  }
};

template <typename T> struct print_type {
  void operator()(const T& t) {
    std::cout << t << '\n';
  }

  template <typename U> void operator()(const U& u) {}
};

int main() {
  typedef boost::tuple<short,int,long> my_tuple;

  boost::tuple<int,short,double> nums(1,2,3.01);

  for_each_element(nums, print());
  for_each_element(nums, print_type<double>());
}
```

函数 `for_each_element` 重用了前面例子中的策略，通过重载函数的一个版本来接受类型为 `null_type` 的参数，表示已来到 `tuple` 元素的结尾，从而不做任何事。让我们来看看完成实际工作的那个函数。

```
template <typename Tuple, typename Function> void
for_each_element(Tuple& t, Function func) {
  func(t.get_head());
  for_each_element(t.get_tail(),func);
}
```

第二个模板的函数参数指定了以 `tuple` 元素为参数进行调用的函数(或函数对象)。`for_each_element` 首先用 `get_head` 返回的元素来调用这个函数(对象)。要注意的是，`get_head` 返回的是 `tuple` 的当前元素。然后，它递归地用 `tuple` 的剩余元素来调用自已本身。第二次调用同样取出头一个元素并用它调用给定的函数(对象)，然后再次递归，一直下去。最后，`get_tail` 发现没有元素了，就返回一个 `null_type` 实例，它匹配了那个非递归的 `for_each_element` 重载版本，从而结束了递归。这就是 `for_each_element` 的全部！

接下来，这个例子给出了两个示例函数对象，它们所用的技术可以在其它情况下重用。一个是 `print` 函数对象。

```
struct print {
  template <typename T> void operator()(const T& t) {
    std::cout << t << '\n';
  }
};
```

这个 `print` 函数对象没什么奇怪的地方，但正如它所做的那样，许多程序员不知道调用操作符可以是模板的！通常，函数对象可以对一个或多个类型进行特化，但对于 `tuples` 这样不行，因为它的元素可以是完全不同的类型。因此，不是对于函数对象本身进行特化，而是对调用操作符进行特化，这样做的另一个好处是它更简单，如下。

```
for_each_element(nums, print());
```

不需要给出类型，而如果使用特化函数对象则需要。把模板参数推给成员函数有些时候是很有用的，而且通常用户也更容易使用。

第二个函数对象打印指定类型的所有元素。这种过滤方法也可以用于取出相容类型的元素。

```
template <typename T> struct print_type {
  void operator()(const T& t) {
    std::cout << t << '\n';
  }

  template <typename U> void operator()(const U& u) {}
};
```

这个函数对象显示了另一个有用的技术，我称之为忽略重载(discarding overload)。它用于忽略传给它的除了 `T` 类型以外的所有元素。窍门就是除了指定类型外的其它类型的重载匹配。这可能会让你想到这种技术与另一个有密切的联系，即 `sizeof` 窍门和省略号(`...`)结构，后者被也用于在编译期进行决议，但它在这里不能使用，在这里，函数确实被调用了，只是没有做任何事情而已。这个函数对象可以这样用：

```
for_each_element(print_type<double>(),nums);
```

很容易用，也容易写，并且它有更多的价值。实际上 Tuple 库使用的函数对象可能没有这么多特性，它们也可以用于这种技术或其它的用法。

Tuple 总结

Tuple 库为C++带来了 tuples 的概念。它是符合直觉和简单明了的，虽然它的主要用途看起来就是用于从函数返回多个返回值，但它也可以用于创建各种逻辑组合，就象在标准库容器中保存一组元素一样。这种方法和为各个不同的返回类型创建一个 `struct` 是相同的，但后者不仅沉闷，而且不可能作出递归的泛型解决方案。使用 Boost.Tuple 就可以解决这些问题。

在本章中，我们看到了如何使用 Tuple 库，以及如何用函数对象扩充它，还有如何对 tuple 使用算法。通过索引来访问元素，以及 `get_head` / `get_tail` 成员函数，提供了使用 `tuple`s 的一致性，使得许多解决方案可以实现，而使用用户定义类型(UDTs)时是不可能的。

Boost.Tuple 的创建者，Jaakko Järvi，应该为这个杰出的库获得荣誉。他的努力有力地证明了，几乎所有C++中缺少的东西都可以由天才的设计者通过库增加进来。

Part III: 函数对象与高级编程

> 以下四个库可能会永远改变你对C++编程的看法。虽然函数对象并不是什么新概念，特别是对于曾长期使用和定制标准库中的算法的人来说，但本书这部分中的几个库的讨论将带给函数对象全新级别的抽象。有一些领域曾经被认为是C++不适用的，在从事某些特定设计时，如表面上看，在使用标准库的算法时，不可避免地会产生很多小的函数对象。但千万不要忘记，在C++中，最好不要只从语言本身来判定，它被设计为可以通过库来弥补本身的缺点；确实，库 Boost.Bind 和 Boost.Lambda 正试图解决前述问题。回调函数是另一个有问题的领域；问题的根本在把库用于更高级别的编程时更为突出，因为存储和延时调用类似于函数的对象成为了一个重要的特性。这正是 Boost.Function 要做的，当然，它与这里提到的两个库(还有其它库)都可以很好地配合。最后一章讨论 Boost.Signals, 这是一个具体化 Observer 模式的库。这些库具有特别的力量，它们可以使程序员写更少的代码、更有表现力的语句，并确实缩短了表达式，使得代码更易读且更易于维护。这些能力同时也带来了负担，因为它也很可能写出不能分析的表达式。对于多数程序员，熟悉这些库将非常有用，我希望对你来说也是这样。

Library 9. Bind

- Bind 库如何改进你的程序？
- Bind 如何适用于标准库？
- Bind
- 用法
- Bind 总结

Bind 库如何改进你的程序？

- 使函数和函数对象适用于标准库算法
- 使用一致语法创建绑定器
- 强大的函数组合

在使用来自于标准库的算法时，你常常需要提供给它们一个函数或一个函数对象。这是对算法的行为进行定制的一个好方法，但你通常需要写一个新的函数对象，因为你没有组合函数或改变参数的顺序等所需的工具。虽然标准库已经提供了一些可用的工具，如 `bind1st` 和 `bind2nd`，但是这不够用。即使功能上够用了，但这通常意味着要忍受笨拙的语法，这些语法通常会让不熟悉这些工具的程序员产生混乱。你需要的是一个解决方案，既具备所需功能，又可以使用普通的语法就地创建函数对象，这正是 Boost.Bind 所要做的。

事实上，泛型绑定器是一种 lambda 表达式，因为通过函数组合，我们可以或多或少在调用点构造一个局部的、无名的函数。在许多情形下这都是需要的，因为它达到了三个目的：减少了代码的数量， 使代码更易懂，还有行为的局部化，这意味着更有效的维护。注意，还有另一个 Boost 库，Boost.Lambda, 它具有更多的特性。Boost.Lambda 将在下一章中讨论。为什么你不直接跳到下一个库？因为多数情况下，Boost.Bind 可以完成你要绑定的所有东西，并且学习曲线没那么陡。

Bind 成功的一个关键是采用统一的语法来创建函数对象，以及对于使用该库的类型只有很少的要求。这种设计使得无需关注如何去写与你的类型一起工作的代码，而只需关注我们最关心的一点，代码如何工作以及它实际上做了什么。使用来自标准库的适配器时，如 `ptr_fun` 和 `mem_fun_ref`，代码很容易变得过分冗长，因为我们必须提供这些适配器以便参数可以符合算法的要求。在 Boost.Bind 里不是这样的，它使用了更为精妙的推断系统，并且在自动推断不能适用时提供了一个简单的语法。使用 Bind 的结果就是，你可以写更少的代码，而且代码更易懂。

Bind 如何适用于标准库？

概念上，Bind 是已有的标准库函数 `bind1st` 和 `bind2nd` 的泛化，其额外的功能就是允许更为精妙的函数组合。它还减少了对函数指针和类成员指针使用适配器的需要，从而缩短了代码，也减少了出错的机会。Boost.Bind 还包含了对C++标准库的一些常用的扩充，如SGI扩充的 `compose1` 和 `compose2`，还有 `select1st` 和 `select2nd` 函数。因此，Bind 非常适用于标准库，而且它也真的非常好用。这些功能被公认为是需要的，最终将被引入到标准库中，也是对STL的扩展。Boost.Bind 已经被即将发布的 Library Technical Report 所接纳。

Bind

头文件: `"boost/bind.hpp"`

Bind 库创建函数对象来绑定到一个函数(普通函数或成员函数)。不需要直接给出函数的所有参数，参数可以稍后给，这意味着绑定器可以用于创建一个改变了它所绑定到的函数的 arity (参数数量) 的函数对象，或者按照你喜欢的顺序重排参数。

函数 `bind` 的重载版本的返回类型是未指定的，即不能保证返回的函数对象的特征是怎样的。有时，你需要将对象存于某处，而不是直接把它传送给另一个函数，这时，你要使用 Boost.Function, 它在 "Library 11: Function 11." 中讨论。弄明白 `bind` 函数返回的是什么的关键在于，理解它发生了什么转换。用 `bind` 函数的一个重载， `template<class R, class F> unspecified-1 bind(F f)` 来作为例子，返回类型就是(引用自在线文档)，"一个函数对象 l，表达式 l(v1, v2, ..., vm) 等同于 f()，隐式转换为 R"。这样，这个被绑定的函数就被保存在绑定器里面，以后对这个函数对象的调用就会得到被绑定的函数的返回值(如果有)，即模板参数 `R`. 我们在这讨论的实现支持最多九个函数参数。

Bind 的实现包括许多函数和类，但作为用户来说，我们不直接使用除了重载函数 `bind` 以外的任何东西。所有绑定通过 `bind` 函数发生，我们可以无须依赖于返回值的类型。使用 `bind` 时，参数占位符(命名为 `_1`, `_2`, 等等)不需要用一个using声明或using指示来引入，因为它们位于匿名名字空间。这样，在使用 Boost.Bind时，没有理由写出以下的代码。

```
using boost::bind;
using namespace boost;
```

前面曾经提到过，当前的 Boost.Bind 实现支持九个占位符(`_1`, `_2`, `_3`, 等等)，也就是说最多九个参数。粗略地过一下大纲对于深入理解如何进行类型推断是有好处的，还可以知道何时/为何它不总是可以工作的。花点时间分析一下成员函数指针与普通函数的署名特征也是很有用的。你将会看到对于普通函数和类成员函数，各有各的重载版本。还有，对于每一个数量的参数，也都有不同的重载。 我不在这里列出所有大纲了，建议你到www.boost.org参考一下 Boost.Bind 的文档。

用法

Boost.Bind 为函数和函数对象提供了一致的语法，对于值语义和指针语义也一样。我们将从一些简单的例子开始，处理一些简单绑定的用法，然后再转移到通过嵌套绑定进行函数组合。弄明白如何使用 `bind` 的关键是，占位符的概念。占位符用于表示提供给结果函数对象的参数，Boost.Bind 支持最多九个参数。占位符被命名为 `_1`，`_2`，`_3`，`_4`，直至 `_9`，你要把它们放在你原先放参数的地方。作为第一个例子，我们定义一个函数，`nine_arguments`，它将被一个 `bind` 表达式调用。

```
#include <iostream>
#include "boost/bind.hpp"

void nine_arguments(
  int i1,int i2,int i3,int i4,
  int i5,int i6,int i7,int i8, int i9) {
  std::cout << i1 << i2 << i3 << i4 << i5
    << i6 << i7 << i8 << i9 << '\n';
}

int main() {
  int i1=1,i2=2,i3=3,i4=4,i5=5,i6=6,i7=7,i8=8,i9=9;
  (boost::bind(&nine_arguments,_9,_2,_1,_6,_3,_8,_4,_5,_7))
    (i1,i2,i3,i4,i5,i6,i7,i8,i9);
}
```

在这个例子中，你创建了一个匿名临时绑定器，并立即把参数传递给它的调用操作符来调用它。如你所见，占位符的顺序是被搅乱的，这说明参数的顺序被重新安排了。注意，占位符可以在一个表达式中被多次使用。这个程序的输出如下。

```
921638457
```

这表示了占位符对应于它的数字所示位置的参数，即 `_1` 被第一个参数替换，`_2` 被第二个参数替换，等等。接下来，你将看到如何调用一个类的成员函数。

调用成员函数

我们来看一下如何用 `bind` 调用成员函数。我们先来做一些可以用标准库来做的事情，这样可以对比一下用 Boost.Bind 的方法。保存某种类型的元素在一个标准库容器中，一个常见的需要是对某些或全部元素调用一个成员函数。这可以用一个循环来完成，通常也正是这样做的，但还有更好的方法。考虑下面这个简单的类，`status`，我们将用它来示范 Boost.Bind 的易用性和强大的功能。

```
class status {
  std::string name_;
  bool ok_;
public:
  status(const std::string& name):name_(name),ok_(true) {}

  void break_it() {
    ok_=false;
  }

  bool is_broken() const {
    return ok_;
  }

  void report() const {
    std::cout << name_ << " is " <<
      (ok_ ? "working nominally":"terribly broken") << '\n';
  }
};
```

如果我们把这个类的实例保存在一个 `vector`，并且我们需要调用成员函数 `report`，我们可能会象下面这样做。

```
std::vector<status> statuses;
statuses.push_back(status("status 1"));
statuses.push_back(status("status 2"));
statuses.push_back(status("status 3"));
statuses.push_back(status("status 4"));

statuses[1].break_it();
statuses[2].break_it();

for (std::vector<status>::iterator it=statuses.begin();
  it!=statuses.end();++it) {
  it->report();
}
```

这个循环正确地完成了任务，但它是冗长、低效的(由于要多次调用 `statuses.end()`)，并且不象使用标准库算法 `for_each` 那样清楚地表明意图。为了用 `for_each` 来替换这个循环，我们需要用一个适配器来对 `vector` 元素调用成员函数 `report`。这时，由于元素是以值的方式保存的，我们需要的是适配器 `mem_fun_ref`.

```
std::for_each(
  statuses.begin(),
  statuses.end(),
  std::mem_fun_ref(&status::report));
```

这是一个正确、合理的方法，它非常简洁，非常清楚这段代码是干什么的。以下是使用 `Boost.Bind` 完成相同任务的代码。[1]

> [1] 要注意的是 `boost::mem_fn`，它也被接纳进入Library Technical Report, 它也可以在这种没有参数的情况下使用。 `mem_fn` 取代了 `std::mem_fun 和 std::mem_fun_ref`.

```
std::for_each(
  statuses.begin(),
  statuses.end(),
  boost::bind(&status::report,_1));
```

这个版本同样的清楚、明白。这是前面所说的占位符的第一个真正的使用，我们同时告诉编译器和代码的读者，`_1` 用于替换这个函数所调用的绑定器的第一个实际参数。虽然这段代码节省了几个字符，但在这种情况下标准库的 `mem_fun_ref` 和 `bind` 之间并没有太大的不同，但是让我们来重用这个例子并把容器改为存储指针。

```
std::vector<status*> p_statuses;
p_statuses.push_back(new status("status 1"));
p_statuses.push_back(new status("status 2"));
p_statuses.push_back(new status("status 3"));
p_statuses.push_back(new status("status 4"));

p_statuses[1]->break_it();
p_statuses[2]->break_it();
```

我们还可以使用标准库，但不能再用 `mem_fun_ref`. 我们需要的是适配器 `mem_fun`, 它被认为有点用词不当，但它的确正确完成了需要做的工作。

```
std::for_each(
  p_statuses.begin(),
  p_statuses.end(),
  std::mem_fun(&status::report));
```

虽然这也可以工作，但语法变了，即使我们想做的事情非常相似。如果语法可以与第一个例子相同，那就更好了，所以我们所关心的是代码要做什么，而不是如何去做。使用 `bind`, 我们就无须关心我们处理的元素是指针了(这一点已经在容器类型的声明中表明了，对于现代的库来说，这样的冗余信息是不需要的)。

```
std::for_each(
  p_statuses.begin(),
  p_statuses.end(),
  boost::bind(&status::report,_1));
```

如你所见，这与我们前一个例子完全一样，这意味着如果我们之前已经明白了 `bind`，那么我们现在也清楚它。现在，我们已决定换用指针了，我们要面对另一个问题，即生存期控制。我们必须手工释放 `p_statuses` 中的元素，这很容易出错，也无须如此。所以，我们可能决定开始使用智能指针，并(再次)修改我们的代码。

```
std::vector<boost::shared_ptr<status> > s_statuses;
s_statuses.push_back(
  boost::shared_ptr<status>(new status("status 1")));
s_statuses.push_back(
  boost::shared_ptr<status>(new status("status 2")));
s_statuses.push_back(
  boost::shared_ptr<status>(new status("status 3")));
s_statuses.push_back(
  boost::shared_ptr<status>(new status("status 4")));
s_statuses[1]->break_it();
s_statuses[2]->break_it();
```

现在，我们要用标准库中的哪个适配器呢？`mem_fun` 和 `mem_fun_ref` 都不适用，因为智能指针没有一个名为 `report` 的成员函数，所以以下代码编译失败。

```
std::for_each(
  s_statuses.begin(),
  s_statuses.end(),
  std::mem_fun(&status::report));
```

不巧，标准库不能帮我们完成这个任务[2]。因此，我们不得不采用我们正想要摆脱的循环，或者使用 Boost.Bind, 它不会抱怨任何事情，而且正确地完成我们想要的。

> [2] 以后将可以这样做，因为 `mem_fn` 和 `bind` 都将成为未来的标准库的一部分。

```
std::for_each(
  s_statuses.begin(),
  s_statuses.end(),
  boost::bind(&status::report,_1));
```

再一次，这段代码与前面的例子完全一样(除了容器的名字不同)。使用绑定的语法是一致的，不论是用于 值语义或是指针语义，甚至是用于智能指针。有时，使用不同的语法有助于理解代码，但在这里，不是这样的，我们的任务是对容器中的元素调用成员函数，没有更 多的也没有更少的事情。语法一致的价值不应被低估，因为它对于编写代码的人，以及对于日后需要维护代码的人都是有帮助的(当然，我们并不真的是在写需要维 护的代码，但为了这个主题，让我们假装是在写)。

这些例子示范了一个非常基本和常见的情形，在这种情形下 Boost.Bind 尤为出色。即使标准库也提供了完成相同工作的一些基本工具，但我们还是看到 Bind 既提供了一致的语法，也增加了标准库目前缺少的功能。

看一下门帘的后面

在你开始使用 Boost.Bind 后，这是无可避免的；你将开始惊讶它到底是如何工作的。这看起来就象是魔术，`bind` 可以推断出参数的类型和返回类型，它又是如何处理占位符的呢？我们将快速地看一下驱动这个东西的机制。它有助于知道一点 `bind` 的 工作原理，特别是在试图解释这惊人的简洁性以及编译器对最轻微的错误给出的直接的错误信息。我们将创建一个

非常简单的绑定器，至少是部分地模仿 Boost.Bind 的语法。为了避免把这个离题的讨论搞成几页那么长，我们只支持一类绑定，即接受单个参数的成员函数。此外，我们不会对cv限定符进行处理；我们只处理最简单的情况。

首先，我们需要能够推断出我们要绑定的函数的返回类型、类的类型、和参数类型。我们用一个函数模板来做到这一点。

```
template <typename R, typename T, typename Arg>
simple_bind_t<R,T,Arg> simple_bind(
  R (T::*fn)(Arg),
  const T& t,
  const placeholder&) {
  return simple_bind_t<R,T,Arg>(fn,t);
}
```

这看起来有点可怕，毕竟这只是在定义整个机器的一部分。但是，这一部分的焦点在于类型推断在哪发生。你会注意到这个函数有三个模板参数，`R`，`T`，和`Arg`. `R`是返回的类型，`T`是类的类型，而`Arg`是(单个)参数的类型。这些模板参数组成了我们的函数的第一个参数，即`R (T::*f)(Arg)`. 这样，传递一个带单个参数的成员函数给`simple_bind`将允许编译器推断出`R`为成员函数的返回类型，`T`为成员函数的类，`Arg`为成员函数的参数类型。`simple_bind`的返回类型是一个函数对象，它使用与`simple_bind`相同的三个类型进行特化，其构造函数接受一个成员函数指针和一个对应类(`T`)的实例。`simple_bind`简单地忽略占位符(即函数的最后一个参数)，我保留这个参数的原因是为了模仿 Boost.Bind 的语法。在一个更好的实现中，我们显然应该使用这个参数，但是现在让我们先不要管它。这个函数对象的实现相当简单。

```
template <typename R,typename T, typename Arg>
class simple_bind_t {
  typedef R (T::*fn)(Arg);
  fn fn_;
  T t_;
public:
  simple_bind_t(fn f,const T& t):fn_(f),t_(t) {}

  R operator()(Arg& a) {
    return (t_.*fn_)(a);
  }
};
```

从`simple_bind`的实现中我们可以看到，构造函数接受两个参数：第一个是指向成员函数的指针，第二个是一个`const T`引用，它会被复制并稍后用于给定一个用户提供的参数来调用其成员函数。最后，调用操作符返回`R`，即成员函数的返回类型，并接受一个`Arg`参数，即传给成员函数的那个参数的类型。调用成员函数的语法稍稍有点晦涩：

```
(t_.*fn_)(a);
```

`.*` 是成员指针操作符，它的第一个操作数是 `class T`；另外还有一个成员指针操作符，`->*`，它的第一个操作数是是一个 `T` 指针。剩下就是创建一个占位符，即用于替换实际参数的变量。我们可以通过在匿名名字空间中包含某种类型的变量来创建一个占位符；我们把它称为 `placeholder`：

```
namespace {
  class placeholder {};
  placeholder _1;
}
```

我们创建一个简单的类和一个小程序来测试一下。

```
class Test {
public:
  void do_stuff(const std::vector<int>& v) {
    std::copy(v.begin(),v.end(),
      std::ostream_iterator<int>(std::cout," "));
  }
};

int main() {
  Test t;
  std::vector<int> vec;
  vec.push_back(42);
  simple_bind(&Test::do_stuff,t,_1)(vec);
}
```

当我们用上述参数实例化函数 `simple_bind` 时，类型被自动推断；`R` 是 `void`，`T` 是 `Test`，而 `Arg` 是一个 `const std::vector<int>` 引用。函数返回一个 `simple_bind_t<void,Test,Arg>` 的实例，我们立即调用它的调用操作符，并传进一个参数 `vec`.

非常不错，`simple_bind` 已经给了你关于绑定器如何工作的一些想法。现在，是时候回到 Boost.Bind 了！

关于占位符和参数

第一个例子示范了 `bind` 最多可以支持九个参数，但了解多一点关于参数和占位符如何工作的情况，可以让我们更好地使用它。首先，很重要的一点是，普通函数与成员函数之间有着非常大的差异，在绑定一个成员函数时，`bind` 表达式的第一个参数必须是成员函数所在类的实例！理解这个规则的最容易的方法是，这个显式的参数将取替隐式的 `this`，被传递给所有的非静态成员函数。细心的读者将会留意到，实际上这意味着对于成员函数的绑定器来说，只能支持八个参数，因为第一个要用于传递实际的对象。以下例子定义了一个普通函数 `print_string` 和一个带有成员函数 `print_string` 的类 `some_class`，它们将被用于 `bind` 表达式。

```
#include <iostream>
#include <string>
#include "boost/bind.hpp"

class some_class {
public:
  typedef void result_type;
  void print_string(const std::string& s) const {
    std::cout << s << '\n';
  }
};

void print_string(const std::string s) {
  std::cout << s << '\n';
}

int main() {
  (boost::bind(&print_string,_1))("Hello func!");
  some_class sc;
  (boost::bind(&some_class::print_string,_1,_2))
    (sc,"Hello member!");
}
```

第一个 `bind` 表达式绑定到普通函数 `print_string`。因为该函数要求一个参数，因此我们需要用一个占位符(`_1`)来告诉 `bind` 它的哪一个参数将被传递为 `print_string` 的第一个参数。要调用获得的函数对象，我们必须传递一个 `string` 参数给调用操作符。参数是一个 `const std::string&`，因此传递一个字面的字符串将引发一个 `std::string` 转型构造函数的调用。

```
(boost::bind(&print_string,_1))("Hello func!");
```

第二个绑定器用于一个成员函数， `some_class` 的 `print_string` 。 `bind` 的第一个参数是成员函数指针。但是，一个非静态成员函数指针并不真的是一个指针[3]。我们必须要有一个对象才可以调用这个函数。这就是为什么这个 `bind` 表达式必须声明绑定器有两个参数，调用它时两个参数都必须提供。

> [3] 是的，我知道这听起来很怪异。但它的确是真的。

```
boost::bind(&some_class::print_string,_1,_2);
```

要看看为什么会这样，就要考虑一下得到的这个函数对象要怎么使用。我们必须把一个 `some_class` 实例和一个 `print_string` 用的参数一起传递给它。

```
(boost::bind(&some_class::print_string,_1,_2))(sc,"Hello member!");
```

这个调用操作符的第一个参数是 `this` ，即那个 `some_class` 实例。注意，这第一个参数可以是一个指针(智能的或裸的)或者是一个引用； `bind` 是非常随和的。调用操作符的第二个参数是那个成员函数要用的参数。这里，我们"延迟"了所有两个参数，即我们定义的这个绑定

器，它的两个参数，对象本身及成员函数的参数，都要在调用操作符时才指定。我们不是一定非这样做不可。例如，我们可以创建一个绑定器，每次调用它时，都是对同一个对象调用 `print_string` ，就象这样：

```
(boost::bind(&some_class::print_string,some_class(),_1))
 ("Hello member!");
```

这次得到的函数对象已经包含了一个 `some_class` 实例，因此它的调用操作符只需要一个占位符(`_1`)和一个参数(一个string)。最后，我们还可以创建一个所谓的无参(nullary)函数，它连那个 string 也绑定了，就象这样：

```
(boost::bind(&some_class::print_string,
 some_class(),"Hello member!"))();
```

这些例子清楚地显示了 `bind` 的多功能性。它可用于延迟它所封装的函数的所有参数、部分参数、或一个参数也不延迟。它也可以把参数按照你所要的顺序进行重排；只要照你的需要排列占位符就行了。接下来，我们将看看如何用 `bind` 来就地创建排序用的谓词。

动态的排序标准

在对容器中的元素进行排序时，我们有时候需要创建一个函数对象以定义排序的标准，如果我们没有提供关系操作符，或者是已有的关系操作符不是我们想要的排序标准时，就需要这样做了。有些时候我们可以使用来自标准库的比较函数对象(`std::greater` , `std::greater_equal` , 等等)，但只能对已有类型进行比较，我们不能就地定义一个新的。我们将使用一个名为 `personal_info` 的类来演示 Boost.Bind 如何帮助我们。 `personal_info` 包含有 first name, last name, 和 age, 并且它没有提供任何的比较操作符。这些信息在创建以后就不再变动，并且可以用成员函数 `name` , `surname` , 和 `age` 来取出。

```
class personal_info {
  std::string name_;
  std::string surname_;
  unsigned int age_;

public:
  personal_info(
    const std::string& n,
    const std::string& s,
    unsigned int age):name_(n),surname_(s),age_(age) {}

  std::string name() const {
    return name_;
  }

  std::string surname() const {
    return surname_;
  }

  unsigned int age() const {
    return age_;
  }
};
```

我们通过提供以下操作符来让这个类可以流输出(OutputStreamable)：

```
std::ostream& operator<<(
  std::ostream& os,const personal_info& pi) {
  os << pi.name() << ' ' <<
    pi.surname() << ' ' << pi.age() << '\n';
  return os;
}
```

如果我们要对含有类型 `personal_info` 元素的容器进行排序，我们就需要为它提供一个排序谓词。为什么开始的时候我们没有为 `personal_info` 提供关系操作符呢？一个原因是，因为有几种排序的可能性，而我们不知道对于不同的用户哪一种是合适的。虽然我们也可以选择为不同的排序标准提供不同的成员 函数，但这样会加重负担，我们要在类中实现所有相关的排序标准，这并不总是可以做到的。幸运的是，我们可以很容易地用 `bind` 就地创建所需的谓词。我们先看看基于年龄(可以通过成员函数 `age` 取得)来进行排序。我们可以为此创建一个函数对象。

```
class personal_info_age_less_than :
  public std::binary_function<
  personal_info,personal_info,bool> {
public:
  bool operator()(
  const personal_info& p1,const personal_info& p2) {
    return p1.age()<p2.age();
  }
};
```

我们让 `personal_info_age_less_than` 公有派生自 `binary_function`. 从 `binary_function` 派生可以提供使用适配器时所需的 `typedef` ，例如使用 `std::not2`. 假设有一个 `vector` , `vec` , 含有类型为 `personal_info` 的元素，我们可以象这样来使用这个函数对象：

```
std::sort(vec.begin(),vec.end(),personal_info_age_less_than());
```

只要不同的比较方式的数量很有限，这种方式就可以工作良好。但是，有一个潜在的问题，计算逻辑被定义 在不同的地方，这会使得代码难以理解。利用一个较长的、描述清晰的名字可以解决这个问题，就象我们在这里做的一样，但是不是所有情况都会这样清晰，有很大 可能我们需要为大于、小于或等于关系提供一堆的函数对象。

那么，Boost.Bind 有什么帮助呢？实际上，在这个例子中它可以帮助我们三次。如果我们要解决这个问题，我们发现有三件事情要做，第一件是绑定一个逻辑操作，如 `std::less`. 这很容易，我们可以得到第一部分代码。

```
boost::bind<bool>(std::less<unsigned int>(),_1,_2);
```

注意，我们通过把 `bool` 参数提供给 `bind` ，显式地给出了返回类型。有时这是需要的，对于有缺陷的编译器或者在无法推断出返回类型的上下文时。如果一个函数对象包含 `typedef` , `result_type` , 就不需要显式给出返回类型[4]。现在，我们有了一个接受两个参数的函数对

象，两个参数的类型都是 `unsigned int`，但我们还不能用它，因为容器中的元素的类型是 `personal_info`，我们需要从这些元素中取出 age 并把它作为参数传递给 `std::less`。我们可以再次使用 `bind` 来实现。

> [4] 标准库的函数对象都定义了 `result_type`，因此它们可以与 `bind` 的返回类型推断机制共同工作。

```
boost::bind(
  std::less<unsigned int>(),
  boost::bind(&personal_info::age,_1),
  boost::bind(&personal_info::age,_2));
```

这里，我们创建了另外两个绑定器。第一个用主绑定器的调用操作符的第一个参数(`_1`)来调用 `personal_info::age`。第二个用主绑定器的调用操作符的第二个参数(`_2`)来调用 `personal_info::age`。因为 `std::sort` 传递两个 `personal_info` 对象给主绑定器的调用操作符，结果就是对来自被排序的 `vector` 的两个 `personal_info` 分别调用 `personal_info::age`。最后，主绑定器传递两个新的、内层的绑定器的调用操作符所返回的 age 给 `std::less`。这正是我们所需要的！调用这个函数对象的结果就是 `std::less` 的结果，这意味着我们有了一个有效的比较函数对象可以用来排序容器中的 `personal_info` 对象。以下是使用它的方法：

```
std::vector<personal_info> vec;
vec.push_back(personal_info("Little","John",30));
vec.push_back(personal_info("Friar", "Tuck",50));
vec.push_back(personal_info("Robin", "Hood",40));

std::sort(
  vec.begin(),
  vec.end(),
  boost::bind(
    std::less<unsigned int>(),
    boost::bind(&personal_info::age,_1),
    boost::bind(&personal_info::age,_2)));
```

我们可以简单地通过绑定另一个 `personal_info` 成员(变量或函数)来进行不同的排序，例如，按 last name 排序。

```
std::sort(
  vec.begin(),
  vec.end(),
  boost::bind(
    std::less<std::string>(),
    boost::bind(&personal_info::surname,_1),
    boost::bind(&personal_info::surname,_2)));
```

这是一种出色的技术，因为它提供了一个重要的性质：就地实现简单的函数。它使得代码易懂且易于维护。虽然技术上可以用绑定器实现基于复杂标准的排序，但那样做是不明智的。给 `bind` 表达式添加复杂的逻辑会很快失去它的清晰和简洁。虽然有时你想用绑定来做更多的事情，但最好是让绑定器与要维护它的人一样聪明，而不是更加聪明。

函数组合，Part I

一个常见的问题是，将一些函数或函数对象组合成一个函数对象。假设你需要测试一个 `int` ，看它是否大于5且小于等于10。使用"常规"的代码，你将这样写：

```
if (i>5 && i<=10) {
  // Do something
}
```

如果是处理一个容器中的元素，上述代码只有放在一个单独的函数时才能工作。如果你不想这样，那么用一个嵌套的 `bind` 也可以获得相同的效果(注意，这时通常不能使用标准库的 `bind1st` 和 `bind2nd`)。如果我们对这个问题进行分解，我们会发现我们需要：逻辑与(`std::logical_and`), 大于(`std::greater`), 和小于等于(`std::less_equal`)。逻辑与看起来就象这样：

```
boost::bind(std::logical_and<bool>(),_1,_2);
```

然后，我们需要另一个谓词来回答 `_1` 是否大于5。

```
boost::bind(std::greater<int>(),_1,5);
```

然后，我们还需要另一个谓词来回答 `_1` 是否小于等于10。

```
boost::bind(std::less_equal<int>(),_1,10);
```

最后，我们需要把它们两个用逻辑与合起来，就象这样：

```
boost::bind(
  std::logical_and<bool>(),
  boost::bind(std::greater<int>(),_1,5),
  boost::bind(std::less_equal<int>(),_1,10));
```

这样一个嵌套的 `bind` 相对容易理解，虽然它是后序的。还有，任何人都可以逐字地阅读这段代码并弄清楚它的意图。我们用一个例子来测试一下这个绑定器。

```
std::vector<int> ints;

ints.push_back(7);
ints.push_back(4);
ints.push_back(12);
ints.push_back(10);

int count=std::count_if(
  ints.begin(),
  ints.end(),
  boost::bind(
    std::logical_and<bool>(),
    boost::bind(std::greater<int>(),_1,5),
    boost::bind(std::less_equal<int>(),_1,10)));

std::cout << count << '\n';

std::vector<int>::iterator int_it=std::find_if(
  ints.begin(),
  ints.end(),
  boost::bind(std::logical_and<bool>(),
    boost::bind(std::greater<int>(),_1,5),
    boost::bind(std::less_equal<int>(),_1,10)));

if (int_it!=ints.end()) {
  std::cout << *int_it << '\n';
}
```

使用嵌套的 `bind` 时，小心地对代码进行正确的缩入非常重要，因为如果一旦缩入错误，代码就会很难理解。想想前面那段清晰的代码，再看看以下这个容易混乱的例子。

```
std::vector<int>::iterator int_it=
  std::find_if(ints.begin(),ints.end(),
    boost::bind<bool>(
    std::logical_and<bool>(),
    boost::bind<bool>(std::greater<int>(),_1,5),
      boost::bind<bool>(std::less_equal<int>(),_1,10)));
```

当然，对于较长的代码行，这是一个常见的问题，但是在使用这里所描述的结构时更为明显，在这里长语句是合理的而不是个别例外。因此，请对你之后的程序员友好些，确保你的代码行正确缩入，这样可以让人更容易阅读。

本书的一位认真的审阅者曾经问过，在前面的例子中，为什么创建了两个相同的绑定器，而不是创建一个绑定器对象然后使用两次？答案是，因为我们不知道 `bind` 所创建的绑定器的精确类型(它是由实现定义的)，我们没有方法为它声明一个变量。还有，这个类型通常都非常复杂，因为它的署名特征包括了函数 `bind` 中所有的类型信息(自动推断的)。但是，可以用另外一个工具来保存得到的函数对象，例如来自 Boost.Function 的工具。相关方法的详情请见 "Library 11: Function 11"。

这里给出的函数组合的要点与标准库的一个著名的扩充相符，即来自SGI STL的函数 `compose2` ，它在 Boost.Compose 库(现在已经不用了)中也被称为 `compose_f_gx_hx` 。

函数组合，Part II

在SGI STL中的另一个常用的函数组合是 `compose1` ，在 Boost.Compose 中是 `compose_f_gx` 。这些函数提供了用一个参数调用两个函数的方法，把最里面的函数返回的结果传递给第一个函数。有时一个例子胜过千言万语，设想你需要对容器中的浮点数元素 执行两个算术操作。我们首先把值增加10%，然后再减少10%；这个例子对于少数工作在财政部门的人来说可能是有用的一课。

```
std::list<double> values;
values.push_back(10.0);
values.push_back(100.0);
values.push_back(1000.0);

std::transform(
  values.begin(),
  values.end(),
  values.begin(),
  boost::bind(
    std::multiplies<double>(),0.90,
    boost::bind<double>(
      std::multiplies<double>(),_1,1.10)));

std::copy(
  values.begin(),
  values.end(),
  std::ostream_iterator<double>(std::cout," "));
```

你怎么知道哪个嵌套的 `bind` 先被调用呢？你也许已经注意到，总是最里面的 `bind` 先被求值。这意味着我们可以把同样的代码写得稍微有点不同。

```
std::transform(
  values.begin(),
  values.end(),
  values.begin(),
  boost::bind<double>(
    std::multiplies<double>(),
    boost::bind<double>(
      std::multiplies<double>(),_1,1.10),0.90));
```

这里，我们改变了传给 `bind` 的参数的顺序，把第一个 `bind` 的参数加在了表达式的最后。虽然我不建议这样做，但它对于理解参数如何传递给 `bind` 函数很有帮助。

bind 表达式中的是值语义还是指针语义?

当我们传递某种类型的实例给一个 `bind` 表达式时，它将被复制，除非我们显式地告诉 `bind` 不要复制它。要看我们怎么做，这可能是至关重要的。为了看一下在我们背后发生了什么事情，我们创建一个 `tracer` 类，它可以告诉我们它什么时候被缺省构造、被复制构造、被赋值，以及被析构。这样，我们就可以很容易看到用不同的方式使用 `bind` 会如何影响我们传送的实例。以下是完整的 `tracer` 类。

```
class tracer {
public:
  tracer() {
    std::cout << "tracer::tracer()\n";
  }

  tracer(const tracer& other) {
    std::cout << "tracer::tracer(const tracer& other)\n";
  }

  tracer& operator=(const tracer& other) {
    std::cout <<
      "tracer& tracer::operator=(const tracer& other)\n";
    return *this;
  }

  ~tracer() {
    std::cout << "tracer::~tracer()\n";
  }

  void print(const std::string& s) const {
    std::cout << s << '\n';
  }
};
```

我们把我们的 `tracer` 类用于一个普通的 `bind` 表达式，象下面这样。

```
tracer t;
boost::bind(&tracer::print,t,_1)
  (std::string("I'm called on a copy of t\n"));
```

运行这段代码将产生以下输出，可以清楚地看到有很多拷贝产生。

```
tracer::tracer()
tracer::tracer(const tracer& other)
tracer::tracer(const tracer& other)
tracer::tracer(const tracer& other)
tracer::~tracer()
tracer::tracer(const tracer& other)
tracer::~tracer()
tracer::~tracer()
I'm called on a copy of t

tracer::~tracer()
tracer::~tracer()  // 译注：原文没有这一行，有误
```

如果我们使用的对象的拷贝动作代价昂贵，我们也许就不能这样用 `bind` 了。但是，拷贝还是有优点的。它意味着 `bind` 表达式以及由它所得到的绑定器不依赖于原始对象(在这里是 `t`)的生存期，这通常正是想要的。要避免复制，我们必须告诉 `bind` 我们想传递引用而不是它所假定的传值。我们要用 `boost::ref` 和 `boost::cref` (分别用于引用和 `const` 引用)来做到这一点，它们也是 Boost.Bind 库的一部分。对我们的 `tracer` 类使用 `boost::ref` ，测试代码现在看起来象这样：

```
tracer t;
boost::bind(&tracer::print,boost::ref(t),_1)(
  std::string("I'm called directly on t\n"));
```

Executing the code gives us this:

```
tracer::tracer()
I'm called directly on t

tracer::~tracer()  // 译注：原文为 tracer::~tracer，有误
```

这正是我们要的，避免了无谓的复制。`bind` 表达式使用原始的实例，这意味着没有 `tracer` 对象的拷贝了。当然，它同时也意味着绑定器现在要依赖于 `tracer` 实例的生存期了。还有一种避免复制的方法；就是通过指针来传递参数而不是通过值来传递。

```
tracer t;
boost::bind(&tracer::print,&t,_1)(
  std::string("I'm called directly on t\n"));
```

因此说，`bind` 总是执行复制。如果你通过值来传递，对象将被复制，这可能对性能有害或者产生不必要的影响。为了避免复制对象，你可以使用 `boost::ref` / `boost::cref` 或者使用指针语义。

虚拟函数也可以绑定

到目前为止，我们看到了 `bind` 如何可以用于非成员函数和非虚拟成员函数，但是 它也可以用于绑定一个虚拟成员函数。通过 Boost.Bind, 你可以象使用非虚拟函数一样使用虚拟函数，即把它绑定到最先声明该成员函数为虚拟的基类的那个虚拟函数上。这个绑定器就可以用于所有的派生类。如果你绑定 到其它派生类，你就限制了可以使用这个绑定器的类[5]。考虑以下两个类 `base` 和 `derived` ：

> [5] 这与声明一个类指针来调用虚拟函数没有什么不同。指针指向的派生类越靠近底层，则越少的类可以绑定到指针。

```
class base {
public:
  virtual void print() const {
    std::cout << "I am base.\n";
  }
  virtual ~base() {}
};

class derived : public base {
public:
  void print() const {
    std::cout << "I am derived.\n";
  }
};
```

我们可以用这两个类对绑定到虚拟函数进行测试，如下：

```
derived d;
base b;
boost::bind(&base::print,_1)(b);
boost::bind(&base::print,_1)(d);
```

运行这段代码可以清楚地看到结果正是我们所希望的。

```
I am base.
I am derived.
```

对于可以支持虚拟函数，你应该不会惊讶，现在我们已经示范了它和其它函数一样运行。有一个相关的注意事项，如果你 `bind` 了一个成员函数而后来它被一个派生类重新定义了，或者一个虚拟函数在基类中是公有的而在派生类中变成了私有的，那么会发生什么呢？还可以正常工作吗？如果可以，你希望是哪一种行为呢？是的，不管你是否使用 Boost.Bind，行为都不会有变化。因面，如果你 `bind` 到 一个在其它类中被重新定义的函数，即它不是虚拟的并且派生类有一个相同特征的成员函数，那么基类中的版本将被调用。如果函数被隐藏，绑定器依然会被执行， 因为它显式地访问类型中的函数，这样即使是被隐藏的成员函数也可以使用。最后，如果虚拟函数在基类中声明为公有的，但在派生类中变成了私有的，那么对一个 派生类实例调用该函数将会成功，因为访问是通过一个基类实例产生的，而基类的成员函数是公有的。当然，这种情况显示出设计的确是有问题的。

绑定到成员变量

很多时候你需要 `bind` 数据成员而不是成员函数。例如，使用 `std::map` 或 `std::multimap` 时，元素的类型是 `std::pair<key const,data>` , 但你想使用的信息通常不是 key, 而是 data. 假设你想把一个 `map` 中的每个元素传递给一个函数，它接受单个 data 类型的参数。你需要创建一个绑定器，它把每个元素(类型为 `std::pair`)的 `second` 成员传给绑定的函数。以下代码举例说明如何实现：

```
void print_string(const std::string& s) {
  std::cout << s << '\n';
}

std::map<int,std::string> my_map;
my_map[0]="Boost";
my_map[1]="Bind";

std::for_each(
  my_map.begin(),
  my_map.end(),
  boost::bind(&print_string, boost::bind(
    &std::map<int,std::string>::value_type::second,_1)));
```

你可以 `bind` 到一个成员变量，就象你可以绑定一个成员函数或普通函数一样。要注意的是，要使得代码更易读(和写)，使用短的、方便的名字是个好主意。在前例中，对 `std::map` 使用一个 `typedef` 有助于提高可读性。

```
typedef std::map<int,std::string> map_type;
boost::bind(&map_type::value_type::second,_1)));
```

虽然需要 `bind` 到成员变量的时候没有象成员函数那么多，但是可以这样做还是很方便的。SGI STL (及其派生的库)的用户可能很熟悉 `select1st` 和 `select2nd` 函数。它们用于选出 `std::pair` 的 `first` 或 `second` 成员，与我们在这个例子中所做的一样。注意， `bind` 可以用于任意类型和任意名字。

绑定还是不绑定

Boost.Bind 库带来了很大的灵活性，但是也给程序员带来了挑战，因为有些时候本应该使用独立的函数对象的，但也会让人倾向于使用绑定器。许多工作可以也应该利用 Bind 来完成，但过度使用也是一种错误，应该在代码开始变得难以阅读、理解和维护的地方画一条分界线。不幸的是，分界线的位置是由分享(阅读、维护和扩展)代码的程序员所决定的，他们的经验决定了什么是可以接受的，什么不是。使用专门的函数对象的好处是，它们通常是无需加以说明的，而使用绑定器来提供同样清楚的信息则是一项我们必须坚持克服的挑战。例如，如果你需要创建一个你都很难弄明白的嵌套 `bind` ，有可能就是你已经过度使用了。让我们用代码来解释一下。

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
#include "boost/bind.hpp"

void print(std::ostream* os,int i) {
  (*os) << i << '\n';
}

int main() {
  std::map<std::string,std::vector<int> > m;
  m["Strange?"].push_back(1);
  m["Strange?"].push_back(2);
  m["Strange?"].push_back(3);
  m["Weird?"].push_back(4);
  m["Weird?"].push_back(5);

  std::for_each(m.begin(),m.end(),
    boost::bind(&print,&std::cout,
      boost::bind(&std::vector<int>::size,
        boost::bind(
          &std::map<std::string,
            std::vector<int> >::value_type::second,_1))));
}
```

上面这段代码实际上做了什么？有的人可以流畅地阅读这段代码[6]，但对于我们多数人来说，需要一些时间才能搞清楚它是干嘛的。是的，绑定器对 `pair` (即 `std::map&lt;std::string,std::vector&lt;int&gt; &gt;::value_type`)的成员 `second` 调用成员

函数 `size` 。这种情况下，简单的问题被绑定器弄得复杂了，创建一个小的函数对象来取代这个让人难以理解的复杂绑定器是更好的选择。一个可以完成相同工作的简单函数对象如下：

> [6] 你好，Peter Dimov.

```
class print_size {
  std::ostream& os_;
  typedef std::map<std::string,std::vector<int> > map_type;
public:
  print_size(std::ostream& os):os_(os) {}

  void operator()(
    const map_type::value_type& x) const {
    os_ << x.second.size() << '\n';
  }
};
```

这种时候使用函数对象的最大好处就是，名字是无需加以说明的。

```
std::for_each(m.begin(),m.end(),print_size(std::cout));
```

我们把这些(函数对象以及实际调用的所有代码)和前面使用绑定器的版本作一下比较。

```
std::for_each(m.begin(),m.end(),
  boost::bind(&print,&std::cout,
    boost::bind(&std::vector<int>::size,
      boost::bind(
        &std::map<std::string,
          std::vector<int> >::value_type::second,_1))));
```

或者，如果我们负点责任，为 `vector` 和 `map` 分别创建一个简洁的 `typedef` ：

```
std::for_each(m.begin(),m.end(),
  boost::bind(&print,&std::cout,
    boost::bind(&vec_type::size,
      boost::bind(&map_type::value_type::second,_1))));
```

这样可以容易点分析，但它还是有点长。

虽然使用 `bind` 版本是有一些好理由，但我想观点是很清楚的，绑定器不是非用不可的工具，使用时应该负责任，要让它们物有所值。这一点在使用标准库的容器和算法时非常、非常普遍。当事情变得太过复杂时，就回到老风格的方法上。

让绑定器把握状态

创建一个象 `print_size` 那样的函数对象时，有几个选项可用。我们在上一节中创建的那个版本中，保存了一个到 `std::ostream` 的引用，并使用这个 `ostream` 来打印 `map_type::value_type` 参数的成员 `second` 的 `size` 函数的返回值。以下是原来的

`print_size` ：

```
class print_size {
  std::ostream& os_;
  typedef std::map<std::string,std::vector<int> > map_type;
public:
  print_size(std::ostream& os):os_(os) {}

  void operator()(
    const map_type::value_type& x) const {
    os_ << x.second.size() << '\n';
 }
};
```

要重点关注的一点是，这个类是有状态的，状态就在于那个保存的 `std::ostream` . 我们可以通过向调用操作符增加一个 `ostream` 参数来去掉这个状态。这意味着这个函数对象将变为无状态的。

```
class print_size {
  typedef std::map<std::string,std::vector<int> > map_type;
public:
  typedef void result_type;
  result_type operator()(std::ostream& os,
    const map_type::value_type& x) const {
    os << x.second.size() << '\n';
  }
};
```

注意，这个版本的 `print_size` 可以很好地用于 `bind` , 因为它增加了一个 `result_type typedef` . 这样用户在使用 `bind` 时就不需要显式声明函数对象的返回类型。在这个新版本的 `print_size` 里，用户需要传递一个 `ostream` 参数来调用它。这在使用绑定器时是很容易的。用这个新的 `print_size` 重写前节中的例子，我们可以得到：

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
#include "boost/bind.hpp"

// 省略 print_size 的定义

int main() {
  typedef std::map<std::string,std::vector<int> > map_type;
  map_type m;
  m["Strange?"].push_back(1);
  m["Strange?"].push_back(2);
  m["Strange?"].push_back(3);
  m["Weird?"].push_back(4);
  m["Weird?"].push_back(5);

  std::for_each(m.begin(),m.end(),
    boost::bind(print_size(),boost::ref(std::cout),_1));
}
```

细心的读者可能觉得为什么 `print_size` 不是一个普通函数，毕竟它已经不带有任何状态了。事实上，它可以是普通函数。

```
void print_size(std::ostream& os,
  const std::map<std::string,std::vector<int> >::value_type& x) {
  os << x.second.size() << '\n';
}
```

还有更多的泛化工作可以做。我们当前版本的 `print_size` 要求其调用操作符的第二个参数是一个 `const std::map<std::string,std::vector<int> >` 引用，这不够通用。我们可以做得更好一些，让调用操作符对这个类型进行泛化。这样， `print_size` 就可以使用任意类型的参数，只要该参数含有名为 `second` 的公有成员，并且该成员有一个成员函数 `size` . 以下是改进后的版本：

```
class print_size {
public:
  typedef void result_type;
  template <typename Pair> result_type operator()
    (std::ostream& os,const Pair& x) const {
    os << x.second.size() << '\n';
  }
};
```

这个版本的用法与前一个是一样的，但它更为灵活。在创建可用于 `bind` 表达式的函数对象时，这种泛化更为重要。因为这样的函数对象可用的情形将显著增加，多数潜在的泛化都是值得做的。既然如此，我们还可以进一步放松对使用 `print_size` 的类型的要求。当前版本的 `print_size` 要求调用操作符的第二个参数是一个类似于 pair 的对象，即一个含有名为 `second` 的成员的对象。如果我们决定只要求这个参数含有成员函数 `size` , 这个函数对象就真的与它的名字相符了。

```
class print_size {
public:
  typedef void result_type;
  template <typename T> void operator()
    (std::ostream& os,const T& x) const {
    os << x.size() << '\n';
  }
};
```

当然，尽管 `print_size` 现在是与它的名字相符了，但是我们也要求用户要做的更多了。象对于我们前面的例子，就需要手工绑定一个 `map_type::value_type::second` .

```
std::for_each(m.begin(),m.end(),
  boost::bind(print_size(),boost::ref(std::cout),
    boost::bind(&map_type::value_type::second,_1)));
```

在使用 `bind` 时，通常都需要这样的折衷，泛化只能到此为止，不要损害到可用性。如果我们走到极端，甚至去掉对成员函数 `size` 的要求，那么我们就转了一圈，回到了我们开始的地方，又回到那个对多数程序员而言都过于复杂的 `bind` 表达式了。

```
std::for_each(m.begin(),m.end(),
  boost::bind(&print<sup class="docfootnote">\[7\]</sup>,&std::cout,
    boost::bind(&vec_type::size,
      boost::bind(&map_type::value_type::second,_1))));
```

> [7] `print` 函数显然也是需要的，没有 lambda 工具。

关于 **Boost.Bind** 和 **Boost.Function**

虽然本章中讨论的内容应该没有遗漏了，但是对于 Boost.Bind 和另一个库，Boost.Function，之间的配合还是值得一提，它可以提供更多的功能。我们将在 "Library 11:Function 11" 看到，不过我还是想给你一些提示。正如我们所看到的，没有一个明显的方法来保存我们的绑定器以备后用，我们只知道它们是带有某些(未知)的特征的兼容函数对象。但是，如果使用 Boost.Function, 保存函数用于以后的调用正是那个库要做的，并且它兼容于 Boost.Bind, 可以把绑定器赋值给函数，保存它们并用于以后的调用。这是一个非常有用的概念，它可以用于适配并提高了松耦合。

Bind 总结

在以下情形时使用 Bind ：

- 你需要绑定一个调用到一个普通函数，使用部分或全部参数
- 你需要绑定一个调用到一个成员函数，使用部分或全部参数
- 你需要嵌套组合函数对象

泛化绑定器的存在对于编写简洁、连贯的代码非常有用。它减少了为了适配函数/函数对象以及函数组合而创建的小函数对象的数量。虽然标准库已经提供了 Boost.Bind 的一小部分功能，但是 Boost.Bind 所具有的重大改进使得它在多数情况下成为了更好的选择。除了对已有功能进行简化外，Bind 还提供了强大的函数组合功能，这为程序员提供了强大的力量而且没有维护上的负作用。如果你已经花了时间学习 `bind1st` , `bind2nd` , `ptr_fun` , `mem_fun_ref` , 等等，那么转换到 Boost.Bind 对你而言几乎没有困难。如果你已经开始使用C++标准库所提供的绑定器，我强烈建议你开始使用 Bind, 因为它更容易学习，而且更强大。

我知道许多程序员通常都有绑定器的经验，特别是函数组合。如果你用过其中之一，我希望本章能够为你提 供某些动力，推动你更进一步。此外，回想一下这种就地声明并定义的函数意味着什么，它意味着无需维护。仅仅为了提供正确的署名和执行简单的小任务而在类的 周围创建一堆小的、看起来很简单的[8]函数对象，会导致代码的分散，与之相比，使用绑定器更容易。

> [8] 但它们并不是那么简单。

Boost.Bind 库由 Peter Dimov 创建并维护，他除了令这个库实现了完整的绑定和函数组合功能之外，还令它可以很好地工作在多数编译器环境下。

Library 10. Lambda

- Lambda 库如何改进你的程序？
- Lambda 如何适用于标准库？
- Lambda
- 用法
- Lambda 总结

Lambda 库如何改进你的程序？

- 对函数和函数对象进行适配，使之可用于标准库算法
- 绑定参数到函数调用
- 将任意的表达式转换为可以兼容标准库算法的函数对象
- 就地定义匿名函数，提高代码的可读性和可维护性
- 在需要的时间和地点实现谓词

在使用标准库或其它采用相似设计的库时，需要依靠函数或函数对象来对算法 进行配置，你通常要编写很多小的函数对象来执行一些非常简单的操作。就象我们在 "Library 9: Bind 9" 看到的那样，这很容易成为一个问题，因为有大量的小类分散在代码中，这样很难进行维护。另外，理解函数对象被调用处的代码会很难，因为有一部分的功能被定 义在别的地方。一个好的解决办法是，想办法就在调用的地方定义这些函数或函数对象。这通常可以使代码写得更快，也更容易维护，因为函数的定义就在它被使用 的地方。这正是 Boost.Lambda 库所要提供的，就地定义匿名函数。Boost.Lambda 可以创建直接定义和调用的函数对象，或者把它保存起来晚一些再调用。这与 Boost.Bind 库所提供的很相似，但 Boost.Lambda 除了可以进行参数绑定，还有其它功能，增加了控制结构、表达式到函数对象的自动转换，还支持在 lambda 表达式中的异常处理。

术语 lambda 表达式或 lambda 函数，来源于函数式编程与 lambda 演算。一个 lambda 抽象概念定义了一个匿名函数。虽然 lambda 抽象概念在函数式编程语言(functional programming language)中非常普遍，但是在象C++这样的命令式编程语言(imperative programming language)中则不是。但是，通过使用象表达式模板这样的先进技术，C++ 可以在语言中增加某种形式的 lambda 表达式。

创建 Lambda 库最早的动机是，可以在标准库算法中使用匿名函数。因为从1998年第一个C++标准发布后，标准库的使用非常广泛，我们对于什么好什么不好的认识快速增 长，而其中一个存在疑问的就是，对于众多小函数对象的定义，好象只需要一个简单的表达式就可以满足了。显然这个库就是定位于解决这些函数对象的问题，但是 对于 lambda 函数的使用还有很大的探索空间。现在，lambda 函数已经可以使用了，我们可以把它应用于以前需要用完全不同的方法来解决的问题。令人着迷和兴奋的是，象C++这样一种成熟的语言还可以探索出新的编程技 术。匿名函数和表达式模板的出现会带来怎样的新用法和新方法呢？事实是，我们不知道，因为我们还没有全力去试验它！尽管如此，这里所关注的是这个库明确要 解决的实际问题，即通过就地定义 lambda 表达式和函数来避免代码膨胀和功能的分散。我们可以用它做出更多惊人的事情，有了它我们可以更为简洁，这可以同时满足程序员和他们的经理，前者可以更加集 中精力在手边的问题，后者可以获得更高生产效率的好处(希望也是更容易维护的代码！)。

Lambda 如何适用于标准库？

这个库用于解决一个使用标准库算法时常会遇见的问题，即需要为了满足算法的要求而定义很多简单的函数对象。几乎所有的标准库算法都有一个接受函数对象的版本，这个函数对象用于执行如排序、等同性检验、转换等操作。标准库通过绑定器 `bind1st` 和 `bind2nd` 支持有限的函数组合。但是，它们能做的事情非常有限，它们只能提供参数绑定，而不能绑定表达式。在 Boost.Lambda 库中，既有对绑定参数的灵活支持，也可以直接从表达式创建函数对象，对于C++标准库来说，这是一个杰出的合作伙伴。

Lambda

头文件: `"boost/lambda/lambda.hpp"`

它包括了本库的核心部分。

```
"boost/lambda/bind.hpp"
```

它定义了 `bind` 函数。

```
"boost/lambda/if.hpp"
```

它定义了相当于 `if` 的 lambda，以及条件操作符。

```
"boost/lambda/loops.hpp"
```

它定义了循环结构(例如，`while_loop` 和 `for_loop`)。

```
"boost/lambda/switch.hpp"
```

它定义了相当于 switch 语句的 lambda。

```
"boost/lambda/construct.hpp"
```

它定义了一些工具，为增加构造函数/析构函数以及 `new` / `delete`。

```
"boost/lambda/casts.hpp"
```

它为提供了转型操作符。

```
"boost/lambda/exceptions.hpp"
```

它定义了在 lambda 表达式中进行异常处理的工具。

```
"boost/lambda/algorithm.hpp" and "boost/lambda/numeric.hpp"
```

它定义了用于嵌套函数调用的C++标准库算法的 lambda 版本(实际上就是函数对象)。

用法

与其它许多 Boost 库一样，这个库完全定义在头文件中，这意味着你不必构建任何东西就可以开始使用。但是，知道一点关于 lambda 表达式的东西肯定是有帮助的。接下来的章节会带你浏览一下这个库，还包括如何在 lambda 表达式中进行异常处理！这个库非常广泛，前面还有很多强大的东西。一个 lambda 表达式通常也称为匿名函数(*unnamed function*)。它在需要的时 候进行声明和定义，即就地进行。这非常有用，因为我们常常需要在一个算法中定义另一个算法，这是语言本身所不能支持的。作为替代，我们通过从更大的范围引 进函数和函数对象来具体定义行为，或者使用嵌套的循环结构，把算法表达式写入循环中。我们将看到，这正是 lambda 表达式可以发挥的地方。本节内有许多例子，通常例子的一部分是示范如何用"传统"的编码方法来解决问题。这样做的目的是，看看 lambda 表达式在何时以及如何帮助程序写出更具逻辑且更少的代码。使用 lambda 表达式的确存在一定的学习曲线，而且它的语法初看起来有点可怕。就象每种新的范式或工具，它们都需要去学习，但是请相信我，得到的好处肯定超过付出的代 价。

一个简单的开始

第一个使用 Boost.Lambda 的程序将会提升你对 lambda 表达式的喜爱。首先，请注意 lambda 类型是声明在 `boost::lambda` 名字空间中，你需要用一个 using 指示符或 using 声明来把这些 lambda 声明带入你的作用域。包含头文件 `"boost/lambda/lambda.hpp"` 就可以使用这个库的主要功能了，对于我们第一个程序这样已经足够了。

```
#include <iostream>
#include "boost/lambda/lambda.hpp"
#include "boost/function.hpp"
int main() {
  using namespace boost::lambda;
  (std::cout << _1 << " " << _3 << " " << _2 << "!\n")
    ("Hello","friend","my");
  boost::function<void(int,int,int)> f=
    std::cout << _1 << "*" << _2 << "+" << _3
      << "=" <<_1*_2+_3 << "\n";
  f(1,2,3);
  f(3,2,1);
}
```

第一个表达式看起来很奇特，你可以在脑子里按着括号来划分这个表达式；第一部分就是一个 lambda 表达式，它的意思基本上是说，"打印这些参数到 `std::cout`，但不是立即就做，因为我还不知道这三个参数"。表达式的第二部分才是真正调用这个函数，它说，"嘿！这里有你要的三个参数"。我们再来看看这个表达式的第一部分。

```
std::cout << _1 << " " << _3 << " " << _2 << "!\n"
```

你会注意到表达式中有三个占位符，命名为 `_1`, `_2`, 和 `_3` [1]。这些占位符为 lambda 表达式指出了延后的参数。注意，跟许多函数式编程语言的语法不一样，创建 lambda 表达式时没有关键字或名字；占位符的出现表明了这是一个 lambda 表达式。所以，这是一个接受三个参数的 lambda 表达式，参数的类型可以是任何支持 `operator<<` 流操作的类型。参数按 1-3-2 的顺序打印到 `cout` 。在这个例子中，我们把这个表达式用括号括起来，然后调用得到的这个函数对象，传递三个参数给它：`"Hello"`, `"friend"`, 和 `"my"`. 输出的结果如下：

[1]你可能没想到象 `_1` 这样的标识符是合法的，但它们的确是。标识符不能由数字打头，但可以由下划线打头，而数字可以出现在标识符的其它任何地方。

```
Hello my friend!
```

通常，我们要把函数对象传入算法，这是我们要进一步研究的，但是我们来试验一些更有用的东西，把 lambda 表达式存入另一个延后调用的函数，名为 `boost::function`. 这个有用的发明将在下一章 "Library 11: Function 11" 中讨论，现在你只有知道可以传递一个函数或函数对象给 `boost::function` 的实例并保存它以备后用就可以了。在本例中，我们定义了这样的一个函数 `f`，象这样：

```
boost::function<void(int,int,int)> f;
```

这个声明表示 `f` 可以存放用三个参数调用的函数和函数对象，参数的类型全部为 `int`. 然后，我们用一个 lambda 表达式把一个函数对象赋给它，这个表达式表示了算法 *X=S*T+U*, 并且把这个算式及其结果打印到 `cout`.

```
boost::function<void(int,int,int)> f=
  std::cout <<
    _1 << "*" << _2 << "+" << _3 << "=" <<_1*_2+_3 << "\n";
```

如你所见，在一个表达式中，占位符可以多次使用。我们的函数 `f` 现在可以象一个普通函数那样调用了，如下：

```
f(1,2,3);
f(3,2,1);
```

运行这段代码的输出如下。

```
1*2+3=5
3*2+1=7
```

任意使用标准操作符(操作符还可以被重载！)的表达式都可以用在 lambda 表达式中，并可以保存下来以后调用，或者直接传递给某个算法。你要留意，当一个 lambda 表达式没有使用占位符时(我们还没有看到如何实现，但的确可以这样用)，那么结果将是一个无参函数(对象)。

作为对比，只使用 `_1` 时，结果是一个单参数函数对象；只使用 `_1` 和 `_2` 时，结果则是一个二元函数对象；当只使用 `_1`，`_2`，和 `_3` 时，结果就是一个三元函数对象。这第一个 lambda 表达式受益于这样一个事实，即该表达式只使用了内建或常用的C++操作符，这样就可以直接编写算法。接下来，我们看看如何绑定表达式到其它函数、类成员函数，甚至是数据成员！

<a class="calibre36" id="ch10lev2sec3">在操作符不够用时就用绑定

到目前为止，我们已经看到如果有操作符可以支持我们的表达式，一切顺利，但并不总是如此的。有时我们需要把调用另一个函数作为表达式的一部分，这通常要借助于绑定；这种绑定与我们前面在创建 lambda 表达式时见过的绑定有所不同，它需要一个单独的关键字，`bind` (嘿，这真是个聪明的名字！)。一个绑定表达式就是一个被延迟的函数调用，可以是普通函数或成员函数。该函数可以有零个或多个参数，某些参数可以直接设定，另一些则可以在函数调用时给出。对于当前版本的 Boost.Lambda, 最多可支持九个参数(其中三个可以通过使用占位符在稍后给出)。要使用绑定器，你需要包含头文件 `"boost/lambda/bind.hpp"`。

在绑定到一个函数时，第一个参数就是该函数的地址，后面的参数则是函数的参数。对于一个非静态成员函数，总是有一个隐式的 `this` 参数；在一个 `bind` 表达式中，必须显式地加上 `this` 参数。为方便起见，不论对象是通过引用传递或是通过指针传递，语法都是一样的。因此，在绑定到一个成员函数时，第二个参数(即函数指针后的第一个)就是将要调用该函数的真实对象。绑定到数据成员也是可以的，下面的例子也将有所示范：

```
#include <iostream>
#include <string>
#include <map>
#include <algorithm>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/bind.hpp"
int main() {
  using namespace boost::lambda;
  typedef std::map<int,std::string> type;
  type keys_and_values;
  keys_and_values[3]="Less than pi";
  keys_and_values[42]="You tell me";
  keys_and_values[0]="Nothing, if you ask me";
  std::cout << "What's wrong with the following expression?\n";
  std::for_each(
    keys_and_values.begin(),
    keys_and_values.end(),
    std::cout << "key=" <<
      bind(&type::value_type::first,_1) << ", value="
      << bind(&type::value_type::second,_1) << '\n');
  std::cout << "\n...and why does this work as expected?\n";
  std::for_each(
    keys_and_values.begin(),
    keys_and_values.end(),
    std::cout << constant("key=") <<
      bind(&type::value_type::first,_1) << ", value="
      << bind(&type::value_type::second,_1) << '\n');
  std::cout << '\n';
  // Print the size and max_size of the container
  (std::cout << "keys_and_values.size()=" <<
    bind(&type::size,_1) << "\nkeys_and_values.max_size()="
    << bind(&type::max_size,_1))(keys_and_values);
}
```

这个例子开始时先创建一个 `std::map`，键类型为 `int` 且值类型为 `std::string`。记住，`std::map` 的 `value_type` 是一个由键类型和值类型组成的 `std::pair`。因此，对于我们的 `map`，`value_type` 就是 `std::pair<int,std::string>`，所以在 `for_each` 算法中，我们传入的函数对象要接受一个这样的类型。给出这个 `pair`，就可以取出其中的两个成员(键和值)，这正是我们第一个 `bind` 表达式所做的。

```
bind(&type::value_type::first,_1)
```

这个表达式生成一个函数对象，它被调用时将取出它的参数，即我们前面讨论的 `pair` 中的嵌套类型 `value_type` 的数据成员 `first`。在我们的例子中，`first` 是 `map` 的键类型，它是一个 `const int`. 对于成员函数，语法是完全相同的。但你要留意，我们的 lambda 表达式多做了一点；表达式的第一部分是

```
std::cout << "key=" << ...
```

它可以编译，也可以工作，但它可能不能达到目的。这个表达式不是一个 lambda 表达式；它只是一个表达式而已，再没有别的了。执行时，它打印 `key=`，当这个表达式被求值时它仅执行一次，而不是对于每个被 `std::for_each` 所访问的元素执行一次。在这个例子中，原意是把 `key=` 作为我们的每一个 `keys_and_values` 键/值对的前缀。在早一点的那些例子中，我们也是这样写的，但那里没有出现这些问题。原因在于，那里我们用了一个占位符来作为

`operator&lt;&lt;` 的第一个参数，这样就使得它成为一个有效的 lambda 表达式。而这里，我们必须告诉 Boost.Lambda 要创建一个包含 `"key="` 的函数对象。这就要使用函数 `constant`，它创建一个无参函数对象，即不带参数的函数对象；它仅仅保存其参数，然后在被调用时返回它。

```
std::cout << constant("key=") << ...
```

这个小小的修改使得所有输出都不一样了，以下是该程序的运行输出结果。

```
What's wrong with the following expression?
key=0, value=Nothing, if you ask me
3, value=Less than pi
42, value=You tell me
...and why does this work as expected?
key=0, value=Nothing, if you ask me
key=3, value=Less than pi
key=42, value=You tell me
keys_and_values.size()=3
keys_and_values.max_size()=4294967295
```

例子的最后一部分是一个绑定到成员函数的绑定器，而不是绑定到数据成员；语法是一样的，而且你可以看 到在这两种情形下，都不需要显式地表明函数的返回类型。这种奇妙的事情是由于函数或成员函数的返回类型可以被自动推断，如果是绑定到数据成员，其类型同样可以自动得到。但是，有一种情形不能得到返回类型，即当被绑定的是函数对象时；对于普通函数和成员函数，推断其返回类型是一件简单的事情[2]，但对于函数对象则不可能。有两种方法绕过这个语言的限制，第一种是由 Lambda 库自己来解决：通过显式地给出 `bind` 的模板参数来替代返回类型推断，如下所示。

[2] 你也得小心行事。我们只是说它在技术上可行。

```
class double_it {
public:
  int operator()(int i) const {
    return i*2;
  }
};
int main() {
  using namespace boost::lambda;
  double_it d;
  int i=12;
  // If you uncomment the following expression,
  // the compiler will complain;
  // it's just not possible to deduce the return type
  // of the function call operator of double_it.
  // (std::cout << _1 << "*2=" << (bind(d,_1)))(i);
  (std::cout << _1 << "*2=" << (bind<int>(d,_1)))(i);
  (std::cout << _1 << "*2=" << (ret<int>(bind(d,_1))))(i);
}
```

有两种版本的方法来关闭返回类型推断系统，短格式的版本只需把返回类型作为模板参数传给 `bind`，另一个版本则使用 `ret`，它要括住不能进行自动推断的 lambda/bind 表达式。在嵌套的 lambda 表达式中，这很容易会就得乏味，不过还有一种更好的方法可以让推断成功。我

们将在本章稍后进行介绍。

请注意，一个绑定表达式可以由另一个绑定表达式组成，这使得绑定器成为了进行函数组合的强大工具。嵌套的绑定有许多强大的功能，但是要小心使用，因为这些强大的功能同时也带来了读写以及理解代码上的额外的复杂性。

<a class="calibre36" id="ch10lev2sec4">我不喜欢 _1, _2, and _3，我可以用别的名字吗？

有的人对预定义的占位符名称不满意，因此本库提供了简便的方法来把它们[3]改为任意用户想用的名字。这是通过声明一些类型为 `boost::lambda::placeholderX_type` 的变量来实现的，其中 `X` 为 1, 2, 或 3. 例如，假设某人喜欢用 `Arg1`，`Arg2`，和 `Arg3` 来作占位符的名字：

[3] 技术上，是增加新的名字。

```
#include <iostream>
#include <vector>
#include <string>
#include "boost/lambda/lambda.hpp"
boost::lambda::placeholder1_type Arg1;
boost::lambda::placeholder2_type Arg2;
boost::lambda::placeholder3_type Arg3;
template <typename T,typename Operation>
void for_all(T& t,Operation Op) {
  std::for_each(t.begin(),t.end(),Op);
}
int main() {
  std::vector<std::string> vec;
  vec.push_back("What are");
  vec.push_back("the names");
  vec.push_back("of the");
  vec.push_back("placeholders?");
  for_all(vec,std::cout << Arg1 << " ");
  std::cout << "\nArg1, Arg2, and Arg3!";
}
```

你定义的占位符变量可以象 `_1`，`_2`，和 `_3` 一样使用。另外请注意这里的函数 `for_all`，它提供了一个简便的方法，当你经常要对一个容器中的所有元素进行操作时，可以比用 `for_each` 减少一些键击次数。这个函数接受两个参数：一个容器的引用，以及一个函数或函数对象。该容器中的每个元素将被提供给这个函数或函数对象。我认为它有时会非常有用，也许你也这样认为。运行这个程序将产生以下输出：

```
What are the names of the placeholders?
Arg1, Arg2, and Arg3!
```

创建你自己的占位符可能会影响其它阅读你的代码的人；多数知道 Boost.Lambda (或 Boost.Bind) 的程序员都熟悉占位符名称 `_1`，`_2`，和 `_3`. 如果你决定把它们称为 `q`，`w`，和 `e`，你就需要解释给你的同事听它们有什么意思。(而且你可能要经常重复地进行解释！)

<a class="calibre36" id="ch10lev2sec5">我想给我的常量和变量命名！

有时，给常量和变量命名可以提高代码的可读性。你也记得，我们有时需要创建一个不是立即求值的 lambda 表达式。这时可以使用 `constant` 或 `var`；它们分别对应于常量或变量。我们已经用过 `constant` 了，基本上 `var` 也是相同的用法。对于复杂或长一些的 lambda 表达式，对一个或多个常量给出名字可以使得表达式更易于理解；对于变量也是如此。要创建命名的常量和变量，你只需要定义一个类型为 `boost::lambda::constant_type<T>::type` 和 `boost::lambda::var_type<T>::type` 的变量，其中的 `T` 为被包装的常量或变量的类型。看一下以下这个 lambda 表达式的用法：

```
for_all(vec,
  std::cout << constant(' ') << _ << constant('\n'));
```

总是使用 `constant` 会很让人讨厌。下面是一个例子，它命名了两个常量，`newline` 和 `space`，并把它们用于 lambda 表达式。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "boost/lambda/lambda.hpp"
int main() {
  using boost::lambda::constant;
  using boost::lambda::constant_type;
  constant_type<char>::type newline(constant('\n'));
  constant_type<char>::type space(constant(' '));
  boost::lambda::placeholder1_type _;
  std::vector<int> vec;
  vec.push_back(0);
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);
  vec.push_back(4);
  for_all(vec,std::cout << space << _ << newline);
  for_all(vec,
    std::cout << constant(' ') << _ << constant('\n'));
}
```

这是一个避免重复键入的好方法，也可以使 lambda 表达式更清楚些。下面是一个类似的例子，首先定义一个类型 `memorizer`，用于跟踪曾经赋给它的所有值。然后，用 `var_type` 创建一个命名变量，用于后面的 lambda 表达式。你将会看到命名常量要比命名变量更常用到，但也有些情形会需要使用命名变量[4]。

[4] 特别是使用 lambda 循环结构时。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "boost/lambda/lambda.hpp"
template <typename T> class memorizer {
  std::vector<T> vec_;
public:
  memorizer& operator=(const T& t) {
    vec_.push_back(t);
    return *this;
  }
  void clear() {
    vec_.clear();
  }
  void report() const {
    using boost::lambda::_1;
    std::for_each(
      vec_.begin(),
      vec_.end(),
      std::cout << _1 << ",");
  }
};
int main() {
  using boost::lambda::var_type;
  using boost::lambda::var;
  using boost::lambda::_1;
  std::vector<int> vec;
  vec.push_back(0);
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);
  vec.push_back(4);
  memorizer<int> m;
  var_type<memorizer<int> >::type mem(var(m));
  std::for_each(vec.begin(),vec.end(),mem=_1);
  m.report();
  m.clear();
  std::for_each(vec.begin(),vec.end(),var(m)=_1);
  m.report();
}
```

这就是它的全部了，但在你认为自己已经明白了所有东西之前，先回答这个问题：在以下声明下 `T` 应该是什么类型？

```
constant_type<T>::type hello(constant("Hello"));
```

它是一个 `char*`？一个 `const char*`？都不是，它的正确类型是一个含有六个字符(还有一个结束用的空字符)的数组的常量引用，所以我们应该这样写：

```
constant_type<const char (&)[6]>::type
  hello(constant("Hello"));
```

这很不好看，而且对于需要修改这个字符串的人来说也很痛苦，所以我更愿意使用 `std::string` 来写。

```
constant_type<std::string>::type
  hello_string(constant(std::string("Hello")));
```

这次，你需要比上一次多敲几个字，但你不需要再计算字符的个数，如果你要改变这个字符串，也没有问题。

<a class="calibre36" id="ch10lev2sec6">*ptr_fun* 和 *mem_fun* 到哪去了？

也许你还在怀念它们，由于 Boost.Lambda 创建了与标准一致的函数对象，所以没有必要再记住这些标准库中的适配器类型了。一个绑定了函数或成员函数的 lambda 表达式可以很好地工作，而且不论绑定的是什么类型，其语法都是一致的。这可以让代码更注重其任务而不是某些奇特的语法。以下例子说明了这些好处：

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/bind.hpp"
void plain_function(int i) {
  std::cout << "void plain_function(" << i << ")\n";
}
class some_class {
public:
  void member_function(int i) const {
    std::cout <<
      "void some_class::member_function(" << i << ") const\n";
  }
};
int main() {
  std::vector<int> vec(3);
  vec[0]=12;
  vec[1]=10;
  vec[2]=7;
  some_class sc;
  some_class* psc=&sc;
  // Bind to a free function using ptr_fun
  std::for_each(
    vec.begin(),
    vec.end(),
    std::ptr_fun(plain_function));
  // Bind to a member function using mem_fun_ref
  std::for_each(vec.begin(),vec.end(),
    std::bind1st(
      std::mem_fun_ref(&some_class::member_function),sc));
  // Bind to a member function using mem_fun
  std::for_each(vec.begin(),vec.end(),
    std::bind1st(
      std::mem_fun(&some_class::member_function),psc));
  using namespace boost::lambda;
  std::for_each(
    vec.begin(),
    vec.end(),
    bind(&plain_function,_1));
  std::for_each(vec.begin(),vec.end(),
    bind(&some_class::member_function,sc,_1));
  std::for_each(vec.begin(),vec.end(),
    bind(&some_class::member_function,psc,_1));
}
```

这里真的不需要用 lambda 表达式吗？相对于使用三个不同的结构来完成同一件事情，我们可以只需向 `bind` 指出要干什么，然后它就会去做。在这个例子中，需要用 `std::bind1st` 来把 `some_class` 的实例绑定到调用中；而对于 Boost.Lambda，这是它工作的一部分。因此，下次你再想是否要用 `ptr_fun`，`mem_fun`，或 `mem_fun_ref` 时，停下来，使用 Boost.Lambda 来代替它！

<a class="calibre36" id="ch10lev2sec7">无须<*functional*>的算术操作

我们常常要按顺序对一些元素执行算术操作，而标准库提供了多个函数对象来执行算术操作，如 `plus`，`minus`，`divides`，`modulus`，等等。但是，这些函数对象需要我们多打很多字，而且常常需要绑定一个参数，这时应该使用绑定器。如果要嵌套这些算术操作，表达式很快就会变得难以使用，而 这正是 lambda 表达式可以发挥巨大作用的地方。因为我们正在处理的是操作符，既是算术上的也是C++术语上的，所以我们有能力使用 lambda 表达式直接编写我们的算法代码。作为一个小的动机，考虑一个简单的问题，对一个数值增加4。然后再考虑另一个问题，完成与标准库算法(如 `transform`)同样的工作。虽然第一个问题非常自然，而第二个则完全不一样(它需要你手工写循环)。但使用 lambda 表达式，只需关注算法本身。在下例中，我们先使用 `std::bind1st` 和 `std::plus` 对容器中的每个元素加4，然后我们使用 `lambda` 来减4。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/bind.hpp"
int main() {
  using namespace boost::lambda;
  std::vector<int> vec(3);
  vec[0]=12;
  vec[1]=10;
  vec[2]=7;
  // Transform using std::bind1st and std::plus
  std::transform(vec.begin(),vec.end(),vec.begin(),
    std::bind1st(std::plus<int>(),4));
  // Transform using a lambda expression
  std::transform(vec.begin(),vec.end(),vec.begin(),_1-=4);
}
```

差别是令人惊讶的！在使用"传统"方法进行加4时，对于未经训练的眼睛来说，很难看出究竟在干什么。从代码中我们看到，我们将一个缺省构造的 `std::plus` 实例的第一个参数绑定到4。而 lambda 表达式则写成从元素减4。如果你认为使用 `bind1st` 和 `plus` 的版本还不坏，你可以试试更长的表达式。

Boost.Lambda 支持C++中的所有算术操作符，因此几乎不再需要仅为了算术函数对象而包含 `&lt;functional&gt;`。以下例子示范了这些算术操作符中某些的用法。`vector vec` 中的每个元素被加法和乘法操作符修改。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "boost/lambda/lambda.hpp"
int main() {
  using namespace boost::lambda;
  std::vector<int> vec(3);
  vec[0]=1;
  vec[1]=2;
  vec[2]=3;
  std::for_each(vec.begin(),vec.end(),_1+=10);
  std::for_each(vec.begin(),vec.end(),_1-=10);
  std::for_each(vec.begin(),vec.end(),_1*=3);
  std::for_each(vec.begin(),vec.end(),_1/=2);
  std::for_each(vec.begin(),vec.end(),_1%=3);
}
```

简洁、可读、可维护，这就是使用 Boost.Lambda 所得到的代码的风格。跳过 `std::plus`，`std::minus`，`std::multiplies`，`std::divides`，和 `std::modulus`；使用 Boost.Lambda，你的代码总会更好。

<a class="calibre36" id="ch10lev2sec8">编写可读的谓词

标准库中的许多算法都有一个版本是接受一个一元或二元的谓词的。这些谓词是普通函数或函数对象，当 然，lambda 表达式也可以。对于会经常用到的谓词，当然应该定义函数对象，但通常，它们只使用一两次并且再不会碰到。在这种情况下，lambda 表达式是更好的选择，这既是因为代码可以更容易理解(所有功能都在同一个地方)，也是因为代码不会被一些极少使用的函数对象搞混。作为一个具体的例子，我 们在容器中查找具有某个特定值的元素。如果该元素类型已经定义了 `operator==`，则可以直接使用算法 `find`，但如果要使用其它标准来查找元素呢？以下给出类型 `search_for_me`，你如何使用 `find` 来查找第一个元素，其满足成员函数 `a` 返回 `"apple"` 的条件？

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
class search_for_me {
  std::string a_;
  std::string b_;
public:
  search_for_me() {}
  search_for_me(const std::string& a,const std::string& b)
    : a_(a),b_(b) {}
  std::string a() const {
    return a_;
  }
  std::string b() const {
    return b_;
  }
};
int main() {
  std::vector<search_for_me> vec;
  vec.push_back(search_for_me("apple","banana"));
  vec.push_back(search_for_me("orange","mango"));
  std::vector<search_for_me>::iterator it=
    std::find_if(vec.begin(),vec.end(),???);
  if (it!=vec.end())
    std::cout << it->a() << '\n';
}
```

首先，我们需要用 `find_if`,[5] 但是标记了 `???` 的地方应该怎样写呢？一种办法是：用一个函数对象来实现该谓词的逻辑。

[5] `find` 使用 `operator==; find_if` 则要求一个谓词函数(或函数对象)。

```
class a_finder {
  std::string val_;
public:
  a_finder() {}
  a_finder(const std::string& val) : val_(val) {}
  bool operator()(const search_for_me& s) const {
    return s.a()==val_;
  }
};
```

这个函数对象可以这样使用：

```
std::vector<search_for_me>::iterator it=
  std::find_if(vec.begin(),vec.end(),a_finder("apple"));
```

这可以，但两分钟(或几天)后，我们想要另一个函数对象，这次要测试成员函数 `b`. 等等...这类事情很快就会变得乏味。正如你确信的那样，这是 lambda 表达式的另一个极好的例子；我们需要某种灵活性，可以在需要的地方和需要的时间直接创建谓词。我们可以这样来写前述的 `find_if` 。

```
std::vector<search_for_me>::iterator it=
  std::find_if(vec.begin(),vec.end(),
    bind(&search_for_me::a,_1)=="apple");
```

我们 `bind` 到成员函数 `a`，并且测试它是否等于 `"apple"`，这就是我们的一元谓词，它就定义在使用的地方。但是等一下，正如它们说的，还有更多的东西。在处理数值类型时，我们可以在所有算术操作符、比较和逻辑操作符中选择。这意味着哪怕是复杂的谓词也可以直接了当地定义。仔细阅读以下代码，看看谓词是如何表示的。

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include "boost/lambda/lambda.hpp"
int main() {
  using namespace boost::lambda;
  std::vector<int> vec1;
  vec1.push_back(2);
  vec1.push_back(3);
  vec1.push_back(5);
  vec1.push_back(7);
  vec1.push_back(11);
  std::vector<int> vec2;
  vec2.push_back(7);
  vec2.push_back(4);
  vec2.push_back(2);
  vec2.push_back(3);
  vec2.push_back(1);
  std::cout << *std::find_if(vec1.begin(),vec1.end(),
    (_1>=3 && _1<5) || _1<1) << '\n';
  std::cout << *std::find_if(vec2.begin(),vec2.end(),
    _1>=4 && _1<10) << '\n';
  std::cout << *std::find_if(vec1.begin(),vec1.end(),
    _1==4 || _1==5) << '\n';
  std::cout << *std::find_if(vec2.begin(),vec2.end(),
    _1!=7 && _1<10) << '\n';
  std::cout << *std::find_if(vec1.begin(),vec1.end(),
    !(_1%3)) << '\n';
  std::cout << *std::find_if(vec2.begin(),vec2.end(),
    _1/2<3) << '\n';
}
```

如你所见，创建这些谓词就象写出相应的逻辑一样容易。这正是我喜欢使用 lambda 表达式的地方，因为它可以被任何人所理解。有时候我们也需要选择 lambda 表达式以外的机制，因为那些必须理解这些代码的人的能力；但是在这里，除了增加的价值以外没有其它了。

<a class="calibre36" id="ch10lev2sec9">让你的函数对象可以与 Boost.Lambda 一起使用

不是所有的表达式都适合使用 lambda 表达式，复杂的表达式更适合使用普通的函数对象，而且会多次重用的表达式也应该成为你代码中的一等公民。它们应该被收集为一个可重用函数对象的库。但是， 你也可能想把这些函数对象用在 lambda 表达式中，你希望它们可以与 Lambda 一起使用；不是所有函数对象都能做到。问题是函数对象的返回类型不能象普通函数那样被推断出来；这是语言的固有限制。但是，有一个定义好的方法来把这个重 要的信息提供给 Lambda 库，以使得 `bind` 表达式更加干净。作为这个问题的一个例子，我们看以下函数对象：

```
template <typename T> class add_prev {
  T prev_;
public:
  T operator()(T t) {
    prev_+=t;
    return prev_;
  }
};
```

对于这样一个函数对象，lambda 表达式不能推断出返回类型，因此以下例子不能编译。

```
#include <iostream>
#include <algorithm>
#include <vector>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/bind.hpp"
int main() {
  using namespace boost::lambda;
  std::vector<int> vec;
  vec.push_back(5);
  vec.push_back(8);
  vec.push_back(2);
  vec.push_back(1);
  add_prev<int> ap;
  std::transform(
    vec.begin(),
    vec.end(),
    vec.begin(),
    bind(var(ap),_1));
}
```

问题在于对 `transform` 的调用。

```
std::transform(vec.begin(),vec.end(),vec.begin(),bind(var(ap),_1));
```

当绑定器被实例化时，返回类型推断的机制被使用...而且失败了。因此，这段程序不能通过编译，你必须显式地告诉 `bind` 返回类型是什么，象这样：

```
std::transform(vec.begin(),vec.end(),vec.begin(),
  bind<int>(var(ap),_1));
```

这是为 lambda 表达式显式设置返回类型的正常格式的缩写，它等价于这段代码。

```
std::transform(vec.begin(),vec.end(),vec.begin(),
  ret<int>(bind<int>(var(ap),_1)));
```

这并不是什么新问题；对于在标准库算法中使用函数对象都有同样的问题。在标准库中，解决的方法是增加 `typedef` s 来表明函数对象的返回类型及参数类型。标准库还提供了助手类来完成这件事，即类模板 `unary_function` 和 `binary_function`，要让我们的例子类 `add_prev` 成为合适的函数对象，可以通过定义所需的 `typedef` s (对于一元函数对象，

是 `argument_type` 和 `result_type`，对于二元函数对象，是 `first_argument_type`，`second_argument_type`，和 `result_type`），也可以通过派生自 `unary_function/binary_function` 来实现。

```
template <typename T> class add_prev : public std::unary_function<T,T>
```

这对于 lambda 表达式是否也足够好了呢？我们可以简单地复用这种方法以及我们已有的函数对象吗？唉，答案是否定的。这种 `typedef` 方法有一个问题：对于泛化的调用操作符，当返回类型或参数类型依赖于模板参数时会怎么样？或者，当存在多个重载的调用操作符时会怎么样？由于语言支持模板的 `typedef`s，这些问题可以解决，但是现在不是这样的。这就是为什么 Boost.Lambda 需要一个不同的方法，即一个名为 `sig` 的嵌套泛型类。为了让返回类型推断可以和 `add_prev` 一起使用，我们象下面那样定义一个嵌套类型 `sig`：

```
template <typename T> class add_prev :
  public std::unary_function<T,T> {
  T prev_;
public:
  template <typename Args> class sig {
  public:
    typedef T type;
   };
// Rest of definition
```

模板参数 `Args` 实际上是一个 tuple，包含了函数对象(第一个元素)和调用操作符的参数类型。在这个例子中，我们不需要这些信息，返回类型和参数类型都是 `T`. 使用这个改进版本的 `add_prev`，再不需要在 lambda 表达式中使用返回类型推断的缩写，因此我们最早那个版本的代码现在可以编译了。

```
std::transform(vec.begin(),vec.end(),vec.begin(),bind(var(ap),_1));
```

我们再来看看 tuple 作为 `sig` 的模板参数是如何工作的，来看另一个有两个调用操作符的函数对象，其中一个版本接受一个 `int` 参数，另一个版本接受一个 `const std::string` 引用。我们必须要解决的问题是，"如果传递给 `sig` 模板的 tuple 的第二个元素类型为 `int`，则设置返回类型为 `std::string`；如果传递给 `sig` 模板的 tuple 的第二个元素类型为 `std::string`，则设置返回类型为 `double`"。为此，我们增加一个类模板，我们可以对它进行特化并在 `add_prev::sig` 中使用它。

```
template <typename T> class sig_helper {};
// The version for the overload on int
template<> class sig_helper<int> {
public:
  typedef std::string type;
};
// The version for the overload on std::string
template<> class sig_helper<std::string> {
public:
  typedef double type;
};
// The function object
class some_function_object {
  template <typename Args> class sig {
    typedef typename boost::tuples::element<1,Args>::type
      cv_first_argument_type;
    typedef typename
      boost::remove_cv<cv_first_argument_type>::type
      first_argument_type;
  public:
    // The first argument helps us decide the correct version
    typedef typename
      sig_helper<first_argument_type>::type type;
  };
  std::string operator()(int i) const {
    std::cout << i << '\n';
    return "Hello!";
  }
  double operator()(const std::string& s) const {
    std::cout << s << '\n';
    return 3.14159265353;
  }
};
```

这里有两个重要的部分要讨论：首先是助手类 `sig_helper`，它由类型 `T` 特化。这个类型可以是 `int` 或 `std::string`，依赖于要使用哪一个重载版本的调用操作符。通过对这个模板进行全特化，来定义正确的 `typedef type`。第二个要注意的部分是 `sig` 类，它的第一个参数(即 tuple 的第二个元素)被取出，并去掉所有的 `const` 或 `volatile` 限定符，结果类型被用于实例化正确版本的 `sig_helper` 类，后者具有正确的 `typedef type`. 这是为我们的类定义返回类型的一种相当复杂(但是必须！)的方法，但是多数情况下，通常都只有一个版本的调用操作符；所以正确地增加嵌套 `sig` 类是一件普通的工作。

我们的函数对象可以在 lambda 表达式中正确使用是很重要的，在需要时定义嵌套 `sig` 类是一个好主意；它很有帮助。

Lambda 表达式中的控制结构

我们已经看到强大的 lambda 表达式可以很容易地创建，但是许多编程上的问题需要我们可以表示条件，在C++中我们使用 `if` - `then` - `else`，`for`，`while`，等等。在 Boost.Lambda 中有所有的C++控制结构的 lambda 版本。要使用选择语句， `if` 和 `switch`，就分别包含头文件 `"boost/lambda/if.hpp"` 和 `"boost/lambda/switch.hpp"`。要使用循环语句， `while`，`do`，和 `for`，就包含头文件 `"boost/lambda/loops.hpp"`. 关键字不能被重载，所以语法与你前面使

用过的有所不同，但是也有很明显的关联。作为第一个例子，我们来看看如何在 lambda 表达式中创建一个简单的 if-then-else 结构。格式是 *if_then_else(*条件*, then-*语句*, else-*语句*)*。还有另外一种语法形式，即 *if*(条件)[then-语句].else*[else-*语句*]* 。

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/bind.hpp"
#include "boost/lambda/if.hpp"
int main() {
  using namespace boost::lambda;
  std::vector<std::string> vec;
  vec.push_back("Lambda");
  vec.push_back("expressions");
  vec.push_back("really");
  vec.push_back("rock");
  std::for_each(vec.begin(),vec.end(),if_then_else(
    bind(&std::string::size,_1)<=6u,
    std::cout << _1 << '\n',
    std::cout << constant("Skip.\n")));
  std::for_each(vec.begin(),vec.end(),
    if_(bind(&std::string::size,_1)<=6u) [
      std::cout << _1 << '\n'
    ]
    .else_[
      std::cout << constant("Skip.\n")
    ] );
}
```

如果你是从本章开头一直读到这的，你可能会觉得上述代码非常好读；但如果你是跳到这来的，就可能觉得惊讶了。控制结构的确增加了阅读 lambda 表达式的复杂度，它需要更长一点的时间来掌握它的用法。当你掌握了它以后，它就变得很自然了(编写它们也一样！)。采用哪一种格式完全取决于你的爱好；它们做得是同一件事。

在上例中，我们有一个 `string` 的 `vector`，如果 `string` 元素的大小小于等于6，它们就被输出到 `std::cout`；否则，输出字符串 `"Skip"`。在这个 `if_then_else` 表达式中有一些东西值得留意。

```
if_then_else(
  bind(&std::string::size,_1)<=6u,
  std::cout << _1 << '\n',
  std::cout << constant("Skip.\n")));
```

首先，条件是一个谓词，它必须是一个 lambda 表达式！其次，*then*-语句必须也是一个 lambda 表达式！第三，*else*-语句必须也是一个 lambda 表达式！头两个都很容易写出来，但最后一个很容易忘掉用 `constant` 来把字符串("Skip\n")变成一个 lambda 表达式。细心的读者会注意到例子中使用了 `6u`，而不是使用 `6`，这是为了确保执行的是两个无符号类型的比较。这样做的原因是，我们使用的是非常深的嵌套模板，这意味着如果这样一个 lambda 表达式引发了一个编译器警告，输出信息将会非常、非常长。你可以试一下去掉这个 `u`，看看你的编译器会怎样！你将看到一个关于带符号类型与无符号类型比较的警告，因为 `std::string::size` 返回一个无符号类型。

控制结构的返回类型是 `void`，除了 `if_then_else_return`，它调用条件操作符。让我们来仔细看看所有控制结构，从 `if` 和 `switch` 开始。记住，要使用 `if`-结构，必须包含 `"boost/lambda/if.hpp"`。对于 `switch`，必须包含 `"boost/lambda/switch.hpp"`。以下例子都假定名字空间 `boost::lambda` 中的声明已经通过using声明或using指令，被带入当前名字空间。

```
(if_then(_1<5,
  std::cout << constant("Less than 5")))(make_const(3));
```

`if_then` 函数以一个条件开始，后跟一个 *then*-部分；在上面的代码中，如果传给该 lambda 函数的参数小于5 (`_1<5`), `"Less than 5"` 将被输出到 `std::cout`. 你会看到如果我们用数值3调用这个 lambda 表达式，我们不能直接传递3，象这样。

```
(if_then(_1<5,std::cout << constant("Less than 5")))(3);
```

这会引起一个编译错误，因为3是一个 `int`，而一个类型 `int` (或者任何内建类型)的左值不能被 `const` 限定。因此，我们在这里必须使用工具 `make_const`，它只是返回一个对它的参数的 `const` 引用。另一个方法是把整个 lambda 表达式用于调用 `const_parameters`，象这样：

```
(const_parameters(
  if_then(_1<5,std::cout << constant("Less than 5"))))(3);
```

`const_parameters` 对于避免对多个参数分别进行 `make_const` 非常有用。注意，使用该函数时，lambda 表达式的所有参数都被视为 `const` 引用。

现在来看另一种语法的 `if_then`。

```
(if_(_1<5)
  [std::cout << constant("Less than 5")])(make_const(3));
```

这种写法更类似于C++关键字，但它与 `if_then` 所做的完全一样。函数 `if_` (注意最后的下划线)后跟括起来的条件，再后跟 *then*-语句。重复一次，选择哪种语法完全取决于你的口味。

现在，让我们来看看 *if-then-else* 结构；它们与 `if_then` 很相似。

```
(if_then_else(
  _1==0,
  std::cout << constant("Nothing"),
  std::cout << _1))(make_const(0));
(if_(_1==0)
  [std::cout << constant("Nothing")].
  else_[std::cout << _1])(make_const(0));
```

使用第二种语法增加 else-部分时，要留意 `else_` 前面的点。

lambda 表达式的返回类型是 `void`，但是有一个版本会返回一个值，它使用条件操作符。对于这种表达式的类型有一些不平常的规则(我在这里略过它们，你可以在 Boost.Lambda 的在线文档或 C++ 标准[§5.16] 找到详细的说明)。这里有一个例子，返回值被赋给一个变量，就象你在使用条件操作符一样。

```
int i;
int value=12;
var(i)=(if_then_else_return
  (_1>=10,constant(10),_1))(value);
```

这个结构没有第二种语法。这些就是 *if-then-else*，我们再看看 *switch*-语句，它与标准C++ switch有些不同。

```
(switch_statement
  _1,
  case_statement<0>
    (var(std::cout) << "Nothing"),
  case_statement<1>
    (std::cout << constant("A little")),
  default_statement
    (std::cout << _1))
  )(make_const(100));
```

对 `switch_statement` 的调用从条件变量开始，即我们这里的 `_1`，lambda 表达式的第一个参数。它后跟(最多九个)表现为整型的 case 常量；它们必须是整型的常量表达式。我们提供了两个这样的常量，0 和 1 (注意，它们可以是任何可作为整型类型的值)。最手，我们加一个可选的 `default_statement`，它在 `_1` 不匹配任何一个常量时被执行。注意，在每一个 case 常量后都隐式地增加了一个 `break`-语句，所以无需从 switch 显式退出(这对于代码的维护是一件好事[6])。

[6] Spokesmen of fall-through case-statements; please excuse this blasphemy.

现在我们来看循环语句，`for`，`while`，和 `do`. 要使用它们中的任意一个，你必须首先包含头文件 `"boost/lambda/loops.hpp"`。Boost.Lambda中与C++的 `while` 相对应的是 `while_loop`。

```
int val1=1;
int val2=4;
(while_loop(_1<_2,
  (++_1,std::cout << constant("Inc...\n"))))(val1,val2);
```

`while_loop` 语句执行到条件为 `false` 止；这里的条件是 `_1&lt;_2`，后跟循环体，即表达式 `++_1,std::cout &lt;&lt; constant("Inc...\n")`. 当然，条件和循环体本身必须是有效的 lambda 表达式。另一种语法更接近C++语法，就象 `if_` 那样。

```
int val1=1;
int val2=4;
(while_(_1<_2)
  [++_1,std::cout << constant("Inc...\n")])(val1,val2);
```

格式是 `while_(` 条件 `)[` 子语句 `]`，它可以节省不少输入...但是我个人认为对于 `while` 而言函数调用语法更容易读，虽然我(不太合理)认为 `if_` 比 `if_then(...)` 容易看。从外表看，`do_while_loop` 与 `while_loop` 非常相似，但它的子语句至少被执行一次(不象 `while`，它的条件在每次执行后求值)。

```
(do_while_loop(_1!=12,std::cout <<
  constant("I'll run once")))(make_const(12));
```

另一种语法是

```
(do_[std::cout <<
  constant("I'll run once")].while_(_1!=12))(make_const(12));
```

最后是 `for` 循环的对应，`for_loop`. 在以下例子中，使用了一个命名的延期变量来让 lambda 表达式更可读。我们在前面介绍 `constant` 和 `var` 时已经介绍过延期变量。命名的延期变量用于避免重复为常量和变量敲入 `constant` 或 `var` 。它们对你要用的东西进行命名并稍后可被引用。常用的循环格式是 `for_loop` (*init-*语句, 条件, 表达式, 语句)，即它与一个普通语句相似，但它是函数的一部分(参数)。

```
int val1=0;
var_type<int>::type counter(var(val1));
(for_loop(counter=0,counter<_1,++counter,var(std::cout)
  << "counter is " << counter << "\n"))(make_const(4));
```

采用另一种语法，语句被分为初始化、条件和表达式。

```
(for_(counter=0,counter<_1,++counter)[var(std::cout)
  << "counter is " << counter << "\n"])(make_const(4));
```

这个例子把延期变量 `counter` 初始化为0，条件为 `counter&lt;_1`, 表达式为 `++counter`.

总结一下本节的控制结构。对于我遇到并使用 lambda 表达式来解决的多数问题，事实上也可以不用它们，但有些时候，它们的确真的是救命者。对于选择哪一种语法版本，最好的办法可能是两种都用，然后感觉一下哪一种最适合于你。要注意的一点是，使用 `switch` 和循环结构时，lambda 表达式很快就会变得很大，如果你还不能熟练使用这个库，你将很难弄懂这样的表达式。这时应当小心，如果一个表达式看起来让你的程序员同事难以分析，不要考虑使用独立的函数对象(或者让他们更加熟练地使用 Boost.Lambda！)。

Lambda 表达式中的类型转换

在 lambda 表达式中有四种特殊的"转型操作符"[7] 来进行类型的转换：`ll_dynamic_cast`, `ll_static_cast`, `ll_reinterpret_cast`, 和 `ll_const_cast`. 这些名字与对应的C++关键字不一样，因为它们不能被重载。要使用这些类型转换，就要包含头文件 `"boost/lambda/casts.hpp"`. 这些函数与相对应的C++转型操作符用法类似；它们带一个显式

的模板参数，即要转成的目标类型，以及一个隐式的模板参数，即源类型。在我们的第一个例子中，我们将使用两个类，名为 `base` 和 `derived`. 我们将创建两个指向 `base` 的指针，一个指向 `base` 实例，另一个指向 `derived` 实例。然后我们尝试使用 `ll_dynamic_cast` 来从这两个指针分别获得一个 `derived*` 。

[7] 技术上，它们是返回函数对象的模板函数。

```
#include <iostream>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/casts.hpp"
#include "boost/lambda/if.hpp"
#include "boost/lambda/bind.hpp"

class base {
public:
  virtual ~base() {}
  void do_stuff() const {
    std::cout << "void base::do_stuff() const\n";
  }
};

class derived : public base {
public:
  void do_more_stuff() const {
    std::cout << "void derived::do_more_stuff() const\n";
  }
};

int main() {
  using namespace boost::lambda;
  base* p1=new base;
  base* p2=new derived;
  derived* pd=0;
  (if_(var(pd)=ll_dynamic_cast<derived*>(_1))
    [bind(&derived::do_more_stuff,var(pd))].
    else_[bind(&base::do_stuff,*_1)])(p1);
  (if_(var(pd)=ll_dynamic_cast<derived*>(_1))
    [bind(&derived::do_more_stuff,var(pd))].
    else_[bind(&base::do_stuff,*_1)])(p2);
}
```

在 `main` 中，我们做的第一件事情是创建 `p1` 和 `p2`； `p1` 指向一个 `base`, 而 `p2` 则指向一个 `derived`. 在第一个 lambda 表达式中，被赋值的 `pd` 变成了条件；它被隐式转换为 `bool`, 如果它为 `true`, *then*-部分被求值。这里，我们 `bind` 到成员函数 `do_more_stuff`. 如果 `ll_dynamic_cast` 失败了，延期变量 `pd` 将为 0, 则 *else*-部分被执行。因此，在我们例子中，lambda 表达式的第一次执行将调用 `base` 上的 `do_stuff` ，而第二次执行则调用 `derived` 上的 `do_more_stuff` ，运行这个程序的输出如下。

```
void base::do_stuff() const
void derived::do_more_stuff() const
```

注意，在这个例子中，参数 `_1` 被解引用，但这并不是必须的；如果需要它会隐式地完成。如果一个 `bind` 表达式的某个参数必须总是一个指针类型，你可以自己强制对它解引用。否则，把这件杂事留给 Boost.Lambda.

`ll_static_cast` 对于避免警告非常有用。不要用它来抑制重要的信息，但可以用来减少噪音。在前面的一个例子中，我们创建了一个 `bind` 表达式来求一个 `std::string` 的长度(使用 `std::string::size`)并将该长度与另一个整数进行比较。 `std::string::size` 的返回类型是一个无符号类型，把它与一个有符号整数进行比较(这很常见)，编译器会产生一个警告，说有符号与无符号的比较是危险的动作。但是，因为这发生 在一个 lambda 表达式中，编译器忠实地跟踪到问题的根源，告诉你某个嵌套模板的某部分应对此严重问题负责。结果是一个非常长的警告信息，由于低信噪比的原因，它可能会掩 盖了其它的问题。在泛型代码中，这有时会成为问题，因为所使用的类型不在我们的控制之内。因而，在评估过可能潜在的问题后，你常会发现使用 `ll_static_cast` 来抑制不想要的警告是有好处的。以下例子包含了示范这种行为的代码。

```
#include <iostream>
#include <string>
#include <algorithm>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/casts.hpp"
#include "boost/lambda/if.hpp"
#include "boost/lambda/bind.hpp"

template <typename String,typename Integral>
void is_it_long(const String& s,const Integral& i) {
  using namespace boost::lambda;
  (if_then_else(bind(&String::size,_1)<_2,
    var(std::cout) << "Quite short...\n",
    std::cout << constant("Quite long...\n")))(s,i);
}

int main() {
  std::string s="Is this string long?";
  is_it_long(s,4u);
  is_it_long(s,4);
}
```

泛型函数 `is_it_long` (请不要关注它是一个有点过于做作的例子)用一个 `Integral` 类型的常数变量引用来执行一个 lambda 表达式。现在，这个类型是有符号的还是无符号的并不在我们的控制之内，因此很有机会某个用户会不小心引发了一个非常冗长的警告，正如这个例子所示范的，因为用了一个有符号整数调用 `is_it_long` 。

```
is_it_long(s,4);
```

确保用户不会无意地引发此事(除了要求只能使用无符号类型)的唯一办法是让这个参数变成无符号整数类型，不管它原来是什么。这正是 `ll_static_cast` 的工作，因此我们把函数 `is_it_long` 改成这样：

```
template <typename String,typename Integral>
void is_it_long(const String& s,const Integral& i) {
  using namespace boost::lambda;
  (if_then_else(bind(&String::size,_1)<
    ll_static_cast<typename String::size_type>(_2),
    var(std::cout) << "Quite short...\n",
    std::cout << constant("Quite long...\n")))(s,i);
}
```

这种情况不会经常发生(至少我没碰到几次)，但只要它发生了，这种解决方法就可用。`ll_const_cast` 和 `ll_reinterpret_cast` 的用法与我们已经看到的相似，所以就不再举例了。小心地使用它们，如果没有非常必要的原因(我想不到有什么原因)，尽量不要使用 `ll_reinterpret_cast` 。这是对称的；如果你需要用它，很大机会你做了一些不应该做的事情。

构造与析构

当有必要在 lambda 表达式中创建或销毁对象时，就需要一些特殊的处理和语法。首先，你不可能获取构造函数或析构函数的地址，也就不可能对它们使用标准的 `bind` 表达式。此外，操作符 `new` 和 `delete` 有固定的返回类型，因此它们对于任何类型都不能返回 lambda 表达式。如果你需要在 lambda 表达式中创建或销毁对象，先要确保包含头文件 `"boost/lambda/construct.hpp"`, 它包含了模板 `constructor`, `destructor`, `new_ptr`, `new_array`, `delete_ptr`, 以及 `delete_array`. 我们来看看如何使用它们，并主要关注 `constructor` 和 `new_ptr`, 它们在构造对象时是最常用的。

我们的第一个例子是一个以智能指针作为元素的容器，我们想在 lambda 表达式中重设智能指针的内容。这通常需要一个对 `operator new` 的调用；例外的情形是使用了客户化的分配机制，或者是某种工厂方法(factory method)。我们要用 `new_ptr` 来做，如果你想要或者需要的话，通常也可以在赋值表达式中使用 `constructor` 。我们两种方法都试一下。我们将定义两个类， `base` 和 `derived`, 以及一个 `boost::shared_ptr<base>` 的 `std::map` ，它以 `std::strings` 为索引。在阅读本例中的 lambda 表达式之前，先来一下深呼吸；它们将是你在本章所见到的最复杂的两个 lambda 表达式。虽然复杂，但是要理解它们是干什么的还是很明白的。只是要花你一点时间。

```
#include <iostream>
#include <map>
#include <string>
#include <algorithm>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/construct.hpp"
#include "boost/lambda/bind.hpp"
#include "boost/lambda/if.hpp"
#include "boost/lambda/casts.hpp"
#include "boost/shared_ptr.hpp"

class base {
public:
  virtual ~base() {}
};

class derived : public base {
};

int main() {
  using namespace boost::lambda;

  typedef boost::shared_ptr<base> ptr_type;
  typedef std::map<std::string,ptr_type> map_type;

  map_type m;
  m["An object"]=ptr_type(new base);
  m["Another object"]=ptr_type();
  m["Yet another object"]=ptr_type(new base);

  std::for_each(m.begin(),m.end(),
    if_then_else(!bind(&ptr_type::get,
      bind(&map_type::value_type::second,_1)),
      (bind(&map_type::value_type::second,_1)=
        bind(constructor<ptr_type>(),bind(new_ptr<derived>())),
        var(std::cout) << "Created a new derived for \"" <<
          bind(&map_type::value_type::first,_1) << "\".\n"),
      var(std::cout) << "\"" <<
        bind(&map_type::value_type::first,_1)
        << "\" already has a valid pointer.\n"));

  m["Beware, this is slightly tricky"]=ptr_type();
  std::cout << "\nHere we go again...\n";

  std::for_each(m.begin(),m.end(),
    if_then_else(!bind(&map_type::value_type::second,_1),
      ((bind(static_cast<void (ptr_type::*)(base*)>
        (&ptr_type::reset<base>),
        bind(&map_type::value_type::second,_1),
        bind(new_ptr<base>()))),
      var(std::cout) << "Created a new derived for \""
        << bind(&map_type::value_type::first,_1)
        << "\".\n"),
      var(std::cout) << "\"" <<
        bind(&map_type::value_type::first,_1)
        << "\" already has a valid pointer.\n"));
}
```

你都看懂了，是吗？以防万一有些混乱，我来解释一下在这个例子中发生了什么。首先，这两个 lambda 表达式做的是同一件事情。它们对 `std::map` 中的每一个当前为 null 的元素设置有效的指针。以下是程序运行的输出：

```
"An object" already has a valid pointer.
"Another object" already has a valid pointer.
"Yet another object" already has a valid pointer.
<small class="calibre41">// 译注：前面这三行在原书中有重复，共六行，与程序运行结果不符</small>

Here we go again...
"An object" already has a valid pointer.
"Another object" already has a valid pointer.
Created a new derived for "Beware, this is slightly tricky".
"Yet another object" already has a valid pointer.
```

输出显示我们设法把有效的对象放入 `map` 的每一个元素，但是这是如何办到的呢？

两个表达式完成了相同的工作，但各自有不同的方法。从第一个开始，我们来把这个 lambda 表达式切开来看看它是如何工作的。当然，第一部分是条件，它很普通：[8]

[8] 它还可以更简单，我们即将会看到。

```
!bind(&ptr_type::get,bind(&map_type::value_type::second,_1))
```

这样更容易看了，对吗？从最里面的 `bind` 开始读这个表达式，它告诉我们将绑定到成员 `map_type::value_type::second` (它是一个 `ptr_type`)，然后我们再绑定成员函数 `ptr_type::get` (它返回 `shared_ptr` 的指向物)，最后我们对整个表达式执行 `operator!` . 由于指针可以隐式转换为 `bool` , 因此这是一个有效的布尔表达式。看完条件部分，我们来看 *then*-部分。

```
bind(&map_type::value_type::second,_1)=
  bind(constructor<ptr_type>(),
    bind(new_ptr<derived>())),
```

这里有三个 `bind` 表达式，第一个(我们从最左边开始，因为这个表达式有一个赋值)取出成员 `map_type::value_type::second` , 它是一个智能指针。这就是我们要赋给它一个新的 `derived` 的那个值。第二和第三个表达式是嵌套的，所以我们从里向外读。里面的 `bind` 负责堆上的一个 `derived` 实例的缺省构造，我们再在它的结果之上 `bind` 一个对 `ptr_type` (智能指针类型)的 `constructor` 的调用，然后把它的结果赋值(使用普通的赋值符)给最先的那个 `bind` 表达式。然后，我们给这个 *then*-部分再加一个表达式，打印出一个简短的信息和元素的键值。

```
var(std::cout) << "Created a new derived for \"" <<
  bind(&map_type::value_type::first,_1) << "\".\n")
```

最后，我们加上语句的 *else*-部分，它打印出元素的键值和一些文字。

```
var(std::cout) << "\"" <<
  bind(&map_type::value_type::first,_1)
  << "\" already has a valid pointer.\n"));
```

把这个表达式分开来读，很明显它们并不是那么复杂，虽然整个 看起来很可怕。很重要的一点是，缩进和分离这些代码可以更容易阅读。我们可以写出类似的表达式来完成这件工作，它与这一个版本非常不同，更难阅读，但是它 的效率稍高一些。这里要注意的是，通常都会有好几种方法来写 lambda 表达式，就象其它编程问题的情况一样。在写代码之前应该多想一下，因为你的选择会影响最终结果的可读性。作为比较，以下是我提到的另一个版本：

```
std::for_each(m.begin(),m.end(),
  if_then_else(!bind(&map_type::value_type::second,_1),
    ((bind(static_cast<void (ptr_type::*)(base*)>
      (&ptr_type::reset<base>),
      bind(&map_type::value_type::second,_1),
      bind(new_ptr<derived>()))),
    var(std::cout) << "Created a new derived for \"" <<
      bind(&map_type::value_type::first,_1) << "\".\n"),
    var(std::cout) << "\"" <<
      bind(&map_type::value_type::first,_1)
      << "\" already has a valid pointer.\n"));
```

这不是好的代码，这些代码由于类型转换和复杂的嵌套 `bind` s 而变得混乱，与前面那个版本相比，这个版本很容易使我们偏离主要逻辑。为了弄明白它，我们再来把这个表达式切开成几个部分。首先，我们有条件部分，它很简单(这个表达式中没有其它东西！)；我们利用对 `shared_ptr` 的了解，它告诉我们有一个到 `bool` 的隐式转换可用。因此我们可以去掉在前一个版本中使用的到成员函数 `get` 的 `bind` 。

```
!bind(&map_type::value_type::second,_1)
```

这个条件部分与原先的表达式一样工作。接下来的部分是：

```
bind(static_cast<void (ptr_type::*)(base*)>
  (&ptr_type::reset<base>),
  bind(&map_type::value_type::second,_1),
  bind(new_ptr<derived>()))
```

这真的很难分析，所以我们先避开它。我们不使用赋值，而是直接使用成员函数 `reset`，它不仅是泛化的而且还是重载的。因此我们需要执行 `static_cast` 来告诉编译器我们要用那个版本的 `reset` 。这里主要是 `static_cast` 让表达式变得复杂，但是从最里面的表达式开始分析，我们就可以弄懂它。我们绑定一个调用到 `operator new`，创建一个 `derived` 实例，然后我们把结果绑定到智能指针(通过成员 `map_type::value_type::second`), 然后再绑定到 `shared_ptr` 的成员函数 `reset`. 结果是，对元素中的智能指针调用 `reset` ，参数是一个新构造的 `derived` 实例。虽然我们在前一个例子中也完成了同样的工作，但这个版本更难以理解。

要记住，通常有一些因素让 lambda 表达式更容易或者更难阅读和理解，要认真考虑这些因素并尽可能选择容易的形式。这对于获得这个库所提供的能力是必要的，而且它会影响到你后面的程序员。

抛出及捕获异常

我们已经来到本章的最后一节，讨论 lambda 表达式中的异常处理。如果你对这个话题的反应是怀疑在 lambda 表达式中是否需要异常处理，这就和我的第一感觉一样了。但是，这可能还不是你的想法。你写过在处理数据的循环中执行局部异常处理的代码吗？是的，手写的循 环可以避免使用 Boost.Lambda 库，因此把异常处理放入 lambda 表达式是很自然的。

要使用 Boost.Lambda 的异常处理工具，就要包含头文件 `"boost/lambda/exceptions.hpp"`. 我们来重用一下前面看到的类 `base` 和 `derived` ，并象我们前面做过的那样执行 `dynamic_cast` ，不过这次我们要对引用执行转型而不是对指针，这意味着如果失败的话， `dynamic_cast` 将抛出一个异常。这使得这个例子比前面那个更直观，因为我们不需要再使用 `if` 语句。

```
#include <iostream>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/casts.hpp"
#include "boost/lambda/bind.hpp"
#include "boost/lambda/exceptions.hpp"

int main() {
  using namespace boost::lambda;

  base* p1=new base;
  base* p2=new derived;

  (try_catch(
    bind(&derived::do_more_stuff,ll_dynamic_cast<derived&>(*_1)),
      catch_exception<std::bad_cast>(bind(&base::do_stuff,_1))))(p1);
  (try_catch(
    bind(&derived::do_more_stuff,
      ll_dynamic_cast<derived&>(*_1)),
        catch_exception<std::bad_cast>(
          bind(&base::do_stuff,_1))))(p2);
}
```

这些表达式示范了把一个表达式包装到一个对 tr `y_catch` 的调用。 `try_catch` 的通常形式为：

```
try_catch(_expression_,
  catch_exception<T1>(_expression_),
  catch_exception<T2>(_expression_,
  catch_all(_expression_))
```

在这段例子代码中，表达式对 `derived&` 使用 `dynamic_cast` 。第一个转型由于 `p1` 指向一个 `base` 实例而失败；第二个转型则由于 `p2` 指向一个 `derived` 实例而成功。请留意对占位符的解引用(`*_1`)。这是必须的，因为我们是传送指针参数给表达式的，而 `dynamic_cast` 要求的是对象或引用。如果你需要 `try_catch` 处理几种类型的异常，就要确保把最特化的类型放在前面，就象普通的异常处理代码一样。[9]

9] 否则，一个更为通用的类型将会匹配该异常而不能查找到更为特殊的类型。具体详情请参考你喜欢的C++书籍。

如果我们想访问捕获的异常，可以用一个特别的占位符，`_e`. 当然，在 `catch_all` 中不能这样做，就象在 `catch (...)` 中没有异常对象一样。继续前面的例子，我们可以打印出令 `dynamic_cast` 失败的原因，象这样：

```
try_catch(
  bind(&derived::do_more_stuff,ll_dynamic_cast<derived&>(*_1)),
  catch_exception<std::bad_cast>
    (std::cout << bind(&std::exception::what,_e))))(p1);
```

在处理一个派生自 `std::exception` 的异常类型时——这是很常见的情形——你可以绑定到虚拟成员函数 `what`, 就象这里所示范的那样。

但是有时候，你不想捕获异常，而是想抛出一个异常。这可以通过函数 `throw_exception` 来实现。因为你需要创建一个异常对象用来抛出，所以从一个 lambda 表达式中抛出异常通常还要使用 `constructor` 。下面这个例子定义了一个异常类， `some_exception`, 它公有派生自 `std::exception`, 如果 lambda 表达式的参数为 `true` ，就创建并抛出一个异常。

```
#include <iostream>
#include <exception>
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/exceptions.hpp"
#include "boost/lambda/if.hpp"
#include "boost/lambda/construct.hpp"
#include "boost/lambda/bind.hpp"

class some_exception : public std::exception {
  std::string what_;
public:
  some_exception(const char* what) : what_(what) {}

  virtual const char* what() const throw() {
    return what_.c_str();
  }
  virtual ~some_exception() throw() {}
};

int main() {
  using namespace boost::lambda;

  try {
    std::cout << "Throw an exception here.\n";

    (if_then(_1==true,throw_exception(
      bind(constructor<some_exception>(),
        constant("Somewhere, something went \
        terribly wrong.")))))(make_const(true));

    std::cout << "We'll never get here!\n";
  }
  catch(some_exception& e) {
    std::cout << "Caught exception, \"" << e.what() << "\"\n";
  }
}
```

运行这段程序将产生以下输出：

```
Throw an exception here.
Caught exception, "Somewhere, something went terribly wrong."
```

最有趣的地方是抛出异常的那段代码。

```
throw_exception(
  bind(constructor<some_exception>(),
    constant("Somewhere, something went \
      terribly wrong."))
```

`throw_exception` 的参数是一个 lambda 表达式。这个例子中，它被创建并绑定到对 `some_exception` 构造函数的调用，我们把一个字符串作为 `what` 参数传给该构造函数。

这就是在 Boost.Lambda 中进行异常处理的全部内容。永远记住，要小心谨慎地使用这些工具，它们可以让生活更轻松或更困难，这取决于你能否很好地并明智地使用它们[10] 。抛出并处理异常在你的 lambda 表达式中并不常见，但有些时候还是需要的。

[10] 小心应了这句格言，"如果你只有锤子，那么所有东西看起来都象钉子"。

Lambda 总结

以下情形时使用 Lambda ：

- 你不想创建一个简单的函数对象
- 你需要在调用函数时调整参数顺序或 arity
- 你想就地创建与标准一致的函数对象
- 你需要灵活并可读的谓词

上述原因只是值得使用本库的几种情形。虽然多数情况下，它会与标准库算法一起用，至少部分原因是由于在其它库(就算是 Boost 库)中这样的设计还不多见。通过函数对象来进行算法配置的需要并不能验证本库的有效性，离完全弄清楚它在哪些地方可以带来好处还有很长一段距离。思考一下这个库可能的应用，一定会提高你当前的设计。

Boost.Lambda 是我最喜欢的库之一，主要是因为它提供语言本身没有的很多可用的功能。要象STL在全世界所有程序员的心中一样，它还缺少一些东西。要让算法高效地工作， 还需要一些函数对象以外的东西。这正是 Boost.Lambda 的推动力，它的丰富特性带来了真正简练的编程风格。有许多地方可以使用 lambda 表达式，但还有很多要探究的地方。对于C++而言，这是某种程度上的函数式编程，它是一种仍在探索的编程范式。这些对 Lambda 库的介绍可以推动你继续对它的探究。公平地说，这种语法与"真正"的函数式编程语言相比，显得有点笨拙，而且对于新的用户来说也需要一点时间来习惯它。但 是，这对于使用本库的任何C++程序员都有巨大的价值！我希望它也能成为你喜欢的库。

非常感谢 Jaakko Järvi 和 Gary Powell, 他们是本库的作者，并且是C++函数式编程的真正先驱！

11. Function

- Function 库如何改进你的程序？
- Function 如何适用于标准库？
- Function
- 用法
- Function 总结

Function 库如何改进你的程序？

- 保存函数指针和函数对象，用于后续的调用

在含有回调的设计中，常常需要保存函数和函数对象，而且某些函数或类也是 通过函数指针或函数对象来配 制其客户化功能。传统上，通常使用函数指针来实现回调及延迟调用的函数。但是，仅仅使用函数指针会有很多限制，更好的方法是采用泛型机制来定义要被保存的函数的署名特 征，而让调用者来决定提供哪一种类型的函数实体(函数指针或 函数对象)。这样就可以使用任何行为类似于函数的实体，例 如，使用 Boost.Bind 和 Boost.Lambda 所返回的结果。这意味着可以给这些被保存的函数 增加状态(因为函数对象是一种类)。这种泛化由 Boost.Function 提供。这个库用于保存并然后调用函数或函数对象。

Function 如何适用于标准库？

本库提供了当前标准库所不具备的功能。在那些对商业逻辑的表示层进行 解耦的框架中，使用泛型回调是非常自然的、常见的。由于C++标 准库不支持保存函数指针和函数对象以供稍后的调用，因此这个工具为标准库提供了非常重要的扩展。而且，本库完全兼容于标准库的绑定器(`bind1st` 和 `bind2nd`)，就象前面所 讨论过的其它绑定器一样，如 Boost.Bind 和 Boost.Lambda.

Function

头文件: `"boost/function.hpp"`

头文件 `"function.hpp"` 包含了带有从0到10个参数的函数原型(这是实现所定义的，在当前实现中缺省的上限就是10[1])。你也可以根据你的需要，只包含相应参数数量的头文件，文件名为 `"function/functionN.hpp"`, 其中N可以从0到10。Boost.Function有两种不同的接口，其中一种的好处在于它的语法接近于函数声明(而且不要求函数的签名包含参数的数量)，另一种接口的好处在于可以在多个编译器中工作。选择哪一种接口，至少部分地取决于你使用的编译器。如果可以，就使用我们称为首选语法(preferred syntax)的那种。在本章中，两种格式都会用到。

> [1] Boost.Function 可以配置为支持多达127个参数。

使用首选语法的声明

一个 `function` 的声明包括该 `function` 所兼容的函数或函数对象的签名以及返回类型。结果以及参数的类型以单个参数的方式全部提供给模板。例如，声明一个 `function` ，它返回 `bool` 并接受一个类型 `int` 的参数，如下：

```
boost::function<bool (int)> f;
```

可以在括号中给出参数列表，以逗号分隔，就象普通的函数声明一样。所以，声明一个没有返回值(`void`)并带有类型分别为 `int` 和 `double` 的两个参数的函数，就象这样：

```
boost::function<void (int,double)> f;
```

使用兼容语法的声明

声明 `function` s 的第二种方法是，分别给出函数调用的返回类型及参数类型作为模板类型参数。并且，要在 `function` 类的名字中加上后缀，后缀是一个表示 `function` 可接受的参数数量的整数。例如，声明一个返回 `bool` 并接受一个类型 `int` 的参数的函数，方法如下：

```
boost::function1<bool,int> f;
```

这里的数字是对应函数可接受的参数数量，在上例中有一个参数(`int`)，所以是 `function1` 。更多的参数就意味着要给出更多的模板类型参数并且改变这个数字后缀。一个 `function` ，返回 `void` 并接受类型为 `int` 和 `double` 的两个参数，如下：

```
boost::function2<void,int,double> f;
```

事实上，这个库由一组类组成，其中每个类带有不同数量的参数。如果包含头文件 `"function.hpp"` 就无需关心这一点，但如果包含带数字的头文件，你就必须包含正确数字的头文件。

首选语法更容易阅读，也更象在声明一个函数，所以你应该尽可能使用它。不幸的是，虽然首选语法是完全 合法的C++并且更易读，但是还不是所有编译器都可以支持它。如果你的编译器正好不能处理这种语法，你就必须使用另一种格式。如果你需要编写最大兼容性的 代码，你也只能选择使用另一种格式。让我们来看一下 `function` 的接口中最重要的部分。

成员函数

```
function();
```

缺省构造函数创建一个空的函数对象。如果一个空的 `function` 被调用，将会抛出一个类型为 `bad_function_call` 的异常。

```
template <typename F> function(F g);
```

这个泛型的构造函数接受一个兼容的函数对象，即这样一个函数或函数对象，它的返回类型与被构造的 `function` 的返回类型或者一样，或者可以隐式转换，并且它的参数也要与被构造的 `function` 的参数类型或者一样，或者可以隐式转换。注意，也可以使用另外一个 `function` 实例来进行构造。如果这样做，并且 `function f` 为空，则被构造的 function 也为空。使用空的函数指针和空的成员函数指针也会产生空的 `function` 。

```
template <typename F> function(reference_wrapper<F> g);
```

这个构造函数与前一个类似，但它接受的函数对象包装在一个 `reference_wrapper` 中，这用以避免通过值来传递而产生函数或函数对象的一份拷贝。这同样要求函数对象兼容于 `function` 的签名。

```
function& operator=(const function& g);
```

赋值操作符保存 `g` 中的函数或函数对象的一份拷贝；如果 `g` 为空，被赋值的函数也将为空。

```
template<typename F> function& operator=(F g);
```

这个泛型赋值操作符接受一个兼容的函数指针或函数对象。注意，也可以用另一个 `function` 实例(带有不同但兼容的签名)来赋值。这同样意味着，如果 `g` 是另一个 `function` 实例且为空，则赋值后的函数也为空。赋值一个空的函数指针或空的成员函数指针也会使 `function` 为空。

```
bool empty() const;
```

这个成员函数返回一个布尔值，表示该 function 是含有一个函数/函数对象或是为空。如果有一个目标函数或函数对象可被调用，它返回 `false` 。因为一个 function 可以在一个布尔上下文中测试，或者与0进行比较，因此这个成员函数可能会在未来版本的库中被取消，你应该避免使用它。

```
void clear();
```

这个成员函数清除 `function`，即它不再关联到一个函数或函数对象。如果 function 已经是空的，这个调用没有影响。在调用后，function 肯定为空。令一个 function 为空的首选方法是赋0给它； `clear` 可能在未来版本的库中被取消。

```
operator safe_bool() const
```

这个转型函数返回一个未指定类型(由 `safe_bool` 表示)，它可以用于布尔上下文中。如果 `function` 为空，返回值为 `false`. 如果 function 中保存了一个函数指针或函数对象，则返回值为 `true`. 注意，使用一个不同于 `bool` 的类型，可以使得这个转型函数十分安全且不会被重载干扰，同时还提供了直接在布尔上下文中测试 `function` 实例的惯用法。它等同于表达式 `!!f`，其中 `f` 为一个 `function` 实例。

```
result_type operator()(Arg1 a1, Arg2 a2, ..., ArgN aN) const;
```

调用操作符是调用 `function` 的方法。你不能调用一个空的 `function` ，那样会抛出一个 `bad_function_call` 异常，即当 `!f.empty()` , `if (f)`，或 `if (!!f)` 返回 `true` 时。调用操作符的执行会调用 `function` 中的函数或函数对象，并返回它的结果。

用 法

要开始使用 Boost.Function, 就要包含头文件 `"boost/function.hpp"`, 或者某个带数字的版本，从 `"boost/function/function0.hpp"` 到 `"boost/function/function10.hpp"`. 如果你知道你想保存在 `function` 中的函数的参数数量，这样做可以让编译器仅包含需要的头文件。如果包含 `"boost/function.hpp"`, 那么就会把其它的头文件也包含进去。

理解被存函数的最佳方法是把它想象为一个普通的函数对象，该函数对象用于封装另一个函数(或函数对象)。这个被存的函数的最大用途是它可以被多次调用，而无须在创建 `function` 时立即使用。在声明 `function`s 时，声明中最重要的部分是函数的签名。这部分即是告诉 `function` 它将保存的函数或函数对象的签名和返回类型。我们已经看到，有两种方法来执行这个声明。这里有一个完整的程序，程序声明了一个 `boost::function`，它可以保存返回 `bool` (或某个可以隐式转换为 `bool` 的类型)并接受两个参数的类函数实体，第一个参数可以转换为 `int`, 第二个参数可以转换为 `double`.

```
#include <iostream>
#include "boost/function.hpp"

bool some_func(int i,double d) {
  return i>d;
}

int main() {
  boost::function<bool (int,double)> f;
  f=&some_func;
  f(10,1.1);
}
```

当 `function f` 首次创建时，它不保存任何函数。它是空的，可以在一个布尔上下文中进行测试。如果你试图调用一个没有保存任何函数或函数对象的 `function`，它将抛出一个类型 `bad_function_call` 的异常。为了避免这个问题，我们用普通的赋值语法把一个指向 `some_func` 的指针赋值给 `f`。这导致 `f` 保存了到 `some_func` 的指针。最后，我们用参数 10 (一个 `int`) 和 1.1 (一个 `double`)来调用 `f` (用函数调用操作符)。要调用一个 `function`, 你必须提供被存函数或函数对象所期望的准确数量的参数。

回调的基础

我们先来看看在没有 Boost.Function 以前我们如何实现一个简单的回调，然后再把代码改为使用 `function`, 并看看会带来什么优势。我们从一个支持某种简单的回调形式的类开始，它可以向任何对新值关注的对象报告值的改变。这里的回调是一种传统的C风格回调，即使 用普通函数。这种回调用可用于象GUI控制这样的场合，它可以通知观察者用户改变了它的值，而不需要对监听该信息的客户有任何特殊的知识。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "boost/function.hpp"

void print_new_value(int i) {
  std::cout <<
    "The value has been updated and is now " << i << '\n';
}

void interested_in_the_change(int i) {
  std::cout << "Ah, the value has changed.\n";
}

class notifier {
  typedef void (*function_type)(int);
  std::vector<function_type> vec_;
  int value_;
public:
  void add_observer(function_type t) {
    vec_.push_back(t);
  }

  void change_value(int i) {
    value_=i;
    for (std::size_t i=0;i<vec_.size();++i) {
      (*vec_[i])(value_);
    }
  }
};

int main() {
  notifier n;
  n.add_observer(&print_new_value);
  n.add_observer(&interested_in_the_change);

  n.change_value(42);
}
```

这里的两个函数，`print_new_value` 和 `interested_in_the_change`，它们的函数签名都兼容于 `notifier` 类的要求。这些函数指针被保存在一个 `vector` 内，并且无论何时它的值被改变，这些函数都会在一个循环里被调用。调用这些函数的一种语法是：

```
(*vec_[i])(value_);
```

值(`value_`)被传递给解引用的函数指针(即 `vec_[i]` 所返回的)。另一种写法也是有效的，即这样：

```
vec_[i](value_);
```

这种写法看起来更好看些，但更为重要的是，它还可以允许你把函数指针更换为 Boost.Function 而没有改变调用的语法。现在，工作还是正常的，但是，唉，函数对象不能用于这个 `notifier` 类。事实上，除了函数指针以外，别的任何东西都不能用，这的确是一种局限。但是，如果我们使用 Boost.Function，它就可以工作。重写这个 `notifier` 类非常容易。

```
class notifier {
  typedef boost::function<void(int)> function_type;
  std::vector<function_type> vec_;
  int value_;
public:
  template <typename T> void add_observer(T t) {
    vec_.push_back(function_type(t));
  }

  void change_value(int i) {
    value_=i;
    for (std::size_t i=0;i<vec_.size();++i) {
      vec_[i](value_);
    }
  }
};
```

首先要做的事是，把 `typedef` 改为代表 `boost::function` 而不是函数指针。之前，我们定义的是一个函数指针；现在，我们使用泛型方法，很快就会看到它的用途。接着，我们把成员函数 `add_observer` 的签名改为泛化的参数类型。我们也可以把它改为接受一个 `boost::function`，但那样会要求该类的用户必须也知道 `function` 的使用方法[2]，而不是仅仅知道这个观察者类型的要求就行了。应该注意到 `add_observer` 的这种变化并不应该是转向 `function` 的结果；无论如何代码应该可以继续工作。我们把它改为泛型的；现在，不管是函数指针、函数对象，还是 `boost::function` 实例都可以被传递给 `add_observer`，而无须对已有用户代码进行任何改动。把元素加入到 `vector` 的代码有一些修改，现在需要创建一个 `boost::function&lt;void(int)&gt;` 实例。最后，我们把调用这些函数的语法改为可以使用函数、函数对象以及 `boost::function` 实例[3]。这种对不同类型的类似函数的"东西"的扩展支持可以立即用于带状态的函数对象，它们可以实现一些用函数很难做到的事情。

> [2] 他们应该知道 Boost.Function,但如果他们不知道呢？我们添加到接口上的任何东西都必须及时向用户解释清楚。
>
> [3] 现在我们知道，一开始我们就应该用这种语法。

```
class knows_the_previous_value {
  int last_value_;
public:
  void operator()(int i) {
    static bool first_time=true;
    if (first_time) {
      last_value_=i;
      std::cout <<
        "This is the first change of value, \
so I don't know the previous one.\n";
      first_time=false;
      return;
    }
    std::cout << "Previous value was " << last_value_ << '\n';
    last_value_=i;
  }
};
```

这个函数对象保存以前的值，并在值被改变时把旧值输出到 `std::cout` 。注意，当它第一次被调用时，它并不知道旧值。这个函数对象在函数中使用一个静态 `bool` 变量来检查这一点，该变量被初始化为 `true` . 由于函数中的静态变量是在函数第一次被调用时进行初始化的，所以它仅在第一次调用时被设为 `true` 。虽然也可以在普通函数中使用静态变量来提供状态，但是我们必须知道那样不太好，而且很难做到多线程安全。因此，带状态的函数对象总是优于带静态变量的普通函数。 `notifier` 类并不关心这是不是函数对象，只要符合要求就可以接受。以下更新的例子示范了它如何使用。

```
int main() {
  notifier n;
  n.add_observer(&print_new_value);
  n.add_observer(&interested_in_the_change);
  n.add_observer(knows_the_previous_value());

  n.change_value(42);
  std::cout << '\n';
  n.change_value(30);
}
```

关键一点要注意的是，我们新增的一个观察者不是函数指针，而是一个 `knows_the_previous_value` 函数对象的实例。运行这段程序的输出如下：

```
The value has been updated and is now 42
Ah, the value has changed.
This is the first change of value, so I don't know the previous one.

The value has been updated and is now 30
Ah, the value has changed.
Previous value was 42
```

在这里最大的优点不是放宽了对函数的要求(或者说，增加了对函数对象的支持)，而是我们可以使用带状态的对象，这是非常需要的。我们对 `notifier` 类所做的修改非常简单，而且用户代码不受影响。如上所示，把 Boost.Function 引入一个已有的设计中是非常容易的。

类成员函数

Boost.Function 不支持参数绑定，这在每次调用一个 `function` 就要调用同一个类实例的成员函数时是需要的。幸运的是，如果这个类实例被传递给 `function` 的话，我们就可以直接调用它的成员函数。这个 `function` 的签名必须包含类的类型以及成员函数的签名。换言之，显式传入的类实例要作为隐式的第一个参数， `this` 。这样就得到了一个在给出的对象上调用成员函数的函数对象。看一下以下这个类：

```
class some_class {
public:
  void do_stuff(int i) const {
    std::cout << "OK. Stuff is done. " << i << '\n';
  }
};
```

成员函数 `do_stuff` 要从一个 `boost::function` 实例里被调用。要做到这一点，我们需要 function 接受一个 `some_class` 实例，签名的其它部分为一个 `void` 返回以及一个 `int` 参数。对于如何把 `some_class` 实例传给 function，我们有三种选择：传值，传引用，或者传址。如何要传值，代码就应该这样写[4]

> [4] 很少会有理由来以传值的方式传递对象参数。

```
boost::function<void(some_class,int)> f;
```

注意，返回类型仍旧在最开始，后跟成员函数所在的类，最后是成员函数的参数类型。它就象传递一个 `this` 给一个函数，该函数暗地里用类实例调用一个非成员函数。要把函数 `f` 配置为成员函数 `do_stuff`, 然后调用它，我们这样写：

```
f=&some_class::do_stuff;
f(some_class(),2);
```

如果要传引用，我们要改一下函数的签名，并传递一个 `some_class` 实例。

```
boost::function<void(some_class&,int)> f;
f=&some_class::do_stuff;
some_class s;
f(s,1);
```

最后，如果要传 `some_class` 的指针[5]，我们就要这样写：

> [5] 裸指针或智能指针皆可。

```
boost::function<void(some_class*,int)> f;
f=&some_class::do_stuff;
some_class s;
f(&s,3);
```

好了，所有这些传递"虚拟 `this` "实例的方法都已经在库中提供。当然，这种技术也是有限制的：你必须显式地传递类实例；而理想上，你更愿意这个实例被绑定在函数中。乍一看，这似乎是 Boost.Function 的缺点，但有别的库可以支持参数的绑定，如 Boost.Bind 和 Boost.Lambda. 我们将在本章稍后的地方示范这些库会给 Boost.Function 带有什么好处。

带状态的函数对象

我们已经看到，由于支持了函数对象，就可以给回调函数增加状态。考虑这样一个类， `keeping_state`, 它是一个带状态的函数对象。 `keeping_state` 的实例记录一个总和，它在每次调用操作符执行时被增加。现在，将该类的一个实例用于两个 `boost::function` 实例，结果有些出人意外。

```
#include <iostream>
#include "boost/function.hpp"

class keeping_state {
  int total_;
public:
  keeping_state():total_(0) {}

  int operator()(int i) {
    total_+=i;
    return total_;
  }

  int total() const {
    return total_;
  }
};

int main() {
  keeping_state ks;
  boost::function<int(int)> f1;
  f1=ks;

  boost::function<int(int)> f2;
  f2=ks;

  std::cout << "The current total is " << f1(10) << '\n';
  std::cout << "The current total is " << f2(10) << '\n';
  std::cout << "After adding 10 two times, the total is "
    << ks.total() << '\n';
}
```

写完这段代码并接着执行它，程序员可能期望保存在 `ks` 的总和是20，但不是；事实上，总和为0。以下是这段程序的运行结果。

```
The current total is 10
The current total is 10
After adding 10 two times, the total is 0
```

原因是每一个 `function` 实例(`f1` 和 `f2`)都含有一个 `ks` 的拷贝，这两个实例得到的总和都是10，但 `ks` 没有变化。这可能是也可能不是你想要的，但是记住， `boost::function` 的缺省行为是复制它要调用的函数对象，这一点很重要。如果这导致不正确的语义，或者如果某些函数对象的复制代价太高，你就必须把函数对象包装在 `boost::reference_wrapper` 中，那样 `boost::function` 的复制就会是一个 `boost::reference_wrapper` 的拷贝，它恰好持有一个到原始函数对象的引用。你无须直接使用 `boost::reference_wrapper` ，你可以使用另两个助手函数， `ref` 和 `cref` 。 这两函数返回一个持有到某特定类型的引用或 `const` 引用的 `reference_wrapper` 。在前例中，要获得我们想要的语义，即使用同一个 `keeping_state` 实例，我们就需要把代码修改如下：

```
int main() {
  keeping_state ks;
  boost::function<int(int)> f1;
  f1=boost::ref(ks);

  boost::function<int(int)> f2;
  f2=boost::ref(ks);

  std::cout << "The current total is " << f1(10) << '\n';
  std::cout << "The current total is " << f2(10) << '\n';
  std::cout << "After adding 10 two times, the total is "
    << ks.total() << '\n';
}
```

`boost::ref` 的用途是通知 `boost::function`，我们想保存一个到函数对象的引用，而不是一个拷贝。运行这个程序有以下输出：

```
The current total is 10
The current total is 20
After adding 10 two times, the total is 20
```

这正是我们想要的结果。使用 `boost::ref` 和 `boost::cref` 的不同之处就象引用与 `const` 引用的差异，对于后者，你只能调用其中的常量成员函数。以下例子使用一个名为 `something_else` 的函数对象，它有一个 `const` 的调用操作符。

```
class something_else {
public:
  void operator()() const {
    std::cout << "This works with boost::cref\n";
  }
};
```

对于这个函数对象，我们可以使用 `boost::ref` 或 `boost::cref`.

```
something_else s;
boost::function0<void> f1;
f1=boost::ref(s);
f1();
boost::function0<void> f2;
f2=boost::cref(s);
f2();
```

如果我们改变了 `something_else` 的实现，使其函数为非 `const`，则只有 `boost::ref` 可以使用，而 `boost::cref` 将导致一个编译期错误。

```
class something_else {
public:
  void operator()() {
    std::cout <<
      "This works only with boost::ref, or copies\n";
  }
};

something_else s;
boost::function0<void> f1;
f1=boost::ref(s); // This still works
f1();
boost::function0<void> f2;
f2=boost::cref(s); // This doesn't work;
                   // the function call operator is not const
f2();
```

如果一个 `function` 包含一个被 `boost::reference_wrapper` 所包装的函数对象，那么复制构造函数与赋值操作就会复制该引用，即 `function` 的拷贝将引向原先的函数对象。

```
int main() {
  keeping_state ks;
  boost::function1<int,int> f1;  // 译注：原文为boost::function<int,int> f1，有误
  f1=boost::ref(ks);

  boost::function1<int,int> f2(f1);  // 译注：原文为boost::function<int,int> f2(f1)，有误
  boost::function1<short,short> f3;  // 译注：原文为boost::function<short,short> f3，有误
  f3=f1;

  std::cout << "The current total is " << f1(10) << '\n';
  std::cout << "The current total is " << f2(10) << '\n';
  std::cout << "The current total is " << f3(10) << '\n';
  std::cout << "After adding 10 three times, the total is "
    << ks.total() << '\n';
}
```

这等同于使用 `boost::ref` 并把函数对象 `ks` 赋给每一个 function 实例。

给回调函数增加状态，可以发挥巨大的能力，这也正是使用 Boost.Function 与使用函数对象相比具有的非常突出的优点。

与 Boost.Function 一起使用 Boost.Bind

当我们把 Boost.Function 与某个支持参数绑定的库结合起来使用时，事情变得更为有趣。Boost.Bind 为普通函数、成员函数以及成员变量提供参数绑定。这非常适合于 Boost.Function, 我们常常需要这类绑定，由于我们使用的类本身并不是函数对象。那么，我们用 Boost.Bind 把它们转变为函数对象，然后我们可以用 Boost.Function 来保存它们并稍后调用。在将图形用户界面(GUIs)与如何响应用户的操作进行分离时，几乎总是要使用某种回调方法。如果这种回调机制是基于函数指针的， 就很难避免对可以使用回调的类型的某些限制，也就增加了界面表现与业务逻辑之间的耦合风险。通过使用 Boost.Function，我们可以避免这些事情，并且当与某个支持参数绑定的库结合使用时，我们可以轻而易举地把上下文提供给调用的函数。这是本库 最常见的用途之一，把业务逻辑即从表示层分离出来。

以下例子包含一个艺术级的磁带录音机，定义如下：

```
class tape_recorder {
public:
  void play() {
    std::cout << "Since my baby left me...\n";
  }

  void stop() {
    std::cout << "OK, taking a break\n";
  }

  void forward() {
    std::cout << "whizzz\n";
  }

  void rewind() {
    std::cout << "zzzihw\n";
  }

  void record(const std::string& sound) {
    std::cout << "Recorded: " << sound << '\n';
  }
};
```

这个磁带录音机可以从一个GUI进行控制，或者也可能从一个脚本客户端进行控制，或者从别的源进行控 制，这意味着我们不想把这些函数的执行与它们的实现耦合起来。建立这种分离的一个常用的方法是，用专门的对象负责执行命令，而让客户对命令如何执行毫无所 知。这也被称为命令模式(Command pattern)，并且在它非常有用。这种模式的特定实现中的一个问题是，需要为每个命令创建单独的类。以下片断示范了它看起来是个什么样子：

```
class command_base {
public:
  virtual bool enabled() const=0;
  virtual void execute()=0;

  virtual ~command_base() {}
};

class play_command : public command_base {
  tape_recorder* p_;
public:
  play_command(tape_recorder* p):p_(p) {}

  bool enabled() const {
    return true;
  }

  void execute() {
    p_->play();
  }
};

class stop_command : public command_base {
  tape_recorder* p_;
public:
  stop_command(tape_recorder* p):p_(p) {}

  bool enabled() const {
    return true;
  }

  void execute() {
    p_->stop();
  }
};
```

这并不是一个非常吸引的方案，因为它使得代码膨胀，有许多简单的命令类，而它们只是简单地负责调用一个对象的单个成员函数。有时候，这是必需的，因为这些命令可能需要实现业务逻辑和调用函数，但通常它只是由于我们所使用的工具有所限制而已。这些命令类可以这样使用：

```
int main() {
  tape_recorder tr;

  // 使用命令模式
  command_base* pPlay=new play_command(&tr);
  command_base* pStop=new stop_command(&tr);

  // 在按下某个按钮时调用
  pPlay->execute();
  pStop->execute();

  delete pPlay;
  delete pStop;
}
```

现在，不用再创建额外的具体的命令类，如果我们实现的命令都是调用一个返回 `void` 且没有参数(先暂时忽略函数 record, 它带有一个参数)的成员函数的话，我们可以来点泛化。不用再创建一组具体的命令，我们可以在类中保存一个指向正确成员函数的指针。这是迈向正确方向[6]的一大步，就象这样：

> [6] 虽然损失了一点效率。

```
class tape_recorder_command : public command_base {
  void (tape_recorder::*func_)();
  tape_recorder* p_;
public:

  tape_recorder_command(
    tape_recorder* p,
    void (tape_recorder::*func)()) : p_(p),func_(func) {}

  bool enabled() const {
    return true;
  }

  void execute() {
    (p_->*func_)();
  }
};
```

这个命令模式的实现要好多了，因为它不需要我们再创建一组完成相同事情的独立的类。这里的不同在于我们保存了一个 `tape_recorder` 成员函数指针在 `func_` 中，它要在构造函数中提供。命令的执行部分可能并不是你要展现给你的朋友看的东西，因为成员指针操作符对于一些人来说可能还不太熟悉。但是，这可以被看为一个低层的实现细节，所以还算好。有了这个类，我们可以进行泛化处理，不再需要实现单独的命令类。

```
int main() {
  tape_recorder tr;

  // 使用改进的命令模式
  command_base* pPlay=
    new tape_recorder_command(&tr,&tape_recorder::play);
  command_base* pStop=
    new tape_recorder_command(&tr,&tape_recorder::stop);

  // 从一个GUI或一个脚本客户端进行调用
  pPlay->execute();
  pStop->execute();

  delete pPlay;
  delete pStop;
}
```

你可能还没有理解，我们已经在开始实现一个简单的 `boost::function` 版本，它已经可以做到我们想要的。不要重复发明轮子，让我们重点关注手边的工作：分离调用与实现。以下是一个全新实现的 `command` 类，它更容易编写、维护以及理解。

```
class command {
  boost::function<void()> f_;
public:
  command() {}
  command(boost::function<void()> f):f_(f) {}

  void execute() {
    if (f_) {
      f_();
    }
  }

  template <typename Func> void set_function(Func f) {
    f_=f;
  }

  bool enabled() const {
    return f_;
  }
};
```

通过使用 Boost.Function，我们可以立即从同时兼容函数和函数对象——包括由绑定器生成的函数对象——的灵活性之中获益。这个 `command` 类把函数保存在一个返回 `void` 且不接受参数的 `boost::function` 中。为了让这个类更加灵活，我们提供了在运行期修改函数对象的方法，使用一个泛型的成员函数， `set_function` .

```
template <typename Func> void set_function(Func f) {
  f_=f;
}
```

通过使用泛型方法，任何函数、函数对象，或者绑定器都兼容于我们的 `command` 类。我们也可以选择把 `boost:: function` 作为参数，并使用 `function` 的转型构造函数来达到同样的效果。这个 `command` 类非常通用，我们可以把它用于我们的 `tape_recorder` 类或者别的地方。与前面的使用一个基类与多个具体派生类(在那里我们使用指针来实现多态的行为)的方法相比，还有一个额外的优点就是，它更容易管理生存期问题，我们不再需要删除命令对象，它们可以按值传递和保存。我们在布尔上下文中使用 `function f_` 来测试命令是否可用。如果函数不包含一个目标，即一个函数或函数对象，它将返回 `false` , 这意味着我们不能调用它。这个测试在 `execute` 的实现中进行。以下是使用我们这个新类的一个例子：

```
int main() {
  tape_recorder tr;

  command play(boost::bind(&tape_recorder::play,&tr));
  command stop(boost::bind(&tape_recorder::stop,&tr));
  command forward(boost::bind(&tape_recorder::stop,&tr));
  command rewind(boost::bind(&tape_recorder::rewind,&tr));
  command record;

  // 从某些GUI控制中调用...
  if (play.enabled()) {
    play.execute();
  }

  // 从某些脚本客户端调用...
  stop.execute();

  // Some inspired songwriter has passed some lyrics
  std::string s="What a beautiful morning...";
  record.set_function(
    boost::bind(&tape_recorder::record,&tr,s));
  record.execute();
}
```

为了创建一个具体的命令，我们使用 Boost.Bind 来创建函数对象，当通过这些对象的调用操作符进行调用时，就会调用正确的 `tape_recorder` 成员函数。这些函数对象是自完备的；它们无参函数对象，即它们可以直接调用，无须传入参数，这正是 `boost::function&lt;void()&gt;` 所表示的。换言之，以下代码片断创建了一个函数对象，它在配置好的 `tape_recorder` 实例上调用成员函数 play 。

```
boost::bind(&tape_recorder::play,&tr)
```

通常，我们不能保存 `bind` 所返回的函数对象，但由于 Boost.Function 兼容于任何函数对象，所以它可以。

```
boost::function<void()> f(boost::bind(&tape_recorder::play,&tr));
```

注意，这个类也支持调用 `record`，它带有一个类型为 `const std::string&` 的参数，这是由于成员函数 `set_function`. 因为这个函数对象必须是无参的，所以我们需要绑定上下文以便 `record` 仍旧能够获得它的参数。当然，这是绑定器的工作。因而，在调用 `record` 之前，我们创建一个包含被录音的字符串的函数对象。

```
std::string s="What a beautiful morning...";
record.set_function(
  boost::bind(&tape_recorder::record,&tr,s));
```

执行这个保存在 `record` 的函数对象，将在 `tape_recorder` 实例 `tr` 上执行 `tape_recorder::record`，并传入字符串。有了 Boost.Function 和 Boost.Bind, 就可以实现解耦，让调用代码对于被调用代码一无所知。以这种方式结合使用这两个库非常有用。你已经

在这个 `command` 类中看到了，现在我们该清理一下了。由于 Boost.Function 的杰出功能，你所需的只是以下代码：

```
typedef boost::function<void()> command;
```

与 Boost.Function 一起使用 Boost.Lambda

与 Boost.Function 兼容于由 Boost.Bind 创建的函数对象一样，它也支持由 Boost.Lambda 创建的函数对象。你用 Lambda 库创建的任何函数对象都兼容于相应的 `boost::function`. 我们在前一节已经讨论了基于绑定的一些内容，使用 Boost.Lambda 的主要不同之处是它能做得更多。我们可以轻易地创建一些小的、无名的函数，并把它们保存在 `boost::function` 实例中以用于后续的调用。我们已经在前一章中讨论了 lambda 表达式，在那一章的所有例子中所创建的函数对象都可以保存在一个 `function` 实例中。 `function` 与创建函数对象的库的结合使用会非常强大。

代价的考虑

有一句谚语说，世界上没有免费的午餐，对于 Boost.Function 来说也是如此。与使用函数指针相比，使用 Boost.Function 也有一些缺点，特别是对象大小的增加。显然，一个函数指针只占用一个函数指针的空间大小(这当然了！)，而一个 `boost::function` 实 例占的空间有三倍大。如果需要大量的回调函数，这可能会成为一个问题。函数指针在调用时的效率也稍高一些，因为函数指针是被直接调用的，而 Boost.Function 可能需要使用两次函数指针的调用。最后，可能在某些需要与C库保持后向兼容的情形下，只能使用函数指针。

虽然 Boost.Function 可能存在这些缺点，但是通常它们都不是什么实际问题。额外增加的大小非常小，而且(可能存在的)额外的函数指针调用所带来的代价与真正执行目标函数所花费的时间相比通常都是非常小的。要求使用函数而不能使用 Boost.Function 的情形非常罕见。使用这个库所带来的巨大优点及灵活性显然超出这些代价。

幕后的细节

至少了解一下这个库如何工作的基础知识是非常值得的。我们来看一下保存并调用一个函数指针、一个成员 函数指针和一个函数对象这三种情形。这三种情形是不同的。要真正看到 Boost.Function 如何工作，只有看源代码——不过我们的做法有些不同，我们试着搞清楚这些不同的版本究竟在处理方法上有些什么不同。我们也有一个 不同要求的类，即当调用一个成员函数时，必须传递一个实例的指针给 `function1` (这是我们的类的名字)的构造函数。 `function1` 支持只有一个参数的函数。与 Boost.Function 相比一个较为宽松的投条件是，即使是对于成员函数，也只需要提供返回类型和参数类型。这个要求的直接结果就是，构造函数必须被传入一个类的实例用于成员函数的调用(类型可以自动推断)。

我们将要采用的方法是，创建一个泛型基类，它声明了一个虚拟的调用操作符函数；然后，从这个基类派生三个类，分别支持三种不同形式的函数调用。这些类负责所有的工作，而另一个类，`function1`，依据其构造函数的参数来决定实例化哪一个具体类。以下是调用器的基类，`invoker_base`.

```
template <typename R, typename Arg> class invoker_base {
public:
  virtual R operator()(Arg arg)=0;
};
```

接着，我们开始定义`function_ptr_invoker`，它是一个具体调用器，公有派生自`invoker_base`. 它的目的是调用普通函数。这个类也接受两个类型，即返回类型和参数类型，它们被用于构造函数，构造函数接受一个函数指针作为参数。

```
template <typename R, typename Arg> class function_ptr_invoker
  : public invoker_base<R,Arg> {
  R (*func_)(Arg);
public:
  function_ptr_invoker(R (*func)(Arg)):func_(func) {}

  R operator()(Arg arg) {
    return (func_)(arg);
  }
};
```

这个类模板可用于调用任意一个接受一个参数的普通函数。调用操作符简单地以给定的参数调用保存在`func_`中的函数。请注意(的确有些奇怪)声明一个保存函数指针的变量的那行代码。

```
R (*func_)(Arg);
```

你也可以用一个`typedef`来让它好读一些。

```
typedef R (*FunctionT)(Arg);
FunctionT func_;
```

接着，我们需要一个可以处理成员函数调用的类模板。记住，它要求在构造时给出一个类实例的指针，这一点与 Boost.Function 的做法不一样。这样可以节省我们的打字，因为是编译器而不是程序员来推导这个类。

```
template <typename R, typename Arg, typename T>
class member_ptr_invoker :
  public invoker_base<R,Arg> {
  R (T::*func_)(Arg);
  T* t_;
public:
  member_ptr_invoker(R (T::*func)(Arg),T* t)
    :func_(func),t_(t) {}

  R operator()(Arg arg) {
    return (t_->*func_)(arg);
  }
};
```

这个类模板与普通函数指针的那个版本很相似。它与前一个版本的不同在于，构造函数保存了一个成员函数指针与一个对象指针，而调用操作符则在该对象(`t_`)上调用该成员函数(`func_`)。

最后，我们需要一个兼容函数对象的版本。这是所有实现中最容易的一个，至少在我们的方法中是这样。通过使用单个模板参数，我们只表明类型 `T` 必须是一个真正的函数对象，因为我们想要调用它。说得够多了。

```
template <typename R, typename Arg, typename T>
class function_object_invoker :
  public invoker_base<R,Arg> {
  T t_;
public:
  function_object_invoker(T t):t_(t) {}

  R operator()(Arg arg) {
    return t_(arg);
  }
};
```

现在我们已经有了这些适用的积木，剩下来的就是把它们放在一起组成我们的自己的 `boost::function`，即 `function1` 类。我们想要一种办法来发现要实例化哪一个调用器。然后我们可以把它存入一个 `invoker_base` 指针。这里的窍门就是，提供一些构造函数，它们有能力去检查对于给出的参数，哪种调用器是正确的。这仅仅是重载而已，用了一点点手法，包括泛化两个构造函数。

```
template <typename R, typename Arg> class function1 {
  invoker_base<R,Arg>* invoker_;
public:
  function1(R (*func)(Arg)) :
  invoker_(new function_ptr_invoker<R,Arg>(func)) {}

  template <typename T> function1(R (T::*func)(Arg),T* p) :
    invoker_(new member_ptr_invoker<R,Arg,T>(func,p)) {}

  template <typename T> function1(T t) :
    invoker_(new function_object_invoker<R,Arg,T>(t)) {}

  R operator()(Arg arg) {
    return (*invoker_)(arg);
  }

  ~function1() {
    delete invoker_;
  }
};
```

如你所见，这里面最难的部分是正确地定义出推导系统以支持函数指针、类成员函数以及函数对象。无论使用何种设计来实现这类功能的库，这都是必须的。最后，给出一些例子来测试我们这个方案。

```
bool some_function(const std::string& s) {
  std::cout << s << " This is really neat\n";
  return true;
}

class some_class {
public:
  bool some_function(const std::string& s) {
    std::cout << s << " This is also quite nice\n";
    return true;
  }
};

class some_function_object {
public:
  bool operator()(const std::string& s) {
    std::cout << s <<
      " This should work, too, in a flexible solution\n";
    return true;
  }
};
```

我们的 `function1` 类可以接受以下所有函数。

```
int main() {
  function1<bool,const std::string&> f1(&some_function);
  f1(std::string("Hello"));

  some_class s;
  function1<bool,const std::string&>
    f2(&some_class::some_function,&s);

  f2(std::string("Hello"));

  function1<bool,const std::string&>
    f3(boost::bind(&some_class::some_function,&s,_1));

  f3(std::string("Hello"));

  some_function_object fso;
  function1<bool,const std::string&>
    f4(fso);
  f4(std::string("Hello"));
}
```

它也可以使用象 Boost.Bind 和 Boost.Lambda 这样的 binder 库所返回的函数对象。我们的类与 Boost.Function 中的类相比要简单多了，但是也已经足以看出创建和使用这样一个库的问题以及相关解决方法。知道一点关于一个库是如何实现的事情，对于有效使用这个库是非常有用的。

Function 总结

在以下情形时使用 Function 库

- 你需要保存一个回调函数或函数对象
- 你想要从实现中解耦函数调用，例如在GUI和实现间的解耦
- 你想要保存由 binder 库创建的函数对象，用于后续的调用或多次调用

Boost.Function 是对标准库的功能的重要补充。在回调机制中使用函数指针这样的著名技术被扩充至可以使用任何行为类似于函数的东西，包括由 binder 库创建的函数对象。通过使用 Boost.Function, 可以很容易地为回调增加状态，也可以把已有的类和成员函数进行适配后用作回调函数。

与使用函数指针相比，使用 Boost.Function 有几个优点：通过兼容的函数对象而不是真实的签名放松了对签名的要求；可以使用绑定器，如 Boost.Bind 和 Boost.Lambda；可以在调用函数之前检测函数是否为空，即是否存在目标函数；可以使用带状态的对象而不仅限于无状态函数。这些优点表明了使用 Boost.Function 替代C风格的回调可以解决这类普遍存在的问题。使用 Boost.Function 比使用函数指针要多付出一点点代价，只有这一点小代价是被禁止时，才应该考虑使用函数指针技术。

Boost.Function 由 Douglas Gregor 创建。它是一个拥有巨大威力的库，具有成熟的设计与实现，可以为用户提供额外的价值。

Library 12. Signals

- Signals 库如何改进你的程序？
- Signals 如何适用于标准库？
- Signals
- 用法
- Signals 总结

Signals 库如何改进你的程序？

- 函数和函数对象的灵活多点回调
- 健壮的触发器及事件处理的机制
- 兼容于函数对象工厂，如 Boost.Bind 和 Boost.Lambda

Boost.Signals 库具体化了信号(signals)和 插槽(slots)，信号指的是某种可被"抛出"的 东西，而插槽是接收该信号的连接者。这是一种著名的设计模式，它还有另外一些名字 *Observer, signals/slots, publisher/subscriber, events* (和 event targets)，这些名字指的都是同一个东西，指的是一些信 息源和某些对这些信息的变化感兴趣的实例之间的一对多关系。这种设计模式的使用有多种情况；最常见的是在GUI代 码中，用于使特定动作(例如，用户单击了一个按钮)与其它动作(按 钮改变它的外观，执行某个商业逻辑)松散连接。信号与插槽 在许多场合都很有用，解耦动作的触发条件(信号)和处理它的代码(一 个或多个插槽)。它可用于动态改变处理代码的行为，允许同 一信号对应多个处理，或者通过一个信号及插槽的类型间的抽象关联来降低类型依赖性。通过使用 Boost.Signals, 可 以创建一些信号来接受任意给定的函数特征的插槽，即插槽接受任意类型的参数。这种方法使得该库非常灵活；它适用于任意范围的信号需求。通过对信号源和处理 者的解耦，系统无论在物理和逻辑依赖上都变得更为健壮。它可以让信号类型对插槽类型完全一无所知，反之亦然。这对于更高层次的可复用性是很有必要的，它有 助于打破依赖性的循环。因此，一个信号与插槽的库不仅仅关系到面向对象的回调，它也关系到使用它的整个系统的健壮性。

Signals 如何适用于标准库？

C++标 准库中没有用于回调的工具，而这种工具显然是需要的。Boost.Signals 使 用了与标准库相同的态度进行设计，它是标准库工具箱的一个杰出的扩展。

Signals

头文件: `"boost/signals.hpp"`

通过单个头文件包含了整个库。

```
"boost/signals/signal.hpp"
```

包含了 `signal` 的定义。

```
"boost/signals/slot.hpp"
```

包含了 `slot` 类的定义。

```
"boost/signals/connection.hpp"
```

包含了类 `connection` 和 `scoped_connection` 的定义。

要使用这个库，可以包含头文件 `"boost/signals.hpp"`，这样可以确保整个库可用，或者可以按照你的需要包含单独的头文件。Boost.Signals 库的核心部分定义在名字空间 `boost` 中，高级特性则定义在 `boost::signals` 中。

以下是 `signal` 的部分内容，其后将对其中最主要的成员函数进行了简要的介绍。如果你需要完整的参考，请见 Signals 的在线文档。

```
namespace boost {

  template<typename Signature,
  // Function type R(T1, T2, ..., TN)
    typename Combiner = last_value<R>,
    typename Group = int,
    typename GroupCompare = std::less<Group>,
    typename SlotFunction = function<Signature> >
  class signal : public signals::trackable,
                 private noncopyable {
  public:
    signal(const Combiner&=Combiner(),
           const GroupCompare&=GroupCompare());

    ~signal();

    signals::connection connect(const slot_type&);
    signals::connection connect(
      const Group&,
      const slot_type&);

    void disconnect(const Group&);

    std::size_t num_slots() const;

    result_type operator()
      (T1, T2, ..., TN);
  };
}
```

类型

我们先来看看 `signal` 的模板参数。除了第一个参数，其它参数都有相应的缺省值，这有助于理解这些参数的基本意思。第一个模板参数是被调用的函数的签名。在这种 `signal`s 的情况下， `signal` 本身就是被调用的实体。声明这个签名时，使用与普通函数签名相同的语法[1]。例如，一个返回 `double` 且接受一个类型 `int` 的参数的函数签名应该象这样：

> [1] 细心的读者可以已经注意到 `boost::function` 也是这样用的。

```
signal<double(int)>
```

`Combiner` 参数表示一个函数对象，它负责逐个对该 `signal` 所有已连接的插槽(slot)进行调用。它同时也决定如何将组合这些调用返回的结果。缺省的类型是 `last_value`, 它只是简单地返回最后一个插槽的调用结果。

`Groups` 参数是用于组合所有连接到 `signal` 的插槽的一种类型。通过连接到不同的插槽组，你可以预设调用插槽的顺序，同时也可以断开插槽组。

`GroupCompare` 参数决定了如何排序 `Groups`，缺省值为 `std::less&lt;Group&gt;`, 通常它都是正确的。如果 `Groups` 使用了定制的类型，就有可能需要其它的排序方法。

最后， `SlotFunction` 参数表示插槽函数的类型，缺省值为 `boost::function`. 我想不出有什么理由改变这个缺省值。这个模板参数用于定义插槽的类型，定义的方法是一个公有的 `typedef slot&lt;SlotFunction&gt; slot_type`.

成员函数

```
signal(const Combiner&=Combiner(),
  const GroupCompare&=GroupCompare());
```

在构造一个 `signal` 时，可以传入一个 `Combiner`，它是一个负责在信号到达时调用相应插槽并对返回值进行处理的对象。

```
~signal();
```

析构函数在析构时断开所有已连接的插槽。

```
signals::connection connect(const slot_type& s);
```

`connect` 函数把插槽 `s` 连接到 `signal`. 函数指针、函数对象、 `bind` 表达式或者 lambda 表达式都可以用作插槽。 `connect` 返回一个 `signals::connection`, 它是代表被创建的连接的句柄。通过使用这个句柄，插槽可以从 `signal` 断开，或者你也可以测试该插槽是否还有连接。

```
signals::connection connect(const Group& g, const slot_type& s);
```

这个 `connect` 的重载版本与前一个作用相似，但是它还把插槽 `s` 连接到组 `g`. 把一个插槽连接到一个组意味着当一个 `signal` 产生时，属于较前面的组的插槽会先被调用(即按组的顺序来调用， `signal` 模板的 `GroupCompare` 参数定义了组的顺序)，而且属于组的所有插槽会在不属于组的插槽之前被调用(可能只有部分插槽是在组中的)。

```
void disconnect(const Group& g);
```

断开所有属于组 `g` 的已连接插槽。

```
std::size_t num_slots() const;
```

返回当前连接到 `signal` 的插槽数量。要测试插槽是否为空，应该调用函数 `empty`，而不要调用 `num_slots` 并测试其返回是否为0，因为 `empty` 的效率更高。

```
result_type operator()(T1, T2, ..., TN);
```

`signal`s 使用调用操作符来调用。当信号产生时，必须传递适当的参数给调用操作符，必须符合 `signal` 的签名(即声明 `signal` 类型时的第一个模板参数)。参数的类型必须可以隐式转换为信号所需的类型，只有这样调用才可以成功。

Boost.Signals 中还有其它的类型，我们将在本章剩余部分讨论它们。我们还将讨论 `signal` 类中有用的 `typedef` s。

用法

当你面对需要用多段代码来处理一个事件的情况时，典型的解决方案有：用函数指针进行回调，或者直接对产生事件的子系统与处理事件的子系统之间的依赖性进行编码。这种设计常常会导致循环的依赖性。通过使用 Boost.Signals, 你将获得灵活性和解耦。要开始使用这个库，首先要包含头文件 `"boost/signals.hpp"` .[2]

<small class="calibre23"></small><small class="calibre23"> [2]</small> <small class="calibre23"></small><small class="calibre23"></small><small class="calibre23">Boost.Signals</small> 库和 <small class="calibre40"></small><small class="calibre40">Boost.Regex</small> 库是本书所讨论的库中仅有的需要编译和链接才能使用的库。编译的过程很简单，在线文档中已有详尽的描述，这里我不再复述。

以下例子示范了 `signal` s 和插槽(slots)的基本特性，包括如何连接它们以及如何产生一个 `signal` . 注意，插槽指的是由你提供的一个兼容于 `signal` 的函数签名的函数或函数对象。在以下代码中，我们既创建了一个普通函数， `my_first_slot` , 也创建了一个函数对象， `my_second_slot` ; 它们两个都将连接到我们创建的一个 `signal` 上。

```
#include <iostream>
#include "boost/signals.hpp"

void my_first_slot() {
  std::cout << "void my_first_slot()\n";
}

class my_second_slot {
public:
  void operator()() const {
    std::cout <<
      "void my_second_slot::operator()() const\n";
  }
};

int main() {
  boost::signal<void ()> sig;

  sig.connect(&my_first_slot);
  sig.connect(my_second_slot());

  std::cout << "Emitting a signal...\n";
  sig();
}
```

我们首先声明一个 `signal` , 它所需的插槽为返回 `void` 且不带参数。然后，我们把两个兼容的插槽类型连接到该 `signal` . 对于第一个插槽，我们用普通函数 `my_first_slot` 的地址调用 `connect` 。对于另一个插槽，我们缺省构造一个函数对象 `my_second_slot` 的实例并把它传给 `connect` 。这些连接意味着当我们产生一个 `signal` (通过调用 `sig`)时，这两个插槽将被立即调用。

```
sig();
```

运行这个程序，输出信息如下：

```
Emitting a signal...
void my_first_slot()
void my_second_slot::operator()() const
```

但是，后两行的顺序不一定是这样的，因为属于同一个组的插槽会以不确定的顺序执行。没有办法确定哪一个插槽会先被调用。如果插槽的调用顺序事关紧要，你就必须把它们放入不同的组。

插槽分组

有时候，某些插槽需要在其它插槽之前调用，例如某些插槽会产生一些副作用而别的插槽需要依赖于这些副作用。分组就是支持这种需求的方法。`signal` 有一个模板参数，名为 `Group`，其缺省值为 `int`. Groups 缺省以 `std::less&lt;Group&gt;` 为排序标准，对于 `int` 就是 `operator&lt;` 。换句话说，属于 group 0 的插槽会在 group 1 的插槽之前调用，等等。但是请注意，同一个组中的插槽的调用顺序是不确定的。要严格控制所有插槽的调用顺序，唯一的办法就是把每个插槽都安排到各自的组中。

把插槽指定到一个组的方法是，传递一个 `Group` 给 `signal::connect`. 一个已连接插槽不能改变其所属的组；要改变一个插槽所属的组，必须先断开它的连接，然后重新把它连接到 `signal` 上并同时指定新组。

作为例子，我们考虑两个插槽，它们带一个类型为 `int&` 的参数；第一个插槽将参数加倍，第二个插槽则把当前值加3。我们要求正确的语义是，先把该值加倍，然后再加3。如果不指定顺序，我们就不能确保按该语义执行。以下方法只能在某些系统的某些时候正确执行(可能是周一或周三而且月圆的时候)。

```
#include <iostream>
#include "boost/signals.hpp"

class double_slot {
public:
  void operator()(int& i) const {
    i*=2;
  }
};

class plus_slot {
public:
  void operator()(int& i) const {
    i+=3;
  }
};

int main() {
  boost::signal<void (int&)> sig;
  sig.connect(double_slot());
  sig.connect(plus_slot());

  int result=12;
  sig(result);
  std::cout << "The result is: " << result << '\n';
}
```

运行这段程序，可能产生以下输出：

```
The result is: 30
```

或者产生以下输出：

```
The result is: 27
```

不使用分组的方法就无法保证正确的行为。我们需要确保 `double_slot` 总是在 `plus_slot` 之前被调用。这就要求我们要指定 `double_slot` 属于一个顺序在 `plus_slot` 所属的组之前的组，即：

```
sig.connect(0,double_slot());
sig.connect(1,plus_slot());
```

这样可以确保得到我们想要的(即 27)。再次提醒，对于同一个组中的插槽，它们被调用的顺序是不确定的。只要你需要插槽以特定的顺序来执行，就必须确保它们使用不同的组。

`Groups` 的类型是类 `signal` 的一个模板参数，所以它可以使用别的类型，如 `std::string` 。

```
#include <iostream>
#include <string>
#include "boost/signals.hpp"

class some_slot {
  std::string s_;
public:
  some_slot(const std::string& s) : s_(s) {}
  void operator()() const {
    std::cout << s_ << '\n';
  }
};

int main() {
  boost::signal<void (),
    boost::last_value<void>,std::string> sig;

  some_slot s1("I must be called first, you see!");
  some_slot s2("I don't care when you call me, not at all. \
It'll be after those belonging to groups, anyway.");
  some_slot s3("I'd like to be called second, please.");

  sig.connect(s2);
  sig.connect("Last group",s3);
  sig.connect("First group",s1);

  sig();
}
```

首先我们定义一个插槽类型，它在执行时输出一个 `std::string` 到 `std::cout` 。然后，我们声明 `signal` . 因为 `Groups` 参数是在 `Combiner` 类型之后的，所以我们必须同时指定 `Combiner` (我们只是按缺省值来声明)。我们把 `Groups` 类型设为 `std::string` 。

```
boost::signal<void (),boost::last_value<void>,std::string> sig;
```

对于剩下的模板参数，我们接受缺省值就可以了。在连接到插槽 `s1` , `s2` , 和 `s3` 时，所创建的组是以字母顺序排序的(因为这是 `std::less<std::string>` 的行为)，因此 `"First group"` 先于 `"Last group"` . 注意，由于字符串常量可以隐式转换为 `std::string` , 所以我们可以把它们直接传递给 `signal` 的 `connect` 函数。运行该程序可以告诉我们正确的结果。

```
I must be called first, you see!
I'd like to be called second, please.
I don't care when you call me, not at all.
It'll be after those belonging to groups, anyway.
```

我们也可以在声明 `signal` 类型时选择别的排序方法，例如 `std::greater` .

```
boost::signal<void (),boost::last_value<void>,
 std::string,std::greater<std::string> > sig;
```

如果我们把它用于前面的例子，输出将变为：

```
I'd like to be called second, please.
I must be called first, you see!
I don't care when you call me, not at all.
It'll be after those belonging to groups, anyway.
```

当然，在这个例子中，`std::greater` 产生的顺序导致了错误的输出，但这是另一回事。分组非常有用，绝对必要，但是给组赋以正确的值并不总是那么简单的事，因为被连接的插槽并不需要在代码的同 一个地方执行。弄清楚某个插槽所应该使用什么组号可能是个问题。有时，这个问题可以用规定来解决，即在代码中增加注释，确保每个人都能看到这些注释，但是 这也只能在代码中不是很多地方要进行组号的赋值以及程序员不偷懒时有用。换句话说，这种方法也不一定管用。所以，你需要一个集中的产生组号的地方，它可以 依据某个给定的值为每个插槽产生唯一的组号，或者如果相关的插槽相互了解，那么也可以由插槽提供它们自己的组号。

现在你已经知道如何解决按顺序调用插槽的问题了，让我们来看看如何让你的 `signal`s 使用不同的签名。你常常需要传递额外的信息给你系统中的重要事件。

带参数的 Signals

通常会有一些额外的数据要传递给 `signal`. 例如，想象一个温度保护器，它报告温度的急剧变化。仅仅知道保护器发现了问题是不够的；插槽可能需要知道当前的温度。虽然保护器(一个 `signal`)和插槽都可以从一个公用的传感器去获取温度值，但是最简单的方式还是让保护器在调用插槽时把当前温度传递给插槽。还有一个例子，想象有多个插槽连接到多个 `signal` 上：插槽很可能需要知道是哪一个 `signal` 调用了它。有很多用例都需要从 `signal` 传递一些信息给插槽。插槽接受的参数是 `signal` 声明中的一部分。 `signal` 类模板的第一个参数就是调用 `signal` 的函数签名，而且这个签名也用于 `signal` 调用那些被连接的插槽。如果我们想这个参数可以修改，我们就要确保它是通过非 `const` 引用或指针来进行传递的，否则我们就可以通过值或 `const` 引用来传递它。注意，这个原始参数除了是可修改或不可修改这么明显的差异之外，对于 `signal` 本身以及插槽可以接受的参数类型还有一些隐喻，如果 `signal` 接受一个传值或传 `const` 引用的参数，那么所有可以隐式转换为该参数类型的类型都可以用于产生一个 `signal`. 对于插槽也一样，如果插槽是通过传值或传 `const` 引用来接受参数的话，这就意味着允许从 `signal` 的真正参数类型隐式转换到这个类型。我们后面将讨论如果在处理信号时正确地传递参数，届时我们将看到更多关于这一点的详细讨论。

想象一个自动停车场监视器，一旦有车进入或离开停车场，监视器将收到一个通知。它需要知道一些关于这 辆车的唯一信息，例如车的登记号码，这样它才可以跟踪每辆车的进入和离开。这个监视器有一个它自己的 `signal`，能够在有人试图进行欺骗时触发警报。这样就需要一些警卫监听这个 `signal`, 我们用一个名为 `security_guard` 来对它们进行建模。最后，我们再增加一个 `gate` 类，它包含一个 `signal` 用于在一辆车进入或离开停车场时产生。(`parking_lot_guard` 显然需要知道这一点)。我们先来看看这个 `parking_lot_guard` 的声明。

```
class parking_lot_guard {
  typedef
    boost::signal<void (const std::string&)> alarm_type;
  typedef alarm_type::slot_type slot_type;

  boost::shared_ptr<alarm_type> alarm_;

  typedef std::vector<std::string> cars;
  typedef cars::iterator iterator;

  boost::shared_ptr<cars> cars_;
public:

  parking_lot_guard();
  boost::signals::connection
    connect_to_alarm(const slot_type& a);
  void operator()(bool is_entering,const std::string& car_id);

private:
  void enter(const std::string& car_id);
  void leave(const std::string& car_id);
};
```

这里有三个特别重要的地方要认真看一下；第一个是警报，即一个返回 `void` 且接受一个 `std::string` (它用于标识一辆车)的 `boost::signal` 。这个 `signal` 的声明值得再好好看一次。

```
boost::signal<void (const std::string&)>
```

它就象是一个函数的声明，只是没有了函数名。如果有怀疑，请记住除此以外没有别的东西了！你可以从外部使用成员函数 `connect_to_alarm` 连接这个 `signal` 。(我们将看到在实现这个类时，如何以及为何我们要发出警报)。下一个要留意的地方是，这个警报以及容纳车辆标识的容器(一个容纳 `std::string` s 的 `std::vector`)两者均保存于 `boost::shared_ptr` 中。这样做的原因是，尽管我们只是打算声明一个 `parking_lot_guard` 实例，但是也可能变成多份拷贝；因为这个监视器类稍后还会连接到其它的 `signal` 上，这样就会创建多份拷贝(Boost.Signals 会复制插槽，所以需要正确地管理生存期)；而我们希望所有的数据都可用，因此我们就要共享它。虽然我们可以避免拷贝，例如通过使用指针或者把插槽的行为外 部化，但是这样做可以发现一些容易掉进去的陷阱。最后还要留意的是，我们声明了一个调用操作符，其原因是我们将要在 `gate` 类(待会定义)中把 `parking_lot_guard` 连接到一个 `signal` in the class .

现在让我们把注意力放到 `security_guard` 类。

```
class security_guard {
  std::string name_;
public:
  security_guard (const char* name);

  void do_whatever_it_takes_to_stop_that_car() const;
  void nah_dont_bother() const;

  void operator()(const std::string& car_id) const;
};
```

`security_guard`s 并不需要做太多事情。这个类有一个调用操作符，用作来自于 `parking_lot_guard` 的警报的一个插槽，另外还有两个函数：一个用于停住引发警报的车辆，另一个不做任何事。下面带来我们的 `gate` 类，它用于在有车辆到达停车场以及车辆离开时进行检查。

```
class gate {
  typedef
    boost::signal<void (bool,const std::string&)> signal_type;
  typedef signal_type::slot_type slot_type;

  signal_type enter_or_leave_;
public:
  boost::signals::connection
    connect_to_gate(const slot_type& s);
  void enter(const std::string& car_id);
  void leave(const std::string& car_id);
};
```

你将留意到，`gate` 类包含一个 `signal`，它在有车辆进入或离开停车场时被触发。有一个公用成员函数(`connect_to_gate`)用于连接这个 `signal`，另两个成员函数(`enter` 和 `leave`)用于在车辆进入或离开时被调用。

现在是时候来实现它们了。让我们从 `gate` 类开始。

```
class gate {
  typedef
    boost::signal<void (bool,const std::string&)> signal_type;
  typedef signal_type::slot_type slot_type;

  signal_type enter_or_leave_;
public:
  boost::signals::connection
    connect_to_gate(const slot_type& s) {
    return enter_or_leave_.connect(s);
  }

  void enter(const std::string& car_id) {
    enter_or_leave_(true,car_id);
  }

  void leave(const std::string& car_id) {
    enter_or_leave_(false,car_id);
  }
};
```

这个实现很简单。多数工作都前转到其它对象。函数 `connect_to_gate` 简单地把调用转为对 `signal enter_or_leave_` 的 `connect` 的调用。函数 `enter` 产生 `signal`，传入一个 `true` (代表有车辆进入)和车辆的标识。`leave` 完成同样的工作，但是传入的是 `false`，代表有车辆离开。简单的类做简单的事。`security_guard` 类也不太复杂。

```
class security_guard {
  std::string name_;
public:
  security_guard (const char* name) : name_(name) {}

  void do_whatever_it_takes_to_stop_that_car() const {
    std::cout <<
      "Stop in the name of...eh..." << name_ << '\n';
  }

  void nah_dont_bother() const {
    std::cout << name_ <<
      " says: Man, that coffee tastes f i n e fine!\n";
  }

  void operator()(const std::string& car_id) const {
    if (car_id.size() && car_id[0]=='N')
      do_whatever_it_takes_to_stop_that_car();
    else
      nah_dont_bother();
  }
};
```

`security_guard`s 知道它们自己的名字，并且可以决定在警报发出时是否要做些事情(如果 `car_id` 以字母 N 打头，它们就会有所动作)。调用操作符就是被调用的插槽函数，`security_guard` 对象是一个函数对象，并且符合 `parking_lot_guard` 的 `alarm_type` 信号的要求。`parking_lot_guard` 稍微复杂一些，但也不是很复杂。

```
class parking_lot_guard {
  typedef
    boost::signal<void (const std::string&)> alarm_type;
  typedef alarm_type::slot_type slot_type;

  boost::shared_ptr<alarm_type> alarm_;

  typedef std::vector<std::string> cars;
  typedef cars::iterator iterator;

  boost::shared_ptr<cars> cars_;
public:

  parking_lot_guard()
    : alarm_(new alarm_type), cars_(new cars) {}

  boost::signals::connection
    connect_to_alarm(const slot_type& a) {
    return alarm_->connect(a);
  }

  void operator()
    (bool is_entering,const std::string& car_id) {
    if (is_entering)
      enter(car_id);
    else
      leave(car_id);
  }

private:
  void enter(const std::string& car_id) {
    std::cout <<
      "parking_lot_guard::enter(" << car_id << ")\n";

    // 如果车辆已经在这，就触发警报
    if (std::binary_search(cars_->begin(),cars_->end(),car_id))
      (*alarm_)(car_id);
    else // Insert the car_id
      cars_->insert(
        std::lower_bound(
          cars_->begin(),
          cars_->end(),car_id),car_id);
  }

  void leave(const std::string& car_id) {
    std::cout <<
      "parking_lot_guard::leave(" << car_id << ")\n";

    // 如果是未登记的车辆，就触发警报
    std::pair<iterator,iterator> p=
      std::equal_range(cars_->begin(),cars_->end(),car_id);
    if (p.first==cars_->end() || *(p.first)!=car_id)
      (*alarm_)(car_id);
    else
      cars_->erase(p.first);
  }
};
```

就是这样了！(当然，我们还没有把插槽连接到 `signal` 上，还要做一些事情。但是这些类对于所要做的事情而言还是非常地简单的)。为了让警报和车辆标识的 `shared_ptr` 有正确的行为，我们实现了缺省构造函数，在其中适当地分配了 `signal` 和 `vector` 。隐式创建的复制构造函数、析构函数以及赋值操作符都可以正确工作(这要归功于智能指针)。函数 `connect_to_alarm` 把调用转到所含的 `signal` 的 `connect` . 调用操作符则检查其布尔参数的值来看是否有车辆进入或离开，并且调用相应的函数 `enter` 或 `leave` . 在函数 `enter` 中，

首先做的是在车辆标识的 `vector` 中进行查找。如果找到该标识则说明有问题；可能有人偷了车号牌。查找采用的是算法 `binary_search`,[3] 它要求容器是有序的(我们必须要确保它总是有序的)。如果我们发现标识已存在，就立即触发警报，即调用 `signal` 。

<small class="calibre23"></small><small class="calibre23"> [3] </small><small class="calibre40"></small><small class="calibre40">`binary_search` </small> 的复杂度为 <small class="calibre23"></small><small class="calibre23"></small><small class="calibre23">`O(logN)` .</small>

```
(*alarm_)(car_id);
```

首先我们需要解引用 `alarm_` ，因为 `alarm_` 是一个 `boost::shared_ptr`,而在调用它时，我们传给它一个表示车辆标识的参数。如果我们没有找到该标识，则一切正常， 我们就把这个车辆标识插入到 `cars_` 的正确位置中。记住我们必须保证容器随时有序，最好的办法就是把元素插入到一个不会影响顺序的位置上。算法 `lower_bound` 可以给我们指出这个位置(该算法同样要求有序序列)。最后一个是函数 `leave`,它在有车辆离开停车场时被调用。 `leave` 先确认车辆的标识是否已登记在我们的容器中。这是通过调用算法 `equal_range` 来实现的，该算法返回一对迭代器，表示了一个元素可以插入且不影响有序性的范围。这意味着我们必须解引用这个返回的迭代器并确认它的值是否等于我们要查找的那个。如果我们没有找到，我们就要再一次触发警报，而如果我们找到了，就只需要简单地把它从 `vector` 中删掉。你也许留意到我们没有给出停车者交费的代码；这种有害的代码超出了本书的范围。

我们的停车场所需的各个参与者都已经定义好了，我们必须连接这些 `signal`s 和这些插槽，否则不会发生任何事情！ `gate` 类不知道任何关于 `parking_lot_guard` 类的东西，同样后者也不知道任何关于 `security_guard` 类的东西。这就是本库的一个特性：产生事件的类型不需要对接收事件的类型有任何了解。回到这个例子上，我们来看看是否可以让这个停车场运作起来。

```
int main() {
  // 创建一些警卫
  std::vector<security_guard> security_guards;
  security_guards.push_back("Bill");
  security_guards.push_back("Bob");
  security_guards.push_back("Bull");
  // 创建两个门
  gate gate1;
  gate gate2;

  // 创建自动监视器
  parking_lot_guard plg;

  // 把自动监视器连接到门上
  gate1.connect_to_gate(plg);
  gate2.connect_to_gate(plg);

  // 把警卫连接到自动监视器上
  for (unsigned int i=0;i<security_guards.size();++i) {
    plg.connect_to_alarm(security_guards[i]);
  }

  std::cout << "A couple of cars enter...\n";
  gate1.enter("SLN 123");
  gate2.enter("RFD 444");
  gate2.enter("IUY 897");

  std::cout << "\nA couple of cars leave...\n";
  gate1.leave("IUY 897");
  gate1.leave("SLN 123");

  std::cout << "\nSomeone is entering twice - \
or is it a stolen license plate?\n";
  gate1.enter("RFD 444");
}
```

这就是你要的，一个具有完整功能的停车场。我们创建了三个 `security_guard` s, 两个 `gate` s, 和一个 `parking_lot_guard` . 它们相互之间一无所知，但我们还是要通过正确的架构把它们联系起来，停车场中发生的重要事件才得以相互传递。这意味着要把 `parking_lot_guard` 连接到两个 `gate` s 上。

```
  gate1.connect_to_gate(plg);
  gate2.connect_to_gate(plg);
```

这样就确保了无论何时 `gate` 实例中产生了 `signal enter_or_leave_` 信号， `parking_lot_guard` 都可以收到这个事件通知。接着，我们再将 `security_guard` s 连接到 `parking_lot_guard` 中的警报 `signal` 上。

```
  plg.connect_to_alarm(security_guards[i]);
```

我们已经设法将这些类型相互之间进行了解耦，它们还是得到了执行它们的职责所需的适量的信息。在前面的代码中，我们让少量的车辆进入和离开，来测试这个停车场。这个真实世界的模拟显示了我们已经让各个模块按要求相互通信了。

```
A couple of cars enter...
parking_lot_guard::enter(SLN 123)
parking_lot_guard::enter(RFD 444)
parking_lot_guard::enter(IUY 897)

A couple of cars leave...
parking_lot_guard::leave(IUY 897)
parking_lot_guard::leave(SLN 123)

Someone is entering twice - or is it a stolen license plate?
parking_lot_guard::enter(RFD 444)
Bill says: Man, that coffee tastes f.i.n.e fine!
Bob says: Man, that coffee tastes f.i.n.e fine!
Bull says: Man, that coffee tastes f.i.n.e fine!
```

可惜的是，拿着车牌 RFD 444 的骗子跑掉了，但是你能做的就是这些。

关于 `signal`s 的参数已经讨论了很长一段篇幅，事实上我们更多是在讨论 Signals 的基本用法，即对产生 `signal`s 的类型和监听它的插槽进行解耦。记住，任何类型的参数都可以传递，而 `signal` 类型的声明决定了插槽函数的签名，该声明看起来就象一个不带函数名的函数声明。我们根本没有提到返回类型，虽然它也是签名的一部分。这个疏忽的原因是返回类型可以有多种不同的处理方法，接下来我们将看到为什么会这样以及如何去做。

对结果进行组合

如果一个 `signal` 的签名以及它的插槽具有非 `void` 的返回类型，显然对于插槽的返回值会有事发生，事实上，那个对 `signal` 的调用将产生某种结果。但是结果是什么呢？`signal` 类模板有一个参数名为 Combiner, 它就是负责组合并返回结果的一个类型。缺省的 Combiner 是 `boost::last_value`, 它是一个类，只负责简单地返回所调用的最后一个插槽的返回值。那么，究竟是哪一个插槽呢？我们真的不知道，因为调用同一个组内的插槽的顺序是不确定的[4]。我们从一个小例子来示范一下缺省的 Combiner 。

> [4] 所以，假设最后一个组中只有一个插槽，我们就可以知道。

```
#include <iostream>
#include "boost/signals.hpp"

bool always_return_true() {
  return true;
}

bool always_return_false() {
  return false;
}

int main() {
  boost::signal<bool ()> sig;

  sig.connect(&always_return_true);
  sig.connect(&always_return_false);

  std::cout << std::boolalpha << "True or false? " << sig();
}
```

有两个插槽，`always_return_true` 和 `always_return_false`，被连接到 `signal sig`，每个都返回一个 `bool` 且不带参数。调用 `sig` 的结果被输出到 `cout`. 它会是 `true` 还是 `false`？不经测试的话，我们无法知道(我试了一上，结果是 `false`)。在实践中，你要么不关心调用 `signal` 所返回的值，要么你就要创建你自己的 Combiner 来提供有意义的、客户化的行为。例如，可能是对所有插槽返回的结果进行处理后得到调用 `signal` 的最终结果。另一种情况，也可能是在某一个插槽返回 `false` 后就不再调用其它的插槽。一个定制的 Combiner 可以做到这些，甚至更多。这是因为 Combiner 可以对插槽进行逐个调用，并根据返回值来决定做什么。

想象一个初始化序列，其中任何失败都将中止整个序列。插槽可以根据它们被调用的次序来指定到组中。没有一个定制的 Combiner 的话，它看起来就象这样：

```
#include <iostream>
#include "boost/signals.hpp"

bool step0() {
  std::cout << "step0 is ok\n";
  return true;
}

bool step1() {
  std::cout << "step1 is not ok. This won't do at all!\n";
  return false;
}

bool step2() {
  std::cout << "step2 is ok\n";
  return true;
}

int main() {
  boost::signal<bool ()> sig;
  sig.connect(0,&step0);
  sig.connect(1,&step1);
  sig.connect(2,&step2);

  bool ok=sig();

  if (ok)
    std::cout << "All system tests clear\n";
  else
    std::cout << "At least one test failed. Aborting.\n";
}
```

以上这段代码没有办法让代码知道其中有一个测试是失败的。你也记得，缺省的 combiner 是 `boost::last_value`，它只是简单地返回最后一个插槽的返回值，即调用 `step2` 的返回值。运行这个例子会给出一个令人失望的输出：

```
step0 is ok
step1 is not ok. This won't do at all!
step2 is ok
All system tests clear
```

显然这不是正确的结果。我们需要一个 Combiner，它应该在某个插槽返回 `false` 时中止处理，并把结果传回给 `signal`. 一个 Combiner 就是一个具有某些额外要求的函数对象。它必须有一个名为 `result_type` 的 `typedef`，用于指定其调用操作符的返回类型。此外，调用操作符必须以它被调用的迭代器类型泛化。我们这里需要的 Combiner 非常简单，因此它恰好是一个好的例子。

```
class stop_on_failure {
public:
  typedef bool result_type;

  template <typename InputIterator>
  bool operator()(InputIterator begin,InputIterator end) const
  {
    while (begin!=end) {
      if (!*begin)
        return false;
      ++begin;
    }
    return true;
  }
};
```

注意，公有的 `typedef result_type`, 它定义为 `bool`. `result_type` 的类型无需与插槽的返回类型相关。(在声明 `signal` 时，你指定了插槽的签名以及 `signal` 的调用操作符的参数。但是，Combiner 的返回类型决定了 `signal` 的调用操作符的返回类型。缺省情况下，它与插槽的返回类型相同，但这不是必须的)。 `stop_on_failure` 的调用操作符以一个插槽迭代器类型所泛化，它对插槽进行逐个迭代并调用；直到我们遇到一个错误为止。对于 `stop_on_failure`, 我们不想在遇到错误的返回值后再继续调用插槽，因此我们对于每次调用都检查其返回值。如果返回值为 `false`, 该函数说立即返回，否则它继续调用下一个插槽。要使用这个 `stop_on_failure`, 我们只需在声明 `signal` 类型时指出即可：

```
boost::signal<bool (),stop_on_failure> sig;
```

如果我们在前面的例子中使用它，则输出的结果就会符合我们的要求了。

```
step0 is ok
step1 is not ok. This won't do at all!
At least one test failed. Aborting.
```

Combiner 的另一个常用类型是，返回所有被调用插槽的返回值中的最大或最小值。还有其它很多有趣的 Combiners，包括：将所有结果保存在一个容器中。本库的(优秀的)在线文档就有这么一个 Combiner 的例子，你应该去读一下！你并不是每天都需要编写自己的 Combiner 类，但偶尔在为一个复杂的问题给出一个漂亮的解决方案时可能会用到。

Signals 决不能复制

我已经提到过，`signal`s 不能被复制，但是值得留意的是，应该怎样实现一个包含 `signal` 的类。这些类也都必须是不可复制的吗？不，它们不必，但必须手工实现其复制构造函数和赋值操作符。因为 `signal` 类将其复制构造函数和赋值操作符声明为私有的，所以一个聚合了 `signal`s 的类必须实现其所需的语义。正确处理复制的一个方法是，在类的多个实例间共享 `signal`s，我们在停车场的例子中就是这么做的。在那个例子中，每一个 `parking_lot_guard` 实例通过 `boost::shared_ptr` 引向同一个 `signal`。对于其它类，可以在拷贝中缺省构造 `signal`，因为该复制语义不包含对插槽的连接。另一种情况是，复制一个含有 `signal` 的类是没有意义的，这种情况下你可以依赖所含 `signal` 的不可复制语义来确保复制与赋值是被禁止的。为了看得更清楚一点，考虑一个类 `some_class`, 它的定义是：

```
class some_class {
  boost::signal<void (int)> some_signal;
};
```

对于这个类，编译器生成的复制构造函数和赋值操作符都是不能使用的。如果代码企图去使用它们，编译器就会抗议。例如，以下例子试图从 `sc1` 复制构造 `some_class sc2` ：

```
int main() {
  some_class sc1;
  some_class sc2(sc1);
}
```

编译这段程序时，编译器生成的复制构造函数试图对 `some_class` 的成员进行逐个成员的复制。由于 `signal` 的私有复制构造函数，编译器会输出以下信息：

```
c:/boost_cvs/boost/boost/noncopyable.hpp: In copy constructor `
 boost::signals::detail::signal_base::signal_base(const
 boost::signals::detail::signal_base&)':
c:/boost_cvs/boost/boost/noncopyable.hpp:27: error: `
 boost::noncopyable::noncopyable(
 const boost::noncopyable&)' is private
noncopyable_example.cpp:10: error: within this context
```

所以，无论你的含有 `signal` 的类需要哪一种复制和赋值，你都必须确保其中不会有对 `signal` 的复制！

管理连接

我们已经讨论了如何连接插槽到 `signal`s, 但我们还没有看到如何断开它们。有许多原因让一个插槽不应该永久地连接到一个 `signal` 上。到现在为止，我们都忽略了它，其实 `boost::signal::connect` 会返回一个 `boost::signals::connection` 实例。通过使用这个 `connection` 对象，就可以从 `signal` 断开一个插槽，也可以测试一个插槽是否已连接到 `signal`. `connection` 是到 `signal` 和插槽间的实际链接的一个句柄。由于 `signal` 和插槽间的连接的信息是由它们两者分别跟踪的，所以插槽并不知道它本身是否被连接。如果一个插槽不想与 `signal` 断开，它只要忽略掉 `signal::connect` 所返回的 `connection` 即可。还

有，对一个插槽所属的组调用 `disconnect`，或者调用 `disconnect_all_slots` 都会断开插槽而无需提供插槽的 `connection`. 如果检查插槽是否还连接着 `signal` 的能力非常重要，你就只能保存 `connection` 并用它来询问 `signal`，别无它法。

`connection` 类提供了 `operator<`, 这使得你可以把连接保存在标准库的容器中。为了完备性，它也提供了 `operator==` 。最后，这个类提供了一个 `swap` 成员函数，用于与另一个 `connection` 交换各自的 `signal` /slot 连接信息。以下例子示范了如何使用 `signals::connection` 类：

```
#include <iostream>
#include <string>
#include "boost/signals.hpp"

class some_slot_type {
  std::string s_;
public:
  some_slot_type(const char* s) : s_(s) {}

  void operator()(const std::string& s) const {
    std::cout << s_ << ": " << s << '\n';
  }
};

int main() {
  boost::signal<void (const std::string&)> sig;

  some_slot_type sc1("sc1");
  some_slot_type sc2("sc2");

  boost::signals::connection c1=sig.connect(sc1);
  boost::signals::connection c2=sig.connect(sc2);

  // 比较
  std::cout << "c1==c2: " << (c1==c2) << '\n';
  std::cout << "c1<c2: " << (c1<c2) << '\n';

  // 检查连接
  if (c1.connected())
    std::cout << "c1 is connected to a signal\n";

  // 交换并断开
  sig("Hello there");
  c1.swap(c2);
  sig("We've swapped the connections");
  c1.disconnect();
  sig("Disconnected c1, which referred to sc2 after the swap");
}
```

在这个例子中有两个 `connection` 对象，我们看到它们可以用 `operator<` 和 `operator==` 来比较。 `operator<` 所实现的顺序关系是不确定的；它的存在是为了支持把 `connection`s 保存到标准库的容器中。而 `operator==` 所表示的等价关系则是有定义的。如果两个 `connection`s 引向同一个物理连接，它们就是等价的。如果两个 `connection`s 不引向任何连接，它们也是等价的。其它的 `connection`s 对都不等价。在这个例子中，我们还断开了一个 `connection`.

```
c1.disconnect();
```

虽然 `c1` 原先是引向 `sc1` 的 `connection`，但是在断开的时候它是引向 `sc2` 的，因为我们用成员函数 `swap` 交换了这两个连接的内容。断开连接意味着在 `signal` 产生时，该插槽不再被通知。以下是该程序的运行结果：

```
c1==c2: 0
c1<c2: 1
c1 is connected to a signal
sc1: Hello there
sc2: Hello there
sc1: We've swapped the connections
sc2: We've swapped the connections
sc1: Disconnected c1, which referred to sc2 after the swap
```

如你所见，最后一次的 `signal sig` 只调用了插槽 `sc1`.

有些时候，一个插槽的 `connection` 的生存期只限于某一段特定代码的范围。这种情况类似于其它资源要求仅限于某个特定范围时，通常可以使用智能指针或其它作用域机制来处理。Boost.Signals 提供了 `connection` 的一个作用域版本，名为 `scoped_connection`. `scoped_connection` 确保该 `connection` 在 `scoped_connection` 被销毁时断开连接。`scoped_connection` 的构造函数用一个 `connection` 对象作参数，它以此方式接受其所有权。

```
#include <iostream>
#include "boost/signals.hpp"

class slot {
public:
  void operator()() const {
    std::cout << "Something important just happened!\n";
  }
};

int main() {
  boost::signal<void ()> sig;
  {
    boost::signals::scoped_connection s=sig.connect(slot());
  }
  sig();
}
```

`boost::signals::scoped_connection s` 被限定在 `main` 内的一个小范围中，在离开该范围后，`signal sig` 被调用。这里不会产生输出，因为 `scoped_connection` 已经断开了插槽与 `signal` 间的连接。使用这样的带作用域的资源可以简化代码及其维护工作。

用 Bind 和 Lambda 创建插槽

你已经看到 Signals 多么有用以及么灵活。但是，当你把 Boost.Signals 与 Boost.Bind 和 Boost.Lambda 结合使用时，你会发现更大的威力。这两个库，它们的详细讨论请见 "Library 9: Bind 9" 和 "Library 10: Lambda 10"，它们有助于就地创建函数对象。这意味着你可以在需要连接到 `signal` 的地方就地创建插槽(以及插槽类型)，不再需要为插槽编写一个特定的、功

能单一的类，然后再创建一个实例并连接它。这样做还可以把插槽的逻辑就放在使用它们的地方，而不是放在源代码的别的地方。最后，这些库甚至可以用于改编一些已有的库，这些已有的库不提供调用操作符，但是有别的合适的方法来处理 `signal`。

在下面的第一个例子中，我们将看到 lambda 表达式如何漂亮地创建出一些插槽类型。这些插槽可以在调用 `connect` 的地方创建。第一个插槽在调用时简单地输出一个信息到 `std::cout`。第二个插槽检查 `signal` 传入的字符串值。如果它等于 `"Signal"`，则输出一个信息；否则它输出另一个信息。(这些例子确实有点做作，但这种表达式可以完成任何有用的计算)。该例子中创建的最后两个插槽完成了本章前面的例子中的 `double_slot` 和 `plus_slot` 所做的工作。你会发现这个 lambda 版本更具可读性。

```
#include <iostream>
#include <string>
#include <cassert>
#include "boost/signals.hpp"
#include "boost/lambda/lambda.hpp"
#include "boost/lambda/if.hpp"

int main() {
  using namespace boost::lambda;

  boost::signal<void (std::string)> sig;

  sig.connect(var(std::cout)
    << "Something happened: " << _1 << '\n');
  sig.connect(
    if_(_1=="Signal") [
      var(std::cout) << "Ok, I've got it\n"]
    .else_[
      std::cout << constant("Yeah, whatever\n")]);

  sig("Signal");
  sig("Another signal");

  boost::signal<void (int&)> sig2;
  sig2.connect(0,_1*=2); // 加倍
  sig2.connect(1,_1+=3); // 加 3
  int i=12;
  sig2(i);
  assert(i==27);
}
```

如果你还不熟悉C++(或其它)中的 lambda 表达式，不要为前面这段代码看起来有点糊涂而着急，你可以先看看 Bind 和 Lambda 那两章，然后再回到这个例子上来。如果你已经了解了 lambda 表达式，我可以肯定你一定会认为使用 lambda 表达式可以带来简洁的代码；而且它避免了把代码分割成多个小的函数对象。

现在让我们来看看使用绑定器来创建插槽类型。插槽必须实现一个调用操作符，但不是所有的类都适合作为 插槽。另一方面，通常可以使用一些已有的类成员函数，用绑定器重新包装它们以用作插槽。绑定器也有助于可读性，它允许处理某个事件的函数(而不是函数对 象)具有一个有意义的名字。最后，有时同一个对象需要对不同的事件作出反应，每一个都有相同的插槽签名，但是反应各有不同。因此，这种对象需要不同的成员 函数来为不同的事件所调

用。在这些情形下，没有一个调用操作符适用于连接到一个 `signal`. 因此，需要一个可配置的函数对象，而 Boost.Bind 正好提供了 (就象 Boost.Lambda 中的 `bind` 工具一样) 需要的方法。

考虑一个 `signal`，它接受一个返回 `bool` 且接受一个类型 `double` 的参数的插槽类型。假设类 `some_class` 有一个成员函数 `some_function`，它具有相符的签名，你如何把 `some_class::some_function` 连接到 `signal` 呢？一个方法是给 `some_class` 增加一个调用操作符，而该调用操作符把调用前转到 `some_function`. 这意味着要修改类的接口，而且它不好扩展。而绑定器可以做得更好。

```
#include <iostream>
#include "boost/signals.hpp"
#include "boost/bind.hpp"

class some_class {
public:
  bool some_function(double d) {
    return d>3.14;
  }

  bool another_function(double d) {
    return d<0.0;
  }
};

int main() {
  boost::signal<bool (double)> sig0;
  boost::signal<bool (double)> sig1;

  some_class sc;

  sig0.connect(
    boost::bind(&some_class::some_function,&sc,_1));
  sig1.connect(
    boost::bind(&some_class::another_function,&sc,_1));

  sig0(3.1);
  sig1(-12.78);
}
```

绑定这种方法有一个有趣的副作用：它避免了不必要的 `some_class` 实例的拷贝。绑定器持有对 `some_class` 实例的指针，而 `signal` 复制的是绑定器。不幸的是，这种方法有一个潜在的生存期管理问题：如果 `sc` 被销毁而后一个 `signal` 被调用，将导致未定义行为。这是因为绑定器将持有一个到 `sc` 的悬空指针。为了避免复制，我们必须负责保证插槽的生存期与(间接)引向它们的 `connection` 的存在一样长。当然，这正是引用计数智能指针的功能，所以这个问题很容易解决。

在使用 Boost.Signals 时，象这样使用绑定器是很常见的。无论你是使用 lambda 表达式来创建插槽，还是使用绑定器来把已有类改编为插槽类型使用，你都可以很快看到 Boost.Signals, Boost.Lambda, 与 Boost.Bind 相互配合的价值所在。它可以节省你的时间，并让你的代码更加美观和简洁。

Signals 总结

以下情形时使用 Signals ：

- 你需要健壮的回调时
- 事件具有多个处理者时
- `signal` 与插槽之间的连接需要在运行时可配置时

Boost.Signals 取代旧有风格的回调现在已经是很清楚了，这个库是当前可用的、最好的 signals/slots 实现之一。这个库所代表的设计模式非常著名，并且已经被研究了很长一段时间，所以这个领域已经非常成熟。一些编程语言已经在语言中直接实现了这种机制，如 .NET 中的delegates 和 events。在C++中，这个问题被库优美地解决了。Signals 和 slots 用于把事件的触发器机制从处理它的代码中分离出去。这种分离解耦了子系统，使它们更易于理解。它还解决了当重要事件发生时更新多个关注方的问题。在典型的程序或库中，有很多地方需要用到 signals 和 slots 。无论你是在编写一个GUI框架，或是一个发电站的入侵检测系统，Signals 都可以满足你的需要。它的用法很容易学习，它还提供了复杂任务所需的高级功能。例如，定制的 Combiners 可用于编写特定领域的事件处理机制。

Boost.Signals 由 Douglas Gregor 编写(他还编写了 Boost.Function)。这是一个伟大的库；谢谢你，Doug！