HZ BOOKS

Modern C++ Design
C++ 设计新思维

C++ 模板元编程

C++ Template Metaprogramming

Concepts, Tools, and Techniques from Boost and Beyond

（美） David Abrahams　Aleksey Gurtovoy　著
荣 耀　译

机 械 工 业 出 版 社
China Machine Press

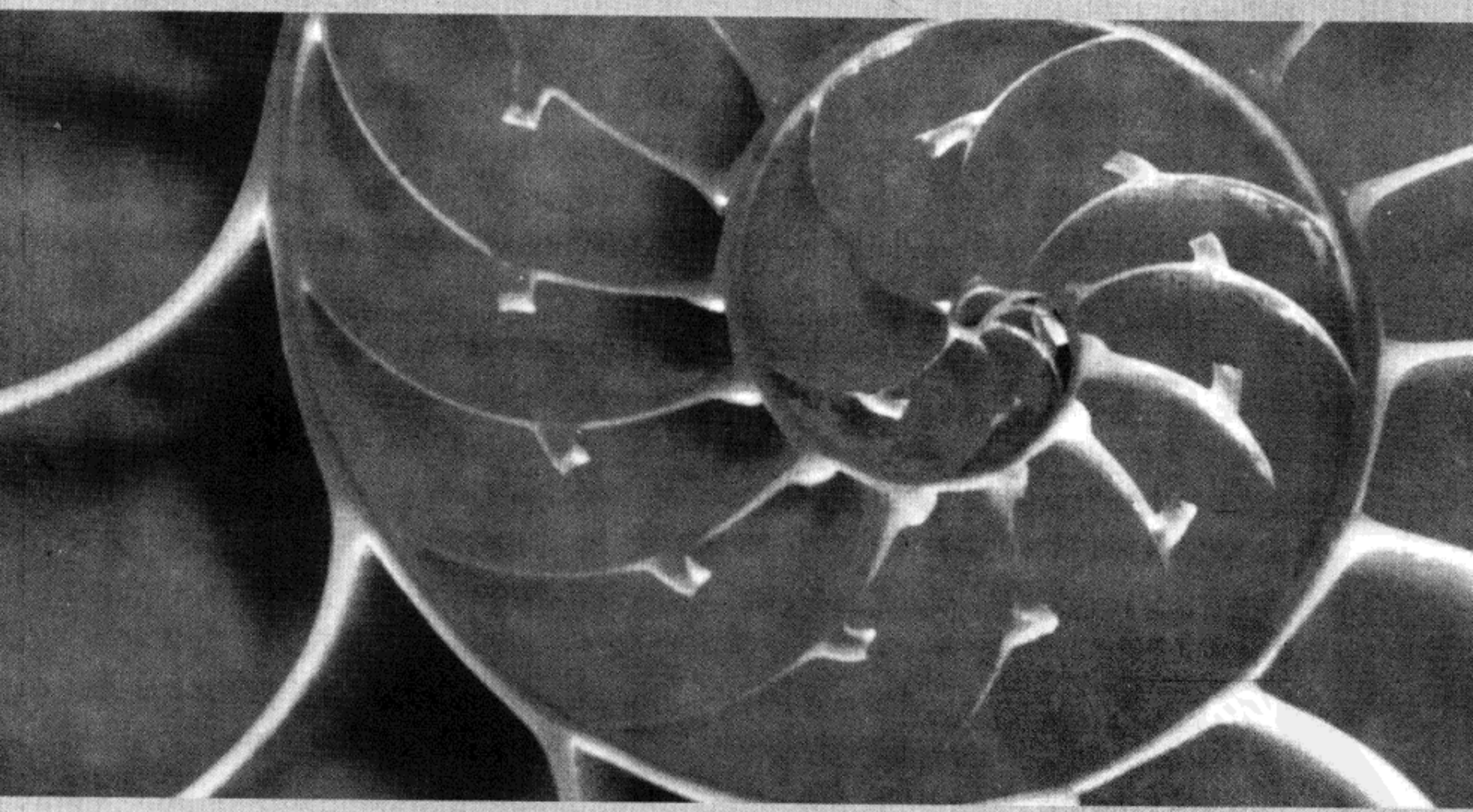
光盘

C++
设计新思维

C++模板元编程

C++ Template Metaprogramming

Concepts, Tools, and Techniques from Boost and Beyond



（美） David Abrahams Aleksey Gurtovoy 著

荣耀 译

机械工业出版社
China Machine Press

本书是关于C++模板元编程的著作。本书主要介绍Traits和类型操纵、深入探索元函数、整型外覆器和操作、序列与迭代器、算法、视图与迭代器适配器、诊断、跨越编译期和运行期边界、领域特定的嵌入式语言、DSEL设计演练，另外附录部分还介绍了预处理元编程、typename和template关键字。本书通过理论联系实践，深入讲解了C++高级编程技术。

本书适合中、高阶C++程序员等参考。

本书版权登记号：图字：01-2006-1938

图书在版编目（CIP）数据

C++模板元编程 /（美）大卫（David, A.）等著；荣耀译. —北京：机械工业出版社，2010.1
（C++设计新思维）

书名原文：C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond

ISBN 978-7-111-26742-3

Ⅰ. C…　Ⅱ. ① 大…　② 荣…　Ⅲ. C语言-程序设计　Ⅳ. TP312

中国版本图书馆CIP数据核字（2009）第048723号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
责任编辑：周茂辉
北京京北印刷有限公司印刷
2010年1月第1版第1次印刷
186mm × 240mm · 18.25印张
标准书号：ISBN 978-7-111-26742-3
　　ISBN 978-7-89451-053-2（光盘）
定价：55.00元（附光盘）

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991；88361066
购书热线：（010）68326294；88379649；68995259
投稿热线：（010）88379604
读者信箱：hzjsj@hzbook.com

译 者 序

作为一种高阶C++编程技术，模板元编程突出编译期决策在整个程序构建和运行过程中的地位，努力将计算从运行期提前至编译期，不但有效地防止程序错误被传播到运行期，而且能够实现以静态代码控制动态代码的目标。使计算尽可能完成于编译期也提高了最终程序的运行性能。

C++模板元编程诞生于十多年前，最初的研究方向是编译期数值计算，后来的实践发展证明，此项技术在类型计算领域可释放出更大的能量。近几年来，由于Andrei Alexandrescu的Loki程序库对元编程的前卫应用，Boost元编程库日益展示出重要的实用价值，C++模板元编程从最初被认为是对模板“过于聪明”的使用，到逐步被学界重视并研究，时至今日，这一高阶编程技术已然为业界所接受。

C++编程书籍不计其数，但涉及模板元编程的书籍屈指可数。作为Loki的传播者，《Modern C++ Design》对元编程的概念和原理的解释不够细致——这不奇怪，那本书的兴趣更多在于元编程在静态设计模式上的应用。David Vandevoorde和Nicolai M. Josuttis所著的《C++ Templates》，以及Krzysztof Czarnecki和Ulrich W.Eisenecker的著作《Generative Programming》，对模板元编程分别做了概述和总结，它们同样不是专注于元编程自身。Boost的创始人之一David Abrahams与Boost MPL的作者Aleksey Gurtovoy的这部著作第一次系统地阐述了模板元编程。

本书从内容上分为理论和实践两部分。前八章和部分附录内容以Boost元编程库为主线介绍模板元编程的概念、技术、工具及陷阱。其余篇幅则主要讨论模板元编程的一个重要的应用：DSEL（Domain-Specific Embedded Languages，领域特定的嵌入式语言）的设计与实现。虽然只有少数C++程序员需要创建DSEL，但了解其原理和实现大有裨益，有利于用好他人创建的DSEL，更重要的是，还可从中领会模板元编程的运用手法以及分析、解决实际问题的方法。

本书阅读门槛较高，适合希望了解模板元编程的中、高阶C++程序员尤其是程序库设计者阅读。如果你缺乏模板元编程必备的基础知识，例如类模板的特化和实例化、双重模板参数、typedef以及模板的继承等，建议参阅侯捷、荣耀和姜宏合译的《C++模板全览》（繁体版）一书，打好基础。

与常规C++编程技术相比，模板元编程技术较为复杂。因此不少C++程序员以为它高不可攀，或以为它只是库设计者的工具。虽然这项技术一直都没有疏远我们，然而我们自己的不作为却使它显得遥不可及。实际上，面向对象编程与泛型编程、运行期与编译期以及动态与静态之间并不互相排斥，而是对立统一的。从更高处审视C++程序设计，将多种编程范型优势互补，无疑可以开发出对程序员和最终用户而言更强大、更美妙的应用。

下一代C++标准C++0x将从语言和程序库两方面进一步增强对模板编程的支持，作为模板编

程的一个高阶子集，模板元编程也将从中受益。实际上，C++0x还将对模板元编程提供更友好的支持，（部分）Boost元编程程序库将会成为C++0x标准库的一个组成部分。模板元编程与普通C++程序员渐行渐近。现在，就让这本书引领你开始奇妙之旅！

感谢刘未鹏先生为第3章和附录A做出的高品质初译协助，感谢机械华章陈冀康先生、周茂辉编辑以及其他所有为本书面世付出贡献的人士，感谢朱艳和荣坤，生活因你们而更加精彩。

祝各位阅读快乐！

荣　耀

2006年6月

南京师范大学中北学院

序　言

1998年，Dave获权参加在德国Dagstuhl Castle举行的泛型编程研讨会。在研讨会临近尾声时，热情的Kristof Czarnecki和Ullrich Eisenecker（在产生式编程领域颇有声望）散发了一些C++源代码，那是采用C++模板编写的完整的Lisp实现清单。那时候，对于Dave而言那不过是个新奇的玩具而已，是对模板系统迷人但不切实际的“劫持”，以证实我们可以编写执行于编译期的程序。他从未想象有朝一日会在自己的大多数日常编程工作中发挥元编程（metaprogramming）的作用。在许多方面，那个模板代码集合是Boost元编程库（Metaprogramming Library, MPL）的先驱：它可能是第一个设计用于将编译期C++从一个特别的“模板技巧”集合转变为正规的、易理解的软件工程的范例的程序库。随着用于编写和理解元程序（metaprogram）的高阶工具的出现，我们发现使用这些技术不但切实可行，而且简单、有趣，并常常带来令人惊讶的威力。

撇开存在许多采用模板元编程和MPL的真实系统不谈，很多人仍然将元编程视作神秘的魔法，并且在日常产品代码中避免使用它。如果你从未进行过任何元编程，你甚至都看不出它和你所做的工作有什么明显的关系。在这本书中，我们希望能够揭开它的神秘面纱，使你不但对如何进行元编程有所理解，对为何（以及何时）进行元编程也会有很好的认知。最好的事情莫过于，尽管有许多神秘将会云开雾散，但你会发现这个主题中仍然蕴藏足够的神奇，让你流连忘返，受到启迪，获得灵感，如同我们一样。

——Dave和Aleksey

前　言

本书一开始的几章为你了解书中其他大多数内容铺设了必需的概念基础，后续各章通常建立于先前各章讲述的内容之上。也就是说，你可以自由地往前跳读，我们已经尽力使你能够这么做，因为当我们在使用早先介绍过的术语时提供了交叉引用。

第10章是“后续各章依赖于先前各章”这一规则的例外。它主要集中于概念之上，放在本书靠后的位置，因为到那时你将已经学到了将领域特定的嵌入式语言应用于现实代码中的工具和技术。如果读完本书你只记得一章的内容，希望就是这一章。

在很多章临近结尾的部分，你都会发现一个名为“细节”的小节，它们总结了关键的思想，而且往往还会添加新材料以深化先前的讨论[⊖]，因此，即使你在第一遍阅读时倾向于忽略它们，我们还是建议你在以后回头查阅它们。

大多数章以“练习”结束，它们设计用于帮助你发展编程和概念的能力。那些标以星号（*）的练习被认为比其他练习题困难一些。并非所有练习都要求你写程序，其中一些可以看做是问答题，并且也不是非得完成它们才能接着往下阅读。不过，我们建议你最好浏览它们一下，稍加思考如何回答每一个问题，并且动手做练习，这对于你巩固已经阅读过的内容是一个非常好的办法。如果你需要提示，可以考虑到boost用户邮件列表讨论问题（参见http://www.boost.org/more/mailing_lists.htm）。本书的Web站点（http://www.boost-consulting.com/mplbook）提供了一组wiki网页链接，其中包含挑选出的练习的答案。

补充材料

本书附带一张CD，以电子形式提供了以下材料：

- 书中的示例代码。
- Boost C++程序库的一个发行版。Boost因高质量、同级复审、可移植、泛型（通用）、可自由（免费）复用而知名。本书中我们广泛地使用了Boost程序库之一：Boost 元编程库（MPL，元编程库），当然我们也讨论了其他一些程序库。
- 一个完整的MPL参考手册，包括HTML和PDF两种格式。
- 本书中讨论到的、尚未成为正式发行版一部分的Boost程序库。

CD中顶层index.html文件为你提供了到其包含的所有内容的方便的引导。附加的和更新的材料，以及必不可少的勘误，将出现于本书的Web站点（http://www.boost-consulting.com/mplbook）

⊖ 我们从Andrew Koenig和Barbara Moo的《Accelerated C++: Practical Programming By Example》[KM00]中借用了这个好方式。

上。你在那儿还会发现一个用于报告你可能会发现的任何错误的地方。

试验

为了编译任何例子，只需将CD中的boost_1_32_0/目录设置到你使用的编译器的#include路径中即可。

我们展示于本书中的程序库已经竭尽全力隐藏不那么完美的编译器的问题，因此，在编译我们展示在这儿的例子过程中不大可能遇到困难。我们将C++编译器大致分为三类：

A. 大体上符合模板标准实现的编译器。在这些编译器上，示例和程序库完全可以工作。2001年后发布的几乎任何编译器，以及一些早于那个时间发布的编译器，都可归入这个范畴。

B. 需在客户代码中做一些迂回处理才可以工作的编译器。

C. 那些太拙劣以致于无法有效地用于模板元编程的编译器。

附录D列出了已知的符合每一个范畴的编译器。对于类型B中的编译器而言，附录D提及了一个移植手法列表。为了避免使大多数读者为之分心，这些迂回处理方式没有出现在正文中。你可以到本书的Web站点（http://www.boost-consulting.com/mplbook）下载补丁代码。

CD中还包含了一个有关移植性的表格，它详尽描述了各种编译器是如何处理示例的。对于许多平台而言，GCC都有可以免费获得的版本，且其最新版在处理列在这儿的代码时不会有任何问题。

即便你手头已经拥有一个处于类型A的相对现代的编译器，获取一份GCC拷贝并使用它来核查代码仍然是个好主意。通常破译难以理解的错误信息的最容易的方式，就是看看其他编译器对你的程序说了些什么。如果你发现当尝试做练习时要与错误信息搏斗，你也许可以往前跳一跳，阅读第8章的头两节，它们讨论了如何阅读和管理诊断信息。

还等什么，让我们开始C++模板元编程之旅吧！

致　谢

首先感谢审稿人Douglas Gregor、Joel de Guzman、Maxim Khesin、Mat Marcus、Jeremy Siek、Jaap Suter、Tommy Svensson、Daniel Wallin以及Leor Zolman，他们使我们保持诚实。特别感谢Luann Abrahams、Brian McNamara和Eric Niebler，他们阅读了每一页并提出建议，通常此时材料仍很粗糙。我们还要感谢Vesa Karvonen和Paul Mensonides，他们详细地审阅了附录A。感谢编辑Peter Gordon和Bjarne Stroustrup，因为他们的信任，我们才得以写出一些有价值的东西。David Goodger和Englebert Gruber构建了ReStructuredText标记语言，本书采用它写作而成。最后，我们感谢Boost社区创建的环境，从而使我们的协作成为可能。

Dave的致谢

2004年2月，我使用本书的一个早期版本为Oerlikon Contraves公司一群勇敢的工程师授课。感谢我所有的学生奋力通过艰苦的部分，并给与这些材料一次很好的彻查。尤其感谢Rejean Senecal逆“无智力投资”潮流为长远的高性能代码付出的投资。

Chuck Allison、Scott Meyers以及Herb Sutter均鼓励我多出版一些作品，谢谢诸位，我希望本书是一个良好的开端。

衷心感谢C++标准委员会和Boost的同事冒着自尊心和名誉受到伤害的危险向我示范（技术），技术人员通过协作可以完成伟大的事情。我很难想象如果没有这些社区今天我的职业生涯是什么样子。我很清楚，如果没有他们就不可能有这本书的问世。

最后，特别的爱送给特别的Luann，感谢她带我去看企鹅，并提醒我在每一章至少会想起它们一次。

Aleksey的致谢

我要特别感谢我的Meta团队伙伴们，过去5年来，这个团队已成为我的“大家庭”，而且还因他们创建和维护最有回报的工作环境。本书中反映的相当数量的知识、概念和思想成型于我们在这儿举行的结对编程（pair programming）会议、研讨会以及非正式的富有洞察力的讨论。

我还要感谢所有以这样或那样的方式对Boost元编程库的开发作出贡献的人，从某种意义上说，本书正是围绕它而展开叙述的。有许多人值得感谢，尤其要感谢以下人士：John R. Bandela、Fernando Cacciola、Peter Dimov、Hugo Duncan、Eric Friedman、Douglas Gregor、David B. Held、Vesa Karvonen、Mat Marcus、Paul Mensonides、aap Suter以及Emily Winch。

我的朋友和家人为我提供了持续的鼓励和支持，在写作本书的过程中这发挥了重要的作用，非常感谢你们！

最后但并非最不重要的是，感谢Julia一个人度过寂寞的时光，感谢对我的信任，感谢为我所做的一切。谢谢你的爱！

目　录

第1章 概 述

你可以将这一章看成是本书后续内容的热身。在本章中，你将有机会小试牛刀，试一试手头的编译器，大致检阅一些基本概念和术语。至本章结束，你至少应该对本书所要讲述的内容有大致的印象，并且（我们希望）届时你将会渴望学习更多的知识。

1.1 起步走

关于模板元程序（template metaprogram）的一个美妙之处在于，它们与传统的优秀系统一样具有一个共同的特性，即：一旦一个元程序（metaprogram）写好了，只要它能够工作，你就大可以放心使用，而不必关心其底层工作机制究竟是怎样的。

为了建立你的信心，让我们以一个C++小程序开始，该程序只是简单地使用了一个采用模板元编程实现的设施：

```
#include "libs/mpl/book/chapter1/binary.hpp"
#include <iostream>

int main()
{
    std::cout << binary<101010>::value << std::endl;
    return 0;
}
```

即使你一向擅长二进制算术，无需真正运行这个程序即可口算出结果，我们还是建议你不要怕麻烦，尝试编译和运行这个例子。这除了有助于建立信心外，还是一个很好的测试，看看你的编译器能否处理本书展示的代码。该程序应该向标准输出打印出二进制值101010的十进制值：

```
42
```

1.2 元程序的概念

如果按字面剖析单词“metaprogram”，它的含义就是“a program about a program”，译成中文，意思就是“一个关于另一个程序的程序”[⊖]。一个不那么有节奏感的说法是，“a metaprogram is a program that manipulates code”，即“元程序就是用于操纵代码的程序”。听起来这个概念好像有点怪怪的，但其实你可能早已熟悉这方面的一些“怪兽”了。你手头的C++编译器就是一个例子：它操纵C++代码来生成汇编语言（assembly language）或机器码（machine code）。

⊖ 在哲学中，并且碰巧在程序设计中，前缀“meta”用于表示“about”或“one level of description higher”的意思，就像源于原始的希腊语含义“beyond”或“behind”一样。

像YACC[Joh79]这样的解析器生成器（parser generators）是另外一类程序操纵程序（program-manipulating program）。YACC[⊖]的输入是一种高阶解析器描述（parser description），这种描述依据文法规则（grammar rule）编写，并且附加有采用封闭的大括号括起来的动作（action）。例如，为了采用通常的优先规则来解析和评估算术表达式，我们可以给YACC如下的文法描述（grammar description）：

```
expression : term
           | expression '+' term { $$ = $1 + $3; }
           | expression '-' term { $$ = $1 - $3; }
           ;

term : factor
     | term '*' factor { $$ = $1 * $3; }
     | term '/' factor { $$ = $1 / $3; }
     ;

factor : INTEGER
       | group
       ;

group : '(' expression ')'
      ;
```

作为响应，YACC将会生成一个C/C++源文件，其中包含有一个yyparse函数（以及一些别的东西），我们可以调用它以便根据文法来解析文本并执行适当的动作[⊖]：

```
int main()
{
  extern int yyparse();
  return yyparse();
}
```

YACC的用户大多在解析器设计领域操作，因此我们将YACC的输入语言称为该系统的领域语言（domain language）。因为用户程序的其余部分通常需要一个通用的编程系统，并且必须和生成的解析器（generated parser）互动，所以YACC将领域语言翻译成宿主语言（host language）——C语言，然后用户编译生成的语言代码，并将其与所编写的其他部分代码进行链接（link）。如此看来，领域语言经历了两个翻译（或转换，translation）步骤，用户总是可以很清醒地意识到它与程序其余部分之间的分界。

⊖ YACC是“Yet Another Compiler Compiler”的缩写，它是一个用于构建解析器（parser）、解释器（interpreter）和编译器（compiler）的工具。详见10.2.3节。——译者注

⊖ 这是基于这样的假设：我们还实现了一个适当的对文本进行标记化（tokenize）的yylex函数。参见第10章，其中有一个完整的例子，你也可以查阅YACC手册。

1.3 在宿主语言中进行元编程

YACC是一个翻译器（translator）的例子，即是这样的一个元程序：其领域语言不同于其宿主语言。在像Scheme[SS75]这样的语言中，还存在一种形式更有意思的元编程。Scheme元编程程序员（metaprogrammer）将他的领域语言定义为Scheme自身的一个合法的子集，元程序在与“处理用户程序的其余部分”相同的翻译步骤中执行。程序员在普通编程、元编程以及在领域语言中编程之间穿梭往来，但他们通常不会意识到这个转换过程，并且能够将多个领域无缝地联合于同一个系统中。

令人惊讶的是，如果你拥有一个C++编译器，以上所述的元编程威力恰好唾手可得。本书的其余部分将讲解如何释放这种威力，并且展示如何以及何时去使用它。

1.4 在C++中进行元编程

在C++中，几乎是在不经意中发现模板机制为原生语言元编程（native language metaprogramming）提供了丰富的设施（[Unruh94], [Veld95b]）。在这一节中，我们将探索元编程的基本机制以及在C++中进行元编程的一些惯用法。

1.4.1 数值计算

最早的C++元程序在编译期执行整数计算，其中最早的元程序之一是Erwin Unruh在一次C++标准委员会会议上所展示的。那实际上是一段不合法的代码，在其编译错误信息中含有一系列计算出来的质数值！[⊖]

由于不合法的代码很难有效地应用于规模较大的系统中，因此让我们来考察一个稍微实际一点的应用。下面的元程序（这是我们上面用于测试编译器的那个小例子的“心脏”代码）将无符号的十进制数值转换为等价的二进制值，允许我们用一种易辨识的形式来表示二进制常数：

```
template <unsigned long N>
struct binary
{
    static unsigned const value
       = binary<N/10>::value * 2        // 将高位位移向低位
         + N%10;
};

template <>                             // 特化
struct binary<0>                        // 终结递归
{
    static unsigned const value = 0;
};

unsigned const one   =    binary<1>::value;
unsigned const three =   binary<11>::value;
unsigned const five  =  binary<101>::value;
```

⊖ 有关此例的详情，参见《C++ 模板全览》（侯捷/荣耀/姜宏 译，台湾 碁峰信息股份有限公司印行）第17章。

```
unsigned const seven =  binary<111>::value;
unsigned const nine  = binary<1001>::value;
```

如果你有这样的疑问："程序在哪儿啊？"，那么请思考一下，当我们访问binary<N>的嵌套成员::value时发生了什么。binary template针对一个较小的数N进行实例化，直到N变为0，并且该特化版被用做终结条件。这个处理过程有我们熟知的递归函数调用的味道，那么程序究竟是什么呢？毕竟这只是一个类模板而已？是这样的，本质上，这里编译器被用于解释（interpret）我们这个小型元程序。

错误检查

这儿没有采取任何措施来阻止用户向binary传递一个诸如678之类的值，即它的十进制表示并不是一个有效的二进制值。所得结果有几分奇怪（其值为6x22 + 7x21 + 8x20）。毋庸置疑，诸如678这样的输入很可能意味着用户的逻辑中出现了臭虫。在第3章，我们将向你展示如何确保只有当N的十进制表示仅由0和1构成时，才可以编译binary<N>::value。

由于C++语言对编译期计算和运行期计算的表达方式强加了一些区别，因此，元程序看起来不同于它们的运行期对应物。如同在Scheme中一样，C++元编程程序员采用"和编写普通程序相同的"语言来编写代码，但是在C++中，程序员只能使用完整语言的一个编译期子集进行元编程。不妨将下面这个直观的运行期版本的binary和先前的例子比较一下：

```
unsigned binary(unsigned long N)
{
    return N == 0 ? 0 : N%10 + 2 * binary(N/10);
}
```

运行期版本和编译期版本之间一个关键的不同在于终结条件的处理方式。我们的meta-binary（元程序版本的binary）使用模板特化（template specialization）来描述当N为0时干些啥。"存在用做终结条件的模板特化"是几乎所有C++ 元程序所具有的共性，尽管在某些情形下，它们隐藏在一个元编程程序库接口的背后。

下面这个使用for循环来取代递归的binary版本，凸现了存在于运行期C++和编译期C++之间的另一个重要区别：

```
unsigned binary(unsigned long N)
{
    unsigned result = 0;
    for (unsigned bit = 0x1; N; N /= 10, bit <<= 1)
    {
        if (N%10)
            result += bit;
    }
    return result;
}
```

尽管这个版本比采用递归的那个版本冗长，然而很多C++程序员却感觉这个版本更舒服，原因并不仅在于运行期的迭代（iteration）有时比递归更高效，还在于这个版本看上去更加直观。

C++语言的编译期组成部分通常称为“纯函数性语言（pure functional language）”，因为它和Haskell这样的语言具有一个共同的特性：（元）数据是不可变的（常性的，immutable），并且（元）函数没有副作用（side effect）。结果导致编译期C++没有任何与用于运行期C++中的非常量变量相对应的东西。由于你无法在不检查其终结条件中一些可变的（mutable）东西的状态的情况下编写一个（有限）循环，因此在编译期无法实现迭代（iteration）。因此，对于C++ 元程序来说，递归（recursion）是惯用的手段。

1.4.2 类型计算

远比在编译期进行数值计算的能力重要得多的是C++“对类型进行计算（compute with types）”的能力。事实上，类型计算（type computation）将在本书余下内容中占据支配性的地位，我们会在下一章的第一节就给出这方面的例子。在我们检视后续内容期间，你可能会把模板元编程想象成“对类型进行计算”。

尽管你可能必须阅读第2章才能了解有关类型计算（type computation）的细节知识，然而在此我们希望给你一个有关其威力的初步认知。还记得我们的YACC表达式求值器（expression evaluator）吗？事实证明我们不需要使用一个翻译器（translator）来获得那种威力和便利。利用来自Boost Spirit程序库的适当的包覆代码（surrounding code），以下合法的C++代码就可以发挥等价的功能：

```
expr =
      ( term[expr.val = _1] >> '+' >> expr[expr.val += _1] )
    | ( term[expr.val = _1] >> '-' >> expr[expr.val -= _1] )
    | term[expr.val = _1]
    ;

term =
      ( factor[term.val = _1] >> '*' >> term[term.val *= _1] )
    | ( factor[term.val = _1] >> '/' >> term[term.val /= _1] )
    | factor[term.val = _1]
    ;

factor =
      integer[factor.val = _1]
    | ( '(' >> expr[factor.val = _1] >> ')' )
    ;
```

每一个赋值都存储有一个函数对象（function object），分别用于对其右侧的文法（grammar）进行解析和评估。当被调用时，每一个被存储的函数对象的行为完全由“用于构造它的表达式”的类型所决定，而每一个表达式的类型则由一个关联有形形色色运算符的元程序进行计算的。

就像YACC一样，Spirit 程序库是一个从文法规范（grammar specification）生成解析器（parser）的元程序。不同于YACC的是，Spirit采用C++自身的一个子集来定义其领域语言（domain language）。如果此刻你还不明白是如何做到这一点的，不必忧心忡忡，到读完这本书时，你就会恍然大悟。

1.5 为何进行元编程

那么，元编程有什么好处呢？毫无疑问，对于我们这里讨论的同类问题来说，还存在更简单的解决方案。让我们看看另外两种途径，考察当被应用于对二进制数值的解释（interpretation）和解析器（parser）的构造时它们是如何运作的。

1.5.1 替代方案1：运行期计算

最直截了当的方案是在运行期而非编译期执行计算。例如，我们可以使用早先展示的binary函数实现之一，也可以使用一个用于在运行期解释（interpret）输入文法（input grammar）的解析系统（parsing system）（解析操作发生于当我们第一次叫它去解析时）。

采用元程序最显而易见的理由是，通过在结果所得程序启动之前做尽可能多的工作，我们可以获得速度更快的程序。当一个文法（grammar）被编译时，YACC执行可观的解析表（parse table）生成和优化步骤，如果在运行期做这些事，将会显著地降低程序的整体性能。类似地，由于binary在编译期做工作，其::value是一个编译期常数，编译器可以将其直接编码（encode）到目的码（object code）中，因而当需要使用它时可以节省内存查找时间。

使用元程序的一个更微妙但也更重要的理由是，这种做法可以使得计算结果和目标语言进行更深入地互动。例如，C++数组的大小只能利用binary<N>::value这样的编译期常数合法地进行指定，而不能通过一个运行期函数返回值进行指定。在YACC文法中用大括号括起来的动作可以包含任意C/C++代码——这些代码作为生成的解析器的一部分而被执行。只有当动作（actions）处理于文法编译期间并传递给目标C/C++编译器时，才有可能做到这一点。

1.5.2 替代方案2：用户分析

不是在运行期或编译期进行计算，我们也可以手动做这件事。毕竟，将二进制数值转换为十六进制值[⊖]从而使它们可被直接用做C++字面常数，是很常见的编程实践，何况YACC和Boost.Spirit程序库所执行的将文法描述转换为一个解析器（parser）的翻译步骤也是众所周知的。

如果替代的是编写一个只被使用一次的元程序，我们就可以辩称用户分析（user analysis）更加方便：手工转换一个二进制数值要比编写一个正确的元程序做这件事当然要容易得多。然而在大多数情况下，这种做法的方便性远不及元程序的方式。一旦一个元程序被写好了，整个C++社群的其他程序员均可受惠于它所带来的方便性。

不管一个元程序被使用多少次，它总是可以让用户编写出更富有表达力的代码，因为用户可以以相应于他的心智模型（mental model）的形式来指定结果。在一个个体位（bit）值有意义的环境（context）中，写binary<101010>::value要比42或传统的0x2a更有意义。类似地，手写的解析器的C源代码通常会使其文法元素之间的逻辑关系变得模糊不清、晦涩难懂。

最后，由于人类易犯错误，而一个元程序的逻辑只需要编写一次，结果所得程序更有可能

⊖ “将二进制数值转换为十六进制值……”，从先前的例子来看，这儿说“将二进制数值转换为十进制值……”更具有一致性。——译者注

正确且维护性更好。转换二进制值是一个如此平常的任务，以至于人们往往太不专心乃至犯错。相对而言，手工编写解析表（parse table）则需要付出过多的注意力（任何做过这件事的人都可以证实这一点）。防止犯错误是使用一个像YACC这样的解析器生成器（parser generator）的充分理由。

1.5.3 为何进行C++ 元编程

在像C++这样的语言中，领域语言（domain language）不过是程序其余部分所使用的语言的一个子集，在这种情形下，元编程甚至具有更大的威力和方便性：

- 用户可以径自使用领域语言，而无需学习一种异质的语法，而且也不会打断原先代码的流程。
- 将元程序与其他代码、尤其是其他元程序进行接口连接，会变得平滑得多。
- 无需附加的生成（build）步骤（就像YACC所强加的那样）。

在传统编程中，程序员常常要努力获得表达性、正确性和效率之间的恰当的平衡。元编程通常允许我们消除这种典型的紧张和压力，这是通过将表达性和正确性所需的计算从运行期转移至编译期而达成的。

1.6 何时进行元编程

你已经看到一些关于为何需要模板元编程的例子，你也已经一瞥如何进行模板元编程，但我们还没有讨论何时适合进行模板元编程。然而，我们其实已经接触了与它使用有关的大部分准则。作为一个指导方针，如果以下条件中有任何三个适合你，可能就适宜采用一个元编程解决方案：

- 你希望代码根据问题领域的抽象进行表达。例如，你可能需要将一个解析器（parser）表达成这样的东西：它看起来像一个正式的文法（formal grammar）而不是充满数值的表（table）或子例程（subroutine）的集合。你希望数组算术使用“针对矩阵或向量对象”的运算符记号编写而成，而不是根据对一系列数值的循环编写而成。
- 如果不使用模板元编程的话，你需要编写大量的刻板的实现代码。
- 你需要基于组件的类型参数的属性来选择组件实现。
- 你希望利用C++中泛型编程（generic programming）的极富价值的特性，例如静态类型检查（static type checking）和行为定制（behavioral customization）能力，同时无需付出效率上的代价。
- 你希望全部在C++语言内做这件事，而无需使用第三方的工具或定制的源代码生成器（source code generator）。

1.7 为何需要元编程程序库

不是白手起家构建元程序，我们将与Boost 元编程库（MPL）提供的高阶设施协作。即使你没有选择这本书来探索MPL，我们还是认为你会发现花在学习MPL上的投资是值得的，因为它可以为你的日常编程工作带来诸多好处：

1. 质量（Quality）。大多数使用模板元编程组件的程序员将其看做“应用于一些更伟大目的”的实现细节。这种看法完全正确。比较而言，MPL作者将开发有用的、高质量的工具和惯用法作为他们的中心使命。一般来讲，Boost 元编程库中的组件要比那些为了别的目标而开发的组件灵活性更大，且有着更好的实现。此外，你还可以期望从将来发布的升级版中获得更多的优化和改善。

2. 复用（Reuse）。所有程序库都将代码封装为可复用的组件。更重要的是，一个设计良好的泛型程序库建立了一个有关概念和惯用法的框架（framework），此框架为解决问题提供了一个可复用的心智模型（reusable mental model）。就像C++ Standard Template Library给与我们迭代器（iterators）和函数对象（function object）协议一样，Boost 元编程库提供了类型迭代器（type iterator）和元函数协议。一个经过深思熟虑的惯用法框架，使得元编程程序员可以集中于自己的设计决策，专注于手头的任务。

3. 移植性（Portability）。一个优秀的程序库可以“掩饰”平台有别的丑陋的现实。尽管在理论上，任何C++ 元程序都不必关心这方面的问题，然而在实践中，甚至在C++被标准化六年后，各家编译器对模板的支持仍然不一致。对此你不应感到意外，因为模板是C++语言力所能及的最复杂的特性，另一方面，这一事实也成就了C++ 元编程的威力。

4. 乐趣（Fun）。一遍遍地重复相同的刻板代码是枯燥乏味的。将高级组件迅速组装成可读性好的、优雅的设计才是有趣的事情！MPL通过消除对最常反复使用的元编程模式的需要，减少了这种无趣的行为。尤其是借助MPL，我们可以容易且优雅地避免编写作为终结条件的模板特化版本（template specializations）和显式递归代码。

5. 生产率（Productivity）。除了个人满足外，我们项目的健康性仰仗于充满乐趣的编程。当我们不再有乐趣时就会感到身心疲倦，效率低下，马虎随便。你知道，充满臭虫的代码甚至比慢慢腾腾地编写代码本身要付出更高的代价。

正如你看到的那样，Boost 元编程库是出于同样的实践考虑所激发的，即成为任何其他程序库的开发基础。我们认为它的出现是一个征兆：模板元编程终将离开高深莫测的王国，并在普通C++程序员的日常必备技能中寻觅一席之地。

最后，我想特别强调一下上面所言的第四个条款：MPL不但使得元编程切实可行且易于使用，而且与之过从亦其乐无穷。在此祝愿诸位就像我们在使用和开发它的过程中享受快乐一样，在学习它的过程中也其乐融融。

第2章　Traits和类型操纵

我们希望第1章对数值问题的偏重不会给你留下这样的印象：大多数元程序实际上都是算术问题。实际上，编译期数值计算是比较罕见的。在这一章中，你将学习一个再次出现主题的基础知识，该主题即是“用做类型计算（type computation）的元编程”。

2.1　类型关联

在C++中，可以在编译期进行操纵的实体称为元数据（metadata），大致可以分为两个范畴：类型（types）和非类型（non-types）。并非巧合，任何种类的元数据都可用做模板参数。第1章中使用的常量整数值（values）就属于一种非类型，此范畴还包括可在编译期知道的几乎任何其他值，包括其他整型、枚举（enums）、函数和“全局”对象的指针和引用，以及指向成员的指针（pointers to members）[⊖]。

很容易想象对某些种类的非类型元数据执行一些计算，但是你可能对还可以对类型进行计算感到惊讶。为了对“这是什么意思？为什么它很重要？”获得一个认知，我们将考察来自C++标准程序库最简单的算法之一：iter_swap。它的卑微的职责就是接受两个迭代器并交换它们各自指向的对象的值。其实现看上去如下：

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 i1, ForwardIterator2 i2)
{
    T tmp = *i1;
    *i1 = *i2;
    *i2 = tmp;
}
```

如果此刻你对T从哪儿来满腹狐疑，恭喜你，你有一双火眼金睛。它还没有被定义，如果我们这么来编写它，那么iter_swap就无法通过编译。当然，一个非正式的说法是，T是当迭代器被解引用（dereferenced）时所得的类型，C++标准（24.1节）说这个东西是“迭代器的值类型（value type）”。好吧，那怎么给这个类型命名呢？

2.1.1　采用一种直接的方式

虽然你已经知道标准程序库作者选择的答案，我们还是建议你暂且忘记这一点，因为我们会研究一些更深入的要点。换个角度，设想我们自个儿正在实现标准程序库并且选择处理迭代器的方法。我们打算最终要编写大量的算法，其中许多需要在迭代器类型和其值类型之间进行

⊖ C++标准还允许模板作为模板参数进行传递。如果这还不让你感到头疼的话，标准又说这些参数是“用做描述目的的类型”。然而，模板并不是类型，并且不可以传递给另一个期望一个类型的模板。

关联。我们可以要求所有迭代器实现提供一个名为value_type的嵌套类型，可以直接访问它：

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 i1, ForwardIterator2 i2)
{
    typename                                    // （参见以下语言注释）
      ForwardIterator1::value_type tmp = *i1;
    *i1 = *i2;
    *i2 = tmp;
}
```

C++语言注释

当使用一个依赖性的名字（dependent name）且该名字表示的是一个类型时，C++标准要求使用typename关键字标明。ForwardIterator1::value_type未必是一个类型，这要视所传递的ForwardIterator1具体情况而定。参见附录B，以便了解关于typename的更多信息。

对于进行类型关联来说，这是一个极佳的策略，但它不是非常具有一般性。特别是，C++中迭代器的设计是模仿指针的设计的，这样做的用意是普通指针也可以用做合法的迭代器。不幸的是，指针没有嵌套类型，只有类[⊖]才有权定义嵌套类型：

```
void f(int* p1, int* p2)
{
    iter_swap(p1,p2); // error: int*没有"value_type"成员
}
```

2.1.2 采用一种迂回方式

我们可以通过引入一个额外的间接层来解决任何问题。

——Butler Lampson

Lampson的思想在程序设计领域是如此具有普遍性，以至于Andrew Koenig[⊖]喜欢将其称为软件工程的基本定理（Fundamental Theorem of Software Engineering，FTSE）。我们也许无法向所有迭代器中添加嵌套的::value_type，但是我们可以将其添加进带有一个迭代器类型的参数的模板。在标准程序库中，这个模板被称为iterator_traits，其签名很简单：

```
template <class Iterator> struct iterator_traits;
```

以下代码展示了iterator_traits是如何被用于iter_swap中的：

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 i1, ForwardIterator2 i2)
{
    typename
      iterator_traits<ForwardIterator1>::value_type tmp = *i1;
    *i1 = *i2;
```

⊖ 这里是广义的类，还包括结构体等。——译者注

⊖ Andrew Koenig是《Accelerated C++》合著者之一，并是C++标准的专案编辑。要想了解对他多年来对C++诸多贡献的公正评价，你可以去查查Bjarne Stroustrup写过的几乎任何一本书。

```
    *i2 = tmp;
}
```

之所以命名为iterator_traits，是因为它描述了其实参的属性（或特性，traits）。在这里，这个被描述的特性（traits）是迭代器的五个相关联的类型：value_type、reference、pointer、difference_type以及iterator_category。

traits模板最重要的功能是为我们提供一种非侵入的方式，将信息与类型进行关联。换句话说，如果你的爱争吵的工作伙伴Hector（泛指虚张声势的家伙）给你某个类似迭代器的类型，称为hands_off，它指向一个int，你可以赋予它一个value_type，而不会扰乱工作组的融洽气氛。你要做的就是添加一个iterator_traits显式特化（explicit specialization），当iter_swap询问关于Hector的迭代器的value_type时，它将会看到类型int[⊖]：

```
namespace std
{
  template <>
  struct iterator_traits<Hector::hands_off>
  {
      typedef int value_type;
      ……另外四个typedefs
  };
}
```

traits的这种非侵入性的方面正好就是使得iterator_traits可以适应于指针的原因：标准程序库包含有以下iterator_traits局部特化，它为所有指针描述了value_type：

```
template <class T>
struct iterator_traits<T*> {
    typedef T value_type;
    另外四个typedefs……
};
```

托“通过iterator_traits而实现的间接层”的福，泛型函数现在可以统一的方式访问迭代器相关联的类型，而不管该迭代器是否碰巧是个指针。

2.1.3 寻找一个捷径

尽管特化是一个相当通用的机制，但它远非是一个向类添加嵌套类型的便利机制。特化伴随着大量的包袱：你可能不得不关闭你正在其中工作的命名空间，并打开traits 模板的命名空间，然后你需要编写traits特化自身的代码。这并不是非常高效的使用键盘输入的方式，因为嵌套的typedef才是惟一真正有价值的的信息。

经过深思熟虑后，标准程序库提供了一个快捷方式，允许迭代器的作者控制嵌套在iterator_traits里头的类型，只要在迭代器里编写成员类型即可。iterator_traits主模板[⊜]“进入”迭代器来攫取其成员类型：

⊖ 本章末尾“细节”一节简要地回顾了模板特化（specialization）和实例化（instantiation）。

⊜ C++标准将与局部特化或显式特化（或完全特化）相对的普通模板的声明和定义称为主模板。

```
template <class Iterator>
struct iterator_traits {
    typedef typename Iterator::value_type value_type;
    ……另外四个typedefs
};
```

这儿你可以看到“额外的间接层”的工作方式：不是直接去访问Iterator::value_type，iter_swap通过询问iterator_traits来获得迭代器的value_type。除非某个特化版本“覆盖”了iterator_traits主模板，否则iter_swap将会看到同样的value_type，就像它直接访问迭代器中一个嵌套类型一样。

2.2 元函数

如果至此你已经开始注意到在traits 模板和普通函数之间存在某些相似性，那很好。traits 模板的参数和嵌套类型在编译期发挥了类似于普通函数的参数和返回值在运行期发挥的作用。第1章中的binary模板当然宛若函数。若你认为和函数相比，iterator_traits所执行的“类型计算(type computation)”看上去有一点儿平庸，我们可以理解，但是我们可以保证事情马上就会变得更有趣。

除了传递和返回的是类型而非值外，traits 模板还展示了两个显著的特征，它们在普通函数身上是看不到的：

- 特化。我们只要通过添加一个特化就可以针对其参数特定的“值”（即类型）非侵入性地修改traits 模板的结果。我们甚至可以通过使用局部特化为整个范围的“值”（例如所有指针）修改结果。如果你可以将特化机制应用于常规的函数，那么特化将会变得不可思议。你可以设想能够添加这样的一个重载的std::abs：只有当其实参为一个奇数时才能调用该特化版本！
- 多个“返回值”。普通函数传递多个实参但只返回一个值（译注：不要偏执地理解作者这个说法……），而traits通常具有不止一个返回结果。例如，std::iterator_traits包含五个嵌套类型：value_type、reference、pointer、difference_type以及iterator_category。traits 模板包含嵌套的常量或静态成员函数的情形也并非不常见，标准程序库中的std::char_traits组件就是一个例子。

尽管如此，类模板足够像函数，我们可以从这种类似中获得一些真正的好处。为了深入理解“类模板用作函数（class templates-as-functions）”的思想，我们将使用术语元函数(metafunctions)。元函数是Boost MPL的一个极为重要的抽象，对它们的形式化（正式化）是发挥其威力的一个关键。我们将在第3章深入讨论元函数，但是在此我们打算探讨一下有关元函数与经典的traits之间的一个重要区别。

标准程序库中的traits 模板都遵从“多个返回值”模型。我们将此类traits 模板称为“blob”，因为它好像是将一把单独的、松散相关的元函数揉碎后放入单个的单元中。我们无论如何都要避免这个惯用法，因为它会造成很大的问题。

首先是效率上的考虑：当我们第一次进入iterator_traits访问其::value_type时，该模板将会被

实例化（instantiated）。这对编译器来说意味着有许多工作要做，但对于我们来说，最重要的事情是，编译器需要计算出模板本体内的每一个依赖于模板参数的声明的含义。对于iterator_traits而言，这意味着不但要计算value_type，还有另外四个相关联的类型——即便我们不打算使用它们。当程序日益壮大时，这些额外的类型计算的代价会变得越发显著，从而降低编译速度。还记得我们说过类型计算将会变得更有意思吗？“更有意思”还意味着你的编译器要做更多的工作，以及你要更长时间地在电脑桌上敲击你的指头，以等待看到你的程序开始工作。

其次，也更重要的是，“the blob”会妨碍我们编写“接收其他元函数作为实参”的元函数的能力。为了使你明白这一点，考虑一个平凡的运行期函数，它接收两个函数实参：

```
template <class X, class UnaryOp1, class UnaryOp2>
X apply_fg(X x, UnaryOp1 f, UnaryOp2 g)
{
    return f(g(x));
}
```

然而，这并非是可以用于设计apply_fg的惟一方式。假定我们将f和g折叠到一个称为blob的单个实参中，如下：

```
template <class X, class Blob>
X apply_fg(X x, Blob blob)
{
    return blob.f(blob.g(x));
}
```

在这里，用于调用f和g的协议类似于访问一个“traits blob”的方式：为了获得“函数”的结果，你进入并访问其成员之一。问题在于，没有单一方式来获得调用这些“blobs”之一的结果。像apply_fg这样的每一个函数将会使用其自己的一套成员函数名，为了将f或g传递给其他此类函数，我们需要将其重新打包在一个具有新名字的外覆器之中。

“The blob”是一个反模式（anti-pattern）（这是应该避免的一种“惯用法”），因为它降低了一个程序的总体互操作能力（interoperability）或其各组件彼此平滑协作的能力。最初的做法，即编写接收两个函数实参的apply_fg的那个版本是一个好的选择，因为它提高了互操作能力。

当传递给apply_fg的可调用实参采用单个协议时，我们可以很容易地交换它们：

```
#include <functional>
float log2(float);

int a = apply_fg(5.0f,     std::negate<float>(), log2);
int b = apply_fg(-3.14f,   log2, std::negate<float>());
```

这种允许不同的实参类型可交换地使用的属性，称为多态（polymorphism）。多态的字面含义是“可以带有多种形式的能力”。

多态

在C++中有两种多态。动态多态（Dynamic polymorphism）允许我们通过单个基类指针或引用处理多个派生类型的对象。本章中我们讨论的静态多态（Static polymorphism），允许不

同类型的对象以同样的方式被操纵，只要它们支持某种共通的语法即可。动态（dynamic）和静态（static）这两个单词分别指示对象的实际类型决定于运行期还是编译期。动态多态，连同“迟绑定（late-binding）”或“运行期派发（runtime dispatch）”（C++借助于虚拟函数（virtual functions）提供的特性），是面向对象编程的关键特性。静态多态（也称为参数化多态（parametric polymorphism））是泛型编程的本质要素。

为了在元函数之间实现多态，我们需要以单一方式来调用它们。Boost 程序库使用的约定如下：

```
metafunction-name<type-arguments...>::type
```

从现在开始，当我们使用术语元函数时，就是指可以采用这种语法进行“调用”的模板。

2.3 数值元函数

如果连产生数值的元函数也按以上方式进行调用，你肯定会感到惊讶。不，我们并没有要求你给那些其实是个数值的东西起一个叫“类型”的名字。一个具有数值结果的元函数的结果::type真的是一个类型，该类型被称为“整型常量外覆器（integral constant wrapper）”，其嵌套的::value成员是一个整型常量。我们将在第4章探索整型常量外覆器的细节，此刻先看一个例子，它可以给你一个关于我们想表达的含义的认知：

```
struct five // 数值5的整型常量外覆器
{
    static int const value = 5;
    typedef int value_type;
    更多的声明……
};
```

这样一来，为了获得一个数值元函数的结果值，我们可以这样写：

```
metafunction-name<type arguments...>::type::value
```

同样，其他整型常量也以类似的外覆器被传递给元函数。乍看上去这个额外的间接层好像不方便，但到如今你也许可以猜到这样做的原因：要求所有的元函数都接收和返回类型，可以使它们更统一、“更多态（polymorphic）”、更具有互操作性。在接下来的一些章节中你将会看到这种对FTSE[⊖]的应用为我们带来回报的大量的例子[⊜]。

撇开所有这些好处不谈，每当你希望计算一个真正的整型常量时都要编写::type::value很快会变得无趣。纯粹是为了方便，一个数值元函数作者可能决定在元函数自身直接提供一个嵌套的::value。我们在本书中讨论到的来自Boost程序库的所有数值元函数都是这么做的。注意，尽管当你知道正在调用的元函数供应有一个::value时，你可以利用这一点，然而一般情况下你不能

⊖ Fundamental Theorem of Software Engineering，软件工程的基本定理。

⊜ 你也许已经注意到元函数协议似乎阻止我们达到目标——该目标正是我们使得元函数成为多态的理由：我们希望能够编写带有其他元函数作为实参的元函数。由于元函数是模板，而不是类型，我们不能够将它们传递给需要类型的地方。眼下我们不得不要求你将你的疑问暂且放一放，读完本章的其余部分，我们承诺将在第3章中讨论这个问题。

指望肯定存在这样的::value，即使你知道被调用的元函数会产生一个数值型的结果。

2.4 在编译期作出选择

如果至此你对类型计算（type computations）的思想仍然心存一丝困惑，我们很难责备你。无可否认，使用元函数来找到一个迭代器的value_type并不比那种美其名曰为“表查找（table lookup）”的做法好到哪里去。如果“类型计算（computation with types）”的思想有生命力，它的含义就不应该仅仅局限于类型关联（type associations）。

2.4.1 进一步讨论iter_swap

为了搞清楚如何将元函数用于现实代码中，让我们回头继续扮演“C++标准程序库实现者”的角色。虽然很遗憾提起这一点——但到如今我们将会收到来自关心性能的客户的大量bug报告，抱怨我们在2.1.2节定义的iter_swap的方式对于某些迭代器来说效率低下得可怕。似乎有一个家伙试图传入一个std::list<std::vector<std::string> >的迭代器，它对字符串向量进行迭代，性能评测器告诉他，iter_swap是性能瓶颈之所在。

按照事后聪明的说法——对此不应该感到特别惊讶：在iter_swap中，第一条语句对其中一个迭代器所引用的值做了一份复制。而复制一个vector则意味着复制其所有元素，每一个字符串元素的复制或赋值又可能导致动态内存分配以及对字符串的字符进行按位复制（bitwise copy），从性能的角度来看，这种做法相当糟糕。

幸运的是，有一个迂回解决方案。由于标准程序库提供了一个可以用于vectors的高效版本的swap，它只交换少量的内部指针（internal pointers），我们可以告诉客户只要解引用（dereference）迭代器并对解引用结果调用swap即可：

```
std::swap(*i1, *i2);
```

然而这样的回答不太令人满意。为什么iter_swap不应该同样高效？灵光乍现之际，我们回想起软件工程的基本定理：为何不加入一个间接层并将效率的责任委托给swap呢？

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 i1, ForwardIterator2 i2)
{
    std::swap(*i1,*i2);
}
```

这看上去挺好，但是运行我们的成套测试（test suite）结果表明，调用swap并非总是能够工作。你注意到iter_swap可以接收两个不同类型的迭代器了吗？看上去有一个测试试图使用iter_swap来将一个int*所指向的值与一个long*所指向的值进行交换。然而，swap函数只能对两个相同类型的对象进行操作：

```
template <class T> void swap(T& x, T& y);
```

当我们试图将其用于int*和long*时，上面的iter_swap实现品会导致一个编译错误，因为不存在匹配实参类型(int, long)的std::swap重载。

我们可以这样来解决这个问题，即让那个慢的iter_swap实现品保持不动，同时加入一个重

载版本：

```
// 通用的（慢速）版本
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 i1, ForwardIterator2 i2)
{
    typename
      iterator_traits<ForwardIterator1>::value_type
    tmp = *i1;
    *i1 = *i2;
    *i2 = tmp;
}

// 当两个迭代器类型相同时能更好地匹配
template <class ForwardIterator>
void iter_swap(ForwardIterator i1, ForwardIterator i2)
{
    std::swap(*i1, *i2); // 有时候更快
}
```

C++函数模板的偏序规则（rules for partial ordering）[⊖]宣称，当进行函数匹配时，上面新加入的重载具有更好的匹配。这也解决了(int*, long*)实参的问题，从而通过成套测试。于是我们的程序可以交付使用了！

2.4.2 美中不足

然而没过多久，又有人注意到我们仍然错过了一个重要的优化契机。考虑当我们对std::vector<std::string>和std::list<std::string>的迭代器调用iter_swap时会发生些什么？这两个迭代器具有同样的value_type（各有自己的高效的swap），但由于迭代器自身的类型不相同，因此快速版本的iter_swap重载将不会被调用。这儿需要的是这样一种方式：使得iter_swap工作于两个有着相同value_type的不同迭代器类型之上。

由于我们正在扮演“标准程序库实现者”的角色，因此我们总是可以尝试改写swap，使其可以处理两个不同的类型：

```
template <class T1, class T2>
void swap(T1& a, T2& b)
{
    T1 tmp = a;
    a = b;
    b = tmp;
}
```

这个简单的修复可以对付用户可能遭遇的绝大多数情形。

⊖ 欲知C++函数模板的偏序规则的详情，可以参考《C++ Templates全览》（侯捷/荣耀/姜宏 译）一书。

2.4.3 另一个美中不足

不幸的是，这种修改对于有一类迭代器来说仍然不能工作，就是其operator*重载产生一个代理引用（proxy reference）的那一种迭代器。事实上，一个代理引用根本不是一个引用，不过是一个试图模拟引用的类而已。对于一个既可读又可写的迭代器来说，一个代理引用不过是一个类，它可以被转换到value_type，也可以将value_type赋值给它。

最广为人知的例子是vector<bool>的迭代器[⊖]。vector<bool>容器以单个位来存储其每一个元素。由于实际上并不存在指向一个位之类的东西，所以使用了一个代理，如此一来，该vector的行为几乎就像其他vector那样了。代理的operator=(bool)向vector写入适当的位，其operator bool()则返回true——当且仅当该位在vector中被设置时。如下：

```
struct proxy
{
   proxy& operator=(bool x)
   {
       if (x)
           bytes[pos/8] |= (1u << (pos%8));
       else
           bytes[pos/8] &= ~(1u << (pos%8));
       return *this;
   }

   operator bool() const
   {
       return bytes[pos/8] & (1u << (pos%8));
   }

   unsigned char* bytes;
   size_t pos;
};

struct bit_iterator
{
   typedef bool value_type;
   typedef proxy reference;
   更多的typedefs……

   proxy operator*() const;
   更多的操作……
};
```

⊖ 在我们的回归测试（regression test）中这个问题很容易被错过。一些人甚至不确信vector<bool>::iterator是一个有效的迭代器。有关vector<bool>及其迭代器如何适合于标准已经招致大量的争论。Herb Sutter甚至为该问题向C++标准委员会提交了两篇论文（[n1185], [n1211]），还写了一个“Guru of the Week [GotW50]”。在委员会中，有关新迭代器概念（new iterator concepts）的系统[n1550]的工作已经开始了，我们希望这项工作有助于解决这个问题。

现在考虑当iter_swap解引用（dereference）一个bit_iterator并试图向std::swap传入一对代理引用时会发生些什么。回想起由于swap会修改其传入的实参，因此它们以非const引用形式被传入。问题在于，operator*返回的代理对象是临时对象，当我们试图将临时对象作为非const引用实参传递时，编译器会报错。大多数情况下，这是正确的决策，因为对临时对象所作的任何改变都将会消失得无影无踪。然而，最初的iter_swap实现品能够很好地处理vector<bool>的迭代器的情形。

2.4.4 “美中不足”之外覆器

最终，我们需要的是这样的一种方式：只有当迭代器具有相同的value_type并且其引用类型是真正的引用而非代理时才去选择快速的iter_swap实现品。为了做出这些选择，我们需要某种方式来询问（并回答）问题：“T是一个真正的引用吗？”，以及“这两个value_type相同吗”？

Boost包含有一个完整的元函数程序库，它被设计用于查询和操纵像“类型的同一性”和“引用性（referenceness）”这样的基础特性（traits）。给定适当的type traits，我们就可以决定到底使用std::swap还是使用我们自己编写的交换操作：

```
#include <boost/type_traits/is_reference.hpp>
#include <boost/type_traits/is_same.hpp>
#include <iterator>   // 针对iterator_traits
#include <utility>    // 针对swap

namespace std {

template <bool use_swap> struct iter_swap_impl; // 见下文

template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 i1, ForwardIterator2 i2)
{
    typedef iterator_traits<ForwardIterator1> traits1;
    typedef typename traits1::value_type v1;
    typedef typename traits1::reference r1;

    typedef iterator_traits<ForwardIterator2> traits2;
    typedef typename traits2::value_type v2;
    typedef typename traits2::reference r2;

    bool const use_swap = boost::is_same<v1,v2>::value
                          && boost::is_reference<r1>::value
                          && boost::is_reference<r2>::value;
```

我们尚未关闭iter_swap的结束大括号，此刻要做的全部事情就是找到一种方式：基于

use_swap的值来选择不同的行为。解决这个问题实际上有许多方式，其中很多方式将在第9章讨论。我们聪明地预见到通过前向声明iter_swap_impl进行派发（dispatching）的需要[⊖]。我们可以在iter_swap_impl的特化（在iter_swap本体之外）中提供两种行为：

```
template <>
struct iter_swap_impl<true>      // "快速的"版本
{
    template <class ForwardIterator1, class ForwardIterator2>
    static void do_it(ForwardIterator1 i1, ForwardIterator2 i2)
    {
        std::swap(*i1, *i2);
    }
};

template <>
struct iter_swap_impl<false>     // 一个总是可以工作的版本
{
    template <class ForwardIterator1, class ForwardIterator2>
    static void do_it(ForwardIterator1 i1, ForwardIterator2 i2)
    {
        typename
          iterator_traits<ForwardIterator1>::value_type
        tmp = *i1;
        *i1 = *i2;
        *i2 = tmp;
    }
};
```

现在iter_swap_impl <use_swap>::do_it为每一个可能的use_swap值提供一个适当的iter_swap实现。由于do_it是一个静态成员函数，所以iter_swap可以调用它而无需构造一个iter_swap_impl实例：

```
iter_swap_impl<use_swap>::do_it(i1, i2);
```

现在可以关闭该大括号了，当所有的回归测试都已通过时我们可以长舒一口气了。交货了，我们为之欣喜！客户得到了一个既快速又正确的iter_swap。

2.5 Boost Type Traits 程序库概览

事实证明，几乎每一个严肃的模板元程序最终都需要Boost Type Traits提供的一类设施。该程序库已经被证明是如此有用，它已经被C++标准委员会第一个“技术报告（Technical Report ([n1424], [n1519])）”所采纳，这预示它将会进入下一个官方标准。如欲了解完整的参考，参见配书光碟中Boost发行版libs/type_traits子目录中的文档，或访问http://www.boost.org/

⊖ 具有一点儿做作的远见卓识是作者的特权！

libs/type_traits。

2.5.1 一般知识

这个程序库总体上说有一些东西是你需要知道的：首先，正如你可能从iter_swap例子猜到的，这个程序库中的所有元函数都位于命名空间boost中，并且对于#include“定义了其中任何一个元函数”的头文件有一个简单的约定：

```
#include <boost/type_traits/     metafunction-name.hpp>
```

其次，正如我们早先暗示的，作为一个便利，Boost中的所有数值元函数都在其自身内直接提供了一个嵌套的::value。将求值为bool的元函数（bool-valued metafunctions，例如is_reference）当作“数值之函数”可能有点奇怪，但C++将bool归入整型一类。同样的约定也适用于其他所有求值为整型的元函数。（C++中的整型还包括char等类型。）

第三，有少数type traits（例如has_trivial_destructor）为了发挥完整的功能需要一些非标准编译器支持。少数编译器，尤其是Metrowerks CodeWarrior 和SGI MipsPro，实际上实现了所需的原语（primitives）。而在另外一些编译器中，这些traits用在某些类型身上通常是正确的，而用在另外一些类型身上时则降级为“得体地、安全地”。说“得体地”，我们的意思是即使traits得不到正确的结果，对它们的使用仍然可以通过编译。

为了理解我们所说的“安全地”的含义，你必须知道这些traits主要是用来决定某种优化能否进行。例如，一个具有平凡析构器（trivial destructor）的类型的存储区可能会被复用而不销毁它里头所包含的对象。然而，如果你无法决定该类型具有一个平凡的析构器，在复用其存储区之前你就必须销毁该对象。当has_trivial_destructor<T>无法决定正确的结果值时，它返回false，于是“泛型代码”（generic code，具有通用性的代码）将总是具有安全的执行路径并调用T的析构器。

最后，要意识到我们接下来描述的类型归类元函数（type categorization metafunctions，例如is_enum<T>）通常会忽视cv修饰符（即const和volatile），因此is_enum<T const>总是和is_enum<T>具有同样的结果。

下面每一个小节都描述了一组不同的traits。

2.5.2 主类型归类（Primary Type Categorization）

这些一元元函数决定C++标准中描述的哪一种基础类型归类（fundamental type categories）可以用做其实参。对于任何给定的类型T，这些元函数（metafunction）中有且仅有一个会产生结果true。

大多数人所熟悉的类型归类一共有9个traits。is_void<T>、is_pointer<T>、is_reference<T>、is_array<T>以及is_enum<T>没有什么好说的，它们不过是做你希望它们做的事情。is_integral<T>识别char、bool以及所有形形色色的signed和unsigned整型。类似地，is_float<T>用于识别浮点类型（floating-point types），包括float、double以及long double等。遗憾的是，如果没有编译器支持，is_union<T>总是返回false，这样，对于类和联合体而言，is_class<T>返回

的都是true[⊖]。

还有两个归类，大多数程序员用到的机会比较少。指向成员函数的指针（Pointers to member functions），具有以下形式：

R (C::)(args...) cv*

以及指向数据成员的指针（pointers to data members），写做

*D C::**

它们通过is_member_pointer<T>进行识别。注意，is_pointer识别不了这些类型，尽管它们称为“指针”。

最后，is_function<T>识别形如以下所示的函数类型：

R (args...)

许多人从未看过一个未经修饰的函数类型（因为函数的命名方式），当未被立即调用时——自动变成如下形式的函数指针或引用：

*R (*) (args...)*或*R (&) (args...)*

表2.1列出了主类型traits。

表2.1 主类型归类（Primary Type Categorization）

主类型Trait	::type::value和::value
`is_array<T>`	true，当且仅当T是一个数组类型
`is_class<T>`	true，当且仅当T是一个类类型，如果缺乏编译器支持，对T是联合体的情形也会报true
`is_enum<T>`	true，当且仅当T是一个枚举类型
`is_float<T>`	true，当且仅当T是一个浮点类型
`is_function<T>`	true，当且仅当T是一个函数类型
`is_integral<T>`	true，当且仅当T是一个整型
`is_member_pointer<T>`	true，当且仅当T是一个指向函数成员或数据成员的指针
`is_pointer<T>`	true，当且仅当T是一个指针类型（但不是一个指向成员的指针）
`is_reference<T>`	true，当且仅当T是一个引用类型
`is_union<T>`	true，当且仅当T是一个联合体，若缺乏编译器支持，总是报false
`is_void<T>`	true，当且仅当T具有cv void形式（cv即为const和volatile）

2.5.3 次类型归类（Secondary Type Categorization）

表2.2所示的traits表示对主类型归类的有用的分组或区分。

⊖ 类也可以使用struct关键字进行声明，但根据C++标准，它们仍然是类。事实上，以下两个声明是可以互换的：

```
class X;
struct X; // 声明了同样的X
```

struct和class的区别仅在于定义上，struct的基类（如果有的话）和成员默认来说是公共的（public）。事实上，“指向成员函数的指针”和“指向数据成员的指针”根本不是指针，而是一种偏移量（offset）。

表2.2 次类型归类

主类型Trait	::type::value和::value
is_arithmetic<T>	is_integral<T>::value || is_float<T>::value
is_compound<T>	!is_fundamental<T>::value
is_fundamental<T>	is_arithmetic<T>::value || is_void<T>::value
is_member_function_pointer<T>	true，当且仅当T是一个指向成员函数的指针
is_object<T>	!(is_function<T>::value || is_reference<T>::value || is_void<T>::value)
is_scalar<T>	is_arithmetic<T>::value || is_enum<T>::value || is_pointer<T>::value || is_member_pointer<T>::value

2.5.4 类型属性

对于那些不是和标准类型归类直接相关的traits，type traits程序库使用“属性（properties）”作为一个包罗万象的的术语来指代它们。这个分组中，最简单的traits是is_const和is_volatile，它们侦测实参的cv修饰符。其余的类型属性（Type Properties）总结在表2.3和表2.4中。

表2.3 类型属性

类型属性	::type::value和::value
alignment_of<T>	T的内存对齐所需大小的正倍数（程序库尽力使得该倍数最小化）
is_empty<T>	true，当且仅当编译器优化掉空基类（empty base classes）占用的空间并且T是一个空类时
is_polymorphic<T>	true，当且仅当T是一个至少具有一个虚拟成员函数（virtual member function）的类时

表2.4 其他类型属性

类型属性	::type::value和::value
has_nothrow_assign<T>	true，只有当T拥有一个不抛异常的赋值操作符时
has_nothrow_constructor<T>	true，只有当T拥有一个不抛异常的默认构造器时
has_nothrow_copy<T>	true，只有当T拥有一个不抛异常的复制构造器时
has_trivial_assign<T>	true，只有当T拥有一个平凡的赋值操作符时
has_trivial_constructor<T>	true，只有当T拥有一个平凡的默认构造器时
has_trivial_copy<T>	true，只有当T拥有一个平凡的复制构造器时
is_pod<T>	true，只有当T是一个POD类型时①
is_stateless<T>	true，只有当T是空的、并且其构造器和析构器是平凡的（trivial）时

① POD代表“plain old data”。信不信由你，这是C++标准中的一个技术术语。标准赋予我们天马行空的自由去对POD类型做出所有种类的特别假定。例如，POD总是占用连续字节的存储空间，其他类型则未必如此。一个POD类型被定义为一个标量（scalar），一个PODs数组或一个这样的结构体或联合体：没有用户声明的构造器，没有用户声明的复制赋值运算符，没有用户声明的析构器，没有私有的或被保护的非静态数据成员，没有基类（base classes），没有非POD、引用或指向成员类型的指针、或这些类型的数组等类型的非静态数据成员，并且没有虚拟函数。

表2.4中的traits对于选择优化来说非常有用。如果编译器提供支持，它们可以被更精确地实现，从而允许表中的“只有当（only if）”被替代为“当且仅当（if and only if （iff））”。

2.5.5 类型之间的关系

程序库包含三个重要的元函数，它们用于指示类型之间的关系。我们已经看到了is_same<T,U>，当T和U是一致的类型时，其::value是true。当且仅当类型T的对象可以被隐式转换为类型U时，is_convertible<T,U>产生true。最后，当且仅当B是D的一个基类时，is_base_and_derived<B,D>::value为true。

2.5.6 类型转化

表2.5中的元函数执行基础类型操纵。注意，不同于我们迄今为止讨论的其他type traits 元函数，这一组元函数产生类型结果而不是布尔值（Boolean values）。你可以将它们看做类型算术运算符。

表2.5 转化类型

转 化	::type
remove_const<T>	T不带任何顶层const。例如，const int变成int，但const int*保持不变
remove_volatile<T>	T不带任何顶层volatile。例如，volatile int变成int
remove_cv<T>	T不带任何顶层cv修饰符。例如，const volatile int变成int
remove_reference<T>	T不带任何顶层reference。例如，int&变成int，但int* 保持不变
remove_bounds<T>	T不带任何顶层数组中括号。例如，int[2][3]变成int[3]
remove_pointer<T>	T不带任何顶层pointer。例如，int*变成int，但int& 保持不变
add_reference<T>	如果T是一个引用类型，结果为T，否则为T&
add_pointer<T>	remove_reference<T>::type*。例如，int和int&都变成int*
add_const<T>	T const
add_volatile<T>	T volatile
add_cv<T>	T const volatile

至此你可能会好奇为什么要为表中最后三个转化而费心，毕竟add_const<T>::type不过是T const的一种更冗长的写作方式。事实证明，能够以元函数的形式来表达哪怕最简单的转化也是非常重要的，因为这样一来它们就可以被传递给其他元函数（正如我们承诺的那样，我们将会在下一章中展示如何操作）。

2.6 无参元函数

在这一章中我们可能做的最重要的事情就是描述“元函数”概念，但还有一个问题我们仍然没有回答：一个无参（0个参数）元函数看起来是什么模样？

从需满足的条件来看，一个无参元函数（nullary metafunction）可以是任何类型，不管它是一个普通的类还是一个类模板特化（提供有嵌套的::type成员）。例如add_const<int>是一个无参元函数，它总是返回同样的结果：int const。

编写无参元函数最容易的方式是借助一个简单的struct：

```
struct always_int
{
    typedef int type;
};
```

2.7 元函数的定义

最终，我们拥有了为元函数编写一个完备的、正式的描述所需的一切东西。

> **定义**
> 一个元函数可以是
> 一个类模板，它的所有参数都是类型，
> 或者
> 一个类
> 带有一个名为“type”的可公开访问的嵌套结果类型。

2.8 历史

Boost Type Traits程序库受到SGI STL实现品中的一个组件的启发，该组件看起来如下：

```
struct true_type {}; struct false_type {};
template <class T> struct type_traits // 什么都没假定（都是false_type……）
{
   typedef false_type has_trivial_default_constructor;
   typedef false_type has_trivial_copy_constructor;
   typedef false_type has_trivial_assignment_operator;
   typedef false_type has_trivial_destructor;
   typedef false_type is_POD_type;
};

template<> struct type_traits<char> // 针对char的特化
{
   typedef true_type has_trivial_default_constructor;
   typedef true_type has_trivial_copy_constructor;
   typedef true_type has_trivial_assignment_operator;
   typedef true_type has_trivial_destructor;
   typedef true_type is_POD_type;
};
接下来是更多的特化……
```

我们可以注意到一个有趣的地方，尽管SGI type traits产生结果类型（result types），但它仍然是一个“blob”，而这杀死了多态（polymorphism）。SGI设计者肯定有别的原因选择使用嵌套的类型而不是bool常量[⊖]。

⊖ 要想获得一个可能的理由的线索，参见9.2.3节。

Boost Type Traits是第一个明确认识到使用单值元函数重要性的C++程序库。Boost拒绝“blob”设计，首先是因为这样可以为单个模板保留一个非常一般化的名字：type_traits。这个名字好像要求任何新的traits应该被吸收在那儿——一个Borg blob！任何希望编写类似的组件的人都会感觉到被迫进入并修改这个模板，从而可能会导致臭虫。当时，这个选择对于效率和互操作性的积极的影响没有被很好地理解。

设计者建立了一个约定，具有布尔结果的traits具有一个::value成员，而那些具有一个类型结果的则具有一个::type成员，这样用户就无需猜测如何调用一个给定的trait。这个选择象征着他们意识到了多态（polymorphism）的价值，即使他们没有得到最终的结论：所有元函数都应该提供一个::type。

事实上，在Boost 元编程库（MPL）的工作开始之前type traits并没有被看做“元函数”。从那时起，Type Traits库中使用的约定变成了MPL元函数使用的统一协议的基础，Boost Type Traits程序库也被相应地加以更新。

2.9 细节

这一章中我们讲述了大量的背景知识。从traits到元函数的旅行将我们从用于最简单的泛型程序中的特别的类型关联（type associations），带到允许将元编程看做一等（first class）编程活动的基础原理。我们还稍加研究了C++ 模板机制，概览了type traits程序库，查看了它的一些组件的用法。在如此开阔的视野下，必然会错过一些重要细节，让我们在回顾本章最重要的知识点的过程中将它们弥补上。

2.9.1 特化

应用于C++模板的特化（specialization）的含义可能较难把握，因为它被用于两种不同的方式。第一个用法指的是填充了具体实参的模板，例如iterator_traits<int*>。换句话说，一个模板特化命名了一个真正的类（若是函数模板特化的情形，则为函数），这是采用具体实参替代模板参数所得的结果。

对特化（specialization）的第二种用法出现于“显式特化（explicit specialization）”或“局部特化（partial specialization）”中。在本章中，我们既展示了iterator_traits的显式特化，也展示了其局部特化。“显式”的说法也许不是一个适当的选择，因为局部特化也是显式的，你可以将“显式特化”理解为“完全特化”而不会有任何理解偏差。

为了记住声明类模板特化的语法规则（上面所述的第二个含义），请牢记如下的形式：

```
template <variable part>
struct template-name<fixed part>
```

在一个显式（或完全）特化中，variable part是空的，而fixed part则由具体的模板参数构成。在一个局部特化中，variable part包含有一个参数列表，fixed part中则至少有一个实参依赖于这些参数。

Primary template（主模板）

不是一个特化（上面所述的第二个含义）的模板声明称为主模板。我们可以认为主模板涵

盖一般的情况，而特化则处理形形色色的特殊情形。

2.9.2 实例化

当编译器需要知道一个模板的更多内容（远比“其实参是什么？”要多，包括诸如其成员的名字或基类的身份等）时，模板将被实例化（instantiated）。在该时刻，编译器为所有模板参数填充实际值，挑选最佳匹配的显式或部分特化（如果有的话），计算出模板本体内的声明中使用的所有类型和常量，并核查这些声明是否有误。然而，不到“定义（definitions）”（例如成员函数本体）被使用时，它并不实例化“定义”。例如：

```
template <class T, class U>
struct X
{
    int f(T* x)                    // 声明
    {
       U y[10];                    // 定义
       return 0;
    }
};

typedef X<int&, char> t1;          // OK，尚未实例化
t1 x1;                             // 错误:指向int&的指针不合法
typedef X<int, char&> t2;
t2 x2;                             // OK，声明合格
int a = x2.f();                    // 错误：元素为char&的数组非法
```

正如你看到的那样，模板实例化不但会影响编译速度，而且还会影响到你的程序从根本上是否编译！

blob

在面向对象的编程文献中将一个带有大量成员（包括成员函数）的类称做“blob”[BMMM98]。类的成员彼此“耦合”，因为它们必须被声明在一起。为了避免耦合并提高模块化程度，应该避免使用这种反模式（anti-pattern）。代替方案是采用独立的元函数来定义单独的traits。

元数据

可被C++编译期系统操纵的“值”可以被认为是元数据。在模板元编程中，两种最常见的元数据是类型和整数（包括bool）常量。C++的编译期部分通常被称为“纯函数式语言（pure functional language）”，因为元数据是不可变的（immutable）并且元函数不可以有任何副作用。

2.9.3 多态

从字面上讲，多态（Polymorphism）就是“具有多种形式的意思”。在计算机语言中，多态意味着通过一个共同接口操纵不同类型的能力。具有一致的接口是“确保代码可复用以及组件可自然地互操作”的最佳方式。由于C++ 模板并非生来就多态地处理不同种类的元数据，因此MPL遵从使用类型外覆器（type wrappers）来包装非类型元数据的约定。尤其是，数值元数据

被表示为一个类型，该类型具有一个名为::value的嵌套数值常量成员。

元函数

一个操作元数据并可以在编译期“调用”的“函数”。在本书的其余部分，一个模板或类只有当它不具有非类型元数据并且返回一个名为type的类型时，才被称为一个元函数。传给类模板的实参就是编译期计算的输入，::type成员则是其返回结果。因此，以下表达式：

```
some_metafunction<Arg1, Arg2>::type
```

类似于以下的运行期计算：

```
some_function(arg1, arg2)
```

数值型元函数

返回一个针对某个数值的外覆器类型的元函数。为了简便，很多数值元函数自身提供了嵌套的::value成员，因此你可以写：

```
some_numerical_metafunction<Arg>::value
```

而不是更一般的：

```
some_numerical_metafunction<Arg>::type::value
```

——如果你希望直接访问数值结果的话。

无参元函数

任何具有可公开访问的::type的类，均可用做一个接受0个实参的元函数。作为这个定义的结果，任何元函数特化（前面所述的第一个含义），例如boost::remove_pointer<char*>，也是一个合法的无参元函数。

Traits

一种通过类模板特化在小片元数据之间建立关联的技术。Traits惯用法的一个关键特性是它是非侵入性的（non-intrusive）：我们可以在不修改被关联项自身的前提下，建立一个新的映射。MPL元函数可看做是traits的一个特例（特殊情况），它们对于任何输入都只有一个结果值。

Type traits

Boost Type Traits程序库是一个元函数程序库。它包含一些用于低阶类型操纵的元函数。例如，add_reference的结果总是一个引用类型。

```
boost::add_reference<char>::type        // char&
boost::add_reference<int&>::type        // int&
```

Type Traits程序库主要由用于回答“关于任何类型的基础属性”的问题的布尔值元函数构成，例如：

```
boost::is_reference<char>::value        // false
boost::is_reference<int&>::value        // true
```

2.10 练习

2-0. 编写一个一元元函数 add_const_ref<T>，如果T是一个引用类型，就返回T，否则返回T const&。编写一个程序来测试你的元函数。提示：可以使用boost::is_same来测试结果。

2-1. 编写一个三元元函数 replace_type<c,x,y>，它接收一个任意的复合类型c作为其第一个参数，并将c中出现的所有type x替换为y：

```
typedef replace_type< void*, void, int >::type t1; // int*
typedef replace_type<
    int const*[10]
  , int const
  , long
>::type t2; // long* [10]

typedef replace_type<
    char& (*)(char&)
  , char&
  , long&
>::type t3; // long& (*)(long&)
```

你可以将所操作的函数类型限制为具有少于两个参数的函数。

2-2. boost::polymorphic_downcast函数模板[⊖]实现一个带检查版本的static_cast，用于将指向多态对象的指针向下转型：

```
template <class Target, class Source>
inline Target polymorphic_downcast(Source* x)
{
    assert( dynamic_cast<Target>(x) == x );
    return static_cast<Target>(x);
}
```

在发行版的软件中，assertion消失并且polymorphic_downcast可以和简单的static_cast一样高效。使用该type traits设施来编写一个模板实现品，使其既可接收指针参数也可接收引用参数：

```
struct A { virtual ~A() {} };
struct B : A {};

B b;
A* a_ptr = &b;
B* b_ptr = polymorphic_downcast<B*>(a_ptr);

A& a_ref = b;
B& b_ref = polymorphic_downcast<B&>(a_ref);
```

2-3. 使用type traits设施实现一个type_descriptor类模板，当被流化（streamed）时，其实例打印其模板参数的类型[⊜]：

```
// 打印“int”
```

⊖ 参见http://www.boost.org/libs/conversion/cast.htm。

⊜ 我们无法使用运行期类型信息（runtime type information，RTTI）来获得同样的效果，因为根据C++98标准18.5.1 [lib.type.info]第7段所述，typeid(T). name()不能保证返回一个有意义的结果。

```
std::cout << type_descriptor<int>();

// 打印“char*”
std::cout << type_descriptor<char*>();

// 打印“long const*&”
std::cout << type_descriptor<long const*&>();
```

你可以假定type_descriptor的模板参数局限于根据以下四种整型构建的复合类型：char、short int、int以及long int。

2-4. 不使用Type Traits程序库，为练习2-3编写一个替代的解决方案，并对两种方案加以比较。

2-5. 修改练习2-3中的type_descriptor 模板，使其输出type的伪英语描述，就像cdecl程序的explain命令所做的那样[⊖]：

```
// 打印“array of pointer to function returning pointer to char”
std::cout << type_descriptor< char *(*[])() >();
```

2-6. 尽管乍看上去Type Traits程序库提供的类型代数集合好像很完备，其实不然。至少有一些类型归类、关系和转化是这个程序库的设施所未涵盖的。例如，它们未提供一个获得signed integer类型的unsigned对应物的方式。

尽可能多地识别这些遗漏的部分，每一个traits归类中至少识别出1个，我们总共可以想出至少11个。为每一个遗漏的traits设计一个接口并给出一个用例（use case）。

2-7. 游览Type Traits程序库的好处之一是我们还游历了C++运行期类型系统。每一个主类型归类（primary type categories），连同const和volatile修饰符，都是可以用于构建任意丰富的类型的基础构建块。

所有可能的C++ types都是类型元数据的可能的“值”，这就带来了一个问题，“C++的编译期类型系统看起来是啥样？”编写一个有关编译期C++的静态类型系统的简短描述。提示：静态类型系统对能够传递给特定函数的值有所限制。

2-8. 根据静态和动态类型检查，描述使得所有元数据多态的效果。

⊖ http://linuxcommand.org/man_pages/cdecl1.html。

第3章　深入探索元函数

有了前面的基础知识作铺垫，我们准备考察模板元编程技术的一个最基本的应用——为传统的不进行类型检查的操作添加静态类型检查。为此，我们将考察一个来自自然科学和工程学的实例（几乎在所有涉及科学计算的代码中都可以找到其应用）。在考察该例子的过程中，你将会学到一些重要的新的概念，并且体会使用MPL进行高阶元编程。

3.1　量纲分析

理论上，物理计算的首要原则是：数值并非是独立的，大多数物理量都附带有量纲(dimensions)。我们一不小心就会将量纲置之脑后，这是很危险的事情。随着计算变得越来越复杂，维持物理量的正确量纲能够避免诸如“将质量赋给长度”以及“将加速度和速度相加”之类不经意间犯下的错误。这意味着要为数值建立一个类型系统。

手工检查类型是项单调乏味的工作，并且容易导致出错。当人们感到厌烦时，注意力就会分散，从而容易犯错误。然而，类型检查这类工作看上去不正是计算机所擅长的吗？如果我们能够为物理量和量纲建立一个C++类型的框架，那么我们从公式中就可以捕获错误，从而不必等它们在现实世界中导致严重问题的时候。

防止量纲不同的物理量互操作并不难：我们可以简单地用类来表现量纲，并且只允许相同的类（量纲）互操作。但问题并不只这么简单，由于不同的量纲可以通过乘法或除法结合起来，从而产生任意复杂的新量纲。例如，牛顿定律（它将力与质量、加速度联系起来）：

$$F = ma$$

由于质量和加速度有着不同的量纲，所以力的量纲必须是二者的结合。实际上，加速度的量纲就已经是一个“混合物”了，它表示单位时间内速度的改变：

$$dv/dt$$

又因速度为“单位时间内（t）经过的距离（l）”，所以加速度的基本量纲是：

$$(l/t)/t = l/t^2$$

由于加速度通常以“米每平方秒”来度量，从而可以推导出力的量纲为：

$$ml/t^2$$

也就是说，力通常以kg(m/s²)或“千克-米每平方秒”来衡量。当我们将质量和加速度相乘时，我们除了将数量相乘之外还必须将量纲相乘，这可以帮助我们确信结果是有意义的。这种（对量纲的）簿记的正式名称为量纲分析（dimensional analysis），我们接下来的任务就是在C++类型系统中实现它。John Barton和Lee Nackman在他们开创性的著作《Scientific and Engineering

C++》[BN94]中第一次展示了如何实现它。我们将沿袭他们的思路，但是以元编程方式重新加以实现。

3.1.1 量纲的表示

国际标准量纲制规定了物理量的基本量纲为：质量、长度或位置、时间、电荷、温度、密度以及物质的量，其他量纲则在此基础上复合而成。为了具有相当程度的通用性，我们的系统必须要能表示七个或七个以上的基本量纲，同时还要能够表示复合量纲，例如像力的量纲，即通过若干基本量纲乘除而构成的复合量纲。

一般来说，一个复合量纲可以看成若干基本量纲的幂的乘积[⊖]。如果要表示这些幂次以便在运行期可以操纵它们，我们可以使用一个数组，七个元素每一个对应一个不同的量纲，其值则表示对应量纲的幂次：

```
typedef int dimension[7]; // m  l  t  ...
dimension const mass      = {1, 0, 0, 0, 0, 0, 0};
dimension const length    = {0, 1, 0, 0, 0, 0, 0};
dimension const time      = {0, 0, 1, 0, 0, 0, 0};
...
```

根据这种表示法，力的量纲表示如下：

```
dimension const force = {1, 1, -2, 0, 0, 0, 0};
```

也就是mlt^{-2}。然而，如果我们想要将量纲融入到类型系统中去，这些数组就无法胜任了：它们的类型全都相同，都是dimension！而我们需要的是自身能够表示数值序列的类型，从而使得两个质量的类型是相同的，而质量和长度的类型则不同。

幸运的是，MPL为我们提供了一组表示类型序列（type sequences）的措施。例如，我们可以如下方式为内建的带符号整型（signed integral types）构建一个序列：

```
#include <boost/mpl/vector.hpp>

typedef boost::mpl::vector<
   signed char, short, int, long> signed_types;
```

那么，我们如何用类型序列来表示数量（numbers）呢？由数值型元函数传递和返回的类型是具有内嵌::value的外覆类型（wrapper types），所以数值序列其实是外覆类型的序列（这又是一个多态的例子）。为了使事情变得简单一些，MPL提供了int_<N>类模板，它以一个内嵌的::value来表现它的整型参数N：

```
#include <boost/mpl/int.hpp>

namespace mpl = boost::mpl; // namespace alias
static int const five = mpl::int_<5>::value;
```

⊖ 除法中的除数可以转换成指数为负的形式，例如可将1/x看成是x的－1次方。

命名空间别名（Namespace Aliases）

```
namespace alias = namespace-name;
```

将alias声明为namespace-name的同义词。本书中的许多例子使用mpl::来指示boost::mpl::，但忽略了对别名的声明。

事实上，MPL程序库包含了一整套整型常量外覆类（integral constant wrappers），如long_和bool_等，每一个外覆类（即类模板）都包装一个不同类型的整型常量。

现在，我们可以将基本量纲构建如下：

```
typedef mpl::vector<
   mpl::int_<1>, mpl::int_<0>, mpl::int_<0>, mpl::int_<0>
 , mpl::int_<0>, mpl::int_<0>, mpl::int_<0>
> mass;

typedef mpl::vector<
   mpl::int_<0>, mpl::int_<1>, mpl::int_<0>, mpl::int_<0>
 , mpl::int_<0>, mpl::int_<0>, mpl::int_<0>
> length;
...
```

你很快就会觉得这写起来实在太累人。更糟糕的是，这样的代码难于阅读和验证。代码的本质信息，也就是每个基本量纲的幂次，被埋藏在重复的语法“噪音”中。因此，MPL相应还提供了整型序列外覆类（integral sequence wrappers），它允许我们写出类似下面的代码：

```
#include <boost/mpl/vector_c.hpp>

typedef mpl::vector_c<int,1,0,0,0,0,0,0> mass;
typedef mpl::vector_c<int,0,1,0,0,0,0,0> length;  // 或position
typedef mpl::vector_c<int,0,0,1,0,0,0,0> time;
typedef mpl::vector_c<int,0,0,0,1,0,0,0> charge;
typedef mpl::vector_c<int,0,0,0,0,1,0,0> temperature;
typedef mpl::vector_c<int,0,0,0,0,0,1,0> intensity;
typedef mpl::vector_c<int,0,0,0,0,0,0,1> amount_of_substance;
```

你可以将这些mpl::vector_c特化与上面那些使用mpl::vector表示的较冗长的版本看成是等价的，尽管它们的类型并不相同。

如果我们愿意，还可以定义一些复合量纲（composite dimensions）：

```
// 基本量纲:          m l  t ...
typedef mpl::vector_c<int,0,1,-1,0,0,0,0> velocity;     // l/t
typedef mpl::vector_c<int,0,1,-2,0,0,0,0> acceleration; // l/(t²)
typedef mpl::vector_c<int,1,1,-1,0,0,0,0> momentum;     // ml/t
typedef mpl::vector_c<int,1,1,-2,0,0,0,0> force;        // ml/(t²)
```

顺带提一句，标量（scalar）[⊖]的量纲（例如pi）可以这样来描述：

⊖ 标量没有量纲。

```
typedef mpl::vector_c<int,0,0,0,0,0,0,0> scalar;
```

3.1.2 物理量的表示

上面所列的类型仍然是纯粹的元数据。要想对真实的计算进行类型检查，我们还需要以某种方式将元数据绑定到运行期数据。一个简单的数值外覆类，即模板参数为数值类型T以及T的量纲，刚好合适：

```
template <class T, class Dimensions>
struct quantity
{
    explicit quantity(T x)
       : m_value(x)
    {}

    T value() const { return m_value; }
 private:
    T m_value;
};
```

现在，我们有了将数值和量纲联系到一起的办法。例如，我们可以说：

```
quantity<float,length> l( 1.0f );
quantity<float,mass> m( 2.0f );
```

注意到在quantity的定义中并没有出现量纲的身影，它只在模板参数列表中出现过，其惟一的作用是确保l和m具有不同的类型。如此一来，我们就不可能错误地将长度赋给质量：

```
m = l;    // 编译期类型错误
```

3.1.3 实现加法和减法

因为参数的类型（量纲）必须是匹配的，所以我们现在可以轻易地写出加法和减法的规则：

```
template <class T, class D>
quantity<T,D>
operator+(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() + y.value());
}

template <class T, class D>
quantity<T,D>
operator-(quantity<T,D> x, quantity<T,D> y)
{
  return quantity<T,D>(x.value() - y.value());
}
```

这些运算符使我们可以写出类似下面的代码：

```
quantity<float,length> len1( 1.0f );
```

```
quantity<float,length> len2( 2.0f );

len1 = len1 + len2;    // OK
```

并阻止我们将量纲不同的量进行相加：

```
len1 = len2 + quantity<float,mass>( 3.7f ); // 错误
```

3.1.4 实现乘法

乘法比加法和减法复杂一些。到目前为止，参与运算的参数和结果的量纲都是一样的，但进行乘法运算时，结果的量纲往往与两个参数的量纲都不相同。对于乘法，以下表达式：

$$(x^a)(x^b) = x^{(a+b)}$$

意味着结果量纲的指数（exponent）为相应参数的量纲的指数和。除法与之类似，但是是指数差（difference）。

我们可以使用MPL的transform算法将两个序列中的对应元素进行加减法运算。transform是这样的一个元函数，它并行地迭代两个输入序列，将两个序列中每一个对应位置的元素传给一个任意的（用户提供的）二元元函数，并将结果存入一个输出序列中。

```
template <class Sequence1, class Sequence2, class BinaryOperation>
struct transform;  // 返回一个序列（Sequence）
```

如果你熟悉STL的transform算法的话，上面的struct transform的签名对于你可能并不陌生，STL的transform算法接收两个运行期输入序列：

```
template <
    class InputIterator1, class InputIterator2
  , class OutputIterator, class BinaryOperation
>
void transform(
    InputIterator1 start1, InputIterator1 finish1
  , InputIterator2 start2
  , OutputIterator result, BinaryOperation func);
```

现在我们只需要向mpl::transform传递一个用于对量纲进行乘除法（通过对两个序列的对应元素相加减）的BinaryOperation。如果你查看MPL的参考手册，你就会知道plus和minus两个元函数刚好可以满足要求：

```
#include <boost/static_assert.hpp>
#include <boost/mpl/plus.hpp>
#include <boost/mpl/int.hpp>
namespace mpl = boost::mpl;

BOOST_STATIC_ASSERT((
    mpl::plus<
        mpl::int_<2>
      , mpl::int_<3>
```

```
    >::type::value == 5
));
```

BOOST_STATIC_ASSERT

是一个宏（macro），如果其参数为false，则会导致一个编译期错误。双层括号是必要的，因为C++预处理器不能解析模板：如果不多加一对括号，那么它会将隔开类模板的逗号当成隔开宏参数的逗号，从而将条件表达式错误地解析为若干宏参数。不同于运行期的assert(...)，BOOST_STATIC_ASSERT也可以用在类作用域中，从而允许我们在元函数中放置断言(assertion)。第8章对此有更深入的讨论。

到目前为止，我们已经有了一个解决方案，像这样：

```
#include <boost/mpl/transform.hpp>

template <class T, class D1, class D2>
quantity<
    T
  , typename mpl::transform<D1,D2,mpl::plus>::type
>
operator*(quantity<T,D1> x, quantity<T,D2> y) { ... }
```

但是很抱歉，这还不够！如果你现在试图使用这个operator*，你会得到一个编译错误，原因在于你将mpl::plus直接传给了mpl::transform，而（MPL）规定元函数参数必须是类型，但mpl::plus却不是一个类型，而是一个类模板。所以我们必须通过某种方式让类似plus这样的元函数满足这种元数据模型。

在元函数和元数据之间引入多态（polymorphism）的一个自然的途径是使用外覆类惯用法(wrapper idiom)，在前面的代码中，这种惯用法曾在类型和整型常量之间引入了多态。现在，不是嵌入一个整型常量，我们将一个类模板嵌入一个所谓的元函数类中：

```
struct plus_f
{
    template <class T1, class T2>
    struct apply
    {
       typedef typename mpl::plus<T1,T2>::type type;
    };
};
```

定义

元函数类是这样的一种类：内嵌有一个可公开访问的名为apply的元函数。

虽然元函数是模板而非类型，但是元函数类却以一个普通的非模板类将其包覆起来，使其成为一个类型。因为元函数操作和返回的都是类型，所以元函数类也可被作为参数传递给另一个元函数，且元函数也可以返回一个元函数类。

最终，我们得到了一个plus_f元函数类，将它作为BinaryOperation传递给mpl::transform不会

导致编译错误：

```
template <class T, class D1, class D2>
quantity<
    T
  , typename mpl::transform<D1,D2,plus_f>::type // 新量纲
>
operator*(quantity<T,D1> x, quantity<T,D2> y)
{
    typedef typename mpl::transform<D1,D2,plus_f>::type dim;
    return quantity<T,dim>( x.value() * y.value() );
}
```

现在，如果我们希望计算一个5公斤重的膝上型电脑的重力（gravity），也就是说，将重力加速度（acceleration due to gravity，值为9.8 米/秒^2）乘以电脑的质量：

```
quantity<float,mass> m(5.0f);
quantity<float,acceleration> a(9.8f);
std::cout << "force = " << (m * a).value();
```

我们自定义的operator*会将这些运行期的值相乘（结果为49.0f），而元程序代码则会通过transform将表现基本量纲的元序列（meta-sequences）进行指数相加，因此结果类型包含一个新的指数列表，它表示一个新的量纲，如下：

```
vector_c<int,1,1,-2,0,0,0,0>
```

然而，如果我们试图写：

```
quantity<float,force> f = m * a;
```

就会遇到一些问题。尽管m*a的结果的确表示重力，其构成量纲的质量、长度、时间的指数分别为1、1、−2，然而transform返回的类型却并非vector_c特化。相反，transform处理它的输入元素，并以适当的元素构建一个新的序列：这个新序列和mpl::vector_c<int,1,1,-2,0,0,0,0>具有几乎相同的属性，但它们却是完全不同的C++类型。如果你想要知道新序列的全名，你可以尝试编译这个例子，然后查看错误信息，不过确切的细节并不重要，问题的关键在于force的类型和新序列的类型不同，所以赋值会失败。

为了解决这个问题，我们可以添加一个从乘法的结果类型到quantity<float,force>的隐式转换。由于我们无法预测介入计算的量纲的确切类型（从而也就无法预测计算的结果的量纲），所以这个转换不得不是模板形式的，如下：

```
template <class T, class Dimensions>
struct quantity
{
    // 转换构造器
    template <class OtherDimensions>
    quantity(quantity<T,OtherDimensions> const& rhs)
      : m_value(rhs.value())
    {
```

```
    }
    ...
```

不幸的是，这样的通用转换彻底违背了我们原来的意图，从而允许写出如下的愚蠢代码：

```
// m*a的结果应该是力（force），而非质量（mass）！

quantity<float,mass> bogus = m * a;
```

我们可以使用另一个MPL算法equal来解决这个问题，equal用于测试两个序列是否具有相同的一组元素：

```
template <class OtherDimensions>
quantity(quantity<T,OtherDimensions> const& rhs)
  : m_value(rhs.value())
{
  BOOST_STATIC_ASSERT((
     mpl::equal<Dimensions,OtherDimensions>::type::value
  ));
}
```

现在，如果两个物理量的量纲不匹配，该断言就会导致一个编译错误，从而及时阻止你的错误行为。

3.1.5 实现除法

除法和乘法类似，乘法将指数相加，而除法将指数相减。显然，做除法的minus_f完全可以按照plus_f的形式来写，但这里我们使用一个小技巧可以使minus_f更简单：

```
struct minus_f
{
    template <class T1, class T2>
    struct apply
      : mpl::minus<T1,T2> {};
};
```

这里，minus_f::apply使用继承将其基类mpl::minus的“type”内嵌类型暴露出来。这样我们就不必写：

```
typedef typename ...::type type
```

这里，我们不用在apply的基类mpl::minus<T1,T2>前加上typename（实际上加了反而是非法的），因为编译器知道在apply的基类列表中的依赖性的名字（dependent names）必定是基类[⊖]。这个强有力的简化代码的手法称为元函数转发（metafunction forwarding），后面还会频繁地用到它[⊜]。

⊖ 为了打消你的狐疑，可以明白地告诉你，plus_f可以如法炮制。鉴于这个技巧有点儿微妙，所以我们首先介绍那个理解起来简单但形式上罗嗦的形式化的实现方式。

⊜ 使用EDG-based编译器的用户应该参考附录C以便了解有关元函数转发的一个告诫。要想知道你手头有无EDG-based编译器，可以通过检查预处理器符号__EDG_VERSION__，因为所有EDG-based编译器都定义了该预处理符号。

尽管有这样的语法技巧来简化代码，但反复编写这些简单至极的外覆类来对现有的元函数进行包装，仍然会很快让人感到厌烦。虽然minus_f远没有plus_f那么臃肿，但你仍要为之键入一堆代码。幸运的是，MPL为我们提供了很简单的办法：我们用不着编写整个元函数类（如minus_f），而是“直接”将元函数传给算法。比如，我们可以这样调用mpl::transform：

```
typename mpl::transform<D1,D2, mpl::minus<_1,_2> >::type
```

看起来有点儿搞笑的那两个实参（_1和_2）称为占位符（placeholders），它们在这里的意思是：当transform的BinaryOperation被调用时，其第一、第二个参数会分别被传递到minus的_1和_2处。整个mpl::minus<_1,_2>类型则被称为占位符表达式（placeholder expression）。

> **注释**
>
> MPL的占位符位于mpl::placeholders命名空间内，定义在boost/mpl/placeholder.hpp文件中。在本书中，我们通常假定你已经写了如下代码：
>
> ```
> #include<boost/mpl/placeholders.hpp>
> using namespace mpl::placeholders;
> ```
>
> 这样，像_1、_2这样的占位符才能够不加修饰地进行访问。

使用占位符表达式实现的operator/如下：

```
template <class T, class D1, class D2>
quantity<
    T
  , typename mpl::transform<D1,D2,mpl::minus<_1,_2> >::type
>
operator/(quantity<T,D1> x, quantity<T,D2> y)
{
   typedef typename
     mpl::transform<D1,D2,mpl::minus<_1,_2> >::type dim;

   return quantity<T,dim>( x.value() / y.value() );
}
```

代码变得相当简洁了（因为如前述，不用额外定义一个minus_f）。我们还可以通过将计算新量纲的代码分解到一个单独的元函数中，从而进一步简化它：

```
template <class D1, class D2>
struct divide_dimensions
  : mpl::transform<D1,D2,mpl::minus<_1,_2> > // 又是元函数转发
{};

template <class T, class D1, class D2>
quantity<T, typename divide_dimensions<D1,D2>::type>
operator/(quantity<T,D1> x, quantity<T,D2> y)
{
   return quantity<T, typename divide_dimensions<D1,D2>::type>(
      x.value() / y.value());
}
```

现在我们可以验证膝上型计算机的重力（force-on-a-laptop）计算是否正确，即通过一个逆向的计算得到质量，然后将其与条件给出的计算机质量进行比较：

```
quantity<float,mass> m2 = f/a;
float rounding_error = std::abs((m2 - m).value());
```

如果一切正常，那么rounding_error应该非常接近0。这类计算虽令人厌烦，但如果它们出错则往往会破坏整个程序（甚至更糟）。如果我们将f/a错写成了a/f，就会得到一个编译错误，从而及时防止错误在整个程序中传播。

3.2 高阶元函数

在前面一节，我们使用了两种不同形式（元函数类和占位符表达式（placeholder expressions））来传递或返回元函数，就像传递和返回任何其他元数据那样。通过把元函数“塞进”一等元数据（first class metadata）中，允许transform执行各种不同的操作，对于我们的例子而言，则是量纲的乘除运算。尽管“使用函数去操纵其他函数”的思想可能看起来比较简单，但却具有非常强大的能力和灵活性[Hudak89]，因此赢得了一个好听的名字：高阶函数式编程（higher-order functional programming）。操纵其他函数的函数被称为高阶函数（higher-order function）。由此得出transform是一个高阶元函数（higher-order），即为一个操纵其他元函数的元函数。

我们既然见识了高阶元函数的强大能力，下面就将尝试创建新的高阶元函数。为了探究其底层机理，先来看一个简单的例子。我们的任务是编写一个名为twice的元函数，twice满足下面的条件：给它一个一元元函数 f和任意的元数据，它将做如下的计算：

$$twice(f, x) := f(f(x))$$

这个例子看上去没什么价值——它的确没有。在现实代码中twice没有多大用处。但是无论如何希望你能容忍我们拿它说事：twice包含了一个“高阶性（higher-orderness）”的所有必要元素，且无令人分神的任何细节，尽管它只是接收并调用了一个元函数。

如果f是个元函数类，那么twice的定义会很直观：

```
template <class F, class X>
struct twice
{
   typedef typename F::template apply<X>::type once;     // f(x)
   typedef typename F::template apply<once>::type type; // f(f(x))
};
```

或者使用元函数转发：

```
template <class F, class X>
struct twice
  : F::template apply<
       typename F::template apply<X>::type
    >
{};
```

C++语言注释

C++标准要求：当我们使用一个依赖性名字（dependent name）且该名字指的是一个成员模板时，我们必须使用模板关键字。F::apply不一定指的是个模板名字，其含义依赖于具体传递的F。关于模板，附录B提供了更多的信息。

每当使用元函数类时都要在apply前加上模板关键字无疑是个负担，若能减轻这种调用语法负担肯定是件好事。一如往常，解决方案是将这种使用模式分解到一个元函数中：

```
template <class UnaryMetaFunctionClass, class Arg>
struct apply1
  : UnaryMetaFunctionClass::template apply<Arg>
{};
```

现在，twice可以简化成这样：

```
template <class F, class X>
struct twice
  : apply1<F, typename apply1<F,X>::type>
{};
```

为了看看twice是如何工作的，可以将它应用到一个利用add_pointer 元函数构建的小型元函数类身上：

```
struct add_pointer_f
{
    template <class T>
    struct apply : boost::add_pointer<T> {};
};
```

现在我们可以使用twice和add_pointer_f来构建“指向指针的指针（pointers-to-pointers）”：

```
BOOST_STATIC_ASSERT((
    boost::is_same<
         twice<add_pointer_f, int>::type
       , int**
    >::value
));
```

3.3 处理占位符

虽然我们的twice实现品已经可以与元函数类一起工作了，但理想情况下，我们还要求它能够与占位符表达式（placeholder expression）一起工作，就像transform允许我们传递任一种形式一样。例如，我们希望能够写出这样的代码：

```
template <class X>
struct two_pointers
    : twice<boost::add_pointer<_1>, X>
{};
```

但是只要考察一下boost::add_pointer的实现就会发现，目前的twice定义显然不能这样工作：

```
template <class T>
struct add_pointer
{
    typedef T* type;
};
```

boost::add_pointer<_1>必须是个元函数类（就像add_pointer_f那样），才能够被twice调用。然而事实上它却是一个无参元函数，返回几乎毫无意义的_1*类型。所有试图使用two_pointers的企图都会失败，因为当apply1要求boost::add_pointer<_1>内嵌的::apply 元函数时，发现其根本不存在。

我们已经断定没有自动得到想要的行为，接下来该怎么办呢？想想看，既然mpl::transform可以做得到，那么我们应该也有办法做到，这不，下面就是。

3.3.1 lambda 元函数

我们可以使用MPL的lambda 元函数，由boost::add_pointer<_1>生成一个元函数类：

```
template <class X>
struct two_pointers
  : twice<typename mpl::lambda<boost::add_pointer<_1> >::type, X>
{};

BOOST_STATIC_ASSERT((
    boost::is_same<
        two_pointers<int>::type
      , int**
    >::value
));
```

后面我们将把add_pointer_f这样的元函数类或boost::add_pointer<_1>这样的占位符表达式（placeholder expressions）统称lambda表达式（lambda expressions）。该术语的含义是“匿名函数对象（unnamed function object）”，它是在20世纪30年代由逻辑学家Alonzo Church引入的，作为被他称为lambda计算（lambda-calculus）[⊖]的计算基础理论中的一部分。之所以使用lambda这个含义有点晦涩的名词，是出于它在函数式编程语言（functional programming languages）中建立的良好先例。

尽管mpl::lambda的首要用途是将占位符表达式转化为元函数类，然而它也可以接收任何lambda表达式，即使该表达式已经是个元函数类。在后一种情况，mpl::lambda原样返回其实参。MPL算法（如transform）在内部调用了mpl::lambda，然后再调用其返回（生成）的元函数类，所以它们和两种lambda表达式都相处得不错。我们可以将相同的策略应用到twice上：

```
template <class F, class X>
```

⊖ 访问http://en.wikipedia.org/wiki/Lambda_calculus以了解有关该主题的深入讨论，包括一个对Church的论文的参考。该论文证明lambda表达式（lambda expressions）的等价性通常是不可决定的。

```
struct twice
   : apply1<
         typename mpl::lambda<F>::type
       , typename apply1<
             typename mpl::lambda<F>::type
           , X
         >::type
     >
{};
```

现在我们可以将twice和元函数类或占位符表达式一起使用了：

```
int* x;

twice<add_pointer_f, int>::type          p = &x;
twice<boost::add_pointer<_1>, int>::type q = &x;
```

3.3.2 apply元函数

调用lambda返回的元函数类是如此常见的模式，以至于MPL提供了一个apply 元函数专门来做这件事。使用mpl::apply，我们的twice会变得更加灵活：

```
#include <boost/mpl/apply.hpp>

template <class F, class X>
struct twice
   : mpl::apply<F, typename mpl::apply<F,X>::type>
{};
```

你可以认为mpl::apply与我们写过的apply1 模板相同，但是mpl::apply还有两个额外的特性：

1. apply1只能操作元函数类，而mpl::apply的第一个参数可以是任意的lambda表达式（包括采用占位符构建的lambda表达式）。

2. apply1只能接收除元函数类之外的1个额外参数，并将这个参数传给元函数类。而mpl::apply可以接受1至5个额外的参数[⊖]，并用它们来调用元函数类。例如：

```
// 将二元的lambda表达式应用到另外两个参数上
mpl::apply<
    mpl::plus<_1,_2>
  , mpl::int_<6>
  , mpl::int_<7>
>::type::value // == 13
```

指导方针

如果你要在元函数中调用某个参数（译注：即将某个参数作为元函数类进行调用），请使用mpl::apply以确保该调用对于两种lambda表达式都是有效的。

⊖ MPL参考手册的Configuration Macros一节描述了如何改变mpl::apply所能接收的参数个数的上限。

3.4 lambda的其他能力

lambda表达式的能力并不止于使元函数成为可传递的参数。下面介绍的另外两种能力使lambda表达式成为几乎每个元编程任务中不可或缺的部分。

3.4.1 偏元函数应用

考虑lambda表达式mpl::plus<_1,_1>：单个的参数会被传递到plus的两个“_1”的位置，也就是说，将一个值与自身相加。因此，在这里，一个二元的元函数 plus被用来创建了一个一元的lambda表达式。换句话说，我们创建了一个全新的运算（译注：plus原先是做加法运算的，但plus<_1,_1>却是将一个值与自身相加，也就是“乘2”运算）！然而，还不止这些，通过将一个普通类型（非占位符，non-placeholder）绑定到plus的其中一个参数，我们可以创建一个一元lambda表达式，其作用是为它的参数加上一个固定值（如42）：

```
mpl::plus<_1, mpl::int_<42> >
```

在函数式编程（functional programming）世界中，将一组实参绑定到某个函数的形参的一个子集的过程被称为偏函数应用（partial function application）。

3.4.2 元函数复合

lambda表达式也可以用于从简单的元函数组装出更有趣的计算。例如，下面的表达式将两个数的和与差相乘（译注：即(a+b)*(a-b)），它是multiplies、plus和minus这三个元函数的复合体（composition）。

```
mpl::multiplies<mpl::plus<_1,_2>, mpl::minus<_1,_2> >
```

当对一个lambda表达式求值时，MPL会先检查它的各个参数以确定它们自身是否为lambda表达式，如果是，则先将它们求值，并将这些（本身为lambda表达式的）参数替换为求值的结果，然后再对外围的lambda表达式求值。

3.5 Lambda的细节

现在你对MPL的lambda设施的语义应该有了一个大致的了解，既然如此，让我们将已有的理解正式化，并且更深入考察一些东西。

3.5.1 占位符

“占位符（placeholder）”的定义可能会吓你一跳：

> **定义**
> 占位符是一个形式为mpl::arg<N>的元函数类。

实现

像_1、_2……_5这些名字不过是为了方便起见，其实它们都是mpl::arg的特化版本的typedefs，mpl::arg<N>的作用是选出（并返回）它的第N个参数[⊖]。占位符的实现像这样：

⊖ MPL默认提供了5个占位符。MPL参考手册的Configuration Macros一节描述了如何改变所提供的占位符的数目。

```
namespace boost { namespace mpl { namespace placeholders {

template <int N> struct arg; // 前置声明
struct void_;

template <>
struct arg<1>
{
    template <
     class A1, class A2 = void_, ... class Am = void_>
    struct apply
    {
     typedef A1 type; // 返回第一个参数
    };
};
typedef arg<1> _1;

template <>
struct arg<2>
{
    template <
      class A1, class A2, class A3 = void_, ...class Am = void_
    >
    struct apply
    {
        typedef A2 type; // 返回第二个参数
    };
};
typedef arg<2> _2;

其他特化版本和typedefs……

}}}
```

前面说过，调用元函数类就是调用其内嵌的::apply 元函数。当一个lambda表达式中的某个占位符被求值时，其实就是以该lambda表达式的实际参数来调用该占位符，然后该占位符会返回参数中的某一个。再后，求值（返回）的结果会替换lambda表达式中该占位符所“占”的位置。如此重复，直到所有的占位符都被替换成它们所表示的（实际的）参数。

匿名占位符

还有一种特殊的占位符称为匿名占位符（unnamed placeholder），其定义如下：

```
namespace boost { namespace mpl { namespace placeholders {

typedef arg<-1> _; // 匿名占位符

}}}
```

其实现细节并不重要。对于匿名占位符，你需要知道的就是：它是被特殊对待的。当一个lambda表达式被mpl::lambda转化为元函数类时，

在某个给定的模板特化中第n个出现的匿名占位符会被替换为_n。

例如，表3.1中的每一行都包含两个等价的lambda表达式：

表3.1 匿名占位符语义

`mpl::plus<_,_>`	`mpl::plus<_1,_2>`

`boost::is_same<`
`  _`
`  , boost::add_pointer<_>`
`>`
	`boost::is_same<`
	`  _1`
	`  , boost::add_pointer<_1>`
	`>`

`mpl::multiplies<`
`  mpl::plus<_,_>`
`  , mpl::minus<_,_>`
`>`
	`mpl::multiplies<`
	`  mpl::plus<_1,_2>`
	`  , mpl::minus<_1,_2>`
	`>`

尤其当用于简单的lambda表达式中时，匿名占位符通常可以消除足够的句法“噪音”，从而显著地提高可读性。

3.5.2 占位符表达式的定义

既然你已经了解占位符的含义了，那我们现在可以给出如下的占位符表达式（placeholder expression）定义：

> **定义**
>
> *一个占位符表达式是：*
>
> • *一个占位符。*
>
> *或者*
>
> • *一个至少有一个参数是占位符表达式的模板特化。*

换句话说，一个占位符表达式总是包含（至少）一个占位符。

3.5.3 Lambda和非元函数模板

关于占位符表达式，尚有一个尚未讨论的细节：为了使普通模板更容易融入元程序，MPL对其使用了特殊的规则。在所有占位符都被相应的实际参数替换后，如果作为结果的模板特化X并没有一个内嵌的::type，那么lambda表达式的结果就是X自身。

例如，mpl::apply<std::vector<_>, T>的结果始终都是std::vector<T>。如果不是由于这个行为，我们就不得不编写一个无趣的元函数用于在lambda表达式中创建普通模板特化了：

```
// 平凡的std::vector产生器
template<class U>
struct make_vector { typedef std::vector<U> type; };
typedef mpl::apply<make_vector<_>, T>::type vector_of_t;
```

由于有了这个特殊规则，我们现在可以简单地写：

```
typedef mpl::apply<std::vector<_>, T>::type vector_of_t;
```

3.5.4 “懒惰”的重要性

回顾前一章给出的always_int的定义：

```
struct always_int
{
    typedef int type;
};
```

无参元函数（Nullary metafunctions）乍看上去可能并不特别重要，因为像add_pointer<int>这样的东西在任何lambda表达式中出现的地方都可被替换为int*。然而，并非所有的无参元函数都是这样简单。例如：

```
struct add_pointer_f
{
    template <class T>
    struct apply : boost::add_pointer<T> {};
};
typedef mpl::vector<int, char*, double&> seq;
typedef mpl::transform<seq, boost::add_pointer<_> > calc_ptr_seq;
```

注意到calc_ptr_seq是个无参元函数，因为它有transform的内嵌::type。但是，对于一个C++模板，只有当我们试图“观察其内部”时，它才会被实例化。仅仅将calc_ptr_seq作为一个typedef名字并不会导致它被求值，因为我们并没有访问它内部的::type。

元函数接收了参数后仍可以被延迟调用。当一个元函数的结果只是被选择性地使用时，我们可以使用惰性评估（lazy evaluation，缓式评估）来减少编译时间。有时，通过命名一个无效的计算而并不实际去执行它，我们还可以避免扭曲程序结构。上面对calc_ptr_seq正是这么做的，原因在于double&*是非法类型。这种“惰性”及其优点将是本书中反复出现的主题。

3.6 细节

到目前为止，你对一般的模板元编程和Boost 元编程库的基本概念和语言应该有了一个相当全面的了解。本节将回顾其中的要点。

元函数转发

使用公有派生将基类元函数中内嵌的::type暴露给用户的技术。

元函数类

将编译期函数形式化（formulate）的最基本方法，由此，编译期函数可以被看做多态元数据（polymorphic metadata），也就是看成一个类型。元函数类是个内嵌有名为apply 元函数的类。

MPL

本书中大部分例子都用到了Boost 元编程库。正如Boost的type traits的头文件一样，MPL头

文件遵循一个简单的约定：

```
#include <boost/mpl/component-name.hpp>
```

然而，如果MPL的某个组件名以下划线结尾，那么对应的MPL头文件名就不包含最后的下划线。例如，mpl::bool_可以在<boost/mpl/bool.hpp>中找到。如果该程序库的哪些地方没有遵循这个约定，我们会为你指出来。

高阶函数（Higher-order function）

一种操作或返回函数的函数。利用其他元数据使元函数成为多态的是高阶元编程（higher-order metaprogramming）中的一个关键成分。

lambda表达式

简单地说，lambda表达式就是可调用的元数据。如果没有某种形式的可调用元数据，高阶元函数（higher-order metafunctions）也不会成为可能。lambda表达式有两个基本形式：元函数类和占位符表达式（placeholder expressions）。

占位符表达式

lambda表达式的一种。通过使用占位符达成偏元函数应用（partial metafunction application）和元函数复合（metafunction composition）。正如在本书中随处可见的那样，这些特性赋予了我们惊人的能力，允许我们从比较原始的元函数构造出任意复杂的类型计算——就在它被使用时：

```
// 确定满足以下条件的类型x在some_sequence中的位置
// x可被转换为int
// 且x不是char类型
// 且x不是浮点类型
typedef mpl::find_if<
      some_sequence
    , mpl::and_<
        boost::is_convertible<_1,int>
      , mpl::not_<boost::is_same<_1,char> >
      , mpl::not_<boost::is_float<_1> >
      >
    >::type iter;
```

占位符表达式使我们不必（为元函数）编写新的（外覆）元函数类，从而很好地实现了算法复用的目的。这种能力在STL的运行期世界里往往严重缺乏，因为倘若撇开标准算法的正确性和效率优势不谈，手写一个循环往往比使用标准算法容易得多。

lambda元函数

将lambda表达式转化为相应元函数类的元函数。要得到关于lambda和lambda求值过程的更为详细的信息，请参考MPL的参考手册。

apply元函数

一个元函数，其行为是：用其余参数去调用第一个参数，后者必须是个lambda表达式。通常，要调用一个lambda表达式，你应该总是将它以及调用它的参数传给mpl::apply，而不是“手工”使用mpl::lambda并调用其结果。

惰性评估（缓式评估，Lazy evaluation）

一种将计算推迟到其结果被要求时的策略，从而可以避免任何不必要的计算以及任何有关的不必要的错误。元函数仅仅在我们访问其内嵌的::type时才会被（真正）调用，所以我们可以在提供了其所有参数的同时却不作任何实质性的计算，将评估尽可能地延迟到最后一刻（或必要时）。

3.7 练习

3-0. 利用BOOST_STATIC_ASSERT为1.4.1节中展示的binary模板添加错误检查功能，使得若N为0和1以外的数字，则binary<N>::value会导致编译错误。

3-1. 使用transform将vector_c<int,1,2,3>变成带有元素(2,3,4)的类型序列。

3-2. 使用transform将vector_c<int,1,2,3>变成带有元素(1,4,9)的类型序列。

3-3. 使用twice两次，将T变成T****。

3-4. 对twice自身使用twice，将T变成T****。

3-5. 3.1节中展示的量纲分析代码仍然有一个问题。提示：当进行如下运算时会发生什么？

```
f = f + m * a;
```

使用本章介绍的技术修复该例子代码。

3-6. 构建一个功能与twice等价的lambda表达式。提示：mpl::apply是一个元函数！

3-7*. 以下各构造的语义是什么：

```
typedef mpl::lambda<mpl::lambda<_1> >::type t1;
typedef mpl::apply<_1,mpl::plus<_1,_2> >::type t2;
typedef mpl::apply<_1,std::vector<int> >::type t3;
typedef mpl::apply<_1,std::vector<_1> >::type t4;
typedef mpl::apply<mpl::lambda<_1>,std::vector<int> >::type t5;
typedef mpl::apply<mpl::lambda<_1>,std::vector<_1> >::type t6;
typedef mpl::apply<mpl::lambda<_1>,mpl::plus<_1,_2> >::type t7;
typedef mpl::apply<_1,mpl::lambda< mpl::plus<_1,_2> > >::type t8;
```

写出用于推出你的答案的步骤，并编写测试来验证你的假设。程序库的行为和你的推理一致吗？若不是，分析失败的测试可以揭示实际的表达式语义。解释为何你的假设是错误的，什么样的行为与你的假设更一致，并说明原因。

3-8*. 我们的量纲分析框架处理了量纲，但完全忽略了单位（units）问题。一个长度（length）可以采用英寸（inches）、英尺（feet）或公尺（meters）来表示。力（force）则采用牛顿（newtons）或kg m/sec^{2}来表示。向框架中添加指定单位的能力，并测试你的代码，使接口在语法上对用户尽可能友好。

第4章　整型外覆器和操作

正如早先提示的那样，MPL供应了一组外覆器模板，比如int_，用于将整数值制作成多态元数据。实际上MPL中的外覆器远比我们前面看到的多，在这一章中，我们将揭示它们的结构细节。我们还将探索一些对它们进行操作的元函数，并讨论如何最佳化地编写返回整型常量的元函数。

4.1　布尔外覆器和操作

bool不仅仅是最简单的整数类型，而且还是最有用的整型之一。如前面提到的那样，大多数type traits评估结果均为bool值，它们在很多元程序中发挥着重要作用。MPL针对bool值的类型外覆器以如下方式进行定义：

```
template< bool x > struct bool_
{
    static bool const value = x;          // 1
    typedef bool_<x> type;                // 2
    typedef bool value_type;              // 3
    operator bool() const { return x; } // 4
};
```

让我们逐行考察上面的注释代码行：

1. 现在这行代码不该有让你感到任何奇怪的地方。正如我们先前所说的那样，每一个整型常量外覆器都包含有一个::value成员。

2. 每一个整型常量外覆器都是一个返回其自身的无参元函数。你很快就会明白有关此项设计决策的原因。

3. 外覆器的::value_type指明其::value的（cv修饰的）类型。

4. 每一个bool_<x>特化都可以非常自然地转化为一个值为x的布尔值（bool）。

这个程序库还供应了两个方便的typedefs：

```
typedef bool_<false> false_;
typedef bool_<true> true_;
```

4.1.1　类型选择

到目前为止，我们只是在编译期通过将决策嵌入专门的类模板特化而做出决策：递归算法（比如我们在第一章编写的binary模板）的终结条件说“如果实参是0，就以这种方式计算结果，否则采用另一种方式（默认方式）计算结果”我们还对iter_swap_impl进行特化，以便选择在iter_swap内部的两种实现之一：

```
iter_swap_impl<use_swap>::do_it(i1,i2);
```

不是专门针对我们所做的每一个选择手工编写一个模板，而是可以利用一个可以作出决策的MPL 元函数：如果C::value是true，则 mpl::if_<C,T,F>::type是T，否则就是F。回到iter_swap例子，我们现在可以使用有助于记忆的名字的类，来代替一个iter_swap_impl 模板：

```
#include <boost/mpl/if.hpp>

struct fast_swap
{
    template <class ForwardIterator1, class ForwardIterator2>
    static void do_it(ForwardIterator1 i1, ForwardIterator2 i2)
    {
        std::swap(*i1, *i2);
    }
};

struct reliable_swap
{
    template <class ForwardIterator1, class ForwardIterator2>
    static void do_it(ForwardIterator1 i1, ForwardIterator2 i2)
    {
        typename
          std::iterator_traits<ForwardIterator1>::value_type
        tmp = *i1;
        *i1 = *i2;
        *i2 = tmp;
    }
};
```

iter_swap中调用iter_swap_impl的do_it成员的那行代码被改写如下：

```
mpl::if_<
    mpl::bool_<use_swap>
  , fast_swap
  , reliable_swap
>::type::do_it(i1,i2);
```

这好像并没有多大的改进：复杂性从iter_swap_impl的定义中移到了iter_swap本体中。然而，这使得代码更清晰，因为这样做就将选择一个iter_swap实现品的逻辑并保持在它的定义内部。

另外一个例子。让我们看看可以如何优化泛型代码中的函数参数传递问题。通常而言，一个实参类型的复制构造器可能是代价高昂的，因此泛型函数应该按引用接收参数。另一方面，通常按引用来传递像标量类型（scalar type）这样琐细的东西是浪费的：在一些编译器上，标量是按值进行传递的，但如果按引用传递，它们就被迫配置在栈上。这就呼唤一个元函数的闪亮登场：param_type<T>，当它是一个标量时，返回T，否则返回T const&。

我们可以用如下方式来使用它：

```
template <class T>
class holder
{
 public:
    holder(typename param_type<T>::type x);
    ...
 private:
    T x;
};
```

holder<int>构造器的参数类型是int，而holder<std:: vector<int> >的构造器则接收一个std::vector<int> const&类型的参数。为了实现param_type，我们可以如下方式使用mpl::if_：

```
#include <boost/mpl/if.hpp>
#include <boost/type_traits/is_scalar.hpp>
template <class T>
struct param_type
  : mpl::if_<
        typename boost::is_scalar<T>::type
      , T
      , T const&
    >
{};
```

遗憾的是，这个实现会阻止我们将引用类型放入holder中：由于形成一个双重引用是非法的，所以实例化holder<int&>会导致错误。Boost.Type Traits提供了一个迂回解决方式，我们可以针对一个引用类型实例化add_reference<T>——在这种情况下它只是返回其实参而已：

```
#include <boost/mpl/if.hpp>
#include <boost/type_traits/add_reference.hpp>

template <class T>
struct param_type
  : mpl::if_<
        typename boost::is_scalar<T>::type
      , T
      , typename boost::add_reference<T const>::type
    >
{};
```

4.1.2 缓式类型选择

上述方式不能令人完全满意，因为它会导致即使T是一个标量，add_reference<T const>也会被实例化，从而浪费编译时间。将计算延缓到绝对需要时称做缓式评估（lazy evaluation）。一些函数式编程语言（functional programming languages），例如Haskell，对每一个计算都采用缓式

评估机制，无需特别的提示。在C++中，我们需要显式地进行缓式评估。延缓add_reference的实例化直到其需要时的方式之一，是使得mpl::if_可以从两个无参元函数中选择一个，然后调用被选择的那一个：

```
#include <boost/mpl/if.hpp>
#include <boost/mpl/identity.hpp>
#include <boost/type_traits/add_reference.hpp>

template <class T>
struct param_type
  : mpl::if_<              // 转发给选定的转换操作
        typename boost::is_scalar<T>::type
      , mpl::identity<T>
      , boost::add_reference<T const>
    >::type
{};
```

注意我们对mpl::identity的使用，这是一个简单地返回其实参的元函数。现在param_type<T>根据T是否为一个标量，返回调用mpl::identity<T>或boost:: add_reference<T const>二者之一的结果。

由于这个惯用法在元程序中是如此常见，乃至MPL提供了一个名为eval_if的元函数，以如下方式定义：

```
template <class C, class TrueMetafunc, class FalseMetafunc>
struct eval_if
  : mpl::if_<C,TrueMetafunc,FalseMetafunc>::type
{};
```

由于if_基于一个条件返回两个实参之一，所以eval_if也基于一个条件调用两个无参元函数实参之一并返回结果。我们现在可以通过直接转发给eval_if稍微简化param_type的定义：

```
#include <boost/mpl/eval_if.hpp>
#include <boost/mpl/identity.hpp>
#include <boost/type_traits/add_reference.hpp>

template <class T>
struct param_type
  : mpl::eval_if<
        typename boost::is_scalar<T>::type
      , mpl::identity<T>
      , boost::add_reference<T const>
    >      // 这儿不再有::type
{};
```

由于所有Boost整型元函数都提供了一个嵌套的::value，它们的特化是有效的整型类型外覆器，并且可被直接传递给mpl::if_，从而带来了另一个简化：

```
template <class T>
```

```
struct param_type
  : mpl::eval_if<
        boost::is_scalar<T>
      , mpl::identity<T>
      , boost::add_reference<T const>
    >
{};
```

返回整型常量外覆器的Boost元函数的特化（例如is_scalar），（恰好）公有派生于那些非常一致的外覆器。结果，这些元函数的特化自身不仅仅是有效的整型常量外覆器，它们还继承了上面为bool_这样的外覆器所勾勒的所有有用的属性：

```
if (boost::is_scalar<X>()) // 调用继承的operator bool()
{
    // 当且仅当X是一个标量类型时，此处的代码才执行
}
```

4.1.3 逻辑运算符

让我们试想一下，我们并没有一个可支配的如此智能的add_reference。如果add_reference只被定义为如下所示，我们将不能靠它来避免形成双重引用：

```
template <class T>
struct add_reference { typedef T& type; };
```

如果是这样的话，我们希望对param_type作一些修改，以避免向add_reference传递引用：

```
template <class T>
struct param_type
  : mpl::eval_if<
        mpl::bool_<
          boost::is_scalar<T>::value
          || boost::is_reference<T>::value
        >
      , mpl::identity<T>
      , add_reference<T const>
    >
{};
```

相当丑陋，不是吗？这种做法使得我们先前版本的语法清洁感大都失去了。如果希望为param_type构建一个lambda表达式（on-the-fly），而不是编写一个新的元函数，我们甚至会碰到更糟糕的问题：

```
typedef mpl::vector<int, long, std::string> argument_types;

// 为实参类型构建一个参数类型列表
typedef mpl::transform<
    argument_types
```

```
    , mpl::if_<
          mpl::bool_<
            boost::is_scalar<_1>::value
            || boost::is_reference<_1>::value
          >
        , mpl::identity<_1>
        , add_reference<boost::add_const<_1> >
      >
>::type param_types;
```

这里就不仅仅是丑陋的问题了，它实际上根本不能正确地工作。因为触及（访问/使用）一个模板的嵌套的::value会强迫该模板被实例化，因此逻辑表达式boost::is_scalar<_1>::value || is_reference<_1>::value被立即评估并且只测试占位符类型_1自身的属性。由于_1既不是一个标量也不是一个引用，因此结果为false，并且我们的lambda表达式等价于add_reference <boost::add_const<_1> >。我们可以利用MPL的逻辑运算符元函数来解决这些问题。使用mpl::or_，我们可以重新获得最初的param_type的语法清洁性：

```
#include <boost/mpl/or.hpp>

template <class T>
struct param_type
  : mpl::eval_if<
        mpl::or_<boost::is_scalar<T>, boost::is_reference<T> >
      , mpl::identity<T>
      , add_reference<T const>
    >
{};
```

因为mpl::or_<x,y>派生于它的结果::type（bool_<n>的一个特化），因此它自身是一个有效的MPL逻辑常量外覆器，我们现在能够完全消除对bool_的显式使用，并且避免访问嵌套的::type。撇开我们不使用运算符记号的事实不谈，代码的可读性确实比以前更好了。

如果我们对lambda表达式进行同样的修改，可以带来类似的好处，而且它可以正确地工作：

```
typedef mpl::transform<
    argument_types
  , mpl::if_<
        mpl::or_<boost::is_scalar<_1>, boost::is_reference<_1> >
      , mpl::identity<_1>
      , add_reference<boost::add_const<_1> >
    >
>::type param_types;
```

如果我们希望对param_type加以修改，使得除了标量外，还可以按值传递所有无状态（stateless）的类类型，该怎么做？我们只要嵌套另一个对or_的调用即可：

```
# include <boost/type_traits/is_stateless.hpp>
```

```
template <class T>
struct param_type
  : mpl::eval_if<
        mpl::or_<
            boost::is_stateless<T>
          , mpl::or_<
                boost::is_scalar<T>
              , boost::is_reference<T>
            >
        >
      , mpl::identity<T>
      , add_reference<T const>
    >
{};
```

尽管这已经可以工作，但我们还可以做得更好。因为mpl::or_可以接收2个到5个实参，因此我们可以写：

```
# include <boost/type_traits/is_stateless.hpp>

template <class T>
struct param_type
  : mpl::eval_if<
        mpl::or_<
            boost::is_scalar<T>
          , boost::is_stateless<T>
          , boost::is_reference<T>
        >
      , mpl::identity<T>
      , add_reference<T const>
    >
{};
```

实际上，大多数操作整型实参的MPL元函数（例如mpl:: plus<...>）都具有同样的属性。

程序库包含有一个类似的and_ 元函数，以及一个用于转置布尔条件的一元not_ 元函数[⊖]。值得指出的是，正如内建的&&和||运算符一样，mpl::and_和mpl::or_展示出了“短路”行为。例如，在上面的例子中，如果T是一个标量，那么boost::is_stateless<T>和is_reference<T>永远都不会被实例化。

4.2 整数外覆器和运算

在量纲分析例子（参见3.1节）中，我们已经使用了MPL的int_外覆器。现在我们可以更细

⊖ 这些名字都以下划线结尾，因为and、or和not是C++关键字，其功能是用做更广为人知的运算符记号&&、||以及!的别名。

致地检视它，从它的定义入手：

```
template< int N >
struct int_
{
    static const int value = N;
    typedef int_<N> type;

    typedef int value_type;

    typedef mpl::int_<N+1> next;
    typedef mpl::int_<N-1> prior;
    operator int() const { return N; }
};
```

正如你看到的那样，int_ 酷似bool_，事实上，主要的区别仅在于它多了::next和::prior成员。本章后面我们将解释它们的用途。程序库还提供了针对long和std::size_t的类似的数值外覆器，分别命名为long_和size_t。

为了表示任何其他整数类型的值，程序库提供了一个泛型外覆器，定义如下：

```
template<class T, T N>
struct integral_c
{
    static const T value = N;
    typedef integral_c<T,N> type;

    typedef T value_type;

    typedef mpl::integral_c<T,N+1> next;
    typedef mpl::integral_c<T,N-1> prior;
    operator T() const { return N; }
};
```

整型序列外覆器，就像我们实现量纲分析（见第3章）的vector_c模板那样，带有一个初始的类型参数T，用于形成它们所包含的integral_c<T, ...> 特化。

如果int_<...>和integral_c<int,...>的并存使你皱起眉头，我们很难说你什么。毕竟，两个其他方面等价的整数外覆器可能是不同的类型。如果我们试图以这种方式比较两个整数外覆器：

```
boost::is_same<mpl::integral_c<int,3>, mpl::int_<3> >::value
```

结果（false）可能有点让人惊讶。下面这个比较的结果也是false可能不那么让人感到惊讶：

```
boost::is_same<mpl::long_<3>, mpl::int_<3> >::value
```

然而，不论你对这两个例子的反应是什么，到目前为止，有一点应该很清楚，整型常量外覆器的值等价性（value equality）做的事情要比简单的类型匹配多。用于测试值的等价性的MPL 元函数称为equal_to，其定义也很简单：

```
template<class N1, class N2>
struct equal_to
  : mpl::bool_<(N1::value == N2::value)>
{};
```

不要将equal_to和equal弄混淆了，这一点很重要，后者比较两个序列中的元素。这两个元函数的名字取自STL中的类似组件。

4.2.1 整型运算符

MPL供应了一整套元函数来操作整型常量外覆器，你已经看到了一些（例如plus和minus）。在我们深入细节之前，我说一下命名约定的问题：当元函数对应于一个内建的C++运算符且语言提供了一个文本性的替代标记名字（例如&&/and）时，MPL 元函数被命名为替代的标记后加一个下划线（例如mpl::and_）。否则，MPL元函数则取相应的STL函数对象的名字，例如mpl::equal_to。

这些运算符可以分成4个组。在下面的表格中，默认n=5。参见MPL参考手册中的Configuration Macros一节，了解有关如何修改n的信息。

评估为布尔值的运算符

这一组中的元函数都具有bool常量结果。我们已经讨论过逻辑运算符，这儿列出它们只是出于完整性的考虑（见表4.1）。

表4.1 逻辑运算符

元函数特化	::value和::type::value
not_<X>	!X::value
and_<T1,T2,...Tn>	T1::value && ... Tn ::value
or_<T1,T2,...Tn>	T1::value || ... Tn ::value

表4.2列出了值比较运算符。

表4.2 值比较运算符

元函数特化	::value和::type::value
equal_to<X,Y>	X::value == Y::value
not_equal_to<X,Y>	X::value != Y::value
greater<X,Y>	X::value > Y::value
greater_equal<X,Y>	X::value >= Y::value
less<X,Y>	X::value < Y::value
less_equal<X,Y>	X::value <= Y::value

评估为整型值的运算符

这一组运算符都有一个整型常量结果，其类型与它们所评估的表达式的类型相同（参见表4.3和表4.4）。换句话说，因为3+2L的type是long，因此

```
mpl::plus<mpl::int_<3>, mpl::long_<2> >::type::value_type
```

也是long。

表4.3 位运算符（Bitwise Operators）

元函数特化	::value和::type::value
bitand_<X,Y>	X::value & Y::value
bitor_<X,Y>	X::value | Y::value
bitxor_<X,Y>	X::value ^ Y::value

表4.4 算术运算符

元函数特化	::value和::type::value
divides<T1,T2,...Tn>	T1::value / ... Tn ::value
minus<T1,T2,...Tn>	T1::value - ... Tn ::value
multiplies<T1,T2,...Tn>	T1::value * ... Tn ::value
plus<T1,T2,...Tn>	T1::value + ... Tn ::value
modulus<X,Y>	X::value % Y::value
shift_left<X,Y>	X::value << Y::value
shift_right<X,Y>	X::value >> Y::value
next<X>	X::next
prior<X>	X::prior

next和prior 元函数有点类似于C++一元运算符++和--。然而，由于元数据是不可变的(immutable)，所以next和prior不能修改它们的实参。事实上，mpl::next和mpl::prior正好类似于两个运行期函数，这两个函数声明于boost命名空间中，它们只是简单地返回它们的实参递增后或递减后的版本：

```
namespace boost
{
  template <class T>
  inline T next(T x) { return ++x; }
  template <class T>
  inline T prior(T x) { return --x; }
}
```

你可能对mpl::next<X>和mpl::prior<X>没有分别被定义为返回针对X::value+1和X::value-1的外覆器而感到奇怪，即便当它们用在整型常量外覆器身上时是以那种方式发挥作用的。在下一章中，当我们讨论用于序列迭代的next和prior时会提示原因。

4.2.2 _c整型速记法

有时我们发现身处这样的一种境况中：显式构建外覆器类型的需要变成了一种不方便。这在我们的量纲分析代码（第3章）中就曾发生过，那儿使用mpl::vector_c<int, ...>而不是mpl::vector<...>来消除为7个基础单位编写int_特化的需要。

当在本章前面工作于param_type 元函数时，我们实际上回避了另一个这种情形。在mpl::or_来拯救我们之前，我们被这个丑陋的定义纠缠住：

```
template <class T>
```

```
struct param_type
  : mpl::eval_if<
        mpl::bool_<
          boost::is_scalar<T>::value
          || boost::is_reference<T>::value
        >
      , mpl::identity<T>
      , add_reference<T const>
    >
{};
```

利用MPL的eval_if_c（也由<boost/mpl/eval_if.hpp>供应），我们可以这样写：

```
template <class T>
struct param_type
  : mpl::eval_if_c<
        boost::is_scalar<T>::value
        || boost::is_reference<T>::value
      , mpl::identity<T>
      , add_reference<T const>
    >
{};
```

到现在你或许已经开始注意到在_c的使用中的共性了，它总是作为这样的模板后缀：带有原始整型常量——而不是外覆器——作为参数。_c后缀可被认为是“constant”或“of integral constants”的缩写。

4.3 练习

4-0. 为mpl::or_和mpl::and_ 元函数编写测试例子，要利用它们的短路行为（short-circuit behavior）。

4-1. 实现名为logical_or和logical_and的二元元函数，它们相应地模仿mpl::or_和mpl::and_的行为。使用练习4-0的测试例子来验证你的实现。

4-2. 扩充练习4-1中的logical_or和logical_and 元函数实现，使其可以接收多至5个实参。

4-3. 消除以下代码片段中不必要的实例化：

1.
```
template< typename N, typename Predicate >
struct next_if
  : mpl::if_<
        typename mpl::apply<Predicate,N>::type
      , typename mpl::next<N>::type
      , N
    >
{};
```

2.
```
template< typename N1, typename N2 >
struct formula
```

```
    : mpl::if_<
          mpl::not_equal_to<N1,N2>
        , typename mpl::if_<
              mpl::greater<N1,N2>
            , typename mpl::minus<N1,N2>::type
            , N1
          >::type
        , typename mpl::plus<
              N1
              , typename mpl::multiplies<N1,
                                         mpl::int_<2> >::type
      >::type
   >::type
  {};
```

编写测试例子来验证转化过的元函数的语义保持不变。

4-4. 使用整型运算符和type traits程序库设施实现以下组合traits：

```
is_data_member_pointer
is_pointer_to_function
is_reference_to_function_pointer
is_reference_to_non_const
```

4-5. 考虑如下函数模板，它被设计用于为std::find提供一个“基于容器的”（和基于迭代器的相对）的接口：

```
template <class Container, class Value>
typename Container::iterator
container_find(Container& c, Value const& v)
{
    return std::find(c.begin(), c.end(), v);
}
```

正如代码所写的那样，container_find不能工作于const容器的情形，对某些容器类型X而言，Container将被推导为const X，但是当我们尝试将std::find返回的Container::const_iterator转换为一个Container::iterator时，编译将会失败。通过使用一个小的元程序来计算container_find的返回类型从而修复这个问题。

第5章　序列与迭代器

如果把STL描述成一个基于运行期算法、函数对象和迭代器的框架，那我们可以说MPL是建立于编译期算法、元函数、序列和迭代器之上的[⊖]。

我们在第3章中非正式地使用了序列和算法来实现量纲分析逻辑。如果你熟悉STL，也许已经猜到在幕后我们也使用了迭代器。然而，程序库到目前为止允许我们愉快地忽视它们的作用，这是借助于其基于序列的算法接口达成的。

在本章中，你将会获得对“编译期STL”的一般理解，并进而形式化（formalize）序列和迭代器的概念，研究它们和算法的互动，考察程序库提供的许多具体实例，并学习如何为它们实现新的范本。

5.1　Concepts

首先我们定义一个重要的术语，该术语源自运行期泛型编程世界，此术语就是concept。一个concept就是一个泛型组件对置于其一个或多个实参上的要求的描述。在本书中我们已经讨论了一些concepts。例如，我们在第3章中编写的apply1 元函数要求第一个参数必须是一个元函数类。

满足一个concept的要求的一个或一组类型被称为模塑（model）了该concept，或称为该concept的一个model。因此plus_f（同样来自于第3章）是元函数类的一个model。一个concept被称为精化（refine）另一个concept——当它的条件要求是另一个concept的条件要求的超集（superset）时。

Concept条件要求（Concept requirements）通常来自于以下范畴（category）。

有效的表达式：必须针对“涉入表达式的对象”编译成功的C++表达式被认为是concept的models。例如，一个Iterator x被预期支持++x和*x表达式。

关联类型：参与到一个或多个有效表达式中，而且可从模塑（modeling）了concept的type(s)计算出来的类型，被称为关联类型（associated types）。通常来说，关联类型可以通过嵌套在用于模型类型的类定义中的typedefs来访问，也可以通过一个traits类来访问。例如，在第2章中所描述的，一个迭代器的值类型通过std::iterator_traits和迭代器相关联。

不变式（Invariants）：不变式是指一个model的实例的运行期特征必须总是为true，换句话说，对一个实例的所有操作必须保持这些特征。不变式通常表现为前条件（pre-conditions）和后条件（post-conditions）的形式。例如，在一个前向迭代器（Forward Iterator）被复制后，复制后的和原始的那份前向迭代器的比较操作必须是相等的。

⊖ 尽管在日常编程中STL容器不可或缺，但它们并非STL概念框架的基础部分，并且它们不和其他STL抽象直接互动。与之相对的是，MPL的序列在其算法接口中起到了直接的作用。

复杂度保证：有效表达式之一的执行需要花费的时间的最大限度，或者它的计算所要使用的各种资源的“数量”的最大限度。例如，对一个迭代器（Iterator）的递增操作，要求具有常量复杂度。

在这一章中，我们将介绍几个新的concepts和对关联类型的精化关系，以及复杂度保证。

5.2 序列和算法

MPL中的大多数算法操作于序列之上。例如，在一个vector中搜索一个类型看起来如下：

```
typedef mpl::vector<char,short,int,long,float,double> types;

// 在types中定位long的位置
typedef mpl::find<types, long>::type long_pos;
```

在这里，find接收两个参数：一个被查找的序列（types），一个被查找的类型（long），并返回一个迭代码，用于指示序列中第一个和long一致的元素的位置。除了mpl::find带有单个序列参数而不是两个迭代器这一事实外，这和你在一个std::list或std::vector中搜索一个值（value）没什么两样：

```
std::vector<int> x(10);
std::vector<int>::iterator five_pos
     = std::find(x.begin(), x.end(), 5);
```

如果不存在匹配元素，mpl::find就返回该序列“超过最后一个元素一位”的那个迭代器，这可以通过mpl::end 元函数相当自然地进行访问：

```
// 断言在序列中发现了long
typedef mpl::end<types>::type finish;
BOOST_STATIC_ASSERT((!boost::is_same<long_pos, finish>::value));
```

一个类似的begin 元函数则返回一个表示序列开头的迭代器。

5.3 迭代器

与STL迭代器一样，MPL迭代器提供的最基础的服务也是访问它们所指向（refer）的序列元素。为了解引用（dereference）一个编译期迭代器，我们不可以简单地应用前缀* operator，因为运行期运算符重载不能用于编译期。取而代之的是，MPL为我们提供了一个有着适当命名的元函数：deref，它接收一个迭代器，并返回所指向（referenced）的元素。

```
typedef mpl::vector<char,short,int,long,float,double> types;

// 在types中定位long的位置
typedef mpl::find<types,long>::type long_pos;

// 对迭代器进行解引用操作
typedef mpl::deref<long_pos>::type x;
// 检查我们是否获得了预期的结果
```

```
BOOST_STATIC_ASSERT((boost::is_same<x,long>::value));
```

一个迭代器还可提供对序列中邻接位置的访问（或遍历，traversal）。在第4章中，我们描述了mpl::next和mpl::prior 元函数，这二者产生它们整型实参的一个增量或减量副本。这些原语同样可以很好地适用于迭代器：

```
typedef mpl::next<long_pos>::type float_pos;
BOOST_STATIC_ASSERT((
    boost::is_same<
        mpl::deref<float_pos>::type
      , float
     >::value
));
```

5.4 迭代器Concepts

在这一节中，我们将定义MPL迭代器concepts。如果你熟悉STL迭代器，你可能注意到MPL迭代器和STL迭代器之间的相似性。当然，它们之间也存在一些差别，这是C++元数据的常性（immutable nature）所导致的直接后果。例如，在MPL中没有单独的输入迭代器和输出迭代器范畴。在后面的讨论中，当我们碰到这些迭代器时再指出这些类似性和差别，以及所有迭代器都具有的一些关键属性（将以粗体字标明）。

就像STL的基础迭代器操作是O(1)复杂度一样（在运行期），本章详述的MPL的基础迭代器操作也是O(1)复杂度（在编译期）[⊖]。

5.4.1 前向迭代器

前向迭代器是最简单的MPL迭代器范畴，它只有三种操作：前向遍历、元素访问和范畴侦测。一个MPL迭代器可以是可递增的（incrementable）和可解引用的（dereferenceable），也可以是逾尾的（past-the-end）。这两种状态是互斥的，因为没有任何迭代器操作可被允许施行在一个逾尾的迭代器上。

由于MPL迭代器是常性的（immutable）的，我们无法像对STL迭代器那样就地（in place）递增它们。取而代之的是，我们将它们传递给mpl::next，后者产生序列中的下一个位置。一个可递增的迭代器的作者可以特化mpl::next来支持他的迭代器类型，或者也可以简单地利用其默认操作，后者就是访问迭代器的::next成员：

```
namespace boost { namespace mpl {
   template <class It> struct next
   {
      typedef typename It::next type;
   };
}}
```

⊖ 在本书中，我们根据所需的模板实例化（template instantiations）数目来度量一个操作的编译期复杂度。当然还有其他一些因素会影响编译程序所需的时间，参见附录C，以便了解更多的细节。

一个可解引用的迭代器支持通过mpl::deref元函数来进行元素访问，与刚才的情况类似，mpl::deref默认实现是访问迭代器的嵌套::type：

```
namespace boost { namespace mpl {
   template <class It> struct deref
   {
      typedef typename It::type type;
   };
}}
```

为了检查迭代器的相等性（equivalence），我们可以使用来自Boost Type Traits程序库的boost::is_same 元函数。只有当两个迭代器具有相同的类型时，它们才是相等的。由于is_same可以工作于任何类型中，因此它同样适用于逾尾迭代器。一个迭代器q可从一个迭代器p获得的条件是：它们是等价的，或者存在某个序列：

```
typedef mpl::next<p>::type p1;
typedef mpl::next<p1>::type p2;
.
.
.
typedef mpl::next<pn-1>::type pn;
```

这样pn是等价于q的。我们将使用“半开区间（half-open range）”记号[p,q)来表示一个区间范围的序列元素，起点为mpl::deref<p>::type，终点为mpl:: deref<pn-1>::type。

表5.1列出了MPL前向迭代器的条件要求，其中p是前向迭代器的一个model。

表5.1 前向迭代器条件要求

表达式	结果	前条件
mpl::next<p>::type	一个前向迭代器	p是可递增的
mpl::deref<p>::type	任何类型.	p是可解引用的
p::category	可被转换为mpl::forward_iterator_tag.	?

5.4.2 双向迭代器

双向迭代器是带有反向遍历序列能力的前向迭代器。一个双向迭代器或者是可递减的，或者指向（refers to）其序列的起点。

给定一个可递减的迭代器，mpl::prior 元函数产生序列中的前一个位置。一个可递减的迭代器的作者可以特化mpl::prior以支持他自己的迭代器类型，或者，也可以简单地利用其默认操作，该操作就是访问迭代器的::prior成员：

```
namespace boost { namespace mpl {
   template <class It> struct prior
   {
      typedef typename It::prior type;
   };
}}
```

表5.2列出了MPL双向迭代器的附加要求，其中p是双向迭代器的一个model。

表5.2 双向迭代器的附加要求

表达式	结果	断言/前条件
`mpl::next<p>::type`	一个双向迭代器	`mpl::prior<mpl::next<p>::type>::type` 等价于p 前条件：p是可递增的
`mpl::prior<p>::type`	一个双向迭代器	前条件：p是可递减的
`p::category`	可被转换为`mpl::bidirectional_iterator_tag`	

5.4.3 随机访问迭代器

随机访问迭代器是这样一种双向迭代器：它还提供了向前或向后移动任意数量的位置操作，以及对同一个序列中的两个迭代器之间的距离进行度量操作，这些操作都具有常量时间的复杂度。

随机访问遍历是使用mpl::advance 元函数而达成的，对于这个元函数来说，给定一个随机访问迭代器p和一个整型常量类型n，并返回同一个序列中的一个步进后的迭代器（advanced iterator）。距离度量则是通过mpl::distance 元函数获得的，对于这个元函数来说，给定同一个序列中的两个随机访问迭代器p和q，返回p和q之间的位置数目。注意，这两个操作之间具有紧密的联系，以下表达式等价于q：

```
mpl::advance<p, mpl::distance<p,q>::type>::type
```

这两种操作均具有常量时间复杂度。

就像STL中的同名函数一样，advance和distance实际上也适用于双向和前向迭代器，然而后两者的情形具有线性的时间复杂度：默认实现只是按需多次调用mpl::next或mpl::prior来完成工作，从而导致随机访问迭代器的作者必须针对他的迭代器来特化advance和distance，以获得常量时间复杂度的效果，否则将无法满足随机访问迭代器的条件要求。

表5.3列出了MPL随机访问迭代器的附加要求。其中p和q表示同一个序列中的两个迭代器，N表示一个整型常量类型，n则是N::value。

表5.3 随机访问迭代器的附加要求

表达式	结果	断言/前条件
`mpl::next<p>::type`	一个随机访问迭代器	前条件：p是可递增的
`mpl::prior<p>::type`	一个随机访问迭代器	前条件：p是可递减的

（续）

表达式	结果	断言/前条件
`mpl::advance< p, N >::type`	如果n>0，等价于对p应用n次mpl::next。否则，等价于对p应用n次mpl::prior	常量时间 `mpl::advance< p , mpl::distance< p,q >::type >::type` 等价于q
`mpl::distance< p, q >::type`	一个整型常量外覆器（integral constant wrapper）	常量时间
`p::category`	可转换为 `mpl::random_access_iterator_tag`	

5.5 序列Concepts

MPL有一个类似于STL序列concepts的序列concepts分类。每一级concept精化都引入了一套新的能力和接口。在这一节中，我们将依次考察每一个concept。

5.5.1 序列遍历Concepts

对于前面三个迭代器遍历中的每一个：前向、双向、随机访问，都存在一个相应的序列concept。如果一个序列的迭代器是前向迭代器，那么它就称为一个前向序列，其他类似。

如果你觉得下面描述的序列遍历concepts有些单薄，那是因为（“可扩充性（extensibility）”除外，稍后我们就会讨论这个问题）一个序列不过是一对指向其元素的迭代器而已。使得一个序列工作所需的大部分东西，是由其迭代器提供的。

前向序列

任何MPL序列（例如mpl::list，我们将会在本章稍后讨论到）都是一个前向序列。

在表5.4中，S表示一个前向序列。

表5.4 前向序列条件要求

表达式	结果	断言
`mpl::begin<S>::type`	一个前向迭代器	
`mpl::end<S>::type`	一个前向迭代器	可从`mpl::begin<S>::type`到达（Reachable）

由于我们可以访问任何序列的begin迭代器，所以我们可以轻而易举地获得其第一个元素。相应地，每一个非空（nonempty） MPL序列还支持以下表达式：

```
mpl::front<S>::type
```

它等价于

```
mpl::deref<
      mpl::begin<S>::type
>::type
```

双向序列

在表5.5中，S表示任意一个双向序列。

表5.5 双向序列的附加要求

表达式	结 果
mpl::begin<S>::type	一个双向迭代器
mpl::end<S>::type	一个双向迭代器

由于我们可以访问任何序列的end迭代器，所以可以轻而易举地获取其最后一个元素——如果其迭代器是双向的话。相应地，每一个非空双向序列还支持以下表达式：

```
mpl::back<S>::type
```

它等价于下面的表达式：

```
mpl::deref<
    mpl::prior<
        mpl::end<S>::type
    >::type
>::type
```

随机访问序列

mpl::vector是一个随机访问序列例子。在表5.6中，S表示任意一个随机访问序列。

表5.6 随机访问序列的附加要求

表达式	结 果
mpl::begin<S>::type	一个随机访问迭代器
mpl::end<S>::type	一个随机访问迭代器

因为一个随机访问序列具有随机访问迭代器，所以我们可以轻而易举地直接访问序列中的任何元素。相应地，每一个随机访问序列还支持以下表达式：

```
mpl::at<S,N>::type
```

它等价于

```
mpl::deref<
    mpl::advance<
       mpl::begin<S>::type
     , N
    >::type
>::type
```

5.5.2 可扩展性

一个可扩展序列是这样的一种序列，它支持insert、erase以及clear操作。自然而然，由于元数据是不可变的，因此这些操作中没有任何一个可以修改原始的序列，相反，它们均返回原始序列的一个修改过的副本（modified copy）。

给定S是一个可扩展序列，pos是S的某个迭代器，finish是一个可从pos到达的迭代器，X表示任意一个类型，表5.7中的表达式返回一个新的序列，它与S一样，模塑（model）了同样的序列concept：

表5.7 可扩展序列条件要求

表达式	结果元素

`mpl::insert<S,pos,X>::type`	`[mpl::begin<S>::type, pos),`
	`X,`
	`[pos, mpl::end<S>::type)`

`mpl::erase<S,pos>::type`	`[mpl::begin<S>::type, pos),` `[mpl::next<pos>::type, mpl::end<S>::type)`

`mpl::erase<`
`S, pos, finish`
`>::type`
	`[mpl::begin<S>::type, pos),`
	`[finish, mpl::end<S>::type)`

`mpl::clear<S>::type`	无

MPL序列中有很多是可扩展的，但它们对于不同的操作具有不同的复杂度。例如，在mpl::list的头部插入（insertion）和擦除（erasure）操作具有O(1)的复杂度（也就是说，占有常量时间和编译器资源），而在list尾部执行这样的操作的复杂度则为O(N)，这意味着该开销与原始list的长度成正比。在mpl::vector尾部插入（insertion）和擦除（erasure）操作的时间复杂度为O(1)，然而在其他任何位置执行这种修改操作则只能保证具有O（N）的复杂度。

MPL还提供了push_front和pop_front 元函数，分别用于在序列的头部插入和擦除单个元素，另外它还提供了push_back和pop_back，用于在序列的尾部干同样的事情。这些操作只对那些可以（以）O(1)复杂度支持该操作的序列是可用的。

5.5.3 关联式序列

一个关联式序列是一个映射，它的一套唯一的键类型被映射到它的一个或多个值类型。每一个序列的元素类型（可通过其迭代器访问）都关联有一个单个的(key, value)对[⊖]。除了像任何前向序列所要求的那样支持begin<S>::type和end<S>::type外，一个关联式序列还支持以下操作。

在表5.8和表5.9中，S是一个关联式序列，pos1和pos2是S的迭代器，t、k和k2可以是任何类型。

⊖ 要想看一些具体的例子，参见5.8节，该例子涉及了mpl::map和mpl::set。

表5.8 关联式序列条件要求

表达式	结 果	前条件/断言
`mpl::has_key< S, k >::value`	true，如果k在S的keys集合中。 否则为false	
`mpl::at< S, k >::type`	和k关联的value type	前条件：k是在S的keys集合中
`mpl::order< S, k >::type`	一个不带符号的整型常量 外覆器	如果 `mpl::order<S,k>::type::value == mpl::order<S,k2>::type::value` 那么k就是和k2一致的。 前条件：k是在S的keys集合中。

`mpl::key_type< S, t >::type`	S将用于一个元素类型t的键
	类型
		如果
		`mpl::key_type< S, mpl::deref<pos1>::type >::type`
		与
		`mpl::key_type< S, mpl::deref<pos2>::type >::type`
		是一致的，那么pos1和pos2就是一致的。

`mpl::value_type< S, t >::type`	S将用于一个元素类型t的值 类型	

注意：对于order元函数而言，除了每一个键都关联有一个唯一的值外，对于该元函数返回的值不再有任何其他保证。特别是，order的值并不要求和迭代器遍历顺序有任何关系。还要注意的是，不同于STL关联式容器，后者总是有一个关联的顺序关系（默认为std::less<KeyType>），一个关联式元序列并没有这样的顺序关系，在迭代期间元素将被遍历的顺序，完全取决于序列的实现。

5.5.4 可扩展的关联式序列

就像普通的可扩展序列一样，一个可扩展的关联式序列（Extensible Associative Sequence）支持insert、erase以及clear操作，每一个操作都生成一个新的序列作为结果。由于在一个关联式序列中的元素的顺序是任意的，因此一个被插入的元素不必最终放在传递给insert元函数的迭代器所指示的位置。在这方面，关联式元序列类似于STL关联式容器（例如std::map和std::set），但在某些方面它们之间区别很大。例如，一个STL序列可以使用一个迭代器作为一个“提示”，来提高插入操作的性能——可将时间复杂度从O(log(N))改善为O(1)，然而一个关联式元序列完全忽略传递给insert的迭代器实参，事实上，此时插入操作的时间复杂度总是O(1)。尽管对于泛型序列算法的作者来说，insert元函数具有统一的形式，即总是带有一个迭代器实参，不仅会带来方便，而且是至关重要的，然而从另一方面来说，每次当你希望向一个set中插入一个新元素时，

都要提供一个迭代器同样也会带来不便。因此，除了mpl::insert<S,pos,t>之外，一个可扩展的关联式序列还必须支持等价的mpl::insert<S,t>形式。

另一个不同于运行期关联式容器的地方在于，擦除（erasures）操作实际上对迭代的效率会有影响：对一个关联式元序列的完整遍历具有最差情形的复杂度O(N+E)，其中N是序列中元素的数目，E是已被擦除的元素数目。当一个元素从一个关联式序列中擦除时，程序库会添加一个特殊的标记元素（marker element），它可以“导致”在迭代过程中跳过被擦除的元素。注意，对一个关联式序列执行clear操作不会招致类似的效率惩罚，因为所得结果是一个崭新的序列。

以下表达式具有常量复杂度，返回一个新的序列S'，它所模塑（models）的MPL序列概念与S所模塑的一样。

表5.9 可扩展关联序列

表达式	结果	备注
mpl::insert< S, pos1,t >::type mpl::insert< S, t >::type	S'等价于S，除了 mpl::at< S' , mpl::key_type<S,t>::type >::type 是 mpl::value_type<S,t>::type 以外。	可能会招致一个擦除惩罚 (erasure penalty)，如果 mpl::has_key< S, mpl::key_type< S, t >::type >::value 是true的话。
mpl::erase< S, pos1 >::type	S' 等价于S，除了mpl::has_key< S' , mpl::key_type< S , mpl::deref<pos1>::type >::type >::value 是false以外。	
mpl::erase_key< S, k >::type	S'等价于S除了 mpl::has_key<S' , k>::value 是false以外	
mpl::clear< S >::type	一个空序列，它具有和S相同的属性	

由于在一个可扩展的关联式序列中的任何地方执行擦除（erasure）操作都具有O(1)的复杂度，因此pop_front和pop_back操作也得到了支持。由于插入操作也是O(1)复杂度，因此mpl::push_front<S,t>和mpl::push_back<S,t>也得到了支持，不过它们都等价于mpl::insert<S,t>，因为mpl::insert<S,pos,t>中的迭代器参数被忽略了。

5.6 序列相等性

记住一点很重要：不要落入依赖于序列类型的一致性（type identity）的陷阱，尤其当处理计算结果时。例如，你不可预期以下断言可以通过：

```
BOOST_STATIC_ASSERT((       // 错误
  boost::is_same<
      mpl::pop_back<mpl::vector<int, short> >::type
    , mpl::vector<int>
  >::value
));
```

对大多数用途来说，以上两个正被比较的类型的行为是相同的，并且在大多数时间里，你不会注意到区别。对一个mpl::vector特化使用mpl::pop_back的结果，不会是另一个mpl::vector特化！

然而，正如你在第3章量纲分析探索中所看到的那样，一个只能采用两个一致类型进行调用的函数模板——如果这些类型是序列的话——就可能无法按预期的那样工作。对于一个只有当两个类型参数是一致时才能工作的类模板局部特化来说，同样如此。

检查序列相等性的正确方式是，总是使用equal算法，如下：

```
BOOST_STATIC_ASSERT((         // OK
  mpl::equal<
      mpl::pop_back<mpl::vector<int, short> >::type
    , mpl::vector<int>
  >::value
));
```

5.7 固有的序列操作

MPL还提供了一个序列元函数范畴，其STL对应物通常采用成员函数的方式来实现。我们已经讨论了begin、end、front、back、push_front、push_back、 pop_front、pop_back、insert、erase以及clear，余下的总结在表5.10中，其中R是任意的一个序列。

表5.10 固有的序列操作

表达式	结 果	最糟情形的复杂度
`mpl::empty<S>::type`	一个布尔常量外覆器，当且仅当序列为空时为`true`	常量
`mpl::insert_range< S, pos, R >::type`	和S一致，不过R的元素被插入于pos位置	和结果序列的长度呈线性关系
`mpl::size<S>::type`	一个整型常量外覆器，其`::value`是S中元素的数目	和S的长度呈线性关系

所有这些元函数均被认为是固有的序列操作，以便和通用的序列算法区分开，因为它们通常需要为每一个新种类的序列单独实现。它们没有像STL中的容器成员函数那样被实现为嵌套元

函数有三个良好的理由：

1. 语法负担。在大多数元编程上下文中，成员模板使用起来很麻烦，因为需要使用额外的template关键字：

```
Sequence::template erase<pos>::type
```

和下面的比较一下：

```
mpl::erase<Sequence,pos>::type
```

如你所知，减少C++ 模板语法的负担是MPL的一个主要设计考虑。

2. 效率。大多数序列都是会以这样或那样的方式被多次实例化的模板。而模板成员的存在——即使它们未被使用——会导致每一次实例化都（可能）带来开销。

3. 方便性。撇开我们将这些操作称为“固有的”不谈，还因为这种方式还可以合理地为它们大多数提供默认实现。举个例子，默认的size实现度量序列的begin和end迭代器之间的距离。如果这些操作采用成员模板的方式进行实现，那么每一个序列的作者都需要把它们全部实现一遍。

5.8 序列类

在这一节中，我们将描述MPL提供的几个具体的序列，并讨论它们是如何符合上面描述的序列概念。

在开始之前，你应该知道，所有MPL序列都有不带数字和带数字的形式。不带数字的形式是你已经熟悉了的，比如mpl::vector<int, long, int>。相应的带数字的形式包括序列的长度作为其模板名字的一部分，例如mpl::vector3<int, long, int>。默认情况下，不带数字的形式的长度被限制为不超过20个元素[⊖]以便减少程序库中的耦合（coupling）并限制编译时间。为了使用长度为N的带数字形式的序列，你必须包含一个相应的“带数字形式的头文件”，长度为N的序列需要包含这样的头文件，其名字末尾的数字N被进制为10的倍数[⊜]。例如：

```
#include <boost/mpl/vector/vector30.hpp> // 28的圆整值

// 声明一个具有28个元素的序列
typedef boost::mpl::vector28<
     char, int, long ... 其他25个types
> s;
```

5.8.1 list

mpl::list是最简单的可扩展的MPL序列，它在结构上酷似运行期单链表（runtime singly-linked list）。由于它是一个前向序列（Forward Sequence），它支持begin、end，当然还有通过front来访问第一个元素。由于它支持在序列的头部进行O(1)复杂度插入和擦除操作，故它还支持push_front和pop_front。

⊖ 参考MPL参考手册的Configuration Macros一节，了解如何修改该限制的详情。

⊜ 不是四舍五入，例如N如果是11，则包含vector20.hpp。

5.8.2 vector

MPL的vector几乎就是STL vector的对应物：它是一个随机访问序列（Random Access Sequence），因此它自然而然有一个随机访问迭代器。由于每一个随机访问迭代器都是一个双向迭代器，并且我们可以访问vector的end迭代器，所以除了front之外，back也得到了支持。就像STL vector一样，MPL的vector也支持高效的push_back和pop_back操作。

除了通常的编译期/运行期区别外，这个序列可能在一个显著的细节方面不同于STL中的那一个：它“可能”有一个最大尺寸限制，这不仅是由通常的编译器资源（例如内存或模板实例化深度）所限制的，而且还由序列的实现方式所决定。在这种情况下，序列通常只能被扩充至包含在翻译单元内带数字最大的序列头文件所指示的数目。例如：

```
#include <boost/mpl/vector/vector10.hpp>

typedef boost::mpl::vector9<
      int[1], int[2], int[3], int[4]
    , int[5], int[6], int[7], int[8], int[9]
> s9;

typedef mpl::push_back<s9, int[10]>::type s10;  // OK
typedef mpl::push_back<s10, int[11]>::type s11; // 错误
```

为了使代码工作，我们必须将#include指令改为：

```
#include <boost/mpl/vector/vector20.hpp>
```

这个局限并不像它听起来的那样严重，原因有二：

1. 在默认情况下，程序库头文件为你提供的带数字的vector形式，允许多达50个元素，并且这个数目可以通过定义一些预处理符号来调整⊖。

2. 由于元代码执行于编译期，因此超过该限制会导致一个编译期错误。除非你在编写被其他元程序员（metaprogrammer）使用的通用元函数，否则你永远都不可能交付可能在客户手中因为这个局限而失败的代码——只要你的代码在你的本地机器上通过了编译。

我们说在这方面“可能”不同，是因为在支持typeof语言扩充的编译器上，最大尺寸限制就消失了，第9章描述了一些使其成为可能的基本技术。

在mpl::vector上的操作往往比那些在mpl::list上操作编译快很多，并且由于其随机访问能力，mpl::vector要灵活得多 。加在一起，这些因素使得mpl::vector成为你的首选——当你选择一个一般用途的可扩展序列时。然而，如果客户将你的代码用于可能需要具有任意长度的序列的编译期计算时，最好还是使用mpl::list。

> **指导方针**
>
> 当需要选择一个一般用途的类型序列时，首选mpl::vector。

⊖ 参见MPL参考手册的Configuration Macros一节，以便了解如何修改这个限制。

5.8.3 deque

MPL的deque在所有方面都酷似mpl::vector，除了deque允许在序列的头部进行高效的push_front和pop_front操作外。不像相应的STL组件，deque的效率非常接近于vector，事实上，在许多编译器上，vector的底层实现就是一个deque。

5.8.4 range_c

range_c是一个“惰性的”随机访问序列，它包含连续的整型常量，也就是说，mpl::range_c<long, N, M>大致等价于：

```
mpl::vector<
    mpl::integral_c<long,N>
  , mpl::integral_c<long,N+1>
  , mpl::integral_c<long,N+2>
  ...
  , mpl::integral_c<long,M-3>
  , mpl::integral_c<long,M-2>
  , mpl::integral_c<long,M-1> // 注意：是M-1而不是M
>
```

我们说range_c是“惰性的”，意思是，它的元素不是被显式表示的：它只是存储端点，并根据需要在区间内产生新的整型常量。当在一个大的整数序列上迭代时，使用range_c不仅方便，还可以比使用非惰性的替代物（比如上面展示的vector）显著地节省编译时间。

获得这种经济实惠的代价是range_c带有一个vector和list所不具有的局限：它不是可扩充的。如果程序库可以支持任意的元素插入到range_c中，元素则需要被显式地表示。尽管不是可扩充的，range_c还是支持pop_front和pop_back，因为缩小一个区间还是很容易做到的。

5.8.5 map

MPL map是一个可扩展关联式序列，它的每一个元素都支持mpl::pair的接口。

```
template <class T1, class T2>
struct pair
{
    typedef pair type;
    typedef T1 first;
    typedef T2 second;
};
```

一个元素的first和second类型分别被当作key和value来处理。为了创建一个map，只要将其元素作为模板参数列在序列中即可。以下例子展示了从内建的整数类型到比它刚好大一级的类型的映射：

```
typedef mpl::map<
    mpl::pair<bool, unsigned char>
  , mpl::pair<unsigned char, unsigned short>
  , mpl::pair<unsigned short, unsigned int>
```

```
    , mpl::pair<unsigned int, unsigned long>
    , mpl::pair<signed char, signed short>
    , mpl::pair<signed short, signed int>
    , mpl::pair<signed int, signed long>
>::type to_larger;
```

就像mpl::vector一样，mpl::map实现品在不支持typeof语言扩充的C++编译器上具有一个圆整的（bounded）最大尺寸，并且，如果你打算增长一个map使其超过其圆整后数字的十的倍数的话，要注意包含适当的带数字的序列头文件。

然而，对于那些手头编译器没有超过C++标准需要的编译器来说，并不全是坏消息：当map具有一个圆整的最大尺寸时，迭代所有元素的时间复杂度为O(N)而非O(N+E)，其中N是map的大小，而E则是已经被擦除的元素的数目。

5.8.6 set

set就像map一样，除了每一个元素的键类型和值类型一致外。键和值类型是一致的，这一事实意味着mpl::at<S,k>::type是一个相当无趣的操作——它只是毫无修改地返回k而已。MPL set的一个主要用途是对mpl::has_key<S,k>::type的成员关系高效地进行测试。set从来不受最大尺寸限制，因而完整的遍历操作总是具有O(N+E)复杂度。

5.8.7 iterator_range

在意图上，iterator_range非常类似于range_c。不是显式地表示其元素，一个iterator_range存储两个用于指示序列端点的迭代器。由于MPL算法在序列上而非迭代器上操作，因此，当你希望操作序列的部分元素时，iterator_range是必不可少之物：一旦你已经找到序列的端点，你就可以形成一个iterator_range并且将其传递给算法，而不是构建原始序列的一个修改版本。

5.9 整型序列外覆器

我们已经讨论了使用vector_c类模板作为编写整型常量列表的捷径。MPL还提供了list_c、deque_c和set_c来表示相应的vector、deque和set。每一个序列都带有如下形式：

```
sequence-type_c<T, n1, n2, ... nk>
```

每一个序列外覆器的第一个参数是将要存储的整数类型T，接下来的参数是将要存储的T类型的值。你可以认为它等价于：

```
sequence-type<
    integral_c<T,n1>
  , integral_c<T,n2>
  , ...
  , integral_c<T,nk>
>
```

换句话说，它们并不是精确相同的类型，并且如我们已经建议的那样，当对序列的相等性进行比较时，你不应该依赖于类型一致性（type identity）。

注意，MPL还提供了_c后缀版本的带数字的序列形式：

```
#include <boost/mpl/vector/vector10_c.hpp>

typedef boost::mpl::vector10_c<int,1,2,3,4,5,6,7,8,9,10> v10;
```

5.10 序列派生

通常来说，任意不带数字形式的序列派生（Sequence Derivation）于相应的带数字的形式，否则会与之共享一个提供了序列实现品的共同基类。例如，mpl::vector以如下方式进行定义：

```
namespace boost { namespace mpl {

  struct void_; // 无参标记

  // 主模板声明
  template <class T0 = void_, class T1 = void_, etc....>
  struct vector;

  // 特化
  template<>
  struct vector<> : vector0<> {};

  template<class T0>
  struct vector<T0> : vector1<T0> {};

  template<class T0, class T1>
  struct vector<T0,T1> : vector2<T0,T1> {};

  template<class T0, class T1, class T2>
  struct vector<T0,T1,T2> : vector3<T0,T1,T2> {};

  //等等
}}
```

类似地，整型序列外覆器（integral sequence wrappers）派生于等价的底层类型序列。

所有内建MPL序列被设计成这样：几乎所有子类函数作为一个等价的类型序列。派生是一种为既有序列家族提供新接口（或只是一个新名字）的极具威力的方式。例如Boost Python库提供如下类型序列：

```
namespace boost { namespace python {

  template <class T0=mpl::void_, ... class T4=mpl::void_>
  struct bases : mpl::vector<T0, T1, T2, T3, T4> {};

}}
```

你可以使用同样的技术来创建是一个MPL类型序列的普通类（plain class）：

```
struct signed_integers
  : mpl::vector<signed char, short, int, long> {};
```

在某些编译器上，使用signed_integers而不是底层的vector，可以大幅提高元程序的效率。参见附录C以了解更多的细节。

5.11 编写你自己的序列

在这一节中我们将向你展示如何编写一个简单的序列。至此你也许会感到奇怪为什么想要干这种事。毕竟，MPL提供的内建设施已经相当完备。原因通常是出于效率的考虑。尽管MPL序列针对一般用途有着良好的优化，但你可能有特别的应用，使用自己专门编写的序列可以工作得更好。例如，我们可以编写一个外覆器，它将一个函数指针的实参类型表示为一个序列[Nas03]。如果出于一些原因，你手头碰巧已经有一个函数指针类型，直接迭代这个外覆器，比装配包含这些类型的另一个序列可以节省相当多的模板实例化。

例如，我们将编写一个容量受限制的随机访问序列，名为tiny，它具有最多三个元素。这个序列非常像MPL（针对符合标准但未供应typeof运算符的编译器的）vector实现品。

5.11.1 构建tiny序列

第一步是选择一个表示法。这方面没有太多的工作要做，只要把它能包含的（至多三个）types编码进序列类型自身即可：

```
struct none {}; // 用于指示没有元素的标签类型(tag type)

template <class T0 = none, class T1 = none, class T2 = none>
struct tiny
{
    typedef tiny type;
    typedef T0 t0;
    typedef T1 t1;
    typedef T2 t2;

    ...
};
```

正如你看到的那样，我们已经提前行动了，已经填充了一些实现：tiny的嵌套的::type回指向该序列自身，这使得tiny成为一种“返回自身的元函数（self-returning metafunction）”。所有MPL序列都干了类似的事情，事实证明这一举措可以带来极大的便利。例如，为了返回一个元函数产生的序列，你只要从希望返回的序列派生该元函数即可。而且，当eval_if的一个分支需要返回一个序列时，你不必将其包装在第4章描述的identity 元函数中。也就是说，给定一个tiny序列S，以下两种形式是等价的：

```
// 移除S的第一个元素，除非S为空
typedef mpl::eval_if<
```

```
    mpl::empty<S>
  , mpl::identity<S>
  , mpl::pop_front<S>
>::type r1;

// 类似地
typedef mpl::eval_if<
    mpl::empty<S>
  , S                       // 当被调用时，S返回S
  , mpl::pop_front<S>
>::type r2;
```

另外三个嵌套的typedefs，即t0、t1、t2，使得任何元函数访问tiny序列的元素，都很容易[⊖]：

```
template <class Tiny>
struct manipulate_tiny
{
    // T0为何物?
    typedef typename Tiny::t0 t0;
};
```

只要我们大家同意不将none用于任何其他目的，只用于标记tiny的空元素的开始，我们现在就拥有了一个容纳三个元素的便利的接口。然而，它还不是一个MPL序列。

回忆序列的最基本需求，我们可以发现每一个序列都必须从MPL的begin和end 元函数返回迭代器。显而易见，我们将需要一个迭代器表示。由于随机访问迭代器可以双向移动，它们必定能访问序列中所有元素。最简单的做法是将整个序列嵌入在迭代器表示中。事实上，通常MPL迭代器将它们遍历的序列的全部或一部分嵌入进来（由于list迭代器只向前移动，所以它们只容纳可访问的那部分list）。

5.11.2 迭代器的表示

一旦迭代器可以访问序列，我们只需要以某种方式表示位置（position）就可以了。一个整型常量外覆器（就是下例中的Pos）即可满足要求：

```
#include <boost/mpl/iterator_tags.hpp>
```

⊖ 替代的方案就是笨重的局部特化（partial specialization）：

```
template <class Tiny>
struct manipulate_tiny;

template <class T0, class T1, class T2>
struct manipulate_tiny<tiny<T0, T1, T2> >
{
    // T0已知
};
```

从长远来看，嵌入元素类型可使我们少写不少代码。

```
template <class Tiny, class Pos>
struct tiny_iterator
{
    typedef mpl::random_access_iterator_tag category;
};
```

任何迭代器最基本的操作是解引用（dereferencing）（经由mpl::deref），以及前向遍历（通过mpl::next）。在这个例子中，通过构建一个新的具有递增的（或递减的）position的tiny_iterator，我们可以处理任一方向的“递增”遍历[⊖]：

```
namespace boost { namespace mpl {

   // 前向迭代器的条件要求
   template <class Tiny, class Pos>
   struct next<tiny_iterator<Tiny,Pos> >
   {
       typedef tiny_iterator<
           Tiny
         , typename mpl::next<Pos>::type
       > type;
   };
    // 双向迭代器的条件要求
   template <class Tiny, class Pos>
   struct prior<tiny_iterator<Tiny,Pos> >
   {
       typedef tiny_iterator<
           Tiny
         , typename mpl::prior<Pos>::type
       > type;
   };

}}
```

对tiny_iterator进行解引用操作有点儿棘手：我们需要某种方式使用迭代器的位置（position）来索引（index）tiny序列。如果你正在想：“等一等，让我们去做实现at操作所要做的事情。”你是正确的，到了该让我们的迭代器单独呆一会儿的时候了。

5.11.3 为tiny实现at

实现at的一个合理的方式是利用局部特化。首先，我们编写一个模板，它基于一个数值参数来选择序列的一个元素：

```
template <class Tiny, int N> struct tiny_at;
```

⊖ 我们还可以利用默认的mpl::next和mpl::prior实现品，并通过简单地提供tiny_iterator（带有相应的嵌套typedefs（::next/::prior））而实现需求。这样做的好处可以减少一些代码键入量，但代价是结果所得元程序较慢。这样的迭代器是一个典型的“Blob”反模式（anti-pattern）实例（见第2章的有关讨论）。

```
// 针对每一个索引的局部特化访问器（partially specialized accessors）
template <class Tiny>
struct tiny_at<Tiny,0>
{
    typedef typename Tiny::t0 type;
};

template <class Tiny>
struct tiny_at<Tiny,1>
{
    typedef typename Tiny::t1 type;
};

template <class Tiny>
struct tiny_at<Tiny,2>
{
    typedef typename Tiny::t2 type;
};
```

注意，如果当第二个参数的值超出0、1、2的范围时，访问tiny_at的嵌套::type，将会得到一个编译错误：非特化模板（或主模板）未定义。

接下来，我们可以简单地为tiny“实例”局部特化mpl::at：

```
namespace boost { namespace mpl {

  template <class T0, class T1, class T2, class Pos>
  struct at<tiny<T0,T1,T2>, Pos>
    : tiny_at<tiny<T0,T1,T2>,Pos::value>
  {
  };

}}
```

从表面看，使用局部特化没有任何错，但是让我们看看是如何使得非特化版本的mpl::at为tiny工作的。MPL所提供的at看上去如下：

```
template<class Sequence, class N>
struct at
  : at_impl<typename Sequence::tag>
      ::template apply<Sequence,N>
{
};
```

在默认情况下，at将其实现转发给at_impl<Sequence::tag>，后者是一个元函数类，它知道如何为所有具有该tag类型的序列执行at功能。因此，我们可以向tiny添加一个::tag（称为tiny_tag），并编写mpl::at_impl的一个显式（完全）特化：

```
struct tiny_tag {};
template <class T0 = none, class T1 = none, class T2 = none>
struct tiny
{
    typedef tiny_tag tag;
    typedef tiny type;
    typedef T0 t0;
    typedef T1 t1;
    typedef T2 t2;
};

namespace boost { namespace mpl {
   template <>
   struct at_impl<tiny_tag>
   {
       template <class Tiny, class N>
       struct apply : tiny_at<Tiny, N::value>
       {};
   };
}}
```

这看上去并不像是对“为tiny序列局部特化at”的明显改进，但确实是。通常而言，编写可以匹配一个特定序列家族所拥有的所有形式的局部特化是不切实际的。等价的序列形式不是同一个模板的实例是很常见的，因此通常每个形式至少需要一个局部特化，例如，你不能编写一个既匹配mpl::vector<int> 又匹配mpl::vector1<int>的局部模板特化。出于同样的原因，对at进行特化，限制了第三方通过派生快速地构建该序列家族新成员的能力。

> **建议**
>
> 为了实现一个固有的序列操作，你总是要提供一个序列标记，以及该操作的_impl 模板的一个特化。

5.11.4 完成tiny_iterator的实现

有了at实现品在握，我们就可以实现tiny_iterator的解引用（dereference）操作：

```
namespace boost { namespace mpl {

   template <class Tiny, class Pos>
   struct deref< tiny_iterator<Tiny,Pos> >
     : at<Tiny,Pos>
   {
   };

}}
```

现在惟一缺少的东西是具有常量时间复杂度的mpl::advance和mpl:: distance 元函数特化：

```
namespace boost { namespace mpl {

    // random access iterator的条件要求
    template <class Tiny, class Pos, class N>
    struct advance<tiny_iterator<Tiny,Pos>,N>
    {
        typedef tiny_iterator<
              Tiny
            , typename mpl::plus<Pos,N>::type
        > type;
    };

    template <class Tiny, class Pos1, class Pos2>
    struct distance<
        tiny_iterator<Tiny,Pos1>
      , tiny_iterator<Tiny,Pos2>
    >
      : mpl::minus<Pos2,Pos1>
    {};

}}
```

注意，我们已经将检查使用错误的工作留给你了（见练习5-0）。

5.11.5 begin和end

最后，我们准备使tiny变成一个真正的序列，剩下要做的全部工作就是供应begin和end。就像mpl::at、mpl::begin和mpl::end使用traits来隔离为一个特定的序列家族的实现一样。这样，编写begin是件很简单的事情：

```
namespace boost { namespace mpl {

    template <>
    struct begin_impl<tiny_tag>
    {
        template <class Tiny>
        struct apply
        {
            typedef tiny_iterator<Tiny,int_<0> > type;
        };
    };
}}
```

编写end比编写begin要困难一些，因为我们需要基于none元素的数量来推导序列的长度。一个直截了当的方式是：

```
namespace boost { namespace mpl {

   template <>
   struct end_impl<tiny_tag>
   {
       template <class Tiny>
       struct apply
        :eval_if<
            is_same<none,typename Tiny::t0>
          , int_<0>
          , eval_if<
               is_same<none,typename Tiny::t1>
             , int_<1>
             , eval_if<
                  is_same<none,typename Tiny::t2>
                , int_<2>
                , int_<3>
               >
            >
         >
     {};
   };
}}
```

不幸的是，以上代码不满足end的O(1)复杂度要求：对于一个长度为N的序列来说，它耗费O(N)复杂度的模板实例化，因为eval_if/is_same对组将会被实例化，直至发现一个none元素为止。为了在常量时间内计算出序列的尺寸，我们只要编写一些局部特化即可：

```
template <class T0, class T1, class T2>
struct tiny_size
  : mpl::int_<3> {};

template <class T0, class T1>
struct tiny_size<T0,T1,none>
  : mpl::int_<2> {};

template <class T0>
struct tiny_size<T0,none,none>
  : mpl::int_<1> {};

template <>
struct tiny_size<none,none,none>
  : mpl::int_<0> {};
namespace boost { namespace mpl {
   template <>
```

```
    struct end_impl<tiny_tag>
    {
        template <class Tiny>
        struct apply
        {
            typedef tiny_iterator<
                Tiny
              , typename tiny_size<
                    typename Tiny::t0
                  , typename Tiny::t1
                  , typename Tiny::t2
                >::type
            >
            type;
        };
    };
}}
```

在这里，每一个后继的tiny_size特化都比前一个“更特化（more specialized)”，对于任何给定的tiny序列来说，只有最适合的版本会被实例化。最佳匹配的tiny_size特化总是直接对应于序列的长度。

如果你对这儿的刻板代码重复的数量之多感到一丝不快（甚至恼怒），我们不能怪你。毕竟，我们不是承诺过元编程有助于避免发生这种情况的吗？唔，我们的确说过。对于该问题我们有两个答案。首先，元编程库使得用户可以避免做这种免重复的事，然而，一旦你开始编写新序列，你就处于程序库设计者的位置了[⊖]。你的用户会对你不怕麻烦感到感激（即使这个用户就是你自己)。其次，正如我们早先所暗示的，还有一些方式使得这种代码的生成变得自动化。你将会看到甚至是程序库设计者都可以免遭自我重复的尴尬（见附录A)。

在这一点很容易做到，我们同样可以实现一个特化的mpl::size。这完全是可选的，MPL默认的size实现品只是测量我们的begin和end迭代器之间的距离，但由于我们是因为追求效率才编写这个tiny的，所以我们可以通过编写自己的版本来节省一些模板实例化：

```
namespace boost { namespace mpl {
    template <>
    struct size_impl<tiny_tag>
    {
        template <class Tiny>
        struct apply
          : tiny_size<
                typename Tiny::t0
              , typename Tiny::t1
```

⊖ 这种对重复的需要（至少在元编程库的层面来看）似乎是C++的一个怪毛病。大多数其他支持元编程的语言都没有遭受类似的局限性，可能是因为它们的元编程设施不仅仅是一个幸运的偶然吧。

```
            , typename Tiny::t2
          >
      {};
  };
}}
```

你现在可能已经理解了同样的标签派发（tag-dispatching）技术一再出现的原因。实际上，MPL所有固有的序列操作都使用了这一技术，因此你总是可以利用它为自己的序列类型定制任何固有的操作。

5.11.6 加入扩充性

在这一小节中，我们将编写一些使tiny能满足可扩展序列需求的操作。我们将不会向你展示全部，因为它们在实质上是类似的。此外，我们还需要留点儿操作当作本章后面的练习题！

首先，让我们处理clear和push_front。对一个满（full）tiny调用push_front是不合法的，因为我们的tiny序列具有一个固定的容量。因此，任何作为第一个实参传递给push_front的有效的tiny<T0, T1, T2>，其长度必须总是<= 2且T2 = none，并且将T2从序列尾部删除掉是可以的[⊖]：

```
namespace boost { namespace mpl {
  template <>
  struct clear_impl<tiny_tag>
  {
      template <class Tiny>
      struct apply : tiny<>
      {};
  };
  template <>
  struct push_front_impl<tiny_tag>
  {
      template <class Tiny, class T>
      struct apply
        : tiny<T, typename Tiny::t0, typename Tiny::t1>
      {};
  };
}}
```

这很简单！注意，因为每一个tiny序列都是一个返回自身的元函数，因此我们能够在apply本体中利用元函数转发（metafunction forwarding）功能。

> **建议**
>
> 为了获得最大程度的MPL互操作性，当编写一个尚未成为元函数的类模板时，你可以考虑通过添加一个嵌套的::type（指向该类模板自身）而使其成为一个元函数。当编写一个将总会返回一个类类型的元函数时，考虑使它派生于那个类，并且使得该元函数返回其自身。

⊖ 实际上，当push_front被调用时，强制实施我们的假定（即序列不是满的（full））被作为一个练习题留给你了。

编写push_back就没有这么容易了：我们应用的转换（transformation）取决于输入序列的长度。不必烦恼，我们已经编写一个“其实现依赖于输入序列的长度”的操作了，就是end。由于我们手头已经有方便的长度计算了，现在所需的全部东西就是一个tiny_push_back 模板，它针对每一个序列长度进行特化：

```
template <class Tiny, class T, int N>
struct tiny_push_back;

template <class Tiny, class T>
struct tiny_push_back<Tiny,T,0>
  : tiny<T,none,none>
{};

template <class Tiny, class T>
struct tiny_push_back<Tiny,T,1>
  : tiny<typename Tiny::t0,T,none>
{};

template <class Tiny, class T>
struct tiny_push_back<Tiny,T,2>
  : tiny<typename Tiny::t0,typename Tiny::t1,T>
{};
```

```
namespace boost { namespace mpl {
   template <>
   struct push_back_impl<tiny_tag>
   {
       template <class Tiny, class T>
       struct apply
         : tiny_push_back<
             Tiny, T, size<Tiny>::value
           >
       {};
   };
}}
```

注意，这里遗漏的和展示的同样重要。通过不为长度为3的序列定义一个tiny_push_back特化，我们向一个满序列（full sequence）中push_back东西会导致编译错误。

5.12 细节

至此，你应该对MPL的输入和产出有一个相当清晰的理解，在未来各章中，你将会看到对类型序列及其实际应用的更多的启示，眼下我们仅仅回顾一下本章阐述的一些核心概念。

序列概念：MPL序列分成三个遍历concept范畴（前向、双向以及随机访问），分别对应于它

们的迭代器能力。一个序列也可能是前向扩展的（front-extensible）的，意味着它支持push_front和pop_front操作，也可能是后向扩展的（back-extensible），意味着它支持迭代器概念push_back和pop_back操作。一个关联式序列表示从类型到类型的映射（具有O(1)查找时间复杂度）。

迭代器概念：MPL迭代器模塑三个遍历concepts之一：前向迭代器、双向迭代器，以及随机访问迭代器。每一个迭代器概念都是对前一个的精化（refine），因此所有双向迭代器也都是前向迭代器，并且所有随机访问迭代器也都是双向迭代器。一个前向迭代器x可以被增量操作（incrementable），并且可以被解引用（dereferenceable），这意味着next<x>::type和deref<x>::type都是有着良好定义的（well-defined），否则会发生逾尾。一个双向迭代器可以是支持减量操作的（decrementable），也可以指向其序列的开头。

序列算法：C++ 模板元编程的纯粹函数式特性其实指MPL算法对序列进行操作而不是对迭代器对（iterator pairs）进行操作。否则，将一个算法的结果传递给另一个算法会变得过于困难。一些人感觉同样的逻辑也适用于STL算法，而且几个用于操作整个运行期序列的算法程序库已经出现了。即将发布的Boost版本中（也许）就有一个。

固有的（Intrinsic）序列操作：并非所有序列操作都可以采用一般化的方式编写出来，例如begin和end就是如此，它们需要专门编写以便能处理特定的序列。这些MPL元函数都使用一个标签派发技术，以使得定制工作变得容易一些。

5.13 练习

5-0. 编写一个测试程序，练习我们已经实现的tiny部分。尝试修改你的程序，使得只有测试成功才能编译通过。

5-1. 编写一个double_first_half 元函数，它接收一个长度为N的整型常量外覆器的随机访问序列作为参数，并且返回前面N/2个元素值倍增后的副本，即使得如下测试结果为true：

```
mpl::equal<
    double_first_half< mpl::vector_c<int,1,2,3,4> >::type
  , mpl::vector_c<int,2,4,3,4>
>::type::value
```

5-2. 注意，如果其tiny参数已经有三个元素了，push_back将无法通过编译。那么对于push_front来说，我们如何实现同样的保证呢？

5-3. 利用我们提供的push_back实现的例子，为tiny序列实现insert操作。重构push_back的实现，使其与insert共享更多的代码。

5-4. 如何减少我们的push_back实现所需要的模板实例化的数目？（提示：请再看看我们在5.11.5节的end的实现）这又如何与前一个练习中的重构互动呢？

5-5. 为tiny实现pop_front、pop_back以及erase算法。

5-6. 编写一个名为dimensions的序列适配器模板，当针对一个数组类型实例化时，以前向、不可扩充的序列的形式呈现该数组的维数（dimensions）：

```
typedef dimensions<char [10][5][2]> seq;
BOOST_STATIC_ASSERT( mpl::size<seq>::value == 3 );
BOOST_STATIC_ASSERT(( mpl::at_c<seq,0>::type::value == 2 ));
BOOST_STATIC_ASSERT(( mpl::at_c<seq,1>::type::value == 5 ));
BOOST_STATIC_ASSERT(( mpl::at_c<seq,2>::type::value == 10 ));
```

可以考虑使用type traits程序库设施来简化实现。

5-7. 修改练习5-6中的dimensions序列适配器，从而提供双向迭代器和push_back、pop_back操作。

5-8. 编写一个fibonacci_series类，它表示Fibonacci数列的一个无限的前向序列：

```
typedef mpl::advance_c< mpl::begin<fibonacci_series>::type, 6 >::type p;
BOOST_STATIC_ASSERT( mpl::deref<i>::type::value == 8 );

typedef mpl::advance_c< p, 4 >::type q;
BOOST_STATIC_ASSERT( mpl::deref<j>::type::value == 55 );
```

每一个Fibonacci数列元素都是前两个元素的和。序列开头几个元素为：0, 1, 1, 2, 3, 5, 8, 13……

5-9. 修改练习5-8的fibonacci_series序列，使其受到序列最大元素数目的限制，并使序列的迭代器成为双向的（bidirectional）：

```
typedef fibonacci_series<8> seq;
BOOST_STATIC_ASSERT( mpl::size<seq>::value == 8 );
BOOST_STATIC_ASSERT( mpl::back<seq>::type::value == 13 );
```

5-10. 编写一个用于组合编译期二叉树数据结构的tree类模板：

```
typedef tree<                          //      double
      double                           //      /    \
    , tree<void*,int,long>             //   void* char
    , char                             //   /   \
    > tree_seq;                        // int   long
```

实现对该tree的元素前序（pre-order）、中序（in-order）以及后序（post-order）遍历的迭代器：

```
BOOST_STATIC_ASSERT(( mpl::equal<
      preorder_view<tree_seq>
    , mpl::vector<double,void*,int,long,char>
    , boost::is_same<_1,_2>
    >::value ));

BOOST_STATIC_ASSERT(( mpl::equal<
      inorder_view<tree_seq>
    , mpl::vector<int,void*,long,double,char>
    , boost::is_same<_1,_2>
    >::value ));
```

```
BOOST_STATIC_ASSERT(( mpl::equal<
      postorder_view<tree_seq>
    , mpl::vector<int,long,void*,char,double>
    , boost::is_same<_1,_2>
    >::value ));
```

重要

扩充练习5-0中的测试，使其涵盖你在练习5-3、5-4以及5-5中实现的算法。

第6章 算 法

STL之父Alexander Stepanov常常强调算法在他的程序库中的中枢角色。MPL也不例外，现在你已经了解算法操作的序列和迭代器，我们准备深入探讨算法的问题。

我们将从算法和抽象之间的关系开始讨论，然后讨论STL算法和MPL算法之间的异同点，尤其是为了对付元数据是不可变的（immutable）这一事实而在MPL中作出的设计决策。然后我们描述MPL的三个算法归类中最有用的算法，并以实现你自己的序列算法的简短一节结束。

6.1 算法、惯用法、复用和抽象

抽象（Abstraction）可定义为远离具体实例或实现，朝着对象或过程的“本质”的一般化。一些抽象，例如STL迭代器，是如此令人熟悉，以至于它们可以被称为惯用法（idiomatic）。在软件设计中，通过惯用的抽象所达成的理想的复用，就像代码复用一样重要。最好的程序库不但提供可复用的代码组件，还提供可复用的惯用法。

由于它们中的大多数在相对低的序列遍历层面操作，以至于我们很容易就会遗漏这个事实：STL算法表示强大的抽象。实际上，通常争论（并非一点理由都没有）对于微不足道的任务来说，算法要比手写循环效率低。例如[⊖]：

```
// "抽象"
std::transform(
    v.begin(), v.end(), v.begin()
  , std::bind2nd(std::plus<int>(),42)
);
// 手写循环
typedef std::vector<int>::iterator v_iter;
for (v_iter i = v.begin(), last = v.end(); i != last; ++i)
   *i += 42;
```

那么，上面对transform的使用有何问题呢？

- 用户必须处理迭代器，哪怕他是想对整个序列进行操作。
- 创建函数对象（function object）的机制笨拙丑陋，并且至少引入了和它所隐藏的低层细节一样多的细节。
- 除非阅读代码的人每日与STL组件形影不离，否则，这种“抽象”实际上使事情变得更加混乱而不清晰。

然而，这些缺点可以很容易地克服。例如，我们可以使用Boost Lambda库（它启发了MPL

⊖ 为了对STL算法表示公平，需要说明一下，这个例子是故意被设计为适于手写循环的。

的编译期lambda表达式）来简化和清晰化运行期函数对象[⊖]：

```
std::transform(v.begin(), v.end(), v.begin(), _1 + 42);
```

或者甚至：

```
std::for_each(v.begin(), v.end(), _1 += 42);
```

这两行语句和我们先前看到的原始循环做了完全相同的事情，一旦你熟悉Lambda库、迭代器和for_each的惯用法，对算法的使用就会清楚许多。

我们可以通过重写（rewriting）STL算法来对整个序列进行操作，从而略微进一步提高抽象层（就象MPL算法所做的那样），但眼下我们在这儿打住。从上面的简化例子可以看出，我们的例子存在的许多问题根本不是算法本身的缺点，真正的罪魁祸首是用于生成算法的函数实参的STL函数对象框架（STL function object framework）。撇开这些问题不谈，我们可以看出这些“平凡的”算法抽象掉一些并不算简单的低层细节：

- 临时迭代器的创建。
- 正确的迭代器类型声明，即使是在泛型代码中。
- 避免已知的低效性[⊜]。
- 利用已知的优化（例如循环开解）。
- 正确的泛化循环终结：for_each使用pos != finish而不是pos < finish来结束循环，后一种做法会受困于随机访问迭代器。

如果你考虑单个循环的话，这些看起来都很容易搞定，但是，当你考虑该模式在一个大的项目里反复出现时，出错的可能性就会迅速增长。上面提及的优化只会加重这种危险，它们通常甚至会引入更多的低层细节。

更重要的是，对for_each的使用获得了利害关系的分离：遍历和修改一个序列中的所有元素的常用模式被以算法的名义优雅地实现了，我们只要指定修改的细节就可以了。在编译期世界，这种劳动分工尤其重要，因为正如你可以从binary模板（见第1章有关讨论）中看到的那样，对哪怕最简单的重复过程进行编码都不会这么简单。能够使用程序库的已编写好的算法是一个很大的优势，因为这样一来，只需添加你正努力解决的问题的细节即可。

当你考虑到隐藏在算法背后的复杂性时，你就很容易明白复用STL算法的价值。例如std::lower_bound，它实现一个定制的二分查找（binary search），或者std::stable_sort，在低内存条件下，它会适当地降低性能。即使我们还没有使你确信，无论何时当需要对一个序列的所有元素进行操作时，你最好去调用std::for_each，我们还是希望你认同这一点：即使最简单的序列算法也提供了一个有用的抽象层。

⊖ 在这些例子中，_1指的是Boost Lambda库（位于boost::lambda命名空间）中的占位符对象（placeholder object）。MPL的占位符类型是受Lambda库的占位符对象启发而来的。

⊜ 当虑及效率时，最好避免使用后缀增量操作（iter++），对于绝大多数迭代器而言都是如此，由于为了返回其原始值，operator++实现在迭代器被增量前必须返回其一个副本。标准程序库实现者知道这个陷阱，因此只要有可能就刻意使用前缀增量运算符（++iter）。

6.2 MPL中的算法

就像STL算法那样，MPL算法实现有用的序列操作，并可用做更复杂的抽象的原始构建块(primitive building blocks)。在MPL算法集中，你将会发现它们就像标准<algorithm>头文件中的那些算法一样，几乎具有类似的命名。

然而，在STL和MPL算法之间也存在一些显著的区别。你已经知道元数据是不可变的(immutable)，因此MPL算法必须返回新的序列而不是就地（in place）改变它们，并且MPL算法直接对序列进行操作而不是对迭代器区间进行操作。选择对序列进行操作除了可以带给我们一个更高层的接口外，这还跟模板元编程的函数式性质有着紧密的关系。当结果序列必须被返回时，将一个操作的结果直接传递给另一个操作会变得很自然。例如：

```
// 给定一个非空序列Seq，返回Seq的一个副本，其中所有float实例被替代为double。
template <class Seq>
struct biggest_float_as_double
  : mpl::deref<
        typename mpl::max_element<
           typename mpl::replace<
               Seq
             , float
             , double
           >::type
             , mpl::less<mpl::sizeof_<_1>, mpl::sizeof_<_2> >
        >::type
  >
{};
```

然而，如果max_element和replace对迭代器而不是序列进行操作，biggest_float_as_double可能看起来如下：

```
template <class Seq>
struct biggest_float_as_double
{
    typedef typename mpl::replace<
    , typename mpl::begin<Seq>::type
    , typename mpl::end<Seq>::type
    , float
    , double
  >::type replaced;

  typedef typename mpl::max_element<
      typename mpl::begin<replaced>::type
    , typename mpl::end<replaced>::type
    , mpl::less<mpl::sizeof_<_1>, mpl::sizeof_<_2> >
  >::type max_pos;
```

```
    typedef typename mpl::deref<max_pos>::type type;
};
```

前一个对整个序列进行操作的结果，增大了互操作性（interoperability），因为一个算法的结果可以被平滑地传递给下一个算法。

6.3 插入器

在MPL和STL算法之间存在另一个重要的区别，也是源于模板元编程的函数式天性。“可构建序列的”STL算法家族，例如copy、transform以及replace_copy_if，都接受一个“指向被写入的结果序列”的输出迭代器。输出迭代器的全部要点是创建一个有状态的改变（stateful change）（例如为了修改一个现有的序列或扩充一个文件），但是在函数式编程中没有状态（state）。你如何写入一个MPL迭代器？结果放在哪儿？我们的例子没有使用有一丁点儿像输出迭代器的东西，它们只是构建一个和某个输入序列（input sequence）相同类型的新的序列。

每一个STL更易性算法（mutating algorithms）都可以将输出写入一个“不同于任何输入序列的类型”的序列中，或者当传递一个适当的输出迭代器时，可以做一些和序列完全无关的事情，比如打印到控制台（console）。MPL目标是使得在编译期可以做同样的事情，允许我们任意定制算法结果被处理的方式，这是通过使用插入器（inserter）而实现的[⊖]。

插入器不过是一个带有两个类型成员的类型：

- ::state，一个正通过算法运送的信息的表示。
- ::operation，一个用于从输出序列元素和现有的::state构建一个新的::state的二元操作。

例如，构建一个新的vector的插入器看起来如下：

```
mpl::inserter<mpl::vector<>, mpl::push_back<_,_> >
```

其中的mpl::inserter被定义为：

```
template <class State, class Operation>
struct inserter
{
    typedef State state;
    typedef Operation operation;
};
```

实际上，基于push_back和push_front而构建的插入器是如此有用，以至于它们已经被赋予熟悉的名字：back_inserter和front_inserter。以下是另一种更有号召力的方式，用于拼写构建vector的插入器：

```
mpl::back_inserter<mpl::vector<> >
```

当传递给一个MPL算法（例如copy）时，它的功能类似于如下STL代码中的std::back_inserter：

⊖ “插入器（inserter）”这个名字是受到STL的创建输出迭代器的函数适配器（output-iterator-creating function adaptors）家族的启发，后者包括std::front_inserter和std::back_inserter。

```
std::vector<any> v; // 起先是个空vector
std::copy(start, finish, std::back_inserter(v));
```

现在让我们通过使用mpl::copy将一个list中的元素复制到一个vector中，来看看一个插入器实际上是怎么工作的。自然，mpl::copy接收一个输入序列（代替std::copy的输入迭代器对组）和一个插入器（代替std::copy的输出迭代器），因此调用起来如下：

```
typedef mpl::copy<
    mpl::list<A, B, C>
  , mpl::back_inserter<mpl::vector<> >
>::type result_vec;
```

在算法的每一步，插入器的二元操作以从上一步得到的结果（在第一步中，则是以插入器的初始类型）作为其第一个参数而进行调用，并且把正常来说被写入输出迭代器的元素作为其第二个参数。算法在最后一步返回结果，因此以上等价于：

```
typedef

  mpl::push_back<              // >-----------------+
                               //                   |
      mpl::push_back<          // >--------------+  |
                               //                |  |
          mpl::push_back<      // >-------------+|  |
              mpl::vector<> //                  ||  |
            , A             //                  ||  |
            >::type         // 第一步        <-+||  |
          , B               //                  ||  |
          >::type           // 第二步         <--+|  |
        , C                 //                   |  |
    >::type                 // 第三步        <-----+

result_vec;
```

由于希望构建和你正在操作的序列同类型的序列非常常见，因此MPL为所有的序列构建算法提供了默认的插入器。这就是为何在3.1.4 和3.1.5小节中，我们能够如此有效地使用transform而无需指定一个插入器的原因。

注意，一个插入器完全可以做跟插入无关的动作。以下例子使用一个插入器来计算一个序列的序列中每个元素的第一个元素的和：

```
typedef mpl::vector<
    mpl::vector_c<int, 1, 2, 3>
  , mpl::vector_c<int, 4, 5, 6>
  , mpl::vector_c<int, 7, 8, 9>
> S;

typedef mpl::transform<
    S                       // 输入序列
```

```
    , mpl::front<_>          // 选择front元素

    , mpl::inserter<
          mpl::int_<0>       // 结果从0开始起算
        , mpl::plus<_,_>     // 累加每一个输出元素
      >
>::type sum; // 0 + 1 + 4 + 7 == 12
```

如果没有插入器，transform将构建一个vector（由S中的每一个序列的初始元素构成），有了插入器，这些初始元素被给予mpl::plus<_,_>，以初始值mpl::int_<0>开始。

6.4 基础序列算法

back_inserter使用的模式（将序列元素“折叠（folding）”进一个结果），本质上和在函数式环境中处理序列的方式一样。Haskell和ML的用户立刻就会意识到这是fold函数所使用的模式（STL中坚用户会认识到它就是std::accumulate使用的模式）。伪代码如下：

```
fold(Seq, Prev, BinaryOp) :=
   if Seq is empty then:
       Prev
   else:          // 将第一个元素与Prev进行“联合（combine）”
       fold(      // 并递归地处理余下的元素
           tail(Seq)
         , BinaryOp(Prev, head(Seq))
         , BinaryOp
       )
```

从调用者的角度来看，Prev可能应该称为InitialType，我们之所以选择Prev这个名字，是因为它可以使理解算法的实现更加容易。除了第一步，在处理过程的每一步中，Prev都是前一步处理过程的结果。

你可以在fold基础上构建很多其他更复杂的序列遍历算法。例如，我们可以采用如下方式反转任何前向可扩展的序列（front-extensible sequence）：

```
template <class Seq>
struct reverse
  : mpl::fold<
        Seq
      , typename mpl::clear<Seq>::type // 初始类型
      , mpl::push_front<_,_>           // 二元操作
    >
{};
```

fold一个值得注意的奇怪属性是，当我们将它与push_front一起使用时，所得结果总是为逆序的方式。由于list只能采用push_front来构建，且使用reverse来按正确的顺序往后移（每当生成一个list结果时）极其低效，MPL还提供了一个reverse_fold 元函数，它以逆序处理元素。对一个

只能前向遍历的序列高效地做这件事，乍看起来颇有点耍小聪明的味道，其实非常简单。不是对序列的第一个元素进行操作并折叠（folding）余下的元素，而是首先fold其余的元素然后操作第一个元素：

```
reverse_fold(Seq, Prev, BinaryOp) :=
   if Seq is empty then:
       Prev
   else:              // 处理序列中其余的元素
       BinaryOp(      // 和第一个元素相“联合”
           reverse_fold(tail(seq), Prev, BinaryOp)
         , head(seq)
       )
```

不是在遍历的入口处理每一个序列元素，而是在遍历的出口处理它们。

既然我们可以在入口或出口处理元素，为什么不能同时做到二者呢？MPL的reverse_fold实际上比我们已经展示给你的更通用一些。第四个可选参数可以用于供应一个“inward”或“forward”操作。因此算法实际上更像如下的模样：

```
reverse_fold(Seq, Prev, OutOp, InOp = _1) :=
   if Seq is empty then:
       Prev
   else:
       OutOp(
           reverse_fold(
               tail(Seq)
             , InOp(Prev,head(Seq)) // 默认情况下就是Prev
             , OutOp
             , InOp)
         , head(Seq)
       )
```

这个一般化（generalization）允许我们充分利用递归序列遍历固有的双向模式。注意InOp默认只是一个返回其首个实参的函数。当我们不供应InOp时，就好像Pre被直接传给递归调用一样。

在我们结束低层迭代算法继续前进之前，在MPL的fold算法家族中还有一个一般化（generalization）需要讨论：不是对序列的元素（elements）进行迭代，我们可以对位置（positions，也就是迭代器的值）进行迭代。这是有用的，例如，如果我们希望处理输入序列的一个连续的子区间。由于我们总是可以获取一个迭代器所引用的元素，采用iter_fold来折叠序列迭代器，而不是采用普通的fold来折叠序列元素，略微一般化一些。iter_fold的伪码定义如下：

```
iter_fold(Seq, Prev, BinaryOp) :=
   if Seq is empty then:
       Prev
   else:              // 将第一个position与Prev相联合
```

```
        iter_fold(     // 并递归地处理其余的position
            tail(Seq)
          , BinaryOp(Prev, begin(Seq))
          , BinaryOp
        )
```

fold和iter_fold之间的主要区别是，BinaryOp第二个参数是一个迭代器而不是一个元素。自然而然，完整的一般化reverse_iter_fold也有提供：

```
reverse_iter_fold(Seq, Prev, OutOp, InOp = _1) :=
    if Seq is empty then:
        Prev
    else:
        OutOp(
            reverse_iter_fold(
                tail(Seq)
              , InOp(Prev, begin(Seq))
              , OutOp
              , InOp)
          , begin(Seq)
        )
```

6.5 查询算法

表6.1描述了MPL的序列查询算法。其中大多数对于STL用户来说应该是很熟悉的，一个可能的例外是contains，它是如此简单有用，也许本来就应该首先成为STL的一个算法。类似于相应的STL算法，compare判断式默认为mpl::less<_,_>，然而，如果你供应了一个判断式的话，那必须导致其参数的严格的弱序（strict weak ordering）。

表6.1 序列查询算法

元函数	结果::type	复杂度
`mpl::find<seq, T>`	一个iterator，指向seq中第一次出现的T，如果没找到，则为mpl::end<seq>::type	线性
`mpl::find_if<seq,T, pred>`	一个iterator，指向seq中第一个满足判断式pred的元素，如果没找到，则为 mpl::end<seq>::type	线性
`mpl::contains<seq, T>`	当且仅当seq包含T时为true	线性
`mpl::count<seq, T>`	T在seq中出现的次数	线性
`mpl::count_if<seq, pred>`	满足判断式pred的元素在seq中出现的次数	线性
`mpl::equal<seq1, seq2>`	当且仅当seq1和seq2包含同样的元素且元素顺序相同时为true	线性

`mpl::lower_bound<`
`  seq, T`
`, compare`
`>`
	序列seq中第一个可供T插入的位置，seq已根据compare规则排序	对于compare的调用为对数复杂度。对于随机访问序列的遍历为对数复杂度，否则为线性复杂度

（续）

元函数	结果::type	复杂度
mpl::upper_bound< seq, T , compare >	序列seq中最后一个可供T插入的位置，seq已根据compare规则排序	对于compare的调用为对数复杂度。对于随机访问序列的遍历为对数复杂度，否则为线性复杂度
mpl::max_element< seq , compare >	seq中的第一个这样的position p：对于seq中的所有positions q，满足 mpl::apply< compare , mpl::deref<p>::type , mpl::deref<q>::type >::type::value == false	线性
mpl::min_element< seq , compare >	seq中的第一个这样的position p：对于seq中的所有positions q，满足 mpl::apply< compare , mpl::deref<q>::type , mpl::deref<p>::type >::type::value == false	线性

6.6 序列构建算法

所有MPL的序列构建算法都遵从同样的模式。这虽有点故意为之，但结果是它们非常易于使用。模式如下，对于任何序列构建算法xxxx都是如此。

- 对于算法xxxx，存在一个相应的算法reverse_xxxx，它接收同样的参数，但以逆序操作输入序列元素。我们称xxxx和reverse_xxxx为配对算法（counterpart algorithm）。
- 算法的最后一个参数为一个可选的插入器。
- 如果未指定插入器：
  - 如果第一个序列参数Seq是向后可扩展的（back-extensible），结果就好像

```
mpl::back_inserter<mpl::clear<Seq>::type>
```

  被作为插入器传入了一样。
  - 否则，结果就好像配对算法被以

```
mpl::front_inserter<mpl::clear<Seq>::type>
```

  作为插入器而被调用。

让我们看看这种模式在实践中运作的情况。在下面的例子中，v123指示一个具有“vector属性”的类型，等价于mpl::vector_c<int, 1,2,3>。类似地，1876指示一个等价于mpl::list_c<int, 8,7,6>的类型。

```
// 起始序列
typedef mpl::vector_c<int, 1, 2, 3> v123;
```

```
typedef mpl::list_c<int, 1, 2, 3>   l123;

// 转换
typedef mpl::plus<_1,mpl::int_<5> > add5;

// 使用默认的插入器
typedef mpl::transform<v123, add5>::type          v678;
typedef mpl::transform<l123, add5>::type          l678;
typedef mpl::reverse_transform<v123, add5>::type  v876;
typedef mpl::reverse_transform<l123, add5>::type  l876;
```

这样，简单的无插入器形式为可向前扩展的和可向后扩展的序列都产生期望的结果。然而，为了使用带有插入器的版本，我们必须知道算法的遍历方向和我们正在构建的序列的属性：

```
// 这个插入器等价于默认的插入器
typedef mpl::transform<
    v123, add5, mpl::back_inserter<mpl::vector<> >
>::type                                           v678;

// 也等价于默认的插入器
typedef mpl::reverse_transform<
    l123, add5, mpl::front_inserter<mpl::list<> >
>::type                                           l678;

// 输入序列的属性不影响结果
typedef mpl::reverse_transform<
    v123, add5, mpl::front_inserter<mpl::list<> >
>::type                                           l678;
```

在构建一个新序列中使用的插入器应该总是通过结果序列的可向前扩展性或可后向扩展性来决定。程序库的默认插入器选择遵从同样的规则，只是当没有用户供应的插入器时，恰巧结果序列的属性与输入序列的属性相同。

表6.2总结了序列构建算法。注意，reverse_形式和带有可选插入器参数的构建算法都没有列出来，但根据上面的描述你应该可以推导它们的存在和行为。在MPL参考手册中对它们也有详细的讨论。我们应该注意到copy和reverse对于命名规则是个例外：它们互为逆序版本，程序库中不存在什么reverse_copy 或reverse_reverse算法。

表6.2 序列构建算法

元函数	结果::type
mpl::copy<seq>	seq的元素
mpl::copy_if<seq, pred>	满足判断式pred的seq的元素
mpl::remove<seq, T>	一个等价于seq的序列，但没有任何和T一致的元素
mpl::remove_if<seq, pred>	等价于seq，但没有任何满足判断式pred的元素
mpl::replace<seq, old, new>	等价于seq，但是其中出现的所有old都被new所取代

（续）

元函数	结果::type
`mpl::replace_if<seq, pred, new>`	等价于seq，但所有满足pred的元素都被new所取代
`mpl::reverse<seq>`	逆序的seq的元素
`mpl::transform<seq, unaryOp>`	对seq的连续元素调用unaryOp的结果，或对seq1和seq2之间的连续元素调用 binaryOp的结果
`mpl::transform<seq1, seq2, binaryOp>`	
`mpl::unique<seq>`	由seq的每一个子区间的初始元素构成的序列，其中每一个子区间的元素均相同。如果提供了等价判定关系equiv，则用它来决定子区间元素的相同性[⊖]
`mpl::unique<seq, equiv>`	

序列构建算法都具有线性复杂度，并且所有算法都默认返回一个（与它们第一个输入序列相同类型的）序列，但是通过使用一个适当的插入器，你可以产生你喜欢的任何种类的结果。

别名下的函数式算法（Functional Algorithms Under Aliases）

很多序列构建算法虽然其名字取自类似的STL算法，但实际上起源于函数式编程世界。例如，两个参数版本的transform，对于函数式程序员来说就是“map”，三个参数版本的transform有时称为“zip_with”，而copy_if也称为“filter”。

由于我们将reverse_算法遗漏于表6.2之外，为了公平，有必要指出，接收一个等价关系的形式的unique 在所有序列构建算法中具有独特性。大多数算法的reverse_形式产生和正常形式相同的元素（以逆序），但是接收相同实参的unique和reverse_unique所产生的序列的元素可能不同。例如：

```
typedef mpl::equal_to<
    mpl::shift_right<_1, mpl::int_<1> >
  , mpl::shift_right<_2, mpl::int_<1> >
> same_except_last_bit;                          // 判断式

typedef mpl::vector_c<int, 0,1,2,3,4,5> v;

typedef unique<
     v, same_except_last_bit
>::type                         v024;           // 0, 2, 4

typedef reverse_unique<
     v, same_except_last_bit
>::type                         v531;           // 5, 3, 1
```

6.7 编写你自己的算法

对于希望实现一个进行低层序列遍历的元函数的任何人，我们给予的首要忠告是“让我们来遍历”。简单地复用MPL算法作为构建高阶算法的“原语（primitive）”通常更加有效。你会说我们在第3章根据transform实现divide_dimensions时已经采用了这种方式。你不仅仅节省编码

⊖ 换句话说，该算法用于移除序列中的“重复”元素。——译者注

工作量：MPL的原始迭代算法已经被专门编写以避免深度模板实例化，后者会急剧减慢编译速度，甚至导致编译失败[⊖]。很多MPL算法最终根据iter_fold来实现，就是出于同样的理由。

由于MPL提供了大量的线性遍历算法保留曲目，因此，如果你发现你必须编写进行自己序列遍历的元函数，很可能是因为你需要一些其他的遍历模式。在这种情况下，你的实现不得不使用我们在第1章介绍的binary模板那样的基本递归形式，使用一个特化版本来终结递归。我们建议你操作迭代器而不是对同一个序列进行连续的增量修改，原因有二：首先，这样做对于较广泛的序列来说都是高效的。并非所有序列都支持O(1)复杂度的 pop_front操作，即使在支持O(1)复杂度的 pop_front操作中，也可能具有相当大的常量因子（constant factor）。其次，正如我们在iter_fold中看到的那样，操作迭代器比操作序列元素略微更具有一般性（通用性）。这种额外的一般性（通用性）会招致细微的实现时间代价，但在算法复用性上可以带来诸多好处。

6.8 细节

抽象

一种强调高级概念而不强调实现细节的思想。运行期C++中的类就是常用于将状态和相关联的过程（associated process）进行打包的抽象。函数是最基础的抽象之一，很明显在任何函数式编程上下文中都很重要。MPL算法是重复过程的抽象，采用元函数实现。MPL中算法抽象的价值通常比相应的STL算法的价值高，因为替代方式在编译时间方面表现得非常糟糕。尽管我们可以采用for循环和一对迭代器来遍历STL序列，但一个手写的编译期序列遍历总是需要至少一个新的类模板和一个显式特化。

折叠（Fold）

一个原始的（简单的）函数抽象，将一个二元函数重复地应用于一个序列的元素和一个附加（额外的）值身上，将每一步函数的结果用做下一步的附加值。STL采用accumulate这个名字来实现同样的抽象。MPL以两种方式将fold一般化：通过操作迭代器而不是元素（iter_fold），以及通过提供双向遍历（reverse_[iter_] fold）。

查询算法（Querying algorithm）

MPL支持多种返回迭代器或简单值的算法，它们一般精确地对应于同名的STL算法。

序列构建算法（Sequence building algorithm）

要求接收输出迭代器实参的STL算法，对应于MPL的前后和后向“序列构建”算法对组（pair）（默认情况下，MPL算法版本构建与它们第一个输入序列同类型的新序列）。它们还接收一个可选的插入器参数，该参数赋予对算法的结果更大的控制。

插入器（Inserter）

按照STL的习惯，一个名字以“inserter”结束的函数，创建一个输出迭代器，用于将元素添加进序列。MPL使用这个术语来指示一个被打包并有一个附加值的二元元函数，后者用作一个算法的结果元素的输出处理器。算法使用默认的插入器是front_inserter<S>和back_inserter<S>，它们分别使用push_front或push_back将结果折叠（fold）进S。对算法使用一个插入器，等价于

⊖ 参见附录C以了解更多的信息。

将fold应用于算法的默认（非插入器）结果、插入器的函数及其初始状态。这样，对一个序列构建算法而言，只有当其插入器的函数组件这么做的时候，它才真正构建序列。比如说，如果使用的是一个mpl::multiplies<_>，那么算法的结果将是一个数字而非一个序列。

6.9 练习

6-0. 使用mpl::copy和一个手工构建的插入器来编写一个smallest 元函数，用于查找类型序列中最小的一个。即：

```
BOOST_STATIC_ASSERT((
    boost::is_same<
        smallest< mpl::vector<int[2], char, double&> >::type
      , char
    >::value
));
```

既然你已经做到了这一点，那么这是解决该问题的好办法吗？请说明原因。

6-1. 使用MPL序列迭代算法之一重新编写1.4.1节中的binary模板，并编写一个测试程序，只有当你的binary模板可以工作时，它才能通过编译。将你编写的代码的数量和第1章展示的手工编写递归版本进行比较。是什么原因导致出现那样的比较结果？

6-2. 由于std::for_each是标准库中最基本的算法，你可能对我们只字未提它的编译期对应物感到奇怪。是这样的，与transform这样的算法不同，for_each没有纯粹的编译期对应物。你能对此提供一个合理的解释吗？

6-3. 编写一个名为binary_tree_inserter的插入器类模板，它利用练习5-10中的tree模板递增地构建一个binary二叉查找树：

```
typedef mpl::copy<
      mpl::vector_c<int,17,25,10,2,11>
    , binary_tree_inserter< tree<> >
    >::type bst;

//        int_<17>
//        /      \
//    int_<10>  int_<25>
//     /    \
// int_<2> int_<11>
BOOST_STATIC_ASSERT(( mpl::equal<
      inorder_view<bst>
    , mpl::vector_c<int,2,10,11,17,25>
    >::value ));
```

6-4. 编写一个名为binary_tree_search的算法元函数，它对使用练习6-3中的binary_tree_inserter构建的树进行二分查找。

```
typedef binary_tree_search<bst,int_<11> >::type pos1;
```

```
typedef binary_tree_search<bst,int_<20> >::type pos2;
typedef mpl::end<bst>::type                     end_pos;
BOOST_STATIC_ASSERT((!boost::is_same< pos1,end_pos >::value));
BOOST_STATIC_ASSERT((boost::is_same< pos2,end_pos >::value));
```

6-5. 列出标准库中所有的算法，并与MPL提供的算法集合进行比较。分析它们之间的区别。哪些算法命名方式不同？哪些算法具有不同的语义？哪些算法在MPL中没有对应物？你认为它们在MPL中缺席的原因是什么？

第7章　视图与迭代器适配器

像transform这样的算法提供了一种操作序列的方式。这一章讨论序列视图（sequence view）的用法。序列视图是一种强大的序列处理惯用法，通常优于使用算法。

首先，我们来看一个非正式的定义：

> **序列视图**
>
> 序列视图（Sequence View），或简称视图（View），是一种惰性适配器（lazy adaptor），它提供有关一个或多个底层序列的变更后的表示。

视图是惰性的（lazy）：它们的元素只在需要时才被计算。我们在第3章讨论无参元函数和在第4章讨论eval_if时看过惰性评估（lazy evaluation）的例子。与其他惰性构造一样，视图可以帮助我们避免那种结果永远都不被使用的计算所导致的过早的错误与低效。而且，序列视图有时比其他途径更适合特定的问题，可以产生更简单、更有表达力、可维护性更好的代码。

在这一章中你将会搞明白视图是如何工作的，我们将讨论如何和何时使用它们。然后我们将探讨MPL中提供的视图类模板，你也将学习如何编写自己的视图类模板。

7.1　一些例子

在下面的几个小节中，我们将探讨一些问题，它们都尤其适合使用视图，对这些问题的探讨将会为你了解有关视图的方方面面提供一个良好的认知。我们希望向你证明视图的思想物有所值，并且在这些情况下使用视图进行编程要么更高效，要么更自然。

7.1.1　对从序列元素计算出来的值进行比较

让我们从一个简单的问题开始，它会告诉你视图是如何工作的：

编写一个元函数 padded_size，给定一个整数MinSize和一个序列Seq，Seq中容纳的是按大小递增的类型，返回Seq中第一个满足条件sizeof(e) >= MinSize的元素的大小。

第一个解决方案

现在让我们使用到目前为止讨论过的工具来尝试解决这个问题。现在的事实是，我们是在一个有序的（已排序的，sorted）序列中查找，这是一个线索，我们希望在解决方案核心使用二分查找算法upper_bound或lower_bound其中之一。“我们希望查找第一个满足该属性的元素”这一事实，将选择方案限制为lower_bound，从而使我们可以勾勒出解决方案的轮廓：

```
template<class Seq, class MinSize>
struct padded_size
  : mpl::sizeof_<                          // 第一个位置的元素的大小
        typename mpl::deref<
```

```
            typename mpl::lower_bound<
                Seq
              , MinSize
              , comparison predicate    // 满足……。
            >::type
        >::type
    >
{};
```

在英语中，这里的含义是返回满足某个条件的第一个位置（第一次出现）的元素的大小”，其中“some condition（某个条件）”是通过传递给lower_bound的comparison predicate来决定的。

我们希望满足的条件（condition）是sizeof(e) >=MinSize。如果你查查MPL参考手册中对lower_bound的描述，你就会明白其简单的描述并不能真正适应这种情形：

返回有序序列（即Seq）中的第一个位置，其中T（即MinSize）可以在不破坏顺序的情况下插入。

毕竟，Seq是按照元素大小排过序的，我们也并不关心整型常量外覆器MinSize的尺寸：我们没打算插入它。这个对lower_bound的简单描述的问题在于，它是专门为同质的比较判断式设计的，其中T是一个可能的序列元素（potential sequence element）。现在，如果你在lower_bound参考中略微再多读一点儿文字，你将会发现如下的条目：

```
typedef lower_bound< Sequence, T, Pred >::type i;
```

返回类型：前向迭代器的一个model

语义：p是序列中最远的迭代器（furthermost iterator），因此，对于

```
[begin<Sequence>::type, i),
```

中的每一个迭代器q，

```
apply<Pred, deref<j>::type, T >::type::value
```

为true。

这意味着，“lower_bound的情景是Sequence中满足如下条件的最后一个位置：对该位置之前的任意元素和T应用判断式，都将产生true。”这个更精确的描述使得lower_bound看上去可以解决我们的问题，我们希望获得这样的最后一个位置：对于前面位置中的所有元素e，sizeof(e) < MinSize::value。因此，判断式（predicate）将为：

```
mpl::less<mpl::sizeof_<_1>, _2>
```

将该判断式（predicate）插入到完整的元函数中，就得到了：

```
template<class Seq, class MinSize>
struct padded_size
  : mpl::sizeof_<
        typename mpl::deref<
            typename mpl::lower_bound<
                Seq
              , MinSize
```

```
                , mpl::less<mpl::sizeof_<_1>, _2>
            >::type
        >::type
    >
{};
```

分析

现在，让我们回头看看已经做的工作。如果你像我们一样，你应该会对代码质量产生一丝隐忧。

首先，编写这么一个简单的元函数可能不应该让我们耗费这么多的时间来查阅MPL参考手册。一般而言，如果你编写一段代码花费了一段困难的时间，那么几乎可以肯定维护代码的人将会花费更困难的一段时间来尝试读懂它。毕竟，代码的作者至少还有知道他自己意图的优势。在这个例子中，lower_bound处理异质比较（heterogeneous comparisons）的方式，以及判断式的参数顺序，都要求要花费不少时间进行研究，而且还不容易被记住。叫后来人熟读手册，以便可以理解我们编写的东西，这太不厚道了。而且，那些“后来人”说不定就是我们自己！

其次，即使撇开需要咨询参考手册这一点不谈，上面的代码中还有一些令人奇怪的地方，我们在lower_bound的调用中计算序列元素的大小，然后再一次在lower_bound返回给我们的位置请求元素的大小。退一步说，自我重复是令人厌烦的。

一个简化

幸运的是，这种重复实际上为我们该如何改善这个状况提供了一个线索。我们在一个按照尺寸（size）排序的元素序列中查找，将每一个元素的尺寸与一个给定的值进行比较，并返回我们找到的元素的尺寸。说到底，我们感兴趣的不是序列中的元素自身，而是这些元素的尺寸。进一步而言，如果我们可以在一个“尺寸序列（a sequence of sizes）”上进行查找，我们就可以使用一个同质的比较判断式（homogeneous comparison predicate）：

```
template<class Seq, class MinSize>
struct padded_size
  : mpl::deref<
        typename mpl::lower_bound<
            typename mpl::transform<
                Seq, mpl::sizeof_<_>
             >::type
          , MinSize
          , mpl::less<_,_>
        >::type
    >
{};
```

事实上，mpl::less<_,_>已经是lower_bound的默认判断式（default predicate），因此我们可以进一步地简化实现：

```
template<class Seq, class MinSize>
```

```
struct padded_size
  : mpl::deref<
        typename mpl::lower_bound<
            typename mpl::transform<
                Seq, mpl::sizeof_<_>
            >::type
          , MinSize
        >::type
 >
{};
```

由于本章旨在为视图构建一个用例，自然，这个简化了的实现品中同样存在一个问题：效率太低。我们第一个实现对mpl::sizeof_的调用只对lower_bound访问的O(log N)个元素进行（在其二分查找过程中），而这一个则使用transform贪婪地计算序列中每一个类型的尺寸。

快速且简单

幸运的是，我们可以将贪婪的尺寸计算转换为一个使用transform_view的惰性的计算，从而实现一个比以上两个都好的解决方案：

```
template<class Seq, class MinSize>
struct first_size_larger_than
  : mpl::deref<
        typename mpl::lower_bound<
            mpl::transform_view<Seq, mpl::sizeof_<_> >
          , MinSize
        >::type
    >
{};
```

transform_view<S,P>是一个元素和transform<S,P>的元素一致的序列，但有两点重要的区别：

1. 其元素总是按需计算，换句话说，这是一个惰性序列（lazy sequence）。

2. 通过它的任何迭代器中的::base成员，我们可以获得一个指向S中相应位置的迭代器。[⊖]

如果我们采取的这种途径看上去有点儿面生，可能是因为人们不经常在运行期C++中这样编写代码。然而，一旦暴露出惰性的优点，你就会迅速发现有一大类算法问题与之类似，使用视图来解决它们是很自然的，即使在运行期也是如此[⊖]。

7.1.2 联合多个序列

只有一个编译期序列构建算法transform对于操作来自两个输入序列的元素对（pairs of elements）有着直接的支持。要不是考虑它的实用性，这个在程序库设计上的不一致性，几乎可以算得上一个审美缺陷：它不过是一个对方便性以及与STL保持一致性的让步。对于其他种类的多序列操作，或者transform三个或更多个输入序列，我们需要一种不同的策略。

⊖ 我们将在7.3节讲述base。

⊖ 参见本章末尾的“历史”小节，以便了解一些关于运行期视图库（runtime views libraries）的参考。

你可以“手工”编码任何新的多序列算法，但正如你可能猜测的那样，我们更鼓励你复用一些为此目的而提供的MPL工具。实际上，有一个组件可以让你使用你信赖的单序列工具来解决任何并行多序列问题。MPL的zip_view将一个具有N个输入序列的序列，转换为一个具有N个元素的序列的序列（由输入序列中选取的元素构成）。因此，如果S是[s1, s2, s3 ...]，T是[t1, t2, t3 ...]，且U是[u1, u2, u3 ...]，那么zip_view<vector<S,T,U> >的元素就是[[s1, t1, u1]、[s2, t2, u2]、[s3, t3, u3] ...]。

例如，三个vector按元素进行求和可以写成：

```
mpl::transform_view<
    mpl::zip_view<mpl::vector<V1,V2,V3> >
  , mpl::plus<
        mpl::at<_, mpl::int_<0> >
      , mpl::at<_, mpl::int_<1> >
      , mpl::at<_, mpl::int_<2> >
    >
>
```

这种做法并不太糟糕，但是我们不得不承认这种使用mpl::at来对vector元素进行解包（unpacking）的做法不但麻烦，而且丑陋。我们可以使用MPL的unpack_args外覆器来整理代码（使得代码整洁一些），该外覆器将一个形如mpl::plus<_,_,_>的具有N个参数的lambda表达式转换为一个一元lambda表达式。当应用于一个具有N个元素的序列时，

```
mpl::unpack_args<lambda-expression>
```

提取每一个序列的N个元素并将它们作为连续的参数传递给lambda表达式。

哦，这个描述真有点绕人，幸运的是，一丁点儿代码也胜千言。以下对按元素求和的等价改写，使用unpack_args可以显著地改善可读性：

```
mpl::transform_view<
    mpl::zip_view<mpl::vector<V1,V2,V3> >
  , mpl::unpack_args<mpl::plus<_,_,_> >
>
```

7.1.3 避免不必要的计算

即使视图在概念上对你没有什么吸引力，你仍然会使用它们来解决能从其惰性特质中获益的问题。现实世界的例子数不胜数，在此我们只列举少数几个：

```
// seq包含int、int&、int const&、int volatile&
// 或int const volatile&吗?
typedef mpl::contains<
    mpl::transform_view<
        seq
      , boost::remove_cv< boost::remove_reference<_> >
    >
  , int
```

```
>::type found;

// 找到阶乘>= n 的最小整数的位置
typedef mpl::lower_bound<
    mpl::transform_view< mpl::range_c<int,0,13>, factorial<_1> >
  , n
>::type::base number_iter;

// 返回一个“由seq1和seq2中所有元素构成”的有序vector
typedef mpl::sort<
    mpl::joint_view<seq1,seq2>
  , mpl::less<_1,_2>
  , mpl::back_inserter<mpl::vector<> >
>::type result;
```

以上最后一个例子使用joint_view，这是一个由其参数的元素首尾相连所组成的序列。在每个例子中，对惰性技术（视图）的使用，使得与相应的饥渴方式（eager approach）相比，节省了显著数量的模板实例化。

7.1.4 选择性的元素处理

使用filter_view——一个惰性版本的filter算法，我们可以处理一个序列元素的一个子集，而无需构建一个中间序列。当一个filter_view的迭代器递增时，序列的一个底层的迭代器被步进，直到该“filter”函数满足条件：

```
// 由Seq中所有指针元素所指向的对象构成的序列
mpl::transform_view<
    mpl::filter_view< Seq, boost::is_pointer<_1> >
  , boost::remove_pointer<_1>
>
```

7.2 视图Concept

到目前为止，你可能对视图有了相当不错的理解，让我们再稍微强化一下这个思想。首先，这一小节的标题可能应该称做“视图concept”，即使用小写的concept，因为通常当我们在C++中说“Concepts”时，我们指的是如第5章中描述的正式的接口要求。视图要比它随意一些。从一个接口角度来看，视图就是序列，之所以是个视图是因为两个实现细节。首先，正如我们已经重复的以致于你的耳朵都长了茧子：视图是惰性的，它们的元素只在需要时才被计算。然而，并非所有惰性序列都是视图。例如，range_c<...>是一个熟知的惰性序列例子，但它看起来不大像“任何东西之上的”一个视图。成为视图的第二个细节要求就是元素必须从一个或多个输入序列中产生。

所谓突现特征（emergent property）就是因为一些更基础的特征而出现的属性。所有视图都拥有两个突现特征。首先（这其实适用于所有惰性序列，因为它们的元素是计算而得），视图是不可扩充的。如果你需要扩充性，需要使用copy算法从视图创建一个可扩充的序列。其次，由

于迭代器“仲裁”所有的元素访问，因此，在实现一个序列视图时涉及的大多数逻辑都被包含于相应的迭代器之中。

7.3 迭代器适配器

序列视图的迭代器是迭代器适配器（iterator adaptor）的例子，它自身是一个重要的concept（小写的c）。正像一个视图是构建于一个或多个底层序列之上的序列那样，一个迭代器适配器是这样的一种迭代器：它适配一个或多个底层迭代器的行为。

在运行期C++中，迭代器适配器是如此有用，以至于有一整个Boost程序库专注于它们。就连STL也包含有几个迭代器适配器，名气最大的当属std::reverse_iterator，它与底层的迭代器遍历同样的元素序列，但遍历的方向恰好相反。mpl::filter_view的迭代器是另一个迭代器遍历适配器（iterator traversal adaptor）例子。而一个迭代器访问适配器（iterator access adaptor）则访问不同于其底层迭代器的元素值，就像mpl::transform_view的迭代器所做的那样。

因为可以通过调用其base()成员函数来访问std::reverse_iterator的底层迭代器，所以MPL适配器也通过嵌套的::base类型提供了对它们的底层迭代器的访问。在所有其他方面，一个迭代器适配器就和任何其他迭代器一样。它可以属于三个迭代器范畴之一（可能不同于其底层迭代器），并且所有普通迭代器的条件要求对它都适应。

7.4 编写你自己的视图

由于大多数序列视图的智能都包含在其迭代器中，因此实现一个视图的大部分工作涉及到实现一个迭代器适配器也就是顺理成章的事情。让我们为zip_view打造一个迭代器，看看它是如何工作的。

由于zip_view操作输入序列的一个序列，因此，其迭代器应该操作那些输入序列中的一个迭代器序列也就是自然而然的了。让我们给zip_iterator一个迭代器序列参数：

```
template <class IteratorSeq>
struct zip_iterator;
```

MPL的zip_iterator模塑（models）了它的任何组件迭代器的最低精化concept，但为了简化，我们的zip_iterator将永远是一个前向迭代器。对于前向迭代器需要满足的惟一条件要求是，采用mpl::deref进行解引用（dereferencing），以及采用mpl::next进行递增操作。为了解引用一个zip_iterator，我们需要解引用其每一个组件迭代器，并将结果打包到一个序列中。利用mpl::deref的默认定义（即以元函数的方式调用实参），zip_iterator本体因此被定义如下：

```
template <class IteratorSeq>
struct zip_iterator
{
    typedef mpl::forward_iterator_tag category;

    typedef typename mpl::transform<
        IteratorSeq
```

```
        , mpl::deref<_1>
    >::type type;
};
```

类似地，为了递增一个zip_iterator，我们需要递增每一个组件迭代器：

```
namespace boost { namespace mpl
{
  // 针对zip_iterator 的next<...> 特化
  template <class IteratorSeq>
  struct next< ::zip_iterator<IteratorSeq> >
  {
      typedef ::zip_iterator<
          typename transform<
              IteratorSeq
            , next<_1>
          >::type
      > type;
  };
}}
```

为了简便，我们希望加入zip_iterator本体中的就只剩下::base成员了，它可以用于访问正被适配的迭代器。在一个用于单个迭代器的迭代器适配器中，::base就是那一个迭代器。但在这个例子中，它是一个底层迭代器序列：

```
template <class IteratorSeq>
struct zip_iterator
{
    typedef IteratorSeq base;
    ...
};
```

zip_view要做的事情很少，它不过是一个使用zip_iterator的序列。实际上，我们可以根据iterator_range来构建zip_view：

```
template <class Sequences>
struct zip_view
  : mpl::iterator_range<
        zip_iterator<
            typename mpl::transform<
                Sequences, mpl::begin<_1>
            >::type
        >
      , zip_iterator<
            typename mpl::transform<
                Sequences, mpl::end<_1>
            >::type
```

```
        >
    >
{};
```

7.5 历史

在程序设计领域，尤其是在函数式编程社群，关于惰性评估和惰性序列有一段很长的历史。第一个知名的C++ “view” concept例子出现于1995年Jon Seymour编写的一个（运行期）程序库中，被恰当地命名为“Views” [Sey96]。有趣的是，视图库的方式更多地是受数据库（database）技术的启发而非函数式编程。关于视图concept一个更完备的实现出现在View Template Library （VTL）中，作者是Martin Wieser和Gary Powell，完成于1999年[WP99, WP00]。在2001年，实现和适配C++迭代器被认为是一项重要的任务，于是Boost迭代器适配器库应运而生[AS01a]。

7.6 练习

7-0. 编写一个测试程序，练习我们的zip_view实现品。设法安排你的程序，使得只有当测试成功时才可以通过编译。

7-1. 我们的zip_iterator实现使用transform来生成其嵌套的::type。在这个例子中，若改用transform_view是否有优势？

7-2. 修改zip_iterator，使得其迭代器范畴反映出由其任何底层迭代器模塑的最低精化的（least-refined） concept。扩充这个迭代器实现品，使得满足计算范畴的所有可能的条件要求。

7-3. 使用mpl::joint_view实现一个rotate_view序列视图，用于提供对一个原始序列的变换过和被包装过的视图：

```
typedef mpl::vector_c<int,5,6,7,8,9,0,1,2,3,4> v;
typedef rotate_view<
    v
  , mpl::advance_c<mpl::begin<v>::type,5>::type
> view;
BOOST_STATIC_ASSERT(( mpl::equal<
    view
  , mpl::range_c<int,0,10>
>::value ));
```

7-4. 设计和实现一个迭代器适配器，它通过提呈所遍历（以一个非负整数索引的序列所决定的顺序）的元素来适配任何随机访问迭代器。将permutation_iterator实现为一个前向迭代器。

7-5. 修改练习7-4中的permutation_iterator，使其遍历范畴由索引序列的范畴所决定。

7-6. 使用你的permutation_iterator来实现一个permutation_ view，使

```
permutation_view<
    mpl::list_c<int,2,1,3,0,2>      // 索引
```

```
  , mpl::vector_c<int,11,22,33,44> // 元素
>
```

产生序列 [33,22,44,11,33]

7-7. 设计并实现一个反向迭代适配器，它具有类似于std::reverse_iterator的语义，其范畴和底层的迭代器范畴相同。使用该迭代器实现一个reverse_view模板。

7-8. 实现一个crossproduct_view模板，它通过“以正确的叉积（cross product）顺序来提呈所有可能的元素对”来适配两个原始序列。

第8章 诊 断

因为C++元程序执行于编译期，所以给调试（debugging）工作带来了特别的挑战。没有调试器允许我们单步跟踪元程序的执行，设置断点，检视数据等等。这类调试工作需要对编译器内部状态的互动式探查。我们能做的全部事情，就是等待编译过程失败，然后破译编译器倾泻到屏幕上的错误信息。C++ 模板的诊断（diagnostics）是一种常见的让人感到挫折的源泉，因为它们通常与导致错误的原因没有明显的关系，并且呈现了大量的无用信息。在本章中，我们将讨论如何理解元编程程序员通常遭遇的错误种类，甚至如何使得这些诊断屈服于我们的“邪恶”的目的。

C++标准将错误报告的具体实现方式完全留给编译器实现者，因此我们将讨论几款不同的编译器的行为，通常是以批评的措辞。因为你的编译器的错误消息是你能得到的全部帮助，所以对工具的选择会对你调试元程序的能力产生巨大的影响。如果你在构建程序库，当出现错误时，你的客户对工具的选择将会影响他们对代码的理解，也会影响到你花在回答问题上的时间。因此，即使当我们在讨论你一般不使用的编译器时，我们也建议你聚精会神，因为你也许会发现你希望将它加入自己的工具箱，或者，希望为可能使用该款编译器的客户提供特别的支持。同样，如果我们看上去是在抨击你喜欢的工具，希望你不要感觉自己受到了冒犯。

8.1 调试错误

这一节的标题实际上取自另一本书[VJ02]，但用在这本书里也极为切题。实际上，模板错误报告通常如此像《战争与和平》，很多程序员忽略它们并且求助于随机的代码并改来改去，希望能碰巧改正确。在这一节中，我们将为你提供工具来剔除那些冗长的诊断信息，并且帮助你找到解决问题的正确途径。

> **注意**
>
> 我们将考察一些错误消息例子，如果我们未作任何修改直接呈现的话，其中不少会因为太宽而不能适应页面。为了便于弄明白这些消息，我们在页面右边对其进行换行，并根据需要，可能在后面加上一个空行，将它与后面的行分隔开。

8.1.1 实例化回溯

让我们以一个简单的（错误的）例子开始。下面的代码定义了一个极其简化的编译期“链接表（linked list）”类型结构，以及一个用于计算一个列表中所有元素的总大小的元函数：

```
struct nil {};                          // 用于表示列表的结束

template <class H, class T = nil>   // 一个列表节点，例如：
```

```
struct node                                  // node<X,node<Y,node<Z> > >
{
    typedef H head; typedef T tail;
};

template <class S>
struct total_size
{
    typedef typename total_size<             // S::tail的总共大小
        typename S::tail
    >::type tail_size;                       // 第17行

    typedef boost::mpl::int_<                // 加上S::head的大小
        sizeof(S::head)
        + tail_size::value                   // 第22行
    > type;
};
```

上面的臭虫在于遗漏了用于终结total_size递归所需的特化版本。如果我们试图以如下方式使用它：

```
typedef total_size<
    node<long, node<int, node<char> > >
>::type x;                                   // 第27行
```

我们就会得到如下所示的错误消息（由GNU C++编译器（GCC）3.2产生）：

```
foo.cpp: In instantiation of 'total_size<nil>':
foo.cpp:17:   instantiated from 'total_size<node<char, nil> >'
foo.cpp:17:   instantiated from 'total_size<node<int,
node<char, nil > > >'
foo.cpp:17:   instantiated from 'total_size<node<long int,
node<int, node<char, nil> > > >'
foo.cpp:27:   instantiated from here
foo.cpp:17: no type named 'tail' in 'struct nil'
还有……
```

对冗长的模板错误消息的适应过程的第一步是，你要认识到编译器实际上是通过倾泻所有信息来帮你忙。你现在看到的称为实例化回溯（instantiation backtrace），大致对应于运行期调用堆栈回溯（call stack backtrace）。错误消息的第一行展示了错误发生处的元函数调用，接下来的每一行展示了前一行调用中的元函数调用。最后，编译器向我们展示了导致错误的底层原因：我们把nil哨位（sentinel）当成了一个node<...>，试图访问其::tail成员，其实nil并没有这一个成员。

在这个例子中，只要阅读最后一行就很容易理解错误信息，但与运行期编程一样，外层调用中的一个错误经常会导致内部很多层发生问题。让整个实例化回溯处于我们的管理之下，有助于我们分析和查明问题的根源。

当然了，此例的展示效果并不完美。编译器通常尝试“恢复”此类错误，并且报告更多的

问题，但要这么做它必须对你的真正意图做些假设。除非错误像遗漏了一个分号那样简单，否则这些假设往往是错误的，从而导致其余错误信息毫无意义：

```
……接上
foo.cpp:22: no type named 'tail' in 'struct nil'
foo.cpp:22: 'head' is not a member of type 'nil'
foo.cpp: In instantiation of 'total_size<node<char, nil> >':
foo.cpp:17:   instantiated from 'total_size<node<int, node<char,
nil> > >'

foo.cpp:17:   instantiated from 'total_size<node<long int, node<
int, node<char, nil> > > >'

foo.cpp:27:   instantiated from here
foo.cpp:17: no type named 'type' in 'struct total_size<nil>'
foo.cpp:22: no type named 'type' in 'struct total_size<nil>'
...这儿省略了多行...
foo.cpp:27: syntax error before ';'token
```

通常而言，最好简单地忽略由编译任何源文件而产生的第一个错误之后的所有错误。

8.1.2 错误消息格式化怪癖

尽管每一款编译器都有不同，然而还是有一些你需要知道的消息格式化方面的共通主题。在这一节中，我们将考察现代编译器的一些高级错误报告特性。

一个更现实的错误

大多数诊断格式变种已经被大块头的类型（当程序员使用STL时，在错误消息中不得不面对这样的类型）所驱动。为了获得一个总的看法，我们将检视由三个不同的编译器针对以下病态的程序所产生的诊断消息：

```
# include <map>
# include <list>
# include <iterator>
# include <string>
# include <algorithm>

using namespace std;
void copy_list_map(list<string> & l, map<string, string>& m)
{
    std::copy(l.begin(), l.end(), std::back_inserter(m));
}
```

尽管这段代码非常简单，但一些编译器会报以让人畏惧的出错消息。如果你和我们一样，你会发现当你面对无用的反馈信息时需要努力保持警醒。如果是这样，我们鼓励你再抓起一杯咖啡并且坚持到底：本节的要点是足够熟悉常见的诊断行为，从而使你可以迅速看透混乱，并在任何错误消息中发现重要的信息。在考察一些例子之后，我们可以保证你会发现事情变得容易了。

接下来，让我们把代码扔到Microsoft Visual C++ (VC++) 6中，看看会发生什么。

```
C:\PROGRA~1\MICROS~4\VC98\INCLUDE\xutility(19) : error C2679:
binary '=' : no operator defined which takes a right-hand operand
of type 'class std::basic_string<char,struct std::char_traits<
char>,class std::allocator<char> >' (or there is no acceptable
conversion)

        foo.cpp(9) : see reference to function template
        instantiation 'class std::back_insert_iterator<class std::
        map<class std::basic_string<char, struct std::char_traits<
        char>,class std::allocator<char> >,class std::basic_string<
        char,struct std::char_traits<char>,class std::allocator<char
        > >,struct std::less<class std::basic_string<char,struct std
        ::char_traits<char>,class std::allocator<char> > >,class std
        ::allocator<class std::basic_string<char,struct std::
        char_traits<char>,class std::allocator<char> > > > > __cdecl
        std::copy(class std::list<class std::basic_string<char,
        struct std::char_traits<char>,class std::allocator<char> >,
        class std::allocator<class std::basic_string<char,struct std
        ::char_traits<char>,class std::allocator<char> > > >::
        iterator,class std::list<class std::basic_string<char,struct
        std::char_traits<char>,class std::allocator<char> >,class
        std::allocator<class std::basic_string<char,struct std::
        char_traits<char>,class std::allocator<char> > > >::iterator
        ,class std::back_insert_iterator<class std::map<class std::
        basic_string<char,struct std::char_traits<char>,class std::
        allocator<char> >,class std::basic_string<char,struct std::
        char_traits<char>,class std::allocator<char> >,struct std::
        less<class std::basic_string<char,struct std::char_traits<
        char>,class std::allocator<char> > >,class std::allocator<
        class std::basic_string<char,struct std::char_traits<char>,
        class std::allocator<char> > > >)' being compiled
    还有很多错误消息……
```

很显然需要对这种状况采取一些措施。我们只是展示了错误消息的头两行（真长！），但尽管如此它们已经几乎不可读了。为了对其进行处理，我们可以将消息复制到一个编辑器中，并采用缩进和换行来安排代码，但它仍然相当难以管理：因为即使没有进行实际的格式化处理，错误消息也将近填充满满一页。

typedef替代

如果近距离地观察，你就会发现那个长类型：

```
class std::basic_string<char, struct std::char_traits<char>,
                        class std::allocator<char> >
```

仅在这头两行错误消息中就被重复了12次。事实证明，std::string恰好是该类型的一个

typedef（别名），因此我们可以借助所使用的编辑器的查找和替换功能迅速地简化这个消息：

```
C:\PROGRA~1\MICROS~4\VC98\INCLUDE\xutility(19) : error C2679:
binary '=' : no operator defined which takes a right-hand operand
of type 'std::string' (or there is no acceptable conversion)

    foo.cpp(9) : see reference to function template instantiation
    'class std::back_insert_iterator<class std::map<std::string,
    std::string,struct std::less<std::string>,class
    std::allocator<std::string> > > __cdecl std::copy(class
    std::list<std::string,class std::allocator<std::string>
    >::iterator,class std::list<std::string,class std::allocator<
    std::string> >::iterator,class std::back_insert_iterator<
    class std::map<std::string,std::string,struct std::
    less<std::string>,class std::allocator<std::string> > >)'
    being compiled
```

这是一个很大的改进。一旦完成此项改动，手工插入换行符和缩进以便我们能够分析消息的工程就更好办了。字符串是一个常见的类型，编译器的作者完全可以实现字符串替代，当然了，std::string并不是世界上惟一的一个typedef。GCC的近期版本通过为我们记忆所有名字空间范围的typedefs（namespace-scope typedefs），使它们能简化诊断，从而将这种转化一般化。例如，对于我们的测试程序，GCC 3.2.2是这么说的：

```
……接下来的消息
/usr/include/c++/3.2/bits/stl_algobase.h:228: no match for '
   std::back_insert_iterator<std::map<std::string, std::string,
   std::less<std::string>, std::allocator<std::pair<const
   std::string, std::string> > > >& = std::basic_string<char,
   std::char_traits<char>, std::allocator<char> >&' operator
后续的消息……
```

注意有一点很有趣：GCC没有对赋值运算符右侧的东西进行替换。然而，正如我们马上要看到的那样，在typedef替换方面保守一些可能并不是什么坏事。

“with”子句

回头看看修订后的VC++ 6错误消息，（即进行std::string替代后的消息）。如果你瞥一眼，第二行有一个对std::copy的调用。为了使该事实更加明显，很多编译器将实际的模板实参从模板特化的名字分离开。例如，上面引述的错误消息之前的最后一行GCC实例化回溯信息如下：

```
/usr/include/c++/3.2/bits/stl_algobase.h:349: instantiated from
'_OutputIter std::copy(_InputIter, _InputIter, _OutputIter)
[with _InputIter = std::_List_iterator<std::string, std::string&,
std::string*>, _OutputIter = std::back_insert_iterator<std::map<
std::string, std::string, std::less<std::string>, std::allocator<
std::pair<const std::string, std::string> > > >]'
后续的消息...
```

保留的标识符

C++标准保留以一个下划线和一个大写字母打头的标识符（例如_InputIter），也保留以双下划线打头和结尾的标识符（例如__function__），仅供语言实现使用。因为我们呈现的是涉及C++标准程序库的诊断，所以在本章中你将会看到相当多的保留标识符。然而，不要误以为这是我们应该仿效的一种约定，标准程序库使用这些名字以便和我们有所区分，如果我们也这么做的话，程序行为就是未定义的（undefined）了。

“with”子句使我们很容易看到std::copy被牵涉进来了。而且，能够看到正式的模板参数名字，可带给我们一个有意义的提醒，即关于copy算法施加于其参数之上的concept条件要求。最后，因为同样的类型被用于两个不同的形式参数（formal parameters），却只在“with”子句中拼写了一次，所以错误消息的整体尺寸减小了。很多构建于Edison Design Group（EDG）前端（front-end）之上的编译器已经采用类似的做法很多年。

微软采用类似的方式对VC++编译器第7版报告的错误消息进行了改善，并且加入了一些有意义的换行符：

```
foo.cpp(10) : see reference to function template instantiation
'_OutIt std::copy(_InIt,std::list<_Ty,_Ax>::iterator,
std::back_insert_iterator<_Container>)' being compiled

with
[
    _OutIt=std::back_insert_iterator<std::map<std::string,std
    ::string,std::less<std::string>,std::allocator<std::pair<
    const std::string,std::string>>>>,

    _InIt=std::list<std::string,std::allocator<std::string>>::
    iterator,

    _Ty=std::string,
    _Ax=std::allocator<std::string>,
    _Container=std::map<std::string,std::string,std::less<std
    ::string>,std::allocator<std::pair<const std::string,std::
    string>>>

]
```

不幸的是，我们也开始注意到VC++ 7.0中一些没有帮助作用的行为。在函数的签名中不是列出_InIt和_OutIt两次，第二、第三个参数类型是被完整地写出的，并且在“with”子句中也被重复写出。这儿有一些连锁反应，因为作为结果，_Ty和_Ax（如果_InIt和_OutIt被一致地用于签名中的话，它们永远都不会出现）也出现在了“with”子句中。

默认模板实参

在版本7.1中，微软修正了这个怪毛病，还给我们看到std::copy的前两个参数具有相同类型

的能力。然而，现在它们展示std::copy特化的完整名字，因此，我们仍然不得不面对一些无用的东西：

```
foo.cpp(10) : see reference to function template instantiation '
_OutIt std::copy<std::list<_Ty>::iterator,std::
back_insert_iterator<_Container> >(_InIt,_InIt,_OutIt)' being
compiled
with
[
    _OutIt=std::back_insert_iterator<std::map<std::string,std::
    string>>,

    _Ty=std::string,
    _Container=std::map<std::string,std::string>,
    _InIt=std::list<std::string>::iterator
]
更多的消息……
```

如果上面突出显示的内容被std::copy<_InIt,_OutIt>代替，那么_Ty也可能会从“with”子句中消除掉。

好消息是已经做出了一个重要的简化：std::list的默认分配器实参，以及std::map的默认分配器和比较参数已经省去了。到写这本书时为止，VC++ 7.1是我们知道的惟一的一款消除了默认模板实参的编译器。

深层typedef替代

很多现代编译器设法记住每一个类型是否并如何通过typedef（不仅仅是名字空间范围的typedef）进行计算的，因此在诊断消息中类型可用这种方式来表示。我们称这个策略为深度typedef替代（deep typedef substitution），因为实例化栈（instantiation stack）[⊖]的深层的typedefs出现在了诊断消息里。例如，下例：

```
# include <map>
# include <vector>
# include <algorithm>
int  main()
{
    std::map<int,int> a;
    std::vector<int> v(20);
    std::copy(a.begin(), a.end(), v.begin());
    return 0;
}
```

在Intel C++ 8.0中产生如下输出：

```
C:\Program Files\Microsoft Visual Studio .NET 2003\VC7\INCLUDE\
xutility(1022): error: no suitable conversion function from "std::
```

⊖ 即实例化回溯信息。

```
allocator<std::pair<const int, int>>::value_type" to "std::
allocator <std::_Tree<std::_Tmap_traits<int, int, std::less<int>,
std::allocator<std::pair<const int, int>>, false>>::key_type=
{std::_Tmap_traits<int, int, std::less<int>, std::allocator<std::
pair<const int, int>>, false>::key_type={int}}>::value_type=
{std::_Allocator_base<std::_Tree<std::_Tmap_traits<int, int,

        std::less<int>, std::allocator<std::pair<const int, int>>,
        false>>::key_type={std::_Tmap_traits<int, int, std::less<
        int>, std::allocator<std::pair<const int, int>>, false>::
        key_type={int}}>::value_type={std::_Tree<std::_Tmap_traits
        <int, int, std::less<int>, std::allocator<std::pair<const
        int, int>>, false>>::key_type={std::_Tmap_traits<int, int,
        std::less<int>, std::allocator<std::pair<const int, int>>,
        false>::key_type={int}}}}" exists

      *_Dest = *_First;
               ^

...
```

这里我们需要知道些什么呢？问题在于你不能将一个pair<int, int>（map的元素）赋值给一个int（vector的元素）。该信息实际上藏身于上面的消息中了，但它的呈现方式很糟糕。将该消息按字面上翻译为某些更像英语的内容就是：

```
No conversion exists from the value_type of an
   allocator<pair<int,int> >
to the value_type of an
   _allocator<

         _Tree<...>::key_type... (which is some
   _Tmap_traits<...>::key_type, which is int)
   >.
Oh, that second value_type is the value_type of an
   _Allocator_base<

         _Tree<...>::key_type... (which is some
   _Tmap_traits<...>::key_type, which is int)
   >,
which is also the key_type of a _tree<...>, which is int.
```

如果仅告诉我们无法将pair<int, int>赋值给int将会有更大的帮助。相反，展示给我们的是大量的关于这些类型在标准库实现内部来龙去脉的信息。

对于同样的错误，VC++ 7.1中报告如下：

```
C:\Program Files \Microsoft Visual Studio .NET 2003\Vc7 \include\
xutility(1022) : error C2440: '=' : cannot convert from 'std::
allocator<_Ty>::value_type' to 'std::allocator<_Ty>::value_type'
```

```
        with
        [
            _Ty=std::pair<const int,int>
        ]
        and
        [
            _Ty=std::_Tree<std::_Tmap_traits<int,int,std::less<
            int>,std::allocator<std::pair<const int,int>>,
            false>>::key_type

        ]
...
```

这个消息要简短得多，但它也没有太多的安慰作用：乍看上去它宣称allocator<_Ty>::value_type不能向自身转换！实际上，两次提到的_Ty分别指的是两个连续方括号中的子句所定义的类型（通过“with”和“and”引入）。即使我们挑出了它，这个诊断消息也具有同样的问题：在std::allocator中，涉及的类型以typedefs进行表达。容易记住std::allocator的value_type和其模板实参是相同的是件好事，否则我们就没有关于这里涉及的类型的线索了。

由于allocator<_Ty>::value_type本质上是一个元函数调用，这类深层typedef替代确实会对我们调试元程序的能力造成负面影响。看看如下简单的例子：

```
# include <boost/mpl/transform.hpp>
# include <boost/mpl/vector/vector10.hpp>

namespace mpl = boost::mpl;
using namespace mpl::placeholders;
template <class T>
struct returning_ptr
{
    typedef T* type();
};

typedef mpl::transform<
      mpl::vector5<int&,char,long[5],bool,double>
    , returning_ptr<_1>
 >::type functions;
```

这段代码的本意是构建一个函数类型序列，它返回指向“一个输入序列中包含的类型”的指针，但是作者忘记考虑了一个事实：在C++中，构建一个指向引用类型（此例为int&）的指针是不合法的。Intel C++ 7.1报告如下：

```
foo.cpp(19): error: pointer to reference is not allowed
      typedef T* type();
               ^

        detected during:
          instantiation of class "returning_ptr<T> [with T=boost::
```

```
mpl::bind1<boost::mpl::quote1<returning_ptr>, boost::mpl
::lambda_impl<boost::mpl::_1, boost::mpl::false_>::type>
::apply<boost::mpl::vector_iterator<boost::mpl::vector5<
int &, char, long [5], bool={bool}, double>::type, boost
::mpl::integral_c<long, 0L>>::type, boost::mpl::void_,
boost::mpl::void_, boost::mpl::void_, boost::mpl::void_>
::t1]" at line 23 of "c:/boost/boost/mpl/aux_/has_type.
hpp"
```

导致这个错误的一般原因是极其清晰的，但是被冒犯的类型却远非如此。我们真的希望知道T是什么，但它却根据一个嵌套的typedef: mpl::bind1<...>::t1进行表达。除非我们准备好慢慢消化那一行中提及的mpl::bind1的定义以及其他MPL 模板，否则我们会困惑不解。微软VC++ 7.1同样无济于事[⊖]：

```
foo.cpp(9) : error C2528: 'type' : pointer to reference is illegal
        c:\boost\boost\mpl\aux_\has_type.hpp(23) : see reference
        to class template instantiation 'returning_ptr<T>' being
        compiled

        with
        [
          T=boost::mpl::bind1<boost::mpl::quote1<returning_ptr>,
          boost::mpl::lambda_impl<boost::mpl::_1>::type>::apply<
          boost::mpl::vector_iterator<boost::mpl::vector5<int &,
          char,long [5],bool,double>::type,boost::mpl::integral_c<
          long,0>>::type >::t1

        ]
```

GCC 3.2仅仅执行“浅（shallow）”的typedef替换，报告如下：

```
foo.cpp: In instantiation of 'returning_ptr<int&>':
此处省略多行消息……
foo.cpp:19: forming pointer to reference type 'int&'
```

这个消息要好懂得多。我们待一会儿就会解释忽略的行。

8.2 使用工具进行诊断分析

尽管编译器厂商的努力有时事与愿违，然而他们显然在解决难以理解的模板错误消息的问题上不怕麻烦，坚持不懈。当一个臭虫从一个嵌套的模板实例化的深层咬你一口时，即使最好的错误消息格式仍然有很大的改进余地。但幸运的是，如果你遵从如下三个建议的话，软件工具可以带来极大的帮助。

⊖ 幸运的是，微软的编译器工程师已经听到了我们的抱怨，下一代编译器的一个评估版只向诊断消息中注入定义于命名空间（namespace）范围的typedefs。如果运气好，这个改变将会幸存下来，从而体现于最终的发布版之中。

8.2.1 听取他者的意见

我们的第一个建议是，出于调试目的，你手头要拥有几款不同的编译器。如果一个编译器发出的错误消息让人费解，那么另一个也许会好一些。当一些事情出错时，编译器为了报告错误会猜测你的意图，而能够拥有好几种不同的猜测是有好处的。而且，很多编译器在错误报告方面具有先天性的不足。例如，尽管Metrowerks CodeWarrior Pro 9在其他方面是一款优秀的编译器，并且在我们的时间测试中也是最快的之一，然而，它却经常不能为其实例化回溯的每个帧（frame）输出文件名字和行号，这会导致难以找到出问题的源代码。如果需要追踪错误的来源，你也许希望试试别的编译器。

> **提示**
>
> 如果你没有投资更多工具的预算，我们建议你找一个运行于你所使用的平台上的GCC的最新版。所有版本的GCC都可以免费获得。Windows用户可以获得MinGW（http://www.mingw.org）或Cygwin（http://www.cygwin.com）变种。如果你无法承受在你的机器上安装另一种编译器，Comeau Computing允许你试验他们编译器的在线版本（http://www.comeaucomputing.com/tryitout）。因为Comeau C++基于高度遵从标准的EDG前端，所以提供了一个优秀的方式，便于你快速弄清代码是否遵从C++标准。

8.2.2 使用导航助手

为了遍历实例化堆栈回溯信息，拥有一个可以帮助你看到与错误消息相关联的源代码行的环境是至关紧要的。如果你通常从命令窗口（command shell）来编译程序，也许希望从某种集成开发环境（integrated development environment，IDE）内部发出那些命令，以避免不得不手工在一个编辑器中打开文件并查找行号。很多IDE允许插入（plugged in）多种工具（编译器），但对元程序的调试来说，重要的是IDE可以方便地在不同格式的编译器诊断消息之间切换。例如，Emacs使用一套扩充的正则表达式（regular expressions）从错误消息中抽取文件名和行号，因此它能被协调和许多编译器协作。

8.2.3 清理场面

最后，我们建议采用一个后处理过滤器（post-processing filter），例如TextFilt（http://textfilt.sourceforge.net）或STLFilt（http://www.bdsoft.com/tools/stlfilt.html）。这些过滤器最初是用来帮助程序员搞清楚STL错误消息中的类型。它们最基本的特性包括自动从已知的模板特化中删除默认实参（default arguments），并且为std::string和std::wstring进行typedef替换。例如，TextFilt将以下乱七八糟的信息：

```
example.cc:21: conversion from 'double' to non-scalar type
'map<vector<basic_string<char, string_char_traits<char>,
__default_alloc_template<true, 0> >,
allocator<basic_string<char, string_char_traits<char>,
__default_alloc_template<true, 0> > > >, set<basic_string<char,
string_char_traits<char>, _ _default_alloc_template<true, 0> >,
```

```
less<basic_string<char, string_char_traits<char>,
__default_alloc_template<true, 0> > >,
allocator<basic_string<char, string_char_traits<char>,
__default_alloc_template<true, 0> > > >,
less<vector<basic_string<char, string_char_traits<char>,
__default_alloc_template<true, 0> >,
allocator<basic_string<char, string_char_traits<char>,
__default_alloc_template<true, 0> > > > >,
allocator<set<basic_string<char, string_char_traits<char>,
__default_alloc_template<true, 0> >, less<basic_string<char,
string_char_traits<char>, _ _default_alloc_template<true, 0> > >,
allocator<basic_string<char, string_char_traits<char>,
__default_alloc_template<true, 0> > > > > >' requested
```

转换为可读性好得多的信息：

```
example.cc:21: conversion from 'double' to non-scalar type
'map<vector<string>,set<string>>' requested
```

TextFilt很有趣，因为它容易被定制。你可以通过编写“规则集（rulesets）”加入对你自己的类型的特殊处理，“规则集”是一套简单的基于正则表达式的（regular expression-based）转换。STLFilt不是那么容易定制（除非你对“折磨”Perl乐在其中），但它包含一些命令行选项，利用它们，你可以调节想看到的信息量。我们发现这两款工具对于模板元编程都是不可或缺的。

1. 对GCC错误消息进行重排。尽管GCC是目前为止我们进行元程序调试的首选编译器，但它绝非完美。它的最大的问题是它在整个实例化回溯信息之后打印真正的出错原因。结果，你通常在看到问题变得明朗之前不得不长途跋涉，穿过整个回溯信息，真正的错误被远远地从最近的实例化信息帧那儿分隔开。这也是为何本章中的GCC错误消息常常被展示以“此处省略多行（many lines omitted...）”的原因。STLFilt具有两个用于GCC消息重排（message reordering）的选项：

- -hdr:LD1:，将实际的错误消息移到实例化回溯信息的顶部。
- -hdr:LD2:，就像-hdr:LD1一样，不过在错误消息之后添加了回溯信息的最后一行代码（即启动实例化的非模板代码）的副本。

2. 表达式包装和缩进。不管做多少工作从错误消息中过滤掉不相关的信息，我们也不能回避某些C++类型和表达式具有固有的复杂性的事实。例如，如果没有默认模板实参和typedefs协作，管控先前的例子将需要我们去解析其嵌套的结构。STLFilt包含一个-meta选项，它根据这本书的约定对错误消息进行格式化。即使禁用默认模板实参删除和typedef替代功能，STLFilt仍然可以帮助我们弄明白消息中都有些啥：

```
example.cc:21: conversion from 'double' to non-scalar type
   'map<
    vector<
        basic_string<
            char, string_char_traits<char>
         , __default_alloc_template<true, 0>
```

```
    >, allocator<
        basic_string<
            char, string_char_traits<char>
          , __default_alloc_template<true, 0>
        >
    >
>, set<
    basic_string<
        char, string_char_traits<char>
      , __default_alloc_template<true, 0>
    >, less<
        basic_string<
            char, string_char_traits<char>
          , __default_alloc_template<true, 0>
        >
    >, allocator<
        basic_string<
            char, string_char_traits<char>
          ' __default_alloc_template<true, 0>
        >
     >
 >, less<
     ……此处省略12行
    >, allocator<
        ……此处省略16行
    >
>' requested
```

尽管该消息仍然巨大，但它的可读性变得好多了：通过扫描最前面的几栏，我们就可以迅速地猜测到这个长长的类型是一个从vector<string>到set<string>的map。

如果施加太多的过滤，任何工具都可能使诊断信息变得晦涩难懂，STLFilt也不例外，因此我们鼓励你检查http://www.bdsoft.com/tools/stlfilt-opts.html处的命令行选项，并且仔细地选择使用。幸运的是，由于这些都是外部工具，你总是可以退回原地，对原生诊断消息进行直接地检视。

8.3 有目的的诊断消息生成

为什么会有人故意要生成诊断信息？在花了本章的大量篇幅从模板错误消息的沼泽中采撷有用的信息后，我们都希望它们滚得远远的。然而，编译器诊断自有它们的地位，即使我们的模板用于装备不良（乃至难于破译出错信息）的用户手中。最终，这都归结于一个简单的思想：

> **指导方针**
>
> 在第一时间报告错误。

即使最丑陋的编译期错误也比运行期的悄无生息的错误行为、崩溃或断言好。此外，如果无论如何都会发生一个编译期诊断的话，那么产生错误消息总是越早越好。模板错误消息常常

不能为实际编程问题的性质和位置提供线索的原因在于，它们出现得太晚了，当实例化已经深入到一个程序库的内部实现细节时才出现。因为编译器自身没有关于程序库的领域的知识，所以它无法在程序库接口边界侦测用法错误并根据程序库抽象来报告它们。例如，我们可能试图编译：

```
#include <algorithm>
#include <list>

int main()
{
    std::list<int> x;
    std::sort(x.begin(), x.end());
}
```

理想情况下，在这个例子中，我们希望编译器在实际编程错误的那一点报告问题，并告诉我们一些有关涉及的抽象（此例中即为迭代器）的东西：

```
main.cpp(7) : std::sort requires random access iterators, but
std::list<int>::iterator is only a bidirectional iterator
```

然而VC++ 7.1却报告：

```
C:\Program Files \Microsoft Visual Studio .NET 2003\Vc7 \include\
algorithm(1795) : error C2784:
'reverse_iterator<_RanIt>::difference_type std::operator -(const
std::reverse_iterator<_RanIt> &,const
std::reverse_iterator<_RanIt>  &)' : could not deduce template
argument for 'const std::reverse_iterator<_RanIt> &' from
'std::list<_Ty>::iterator'
        with
        [
            _Ty=int
        ]
更多的信息……
```

注意，错误报告位于标准程序库<algorithm>头文件里的某个operator-实现品内部，而不是错误实际发生的main()中。这个问题的原因被std::reverse_iterator的出现弄得晦涩不清了，因为它与我们编写的代码并没有明显的关系。甚至对operator-的使用（它暗示需要使用随机访问迭代器）和程序员正尝试做的事情也没有什么直接的关系。如果在std::list<int>::iterator和std::sort对随机访问迭代器的要求之间的不匹配被较早地侦测出来了（理想的情况是在std::sort被调用点），编译器就有可能直接报告出问题。

理解这一点很重要：上面的劣质错误消息的过失责任并不在于编译器。实际上，这要归结于C++语言的限制：尽管普通函数的签名清晰地声明了其参数的类型要求，然而对于泛型函数则不能这么说[⊖]。另一方面，程序库作者对限制该损害所能做的事情很少。在这一节中，我们将讨

⊖ C++委员会的一些成员目前正努力工作力图克服该限制（通过将concepts用作C++类型系统的一等公民）。在此期间，程序库解决方案[SL00]将用来满足我们的需求。

论在自己的程序库中使用的一些技术，用于较早地生成诊断消息，并对其内容施加更多的控制。

8.3.1 静态断言

我们已经看过当代码被侦测到误用时一种生成错误的方式

```
BOOST_STATIC_ASSERT(integral-constant-expression);
```

如果该表达式为false（或0），就会产生一个编译器错误。断言的最佳用途是作为一种“完整性检查（sanity check）”，以确保代码编写所依赖的假定真正成立。让我们使用一个经典的factorial 元函数作为例子：

```
#include <boost/mpl/int.hpp>
#include <boost/mpl/multiplies.hpp>
#include <boost/mpl/less_equal.hpp>
#include <boost/mpl/eval_if.hpp>
#include <boost/mpl/prior.hpp>
#include <boost/static_assert.hpp>

namespace mpl = boost::mpl;

template <class N>
struct factorial
    : mpl::eval_if<
        mpl::less_equal<N,mpl::int_<0> >  // 检查N <= 0
      , mpl::int_<1>                      // 0! == 1
      , mpl::multiplies<                  // N! == N * (N-1)!
            N
          , factorial<typename mpl::prior<N>::type>
        >
    >::type
{
   BOOST_STATIC_ASSERT(N::value >= 0); // 确保N非负
};
```

只有当N是非负整数时计算N!才有意义，factorial的编写基于这样的一个假定：其实参满足该约束。断言用于检查假定，如果我们违反它：

```
int const fact = factorial<mpl::int_<-6> >::value;
```

在Intel C++ 8.1中会得到如下诊断消息：

```
foo.cpp(22): error: incomplete type is not allowed
      BOOST_STATIC_ASSERT(N::value >= 0);
      ^

          detected during instantiation of class "factorial<N>
      [with N=mpl_::int_<-6>]" at line 25
```

注意，当违反条件时，我们得到一个错误消息，它指向包含该断言的那行源代码。

程序库基于你所使用的编译器选择BOOST_STATIC_ASSERT的实现，以便确保该宏（macro）能被可靠地用在类、函数或名字空间作用域中，并且诊断信息总是指向断言被触发的那一行代码。当该断言在Intel C++中失败时，它通过误用一个不完全类型来产生诊断消息（因而报出消息“incomplete type is not allowed”），当然，你也许发现在其他编译器生成的错误有别于此。

8.3.2 MPL静态断言

不可能提供比上面的诊断内容信息含量更大的诊断了：不但显示了源代码行，还看到了有问题的条件以及传给factorial的参数。然而，通常而言，你不能依赖从BOOST_STATIC_ASSERT得到这样有帮助作用的结果。在这个例子中，与其说是设计如此，不如说是幸运的偶然，使我们得到了这些有帮助的诊断。

1. 如果断言中正被测试的值（−6）没有呈现在外覆（enclosing）模板的类型中，它就不会被显示出来。

2. 这款编译器在一个错误点只显示一个源代码行。如果宏（macro）调用跨越多行代码，那么被测试的条件至少会被部分地隐藏起来。

3. 很多编译器在错误消息中不显示任何源代码行。例如GCC 3.3.1报告如下：

```
foo.cpp: In instantiation of 'factorial<mpl_::int_<-6> >':
foo.cpp:25:   instantiated from here
foo.cpp:22: error: invalid application of 'sizeof' to an
incomplete type
```

在这里，失败条件丢失了。

MPL供应了一套静态断言宏（static assertion macro），它们真正用来生成有用的错误消息。在这一节中，我们将其用在factorial元函数中挨个探索它们。

基本版本

这些断言中最简单的一个用法如下：

```
BOOST_MPL_ASSERT((bool-valued-nullary-metafunction))
```

注意，双小括号是必不可少的，即使在条件中没有逗号出现。

为了将该宏应用于factorial例子，以下是我们可能作出的修改：

```
...
#include <boost/mpl/greater_equal.hpp>
#include <boost/mpl/assert.hpp>

template <class N>
struct factorial
  ...
{
    BOOST_MPL_ASSERT((mpl::greater_equal<N,mpl::int_<0> >));
};
```

BOOST_MPL_ASSERT的优势在于，它将其实参元函数的名字放入诊断消息中。GCC现在报告：

```
foo.cpp: In instantiation of 'factorial<mpl_::int_<-6> >':
foo.cpp:26   :instantiated from here
foo.cpp:23: error: conversion from '
   mpl_::failed**********boost::mpl::greater_equal<mpl_::int_<-6>,
   mpl_::int_<0> >::***********' to non-scalar type '
   mpl_::assert<false>' requested
foo.cpp:23: error: enumerator value for '
   mpl_assertion_in_line_23' not integer constant
```

注意，被违反的条件现在被突出地显示出来，以连续的星号包围起来，所有支持MPL的编译器都支持这个功能。

一个更合适的断言

说句实话，上面的诊断仍然包含很多我们不关心的字符，但那要归因于使用模板来表达失败的条件（−6 >= 0）而不是什么别的原因。BOOST_MPL_ASSERT实际上更适合检查其他种类的条件。例如，我们可以尝试强制N服从于整型常量外覆器协议，如下：

```
BOOST_MPL_ASSERT((boost::is_integral<typename N::value_type>));
```

为了触发这个断言，我们可以写：

```
// 试图制作一个“浮点常量外覆器”
struct five : mpl::int_<5> { typedef double value_type; };
int const fact = factorial<five>::value;
```

产生如下的诊断消息，它比我们的非负测试具有更好的信噪比：

```
...
foo.cpp:24: error: conversion from
'mpl_::failed***********boost::is_integral<double>::***********'
to non-scalar type 'mpl_::assert<false>' requested
...
```

否定断言（Negative Assertions）

使用BOOST_STATIC_ASSERT否定一个条件测试是非常简单的，只要在前面加上一个!即可，但使用BOOST_MPL_ASSERT做同样的事情，我们则需要采用mpl:: not_<...>来包装判断式。为了简化否定断言，MPL提供了一个BOOST_MPL_ASSERT_NOT，它代我们做好了包装操作。下面是对先前断言N为非负的改写：

```
BOOST_MPL_ASSERT_NOT((mpl::less<N,mpl::int_<0> >));
```

如你所见，结果错误消息包含有mpl::not_<...>外覆器：

```
foo.cpp:24: error: conversion from 'mpl_::failed
***********boost::mpl::not_<boost::mpl::less<mpl_::int_<-5>,
mpl_::int_<0> > >::***********' to non-scalar type
'mpl_::assert<false>' requested
```

断言数值关系

我们认为BOOST_MPL_ASSERT不是特别适合检查数值条件，因为不但是诊断消息，就是断言本身也往往会招致繁杂的语法负担。不可否认，为了表达x >= y而去编写mpl::greater_equal<x,y>，有拐弯抹角之嫌。对于这类数值比较，MPL提供了一个专门的宏：

```
BOOST_MPL_ASSERT_RELATION(
  integral-constant, comparison-operator, integral-constant);
```

为了将它应用于我们的factorial 元函数，只要写：

```
BOOST_MPL_ASSERT_RELATION(N::value, >=, 0);
```

就可以了。在这种情况下，不同编译器生成的错误消息的内容不太一样。GCC报告：

```
...
foo.cpp:30: error: conversion from
'mpl_::failed************mpl_::assert_relation<greater_equal, -5,
0>::*************' to non-scalar type 'mpl_::assert<false>'
requested
...
```

而Intel则说：

```
foo.cpp(30): error: no instance of function template
"mpl_::assertion_failed" matches the argument list
    argument types are: (mpl_::failed
    ************mpl_::assert_relation<  mpl_::operator>=, -5L, 0L
    >::*************)

    BOOST_MPL_ASSERT_RELATION(N::value, >=, 0);
    ^

        detected during instantiation of class "factorial<N>
    [with N=mpl_::int_<-5>]" at line 33
```

尽管有这些区别，但被违反的关系以及涉及的两个整型常量在两个诊断消息中均清晰可见。

定制的断言消息

我们到目前为止看到的断言宏（assertion macros）对于程序库内部“完整性检查”来说，已经很好了，但它们并非总是为程序库的用户生成形式最适当的信息。factorial 元函数可能无法很好地描绘这一事实，因为触发错误（N < 0）的判断式太简单。计算N!的一个先决条件是N必须是非负的，任何用户都可能意识到对N >= 0条件不成立的抱怨是对该约束的一个直接表达。

然而，并非所有静态断言都有这个属性，通常来说，断言反映了程序库实现的低层细节，而不是用户正在处理的抽象。第3章量纲分析代码中有一个例子，用BOOST_MPL_ASSERT改写如下：

```
template <class OtherDimensions>
quantity(quantity<T,OtherDimensions> const& rhs)
```

```
  : m_value(rhs.value())
{
    BOOST_MPL_ASSERT((mpl::equal<Dimensions,OtherDimensions>));
}
```

如果这个断言失败，我们将在诊断消息中看到，在两个序列（包含有整型常量外覆器）之间不相等。这个信息，连同源代码行，开始提示真正的问题，但它并没有说到点子上。用户需要知道的第一件事是，当这个断言失败时，存在量纲不匹配的情况。其次，知道第一个没有被匹配的基础量纲的身份以及有关指数的值，可能是有帮助作用的。然而，这些信息从实际生成的诊断消息中很难立刻体现出来。

有了对诊断消息的更多控制，我们就可以生成对用户而言更适合的消息。我们将为量纲分析生成错误的具体问题留一个练习，现在让我们回到factorial问题探索一些技术。

定制判断式

为了显示一个定制的消息，我们可以利用这个事实：BOOST_MPL_ASSERT将它的判断式的名字放入诊断输出中。只要编写一个具有适当命名的判断式，我们就可以使编译器说出我们希望它说出的东西——只要我们要说的话可以一个类模板名字的方式进行表达即可。例如：

```
// 特化是计算n>0的无参元函数
template <int n>
struct FACTORIAL_of_NEGATIVE_NUMBER
  : mpl::greater_equal<mpl::int_<n>, mpl::int_<0> >
{};
template <class N>
struct factorial
  : mpl::eval_if<
        mpl::equal_to<N,mpl::int_<0> >
      , mpl::int_<1>
      , mpl::multiplies<
            N
          , factorial<typename mpl::prior<N>::type>
        >
    >
{
    BOOST_MPL_ASSERT((FACTORIAL_of_NEGATIVE_NUMBER<N::value>));
};
```

现在GCC报告：

```
foo.cpp:30: error: conversion from 'mpl_::failed
************FACTORIAL_of_NEGATIVE_NUMBER<-5>::************' to
non-scalar type 'mpl_::assert<false>' requested
```

这种方式存在一个小问题，仅仅出于显示一个错误消息的目的，代码流程就要被打断以便在命名空间（namespace）范围内编写一个判断式。这个策略还具有一个更严重的缺点：代码现在看上去在断言N::value须是一个负数，实际情况其实相反。这不但会使得代码的维护者犯糊涂，

而且还会弄糊涂代码的用户。不要忘记一些编译器（此例中为Intel C++）将会显示包含断言的行号：

```
foo.cpp(30): error: no instance of function template
"mpl_::assertion_failed" matches the argument list

     argument types are: (mpl_::failed
     ************FACTORIAL_of_NEGATIVE_NUMBER<-5>::************)

      BOOST_MPL_ASSERT((FACTORIAL_of_NEGATIVE_NUMBER<N::value>));
      ^
```

如果我们更仔细地选择要显示的消息文本，我们就可以消除这个可能的混淆之源：

```
template <int n>
struct FACTORIAL_requires_NONNEGATIVE_argument
  : mpl::greater_equal<mpl::int_<n>, mpl::int_<0> >
{};
...
    BOOST_MPL_ASSERT((
        FACTORIAL_requires_NONNEGATIVE_argument<N::value>));
```

然而，这类在语言表达上的煞费苦心，可能会变得笨拙不实用，而且也并非总是可行。

内嵌消息生成（Inline Message Generation）

MPL提供了一个用于生成定制消息的宏，由于它不依赖一个单独编写的判断式类（predicate class），因此不要求在精确的措辞上下多少功夫。用法如下：

```
BOOST_MPL_ASSERT_MSG(condition, message, types);
```

其中的condition是一个整型常量表达式，message是一个合法的C++标识符，types是一个合法的函数参数列表。例如，为了将BOOST_MPL_ASSERT_MSG应用于factorial，我们可以这么写：

```
BOOST_MPL_ASSERT_MSG(
    N::value >= 0, FACTORIAL_of_NEGATIVE_NUMBER, (N));
```

在GCC中产生如下消息：

```
foo.cpp:31: error: conversion from 'mpl_::failed
*****************(factorial<mpl_::int_<-5>
>::FACTORIAL_of_NEGATIVE_NUMBER::*****************)
(mpl_::int_<-5>)' to non-scalar type 'mpl_::assert<false>'
requested.
```

在上面的诊断消息中，我们已经突出显示了message和types实参。在这个例子中，types不是特别有意思，因为它只是复述mpl_::int_<-5>，消息中其他地方已经出现了它。我们因此可以将断言中的(N)替换为空的函数参数列表()，从而获得如下消息：

```
foo.cpp:31: error: conversion from 'mpl_::failed
*****************(factorial<mpl_::int_<-5>
```

```
>::FACTORIAL_of_NEGATIVE_NUMBER::*****************)
()' to non-scalar type 'mpl_::assert<false>'
requested.
```

一般而言，即便使用BOOST_MPL_ASSERT_MSG也需要小心，因为types实参被用做函数参数列表，有一些我们希望显示的类型在该上下文中具有特殊的含义。例如，void参数会从大多数诊断中忽略，因为int f(void)和int f()相同。进一步而言，void只能使用一次，int f(void, void)的语法是非法的。此外，数组和函数类型分别被解释为指针和函数指针类型：

```
int f(int x[2], char* (long))
```

和下面的相同

```
int f(int *x, char* (*)(long))
```

为了提早知道types能否被正确地显示，你可以使用如下形式，其中最多可以有四个types：

```
BOOST_MPL_ASSERT_MSG(condition, message, (types<types >));
```

例如，我们可以向factorial中添加如下断言，基于所有整型常量外覆器都是类的事实：

```
BOOST_MPL_ASSERT_MSG(
    boost::is_class<N>::value
  , NOT_an_INTEGRAL_CONSTANT_WRAPPER
  , (types<N>));
```

如果我们试图去实例化factorial<void>，VC++ 7.1报告：

```
foo.cpp(34) : error C2664: 'mpl_::assertion_failed' : cannot
    convert parameter 1 from 'mpl_::failed
    ****************(__thiscall
    factorial<N>::NOT_an_INTEGRAL_CONSTANT_WRAPPER::*
    ***************                    )(mpl_::assert_::types<T1>)
    ' to 'mpl_::assert<false>::type'
    with
    [
        N=void,
        T1=void
    ]
```

由于types可以接收最多四个参数，这里的诊断消息要比在没有消除默认模板实参的编译器中的好。例如，Intel C++ 8.0产生的诊断消息为：

```
foo.cpp(31): error: no instance of function template
"mpl_::assertion_failed" matches the argument list

     argument types are: (mpl_::failed ***************
     (factorial<void>::NOT_an_INTEGRAL_CONSTANT_WRAPPER::
     ****************)(mpl_::assert_::types<void, mpl_::na,
     mpl_::na, mpl_::na>))

      BOOST_MPL_ASSERT_MSG(
      ^
```

```
        detected during instantiation of class "factorial<N>
        [with N=void]" at line 37
```

还有一点值得注意，尽管刚才用在BOOST_MPL_ASSERT中的定制判断式编写在命名空间范围内，然而BOOST_MPL_ASSERT_MSG产生的消息却以断言所产生的作用域（本例中为factorial<void>）的资格限定成员而出现。结果，进行深度typedef替换的编译器多一个机会在诊断消息中插入难以理解的类型膨胀信息。例如，如果我们实例化：

```
mpl::transform<mpl::vector<void>, factorial<mpl::_> >
```

Intel C++ 8.0产生如下诊断：

```
foo.cpp(34): error: no instance of function template
"mpl_::assertion_failed" matches the argument list

    argument types are: (mpl_::failed
    ****************(factorial<boost::mpl::bind1<
    boost::mpl::quote1<factorial, boost::mpl::void_>,
    boost::mpl::lambda<mpl_::_,
    boost::mpl::void_>::result_>::apply<
    boost::mpl::bind1<factorial<mpl_::_>,
    mpl_::_2>::apply<boost::mpl::aux::fold_impl<1,
    boost::mpl::begin<boost::mpl::vector<void, mpl_::na,
    mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na,
    mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na, mpl_::na,
    mpl_::na, mpl_::na, mpl_::na,
    还有4行类似的消息被省略了……

      mpl_::na>::tl>::NOT_an_INTEGRAL_CONSTANT_WRAPPER::
      *****************)(mpl_::assert_::types<boost::mpl::bind1<
      此行其余内容……

      还有5行类似的消息被省略了……
    BOOST_MPL_ASSERT_MSG(
    ^
```

上面省略了9行输出，大大提高了消息的可读性，你可以想象出如果阅读全部的信息会怎样。

选择一个策略

我们在这儿讨论的两种定制错误生成的方式各有利弊：BOOST_MPL_ASSERT_MSG方便、小巧，并且对其意图具有高度的表达性，但如果用它去显示void、数组和函数类型，你就需要小心一些。而且它可能带来可读性问题，尤其当面临深度typedef替换时。对BOOST_MPL_ASSERT使用定制的判断式提供了对消息格式的更多控制，当然，这需要多做一些工作，并导致代码变得稍微复杂一些，而且除非小心地选择判断式名字，否则可能会导致混乱。毫无疑问，不存在对于所有需求而言都完美的策略，因此在作出选择之前，你要仔细地权衡一番。

8.3.3 类型打印

当一个模板元程序行为不端时，它就像一个令人费解的的黑盒，尤其当问题自身未在编译错误中得以呈现时，或者错误在真正的问题出现之后好久才出现时。出于诊断用途的考虑，有时故意生成一个诊断消息是有帮助的。对于大多数情形，这个简单的工具就足够：

```
template <class T> struct incomplete;
```

如果在任何地方需要知道T的类型，我们要做的就是导致incomplete<T>的实例化，例如：

```
template <class T>
struct my_metafunction
{
    incomplete<T> x; // 临时性的诊断
    typedef ... type;
 };
```

大多数C++编译器——实际上我们看到的所有编译器，都会生成一个错误消息告诉我们T是什么[⊖]。这项技术易受一个常见的告诫所影响：进行深度typedef替换的编译器可能会为T显示一个任意复杂的名字，这取决于T是如何计算的。

用于调试C/C++程序的一个历史悠久的技术是“将printfs插入代码中”，并检查结果所得的执行日志。然而，incomplete<T>技术更像是一个运行期断言：它展示给我们有问题的和导致错误的程序语句。记得我们说过大多数C++编译器不能很好地从错误中恢复吗？即使你的编译器在实例化incomplete<T>后继续向前推进，结果的可靠性类似于从一个已经报告运行期数据破坏的程序获得预期结果。

为了生成一个编译期执行日志，我们需要一种生成非错误诊断消息（即警告）的方式。由于没有哪一个构造可以导致所有编译器生成警告（实际上，大多数编译器允许你完全禁用警告），因此MPL提供了一个print 元函数，它很像identity，被调节用于在形形色色的流行的编译器上（在各自“常规设置”下）产生警告。例如，以下程序：

```
template <class T, class U>
struct plus_dbg
{
    typedef typename
      mpl::print< typename mpl::plus<T,U>::type >::type
    type;
};

typedef mpl::fold<
    mpl::range_c<int,1,6>
  , mpl::int_<0>
  , plus_dbg<_1,_2>
>::type sum;
```

⊖ 注意，我们不是写typedef incomplete<T> x;，因为这样写并不会导致incomplete<T>实例化，参见第2章的描述。

在GCC中生成如下诊断（以及其他一些信息）[⊖]：

```
foo.cpp: In instantiation of
'boost::mpl::print<boost::mpl::integral_c<int, 1> >':
...
foo.cpp:72: warning: comparison between signed and unsigned
integer expressions

foo.cpp: In instantiation of
'boost::mpl::print<boost::mpl::integral_c<int, 3> >':
...
foo.cpp:72: warning: comparison between signed and unsigned
integer expressions

foo.cpp: In instantiation of
'boost::mpl::print<boost::mpl::integral_c<int, 6> >':
...
foo.cpp:72: warning: comparison between signed and unsigned
integer expressions

foo.cpp: In instantiation of
'boost::mpl::print<boost::mpl::integral_c<int, 10> >':
...
foo.cpp:72: warning: comparison between signed and unsigned
integer expressions

foo.cpp: In instantiation of
'boost::mpl::print<boost::mpl::integral_c<int, 15> >':
...
foo.cpp:72: warning: comparison between signed and unsigned
integer expressions
```

自然而然，这些消息被混杂在编译器的实例化回溯信息中。这是另一个诊断过滤工具可以帮上忙的领域：STLFilt有一个选项（/showback:N），可以消除上面回溯信息中以省略号显示的内容，于是我们就得到一个简化了的编译期执行回溯。当然，如果你可以使用UNIX工具，将错误消息交由“grep print”处理同样可以使工作容易些。

⊖ GCC有一个罕见的怪癖，使用元函数转发会稍微干扰诊断。如果我们这么写：

```
template <class T, class U>
struct plus_dbg
  : mpl::print< typename mpl::plus<T,U>::type >
{};
```

以“In instantiation of...”打头的诊断消息将带有一个“某个MPL实现头文件而非foo.cpp”的文件名标签。尽管这个问题不足以阻止我们推荐在GCC中使用元函数 转发，但它值得你留意。

8.4 历史

在C++中有目的地生成编译期错误已有很长的历史了。正如在第1章中提到的那样，最早的C++ 模板元程序是Erwin Unruh编写的一个新奇的东西，它在模板错误消息中打印出一系列质数[Unruh94]。

我们第一次（1998年）是从Dietmar Kuehl那儿听说“将可读的错误消息编码进类型和函数的名字”的。到了2000年，BOOST_STATIC_ASSERT [Mad00]出现了，至少有两个工作将Kuehl的技术应用于改善STL生成的错误消息：“Static Interfaces”（由Brian McNamara和 Yannis Smaragdakis设计[MS00a]）和“Boost Concept Checking Library”（由Jeremy Siek设计[SL00]）。

8.5 细节

实例化回溯信息（Instantiation backtraces）

当模板没有通过编译时所得的那些冗长的错误消息，实际上是运行期调用堆栈的编译期等价物：它们常常包含有价值的信息，有助于将你带到一个问题的源头——如果你可以保持镇定而不被它吓倒。编译器厂商已经采取了很多措施，包括使用“with”子句和消除默认模板参数，使它们更具可读性。

typedef替换

很多编译器，包括微软Visual C++ 7/7.1以及大多数基于EDG（EDG-based）的编译器，试图通过以类型在代码中的原始命名来呈现它们从而改善错误消息。例如，它们可能显示一个typedef名字，而不是呈现该typedef所指向的那个底层类型。我们认为对类模板作用域的typedefs的替换对元程序除错带来的伤害大于帮助，因为元函数的结果总是通过嵌套的typedefs访问的。我们建议你手头至少拥有一款不进行深度typedef替换的编译器。GCC就是这样的一种编译器，并且它是免费的。

辅助工具

因为实例化回溯信息在一个程序的很多行报告错误，我们建议你获取某种IDE，它可以自动地显示与错误消息中的行号相关联的程序文本，从而允许你迅速检视位于一个实例化堆栈回溯的每一层代码。我们还建议你使用一个后处理过滤器（post-processing filter，例如STLFilt）来改善你的模板错误消息的可读性。

静态断言

BOOST_STATIC_ASSERT、BOOST_MPL_ASSERT_RELATION以及直截了当地使用BOOST_MPL_ASSERT，是向元程序添加“完整性检查”的极好的工具。它们对于编写元程序测试（只有当代码正确时才能通过编译）来说也非常有用。为了对你的元程序的使用方式强制执行约束，我们建议使用一些可以生成更可读的错误消息的工具。

定制的错误

我们只知道一种当模板实例化时产生具体消息的可移植的方式：即将其嵌入一个将会显示于一个实际编译器诊断中的类型或函数的名字之中。我们讨论了两种方式：BOOST_MPL_ASSERT与手工编写的判断式元函数（predicate metafunctions）的结合运用，以及使用

BOOST_ MPL_ASSERT_MSG。它们各有利弊。尽管都可以工作，但没有一个算得上是干净的解决方案。将来，我们希望有用于定制诊断的直接的语言支持。

类型打印

“定制的错误消息”技术可以扩充为警告（warnings），如果你需要在不打扰元程序执行的前提下检视一个类型的话。mpl::print<T>类模板可用于在广泛的编译器上生成这样的一个警告（取决于你设定的编译选项）。

8.6 练习

8-0. 编写并测试一个元程序，它使用mpl::print打印一系列质数。将你的程序和Erwin Unruh的原始代码（http://www.erwin-unruh.de/primorig.html）加以比较。

8-1. 改写第3章中的量纲分析代码中的断言，为程序库用户优化诊断消息。分析几款不同的编译器产生的结果消息。

8-2. MPL包含一些特殊的宏（macros）用于断言数值关系。因为，当可适用时，它们提供了比普通布尔断言（Boolean assertions）更便利的接口和更高质量的错误消息。还有其他种类的测试可从类似的方式获益吗？设计一个接口，用于处理这些情况，并且描述你希望看到它产生的那种输出。

8-3. 使用本章讨论的两种定制消息产生技术之一来实现你在练习8-2中设计的接口。

8-4. 修正第8.3.3节中的returning_ptr中手写的错误报告机制，使得在GCC中重要的信息显示在诊断消息的第一行。

第9章　跨越编译期和运行期边界

还记得运行期执行吗？我们认识到已经在编译期编程的同温层世界逗留好久了，现在邀请你与我们一道回到硬地上。最终，任何有趣的程序必须在运行期做一些事情。这一章是关于跨越编译期C++和运行期C++的边界的（如果你愿意，可以称之为“臭氧层”），从而使我们的元程序可以在现实用户的生活中发挥重要的作用。在C++中要做到这一点可能有数不清的方式，但有一些方式已经被证明比其他方式更有用。我们接下来将讨论一些最常用的技术。

9.1　for_each

最简单的STL算法应该有一个MPL对应物，事实正是如此。回顾一下，std::for_each遍历一个（运行期）序列，并且对每一个元素调用某个（运行期）函数对象（function object）。类似地，mpl::for_each遍历一个编译期序列并对之调用某个运行期函数。尽管std::for_each仅操作于运行期，但mpl::for_each却是个混血儿，它跨越编译期和运行期世界。

为什么是运行期函数对象

如果你在好奇为何mpl::for_each接收一个运行期函数对象而不是一个元函数，不妨这么考虑一下：正常来说，与std::for_each协同使用的函数对象返回void——即使它返回一个结果，该结果也会被舍弃掉。换句话说，该函数对象，如果它做了一些事情的话，就不得不修改程序的某些状态。由于函数式编程天生是无状态的，而模板元程序是函数式的，因此，除非我们打算对结果做一些事情，否则对序列的每一个元素调用一个元函数没有什么意义。

9.1.1　类型打印

你是否对如何查看你的类型序列的内容好奇过？如果使用一个根据std::type_ info::name生成有意义的字符串的编译器，我们可以用如下方式打印出类型序列中的每一个元素：

```
struct print_type
{
    template <class T>
    void operator() (T) const
    {
        std::cout << typeid(T).name() << std::endl;
    }
};

typedef mpl::vector<int, long, char*> s;
int main ()
```

```
{
    mpl::for_each<s>(print_type());
}
```

关于这段代码，有些东西是我们需要注意的。首先，print_type的函数调用运算符是模板化（templatized）的，因为它必须处理可能会出现在我们序列中的任何类型。除了你希望处理一个“所有元素都可被转化为一种类型”的序列，否则你的mpl::for_each函数对象将需要一个模板化的（templated）（退一万步来说，至少得是重载的）函数调用运算符。

其次，注意for_each以相应元素类型的值初始化（value-initialized）对象的方式来传递每一个序列元素[⊖]。如果你正在迭代一个整型常量外覆器序列，这种形式尤其方便，还记得吗？整型常量外覆器可以隐式地转换为其对应的运行期常量。另一方面，当迭代一个普通的类型序列时，需要一些特殊的关照：如果元素是一个引用类型（reference type），一个不带默认构造器的类类型（class type），或者是一个void，算法将无法通过编译，因为这些类型都不能被“值初始化（value-initialized）”。

我们可以使用一个小型外覆器模板（wrapper template）来转换该序列从而避免这个缺陷：

```
template <class T>
struct wrap {};

// 包含引用
typedef mpl::vector<int&, long&, char*&> s;
mpl::for_each<
    mpl::transform<s, wrap<_1> >::type
>(print_type());
```

我们还需要调整函数对象的签名，使其适应将被传递的参数类型的改变：

```
struct print_type
{
    template <class T>
    void operator()(wrap<T>) const    // 推导T
    {
        std::cout << typeid(T).name() << std::endl;
    }
};
```

⊖ 值初始化的概念已被添加到了C++标准中（添加到C++第一个技术勘误表中（technical corrigendum，TC1））。值初始化一个类型为T的对象，意味着：

- 如果T是一个带有用户声明的构造器（12.1）的类类型（条款9），那么T的默认构造器就会被调用。
- 如果T是一个不带用户声明的构造器的非联合体（non-union）类类型，那么T的每一个非静态数据成员和基类组件（base-class component）都是值初始化的（value-initialized）。
- 如果T是一个数组类型（array type），那么其每一个元素都是值初始化的（value-initialized）。
- 否则，对象就是零初始化的（zero-initialized）。

由于这是一个很常用的手法，所以MPL提供了for_each的第二种形式，它带有一个转换元函数作为附加模板参数。通过使用这第二种形式，我们可以避免构建一个全新的wrap特化序列。这样：

```
mpl::for_each<
    mpl::transform<s, wrap<_1> >::type
>(print_type());
```

就变成了：

```
mpl::for_each<s, wrap<_1> >(print_type());
```

对于s的每一个元素T，print_type对象将以一个wrap<T>实参而被调用。

9.1.2 类型探访

为了获得关于在函数边界“平滑类型问题”的更一般化的解决方案，我们可以应用Visitor模式[GHJV95]：

```
struct visit_type     // 一般化的visitation函数对象
{
    template <class Visitor>
    void operator()(Visitor) const
    {
        Visitor::visit();
    }
};

template <class T>    // 用于类型打印的具体visitor
struct print_visitor
{
    static void visit()
    {
        std::cout << typeid(T).name() << std::endl;
    }
};

int main()
{
    mpl::for_each<s, print_visitor<_1> >(visit_type());
}
```

在这里，visit_type函数对象期望其参数类型具有一个静态的visit成员函数，而且我们可以出于任何目的构建一个新的visitor对象。这是对我们早期for_each例子的一个微妙的改变，但请注意：print_visitor::visit中永远不会传入一个T对象。相反，for_each将一个print_visitor<T>实例（对于序列中的每一个T）传递给visit_type。有关T类型的信息被传输到print_visitor的模板参数中。

9.2 实现选择

在本小节中，我们将讨论一些基于编译期计算的结果来选择不同运行期行为或接口的不同方式。

9.2.1 if语句

控制一个运行期函数模板实现的最简单的方式，是测试if语句中的一个静态条件，如下：

```
template <class T>
void f(T x)
{
    if (boost::is_class<T>::value)
    {
        ...implementation 1...
    }
    else
    {
        ...implementation 2...
    }
}
```

由于条件可以完全地在编译期决定，很多编译器会优化掉is_class测试，并仅为被选择的if语句分支生成代码。

这种方式简单清晰，只要它可以工作，其概念上开销就非常小甚至没有。不幸的是，这项技术并非普遍适应。例如，考虑上面的函数被实现为下面的样子时会发生什么：

```
template <class T>
void f(T x)
{
    if (boost::is_class<T>::value)
    {
        std::cout << x::value;    // 处理整型常量外覆器
    }
    else
    {
        std::cout << x;           // 处理非外覆器
    }
}
```

这里的意图是使f能够打印一个整数类型（例如int）的值，或者一个整型常量外覆器（例如long_<5>）的值。然而，如果调用f(42)，我们将会得到一个编译错误。问题在于整个函数体都需要进行类型检查，包括if的全部两个分支，但我们无法访问int的本不存在的::value成员，不是吗？

9.2.2 类模板特化

我们可以将每一个if分支语句移到不同的函数中来解决刚才的问题，即移入一个类模板的静态成员函数中。通过对类模板进行特化，我们可以决定使用哪一个函数实现：

```
template <bool> // 处理整型常量外覆器
struct f_impl
{
    template <class T>
    static void print() { std::cout << T::value; }
};

template <>     // 针对非外覆器的特化
struct f_impl<false>
{
    template <class T>
    static void print(T x) { std::cout << x; }
};

template <class T>
void f(T x)
{
    f_impl<boost::is_class<T>::value>::print(x);
};
```

这种方式与我们在第2章中实现iter_swap所使用的方式类似，第4章中介绍的使用mpl::if_的版本是同样主题的变种。等我们在本章后面讨论结构选择时，还会看到同样的基本思想的进一步演化。

9.2.3 标签分派

我们已经从第5章对tiny序列所做的工作中获得了对标签分派概念的体验，其实它的基本思想是从运行期领域的泛型编程借用来的。运行期标签分派（Tag Dispatching）使用函数重载基于一个类型的属性来生成可执行代码。

我们可以在大多数C++标准程序库实现品中的advance算法中发现很好的例子。尽管概念上很简单：advance用来将一个迭代器i移动n个位置，但实际上编写该算法的工作是相当复杂的。取决于迭代器的遍历能力，可能需要完全不同的实现策略。例如，如果i支持随机访问，那么advance可以采用i += n实现，并且非常高效——常量时间复杂度。其他迭代器必须向前步进多步，从而使操作跟n呈线性关系。如果i是双向迭代器，那么n就可以是负数，因此我们必须在运行期决定是否递增或递减该迭代器。然而，如果传递的迭代器仅支持前向遍历的话，那么任何一个递减迭代器的函数将无法通过编译。因此，advance需要至少三个不同的实现。

为了在它们之间进行选择，我们必须使用以下归类标签类型（category tag types）中的concept信息：

```
namespace std
```

```
{
  struct input_iterator_tag { };
  struct forward_iterator_tag
    : input_iterator_tag { };

  struct bidirectional_iterator_tag
    : forward_iterator_tag { };

  struct random_access_iterator_tag
    : bidirectional_iterator_tag { };
}
```

一个标签（tag）仅仅是一个空类（empty class），其惟一的用途是在编译期传达一些信息，在这个例子中，该迭代器concept通过一个给定的迭代器类型进行模塑。每一个迭代器类型I具有一个相关联的归类标签（category tag），后者可以如下方式进行访问：

```
std::iterator_traits<I>::iterator_category
```

注意，在这个例子中，标签（tags）属于一个继承层次结构——它反映了这些标签（tags）所表示的concepts的精化层次结构（refinement hierarchy）。例如，每一个双向迭代器也都是一个前向迭代器，因此bidirectional_iterator_tag派生于一个forward_iterator_tag。

再一次，我们将三个实现分离进不同的函数体中，但是这一次我们通过传递一个“迭代器的空tag类型”的实例作为参数，使用重载来选择正确的那一个。

```
namespace std
{
  template <class InputIterator, class Distance>
  void __advance_impl(
      InputIterator& i
    , Distance n
    , input_iterator_tag)
  {
      while (n--) ++i;
  }

  template <class BidirectionalIterator, class Distance>
  void __advance_impl(
      BidirectionalIterator& i
    , Distance n
    , bidirectional_iterator_tag)
  {
      if (n >= 0)
        while (n--) ++i;
      else
        while (n++) --i;
  }
```

```
    template <class RandomAccessIterator, class Distance>
    void __advance_impl(
        RandomAccessIterator& i
      , Distance n
      , random_access_iterator_tag)
    {
        i += n;
    }

    template <class InputIterator, class Distance>
    void advance(InputIterator& i, Distance n)
    {
        typedef typename
          iterator_traits<InputIterator>::iterator_category
        category;

        __advance_impl(i, n, category());
    }
}
```

外部的advance函数调用和标签（tag）有着最佳匹配的__advance_impl重载，其他可能使用一个给定的迭代器所未实现的操作的重载则永远不会被实例化。这里用于迭代器标签（tags）的继承层次结构为我们带来好处：没有专门为forward_iterator_tag范畴的迭代器编写的__advance_impl，但由于forward_iterator_tag派生于input_iterator_tag，编译器会为输入迭代器和前向迭代器选择input_iterator_tag版本。如果我们使用标签（tags）类型的特化来选择实现的话，是不可能做到这一点的。

注意mpl::true_和mpl::false_产生了很好的分派标签。在下面的例子中，desperate_cast<T>(x)等价于static_cast<T>(x)，除非x恰好是一个多态类类型的对象（指针），在这种情况下，desperate_cast<T>(x)等价于dynamic_cast<T>(x)。

```
// 针对多态类型的实现
template <class T, class U>
T desperate_cast_impl2(U& x,  mpl::true_)
{
    return dynamic_cast<T>(x); // 当且仅当U是多态类型时合法
}

// 针对非多态类型的实现
template <class T, class U>
T desperate_cast_impl2(U& x,  mpl::false_)
{
    return static_cast<T>(x);
}

// 分派器（dispatcher）
```

```
template <class T, class U>
T desperate_cast_impl(U& x)
{
    return desperate_cast_impl2<T>(
        x
      , boost::is_polymorphic<
            typename boost::remove_pointer<U>::type
        >()
    );
}

// 共通接口
template <class T, class U>
T desperate_cast(U const& x) { return desperate_cast_impl<T>(x); }

template <class T, class U>
T desperate_cast(U& x) { return desperate_cast_impl<T>(x); }
```

因为整型type traits派生于它们的结果类型，所以我们只需创建一个整体的元函数特化boost::is_polymorphic<...>()的对象，来生成一个匹配mpl::true_或mpl::false_的tag。

9.3 对象生成器

至此，你可能已经对冗长的嵌套模板参数列表（nested template argument lists）感到不那么难受了，但我们认为你现在还不会忘记它们可能会变得多么笨拙难用。对象生成器（object generator）是一种用于推导类型信息的泛型函数，否则这些类型信息就必须以冗长的方式写出来。

为了看它是如何工作的，考虑如下模板，它组合两个可调用对象f和g，结果是一个新的函数对象，当对参数x进行调用时，计算f(g(x))，产生一个类型为R的值：

```
template <class R, class F, class G>
class compose_fg
{
 public:
    compose_fg(F const& f, G const& g)
      : f(f), g(g)
    {}

    template <class T>
    R operator()(T const& x) const
    {
        return f(g(x));
    }

 private:
    F f;
```

```
    G g;
};
```

下面的例子使用compose_fg为一个序列中的每一个元素计算$-\sin^2(x)$。

```
#include <functional>
#include <algorithm>
#include <cmath>

float input[5] = {0.0, 0.1, 0.2, 0.3, 0.4};
float output[5];

float sin_squared(double x) { return std::sin(std::sin(x)); }

float* ignored = std::transform(
    input, input+5, output,
  , compose_fg<float,std::negate<float>,float(*)(float)>(
       std::negate<float>(), &sin_squared
    )
);
```

哦，那个compose_fg实例化当然扎眼！它可以工作，但手工编写一个neg_sin_squared函数用于这个目的可能比使用compose_fg更容易一些。至少结果会比上面的方式有更好的可读性。幸运的是，如果拥有一个辅助的对象生成器（object generator）函数，我们就可以避免为compose_fg写出大多数模板参数：

```
template <class R, class F, class G>
compose_fg<R,F,G> compose(F const& f, G const& g)
{
    return compose_fg<R,F,G>(f,g);
}
```

compose的全部用途就是充当函数模板实参推导机制的一个工具。现在transform调用可以改写为：

```
float* ignored = std::transform(
    input, input+5, seq2
  , compose<float>(std::negate<float>(), &sin_squared)
);
```

因为编译器可以从传递给compose的实参的类型推导出所需的compose_fg特化的类型，因此不需要显式地写出类型。C++标准程序库的bind1st和bind2nd函数模板都是类似的生成器，它们分别产生binder1st和binder2nd类型的对象[⊖]。

一旦熟悉了它们完全的潜能，对象生成器允许用户生成一些真正可怕的——但威力极大的——模板类型，后者具有最低限度的混乱语法。当我们讨论本章后面的类型擦除（type erasure）时，

⊖ Boost Bind程序库——C++标准技术报告（technical report，TR1）的基础条目——提供了更好的方式来做同样的事情。

将学到关于它是如何工作的更多知识。

9.4 结构选择

你已经知道如何使用元函数来影响类成员个体的类型了：

```
template <class T>
struct X
{
    static int const m1 = metafunction1<T>::type::value;
    typedef typename metafunction2<T>::type m2;
    int m3(typename metafunction3<T>::type p);
    ...
};
```

在这个例子中，元程序计算m1的值，类型m2，以及m3的参数类型。然而，假定我们希望控制m2是否出现在一个给定的X特化中，该如何做呢？上面使用的方式允许我们管理一个给定的类成员（class member）的细节，但对于类的基础结构上的改变，还需要一种更有威力的技术。

结构选择（Structure selection）是指将类结构的可变部分放入公共基类或基类模板并使用一个元程序在它们之间进行选择。为了明白这是如何工作的，让我们修复compose_fg中的一个问题。compose_fg当前被定义为：

```
template <class R, class F, class G>
class compose_fg
{
 public:
    compose_fg(F const& f, G const& g)
      : f(f), g(g)
    {}

    template <class T>
    R operator()(T const& x) const
    {
        return f(g(x));
    }
 private:
    F f;
    G g;
};
```

你可能会好奇这里究竟存在什么样的问题：compose_fg是如此简单，我们一眼就可以看出它的正确性，进一步而言，它确实能够工作。问题不属于正确性的问题，而是效率的问题。在我们早先的例子中，我们生成如下类型的一个对象：

```
compose_fg<float,std::negate<float>,float(*)(float)>
```

因此F为std::negate<float>。在大多数实现中，std::negate的惟一成员就是其函数调用运算符：

```
T operator()(const T& x) const { return -x; }
```

换句话说，它是一个空类（empty class）。然而，C++标准要求每一个compose_fg的数据成员必须占有至少一个字节。在一个典型的类布局模式里[⊖]，第一个字节是属于f的，即使negate的特化没有数据成员。接下来是一些填充字节（比如说，3个字节），这是为了满足函数指针适当的内存对齐（memory alignment）的需要，接下来是g的内存（比如4个字节），从而生成一个具有8个字节的对象。如果我们可以完全消除掉f的存储，那么尺寸将会下降为4个字节。如果G也是一个空类（empty class），那么从理论上说，compose_fg对象的全部尺寸将会降至1个字节。我们不能比这做得更好了，因为规则说即使一个空类也必须具有非零的大小。

消除空类存储的方法之一是侦测它们（使用第2章描述的boost::is_empty type trait），并简单地忽略相应的数据成员。然而，这种方法有一些问题：

1. 它不透明：即使空类也可能有非平凡的构造器和析构器，如果我们不存储f和g的副本，compose_fg行为中的不同可能会让人惊讶。

2. 为了实现operator()我们仍然需要F和G对象，如果它们没有被存储，我们就需要构建它们，并且它们可能没有默认构造器（default constructors）。

幸运的是，存在一个更好的解决方案。编译器可以实现空基类优化（Empty Base Optimization，EBO），它允许一个空基类放置在和任何子对象相同的地址处，只要同一个类型的两个不同的子对象占有不同的地址即可。例如：

```
compose_fg<float,std::negate<float>,float(*)(float)>
```

可能具有理想的大小，如果compose_fg以这种方式编写的话：

```
template <class R, class F, class G>
class compose_fg  : F    // 如果为空，F可能会和g重叠
{
 public:
    typedef R result_type;

    compose_fg(F const& f, G const& g)
      : F(f), g(g)     // 采用f来初始化base（基类子对象）
    {}

    template <class T>
    R operator()(T const& x) const
    {
        F const& f = *this;    // 获取F子对象
        return f(g(x));
    }
 private:
```

⊖ 标准对大多数类的布局几乎没有施加任何约束，除了一个给定的类型的每一个基类子对象或成员子对象必须具有不同的地址且成员不可互相重叠外。对此规则的惟一例外是当类是一个“plain old data(POD)”时，2.5.4节给出了POD的技术定义。在这种情况下，类的布局遵循较可预知的一套规则。

```
    G g;
};
```

自然，我们不能将该结构用于所有的compose_fg特化：如果F是一个函数指针，将会得到一个编译错误，因为函数指针不是合法的基类。此外，我们不想在所有情况下都使用该结构：当G为空但F不为空时，我们希望从G派生compose_fg<R,F,G>。结构性变化的需要意味着结构选择是一个技术选择问题。

应用结构选择的第一步是委托对类结构可变部分的控制。本例中，F和G的存储方式不同，因此我们可以写：

```
// 稍后给出基类模板的定义
template <class F, bool F_empty, class G, bool G_empty>
class storage;
template <class R, class F, class G>
class compose_fg
  : storage<
    F,boost::is_empty<F>::value
  , G,boost::is_empty<G>::value
>{
   typedef
     storage<
    F,boost::is_empty<F>::value
  , G,boost::is_empty<G>::value
   > base;

public:
   compose_fg(F const& f, G const& g)
     : base(f, g)
   {}

   template <class T>
   R operator()(T const& x) const
   {
       F const& f = this->get_f();
       G const& g = this->get_g();
       return f(g(x));
    }
};
```

现在我们只要写storage，对于每一对F_empty和G_empty的组合（共4对），它都有着正确的结构，并通过get_f和get_g成员暴露对存储的F和G的访问[⊖]：

```
template <class F, class G>
class storage<F,false,G,false> // F和G均不为空
{
```

⊖ 如果你注意到这段代码在某些情况下无法工作，别担心，本章练习题会要求你给出修复解决方案。

```
 protected:
      storage(F const& f, G const& g)
        : f(f), g(g)
      {}
      F const& get_f() { return f; }
      G const& get_g() { return g; }
 private:
      F f;
      G g;
};

template <class F, class G>
class storage<F,false,G,true> // G为空
  : private G
{
 protected:
      storage(F const& f, G const& g)
        : G(g), f(f)
      {}
      F const& get_f() { return f; }
      G const& get_g() { return *this; }
 private:
      F f;
};

template <class F, class G>
class storage<F,true,G,false> // F为空
  : private F
{
 protected:
      storage(F const& f, G const& g)
        : F(f), g(g)
      {}
      F const& get_f() { return *this; }
      G const& get_g() { return g; }
 private:
      G g;
};

template <class F, class G>
class storage<F,true,G,true> // F和G均为空
  : private F, private G
{
 protected:
      storage(F const& f, G const& g)
        : F(f), G(g)
```

```
    {}
    F const& get_f() { return *this; }
    G const& get_g() { return *this; }
};
```

由于EBO是可选的，因此没有任何保证能使得这些发生作用。话虽这么说，通过在不同的基类中进行选择，我们至少给编译器一个机会来优化掉用于空子对象的存储，并且大多数编译器都实现了EBO，因此可以利用这一点（参见练习以便了解更多的信息）。你也许还想看看Boost compressed_pair模板[CDHM01]，它实现了我们这里使用的EBO模式的一般化版本。

9.5 类复合

如果我们可以使用结构选择来一次控制一个类的结构，那我们就可以细粒度的步骤反复使用它来创建类结构。例如，为了生成一个struct，其成员具有由“一个类型序列给定的”类型，我们可以应用fold算法：

```
// 细粒度的struct元素：存储一个T（类型的value），并继承于More
template <class T, class More>
struct store : More
{
    T value;
};

typedef mpl::vector<short[2], long, char*,  int> member_types;

struct empty {};

mpl::fold<
    member_types, empty, store<_2,_1>
>::type generated;
```

生成一个generated对象，其类型为：

```
store<int
  , store<char*
      , store<long
          , store<short[2], empty> > > >
```

上面展示的每一个store特化都表示继承结构中的一层，它们分别包含一个成员，其类型为member_types中的类型之一。

实际上，使用这种方式复合的类可能很棘手，除非它们被小心地加以构建。尽管generated实际上包含member_types中的每一个类型的成员，但它们仍很难被访问。最明显的问题是它们都叫value：除了第一个外，我们无法直接访问其他任何一个，因为其余的都被继承结构的层（layers）掩盖了。不幸的是，我们对于这种重复没有任何办法，这是在应用类复合（class composition）时一个无法更改的事实，因为尽管我们可以很容易地产生成员类型（member

types），但没有什么办法来使用模板产生成员名字（member names）[⊖]。

此外，很难去访问一个给定类型的value成员，即便它转换成一个适当的基类。为了弄清楚原因，考虑当访问存储在generated中的long值时涉及到哪些事情。因为每一个store特化都派生于它的第二个参数，我们不得不写：

```
long& x = static_cast<
            store<long, store<short[2], empty> >&
          >(generated).value;
```

换句话说，访问store的任何成员都需要知道原始序列中在它的类型之前的所有类型。我们可以让编译器的函数实参推导（function argument deduction）机制为我们做推断基类链（base class chain）的工作：

```
template <class T, class U>
store<T,U> const& get(store<T,U> const& e)
{
    return e;
}

char* s = get<char*>(generated).value;
```

在上面的例子中，get的第一个模板实参被限制为char*，从而有效的函数参数变为store<char*,U> const&，它匹配包含一个char*成员的generated的基类。

还有一个稍微不同的模式，允许我们略微更整洁地解决这个问题。一如既往，应用软件工程的基本理论（Fundamental Theorem of Software Engineering）[⊜]。我们只要加入一个间接层：

```
// 细粒度的struct元素：包装一个T
template <class T>
struct wrap
{
    T value;
};

// 另一个间接层
template <class U, class V>
struct inherit : U, V
{};

typedef mpl::vector<short[2], long, char*, int> member_types;

struct empty {};

mpl::fold<
```

⊖ 使用预处理元编程（preprocessor metaprogramming）可以生成成员名字，参见附录A以便了解更多的信息。

⊜ 参见第2章了解该术语的由来。

```
    member_types, empty, inherit<wrap<_2>,_1>
>::type generated;
```

现在generated的类型为：

```
inherit<wrap<int>
  , inherit<wrap<char*>
      , inherit<wrap<long>
          , inherit<wrap<short[2]>
              , empty
            >
        >
    >
>
```

由于inherit<U,V>同时派生于U和V，因此上面的类型（间接地）派生于“序列中每一个T所对应的”wrap<T>。我们现在可以用如下方式访问一个类型为long的value成员：

```
long& x = static_cast<wrap<long> &>(generated).value;
```

由于以这种方式的类生成（Class generation）是一个常见的元编程活动，因此MPL提供了用于该目的的现成工具。尤其是，我们可以用mpl::empty_base和mpl::inherit来取代empty和inherit。这个程序库还包含一个适当命名的inherit_linearly元函数，它为我们调用fold，inherit_linearly带来一个默认的初始化类型mpl::empty_base：

```
template <class Types, class Node, class Root = empty_base>
struct inherit_linearly
  : fold<Types,Root,Node>
{
};
```

有了这些工具在手，我们就可以更方便地将刚才的例子改写如下：

```
#include <boost/mpl/inherit.hpp>
#include <boost/mpl/inherit_linearly.hpp>
#include <boost/mpl/vector.hpp>

// 细粒度的struct元素
template <class T>
struct wrap
{
    T value;
};

typedef mpl::vector<short[2], long, char*, int> member_types;
mpl::inherit_linearly<
    member_types, mpl::inherit<wrap<_2>,_1>
>::type generated;
```

Andrei Alexandrescu [Ale01] 已经广泛地探索这些类复合（class composition）模式在实践中

的应用。例如，他使用类复合为一个泛型多分派框架（generic multiple dispatch framework）生成visitor类。

9.6 （成员）函数指针作为模板实参

整型常量并非惟一种类的非类型模板参数。实际上，几乎任何种类的可在编译期决定的值都是允许的，包括：

- 指向具体函数的指针和引用（Pointers and references to specific functions）
- 指向静态存储的数据的指针和引用（Pointers and references to statically stored data）
- 指向成员函数的指针（Pointers to member functions）
- 指向数据成员的指针（pointers to data members）

通过使用这些种类的模板参数，我们可以获得极大的效益。当早先的compose_fg 类模板用在两个函数指针上时，它总是至少和指针自身一样大，因为它需要存储那些值。然而，当一个函数指针被当作一个参数传递时，就根本不需要存储区。

为了示范这种技术，让我们构建一个新的复合函数对象模板（composing function object template）：

```
template <class R, class F, F f, class G, G g>
struct compose_fg2
{
    typedef R result_type;

    template <class T>
    R operator()(T const& x) const
    {
        return f(g(x));
    }
};
```

尤其要注意的是，compose_fg2没有数据成员。我们可以用它为一个序列中的各个元素计算$\sin^2(\log^2(x))$：

```
#include <functional>
#include <algorithm>
#include <cmath>

float input[5] = {0.0, 0.1, 0.2, 0.3, 0.4};
float output[5];

float log2(float x) { return std::log(x)/std::log(2.0f); }
float sin_squared(float x) { return std::sin(std::sin(x)); }

typedef float (*floatfun)(float);

float* ignored = std::transform(
```

```
    input, input+5, output
  , compose_fg2<float, floatfun,sin_squared, floatfun,log2>()
);
```

不要被这里涉及有函数指针这一事实所欺骗：在大多数编译器上，你不会付出一个间接函数调用代价。因为它知道f和g所指示的函数的精确身份，编译器应该能优化掉传递给std::transform的compose_fg2空对象（empty compose_fg2 object），并在实例化的transform算法的本体中生成对log2 和sin_squared的直接调用。

尽管带来了效率上的好处，compose_fg2也伴随有一些显著的局限性：

- 因为类类型的值（values of class type）不是合法的模板参数，所以compose_fg2不能用于复合任意的函数对象（请参见练习9-2）。
- 没有什么方式可以用于为compose_fg2构建一个对象生成器函数（object generator function）。对象生成器必须以函数实参的方式接受被复合的函数，并使用这些值作为compose_fg2模板的实参：

```
template <class R, class F, class G>
compose_fg2<R,F,f,G,g> compose(F f, G g)
{
    return compose_fg2<R,F,f,G,g>();    // 错误
}
```

不幸的是，任何被传递给函数的值都会无法挽回地进入运行期世界。在这一点，没有什么办法使它作为一个类模板的实参而不导致编译错误[⊖]。

9.7 类型擦除

尽管这本书的大多数例子在强调静态类型信息的价值，然而有时将那种信息抛弃掉会更合适一些。为了让你明白我们的意思，考虑如下两个运行期表达式：

```
1. compose<float>(std::negate<float>(), &sin_squared)
2. std::bind2nd(std::multiplies<float>(), 3.14159)
```

即使这些表达式的结果具有不同的类型（相应地是compose_fg<float,std::negate<float>, float(*)(float)>和 std::binder2nd<std::multiplies<float> >），它们拥有一个共同的本质东西：可用类型float的实参调用任何一个表达式并获得一个float结果。在一个泛型函数调用中，这个“允许任一个表达式被另一个所替换”的共通的接口，就是一个经典的静态多态（static polymorphism）例子：

```
std::transform(
    input, input+5, output
  , compose<float>(std::negate<float>(), &sin_squared)
);
```

⊖ C++标准社区（比如comp.lang.c++.moderated）正在讨论可以绕过这个限制的语言扩充，让我们关注接下来几年的进展吧。

```
std::transform(
    input, input+5, output
  , std::bind2nd(std::multiplies<float>(), 3.14159)
);
```

然而，函数模板并非总是处理多态的最佳方式。

- 结构在运行期改变的系统（例如图形用户界面（graphical user interfaces，GUI））通常要求运行期分派（runtime dispatching）。
- 函数模板无法被编译成目标代码并以程序库的方式交付。
- 通常来说，函数模板的每一次实例化都会生成新的机器码。当函数处于你的程序的关键路径上或函数极小时，这是件好事，因为代码可以被内联（inlined）和局部化（localized）。然而，如果该调用不是一个显著的瓶颈，你的程序可能会变得更大并且有时甚至更慢。在这小节中，我们将展示单个非模板函数是如何操作所有符合给定接口类型的对象的，在这个例子中，其实例可以用一个float进行调用的类型（types），产生一个float结果。

9.7.1 一个例子

设想我们已经塑造了一个算法原型，用于实现“令人惊讶”的屏幕保护效果，为了使用户感兴趣，我们希望寻找一个方式，使得用户可以定制其行为。生成屏保的算法相当复杂，但它很容易调节：通过替换一个简单的数值函数——在算法的核心它被每一帧调用一次，我们可以使其生成完全不同的模式。仅仅为了允许这个参数化从而将screensaver整体模板化的做法显得有点浪费，因此我们决定使用一个指向转换函数的指针：

```
typedef float (*floatfunc)(float);

class screensaver
{
 public:
    explicit screensaver(floatfunc get_seed)
      : get_seed(get_seed)
    {}

    pixel_map next_screen() // 主要算法
    {
        float center_pixel_brightness = ...;
        float seed = this->get_seed(center_pixel_brightness);
        使用seed的复杂计算……
    }

 private:
    floatfunc get_seed;
    其他成员…….
};
```

我们花了若干天实现了一个有趣的定制函数菜单，并且建立一个用户界面在它们之间进行

选择。然而，正当准备交货时，我们发现了一个新的定制家族，它允许生成很多新的令人惊讶的样式。这些新的定制需要我们维持一个具有128个整型参数的状态向量，该向量在每一次next_screen()被调用时被修改。

9.7.2 一般化

我们可以向screensaver添加一个std::vector<int>成员，并且修改next_screen，将该成员以一个附加参数的形式传递给get_seed函数，从而整合我们的发明：

```
class screensaver
{
    pixel_map next_screen()
    {
        float center_pixel_brightness = ...;
        float seed = this->get_seed(center_pixel_brightness,
                                    state);
        ...
    }
 private:
    std::vector<int> state;
    float (*get_seed)(float, std::vector<int>& s);
    ...
};
```

如果那么做，我们将被迫改写现有的转换来接收一个它们不需要的state向量。此外，看上去好像我们使用新的有趣方式来定制算法，因此这种硬编码的定制接口选择缺乏吸引力。毕竟，我们的下一个定制可能需要一个完全不同类型的状态数据。如果我们采用定制类（customization class）来替换定制函数指针（customization function pointer），我们就可以将状态和该类实例（class instance）进行绑定，并且消除screensaver对一个特定类型的状态（state）的依赖性：

```
class screensaver
{
 public:
    struct customization
    {
        virtual ~customization() {}
        virtual float operator()(float) const = 0;
    };

    explicit screensaver(std::auto_ptr<customization> c)
      : get_seed(c)
    {}
    pixel_map next_screen()
    {
        float center_pixel_brightness = ...;
        float seed = (*this->get_seed)(center_pixel_brightness);
```

```
        ...
    }

 private:
    std::auto_ptr<customization> get_seed;
    ...
};
```

9.7.3 “手工”类型擦除

现在我们可以编写一个类，它持有额外的状态（state）作为一个成员，并且在其operator()中实现我们的定制（customization）：

```
struct hypnotic : screensaver::customization
{
    float operator()(float) const
    {
        使用this->state……
    }
    std::vector<int> state;
};
```

为了使得“不需要一个状态向量的定制（customizations）”适合这个新框架，我们需要将它们包装在派生于screensaver::customization的类中：

```
struct funwrapper : screensaver::customization
{
    funwrapper(floatfunc pf)
      : pf(pf) {}

    float operator()(float x) const
    {
        return this->pf(x);
    }

    floatfunc pf; // 存储的函数指针
};
```

现在我们开始看到运作中的类型擦除（type erasure）的第一个线索。运行期多态基类screensaver::customization用来“擦除”两个派生类的细节（从screensaver、hypnotic和funwrapper是不可见的角度来看），包括存储的状态向量和函数指针类型。

如果你抗议我们已经向你展示的不过是相当老式的面向对象编程，你是正确的。然而，故事还没有结束：有许多其他类型其实也可以用一个float实参进行调用，并产生另一个float（结果）。如果我们希望采用一个预先存在的、接收一个double参数的函数来定制screensaver，我们需要制作另一个外覆器（wrapper）。对于任何可调用的类来说都是这样，即使其函数调用运算符精确

地匹配float (float)签名，也是如此。

9.7.4 自动类型擦除

难道自动化外覆器的构建不是好得多吗？通过模板化（templatizing）派生的定制（derived customization）和screensaver的构造器，我们可以做到这一点：

```
class screensaver
{
 private:
    struct customization
    {
        virtual ~customization() {}
        virtual float operator()(float) const = 0;
    };

    template <class F>                          // F的一个外覆器（wrapper）
    struct wrapper : customization
    {
        explicit wrapper(F f)
          : f(f) {}                             // 存储一个F

        float operator()(float x) const
        {
            return this->f(x);                  // 委托给存储的F
        }
      private:
          F f;
    };

 public:
    template <class F>
    explicit screensaver(F const& f)
      : get_seed(new wrapper<F>(f))
   {}
   ...
 private:
    std::auto_ptr<customization> get_seed;
    ...
};
```

我们现在可以传递任何函数指针或函数对象给screensaver的构造器，只要传递的东西可以一个float实参进行调用、并且结果可被转换回一个float即可。构造器“擦除”包含在其实参中的静态类型信息，同时保持通过定制的虚拟函数调用运算符来访问其本质功能的能力（即采用一个float来调用它并返回一个float结果的能力）。然而，为了使得类型擦除真正具有说服力，我们不得不将它同screensaver完全分离而进一步执行这一步骤。

9.7.5 保持接口

对类型擦除（type erasure）最完整的表达是，它是这样的一个过程：将具有一个共通接口的形形色色的类型变成具有同样接口的“一个”类型。到目前为止，我们已经将多种函数指针和对象类型变成一个auto_ptr<customization>，然后我们将其作为screensaver的一个成员存储起来。然而，该auto_ptr不是可调用的：只有它指向的东西（pointee）才是可调用的。但是，我们距离拥有一个一般化的float-to-float函数并不远，实际上可以向screensaver自身添加一个函数调用运算符，但我们采用的是另一种方式：将整个函数外覆设施重构为一个单独的float_function类，从而我们可以将其应用于任何项目中。这样，我们就可以将screensaver类重构如下：

```
class screensaver
{
 public:
    explicit screensaver(float_function f)
      : get_seed(f)
    {}

    pixel_map next_screen()
    {
        float center_pixel_brightness = ...;
        float seed = this->get_seed(center_pixel_brightness);
        ...
    }

 private:
    float_function get_seed;
    ...
};
```

上面的重构揭示了所有函数对象的共通接口的另一个部分（现在我们已经认为是理所当然的），那就是可复制性（copyability）。为了使得复制float_function对象并将其存储于screensaver中成为可能，我们采用被包装类型的复制构造器完成了同样的“虚拟化”过程，这也解释了以下实现品中的clone函数的必要性：

```
class float_function
{

 private:
    struct impl
    {
        virtual ~impl() {}
        virtual impl* clone() const = 0;
        virtual float operator()(float) const = 0;
    };
```

```
    template <class F>
    struct wrapper : impl
    {
        explicit wrapper(F const& f)
          : f(f) {}

        impl* clone() const
        {
            return new wrapper<F>(this->f);    // 委托
        }

        float operator()(float x) const
        {
            return this->f(x);                 // 委托
        }

     private:
        F f;
    };

public:
    // 从F隐式转换
    template <class F>
    float_function(F const& f)
      : pimpl(new wrapper<F>(f)) {}
    float_function(float_function const& rhs)
      : pimpl(rhs.pimpl->clone()) {}

    float_function& operator=(float_function const& rhs)
    {
        this->pimpl.reset(rhs.pimpl->clone());
        return *this;
    }

    float operator()(float x) const
    {
        return (*this->pimpl)(x);
    }

 private:
    std::auto_ptr<impl> pimpl;
};
```

现在我们就拥有了一个类，它可以“满足”任何这样的类型功能：这种类型可以通过一个float进行调用，且其返回类型可以转换为一个float。这个基本的模式在Boost函数库（另一个出现在TR1中的程序库）中处于核心位置，在该程序库中，它被泛化为支持任意类型的参数和返回

类型。实际上，我们的float_function整个定义可用如下typedef取代：

```
typedef boost::function<float (float x)> float_function;
```

传递给boost::function的模板实参是一个函数类型，它指定了结果函数对象的参数和返回类型。

9.8 奇特的递归模板模式

这一小节的标题所命名的模式首先是由James Coplien [Cop96]确定为“奇特的递归(curiously recurring)”，因为看上去它是如此频繁地出现。我们就不兜圈子了，以下就是。

奇特的递归模板模式（Curiously Recurring Template Pattern，CRTP）
类X具有一个模板特化作为基类，该模板特化将X自身作为实参：

```
class X
  : public base<X>
{
   ...
};
```

因为X派生于一个了解X自身的类，所以此模式有时也被称为奇特的递归（curiously recursive）。

CRTP威力巨大，源于模板实例化的工作方式：尽管当派生类被声明（或被实例化，如果它也是模板的话）时基类模板中的声明被实例化，但只有当编译器知道派生类（derived class）的整个声明后基类模板的成员函数的本体才被实例化。结果，这些成员函数可以使用派生类的实现细节。

9.8.1 生成函数

下面的例子展示了CRTP如何为任何支持前缀operator<的类生成一个operator>：

```
#include <cassert>

template <class T>
struct ordered
{
    bool operator>(T const& rhs) const
    {
        // locate full derived object
        T const& self = static_cast<T const&>(*this);
        return rhs < self;
    }
};

class Int
  : public ordered<Int>
{
```

```
 public:
    explicit Int(int x)
      : value(x) {}

    bool operator<(Int const& rhs) const
    {
        return this->value < rhs.value;
    }

    int value;
};

int main()
{
    assert(Int(4) < Int(6));
    assert(Int(9) > Int(6));
}
```

这种使用static_cast和CRTP来访问派生对象的技术有时被称为“Barton和Nackman技巧”，因为它第一次出现于John Barton和Nackman的《Scientific and Engineering C++》[BN94]一书中。尽管该书写于1994年，然而Barton和Nackman的书开拓了泛型编程和元编程技术，直到今天它们仍然被认为是高级技术。我们高度推荐这本书。

CRTP和类型安全

一般而言，强制转型（casts）打开了一个类型安全漏洞，但在这个例子中，这个漏洞不算大，因为只有T派生于ordered<T>时static_cast才能通过编译。陷入困境的惟一可能情况是从ordered的同一个特化派生出两个不同的类：

```
class Int : public ordered<Int> { ... };
class bogus : public ordered<Int> {};
bool crash = bogus() > Int();
```

在这个例子中，因为Int已经派生于ordered<Int>，operator>可以编译，但static_cast试图将一个指向bogus实例的指针转型为一个指向Int的指针，这将导致未定义行为。

这个技巧的另一个变种可用于在基类的命名空间内定义非成员友元函数（non-member friend functions）：

```
namespace crtp
{
  template <class T>
  struct signed_number
  {
      friend T abs(T x)
      {
          return x < 0 ? -x : x;
      }
```

```
  };
}
```

如果signed_number<T>用做任何支持一元operator-和operator<的类的基类，那么它将自动获得一个非成员abs函数（non-member abs function）：

```
class Float : crtp::signed_number<Float>
{
 public:
    Float(float x)
      : value(x)
    {}

    Float operator-() const
    {
        return Float(-value);
    }

    bool operator<(float x) const
    {
        return value < x;
    }

    float value;
};

Float const minus_pi = -3.14159265;
Float const pi = abs(minus_pi);
```

这里的abs函数被发现于命名空间crtp中，这是通过实参相依的查找（argument-dependent lookup，ADL）达成的。只有非限定性调用才服从于ADL，ADL在“函数实参及其基类”的命名空间内查找可行的函数重载。

定义在类模板本体内的友元函数（friend functions）具有一个奇怪的属性，那就是，除非也声明在本体的外面，否则它们只能通过ADL被发现。显式的资格限定无法工作：

```
Float const erroneous = crtp::abs(pi); // 错误
```

当使用CRTP生成自由函数时，不要忘记这个限制。

9.8.2 管理重载决议

CRTP最简单的形式用于在其他方面无关的类之间建立一个继承关系（用于重载决议的目的），以及用于避免过于一般化的函数模板实参（function template arguments）。例如，如果我们编写一个泛型函数drive操作Vehicles（其中Vehicle是一个Concept），我们可以写：

```
template <class Vehicle>
void drive(Vehicle const& v)
{ ... }
```

这个定义相当好，直到有人编写了一个名为“drive”的泛型函数来操作Screws，问题就来了：

```
template <class Screw>
void drive(Screw const& s)
{ ... }
```

问题在于尽管Vehicle和Screw这两个标识符对我们来说有意义，但对于编译器来说它们并没有区别。如果这两个drives位于同一个命名空间中，两个声明就指向同一个实体。如果两个函数本体都可见，我们就会得到一个编译错误，但是，如果只有一个本体可见，我们又无形中违反了标准的“一次定义规则（One Definition Rule）”，从而导致未定义行为。

即使它们不在同一个命名空间中，对drive的非资格限定的调用也可能会导致歧义[⊖]，甚至更糟，即可能导致调用错误的函数[⊜]。因为ADL无声无息地将无关的函数加入到重载函数集中，并且因为非资格限定的函数调用是如此自然，编写“参数可以匹配所有类型”的完全一般化的函数模板是极其危险的。考虑如下人为捏造的例子：

```
#include <list>

namespace utility
{
  // 采用0填充区间
  template <class Iterator>
  Iterator clear(Iterator const& start, Iterator const& finish);
  // 对序列执行一些转换
  template <class Iterator>
  int munge(Iterator start, Iterator finish)
  {
      // ...
      start = clear(start, finish);
      // ...
  }
}

namespace paint
{
  template <class Canvas, class Color> //一般化的template
  void clear(Canvas&, Color const&);

  struct some_canvas { };
  struct black { };

  std::list<some_canvas> canvases(10);
  int x = utility::munge(canvases.begin(), canvases.end());
}
```

⊖ 译注：编译期错误。

⊜ 译注：运行期错误。

实际上，munge的实例化通常无法通过编译，因为list迭代器是采用paint::some_canvas而参数化的类模板。ADL（Argument-dependent lookup）看到那个参数并在命名空间paint中发现一个clear定义，它被添加到重载集中。在munge内部，对于传入的参数来说，paint::clear恰巧比utility::clear有着略好的匹配。幸运的是，paint::clear返回void，因此赋值失败。但是，请设想paint::clear返回Canvas&的情况。在这种情况下，代码可能会“干脆利落地”通过编译，但它可能会神不知、鬼不觉地做一些违背我们本意的事情。

为了解决这个问题，我们可以使用CRTP来识别Vehicle和Screw concepts模型。我们只需添加这样的条件要求：它模塑每一个公开派生于相应CRTP基类的concept：

```
template <class Derived>
struct vehicle
{};

template <class Derived>
struct screw
{};
```

现在我们的drive函数模板可改写为更具有识别力。应用通常的向下转型：

```
template <class Vehicle>
void drive(vehicle<Vehicle> const& v)
{
    Vehicle const& v_= static_cast<Vehicle const&>(v);
    ...
};

template <class Screw>
void drive(screw<Screw> const& s)
{
    Screw const& s_= static_cast<Screw const&>(s);
    ...
};
```

9.9 显式管理重载集

有时候，CRTP对于限制一般化的函数模板实参的范围是不够的。例如，我们可能希望函数模板操作内建类型（它们没有基类），或者操作已有的第三方类型（我们不想去修改它们）。幸运的是，如果我们能在编译期决定一个参数类型的适当性，Boost的enable_if 模板家族将允许我们非侵入地（non-intrusively）管理重载集。

例如，下面的函数模板仅应用于算术类型的迭代器上。本小节中的例子使用boost::iterator_value，这是一个元函数，用来获取一个迭代器的value_type。

```
#include <iterator>
#include <boost/utility/enable_if.hpp>
#include <boost/type_traits/is_arithmetic.hpp>
```

```
#include <boost/iterator/iterator_traits.hpp>
template <class Iterator>
typename boost::enable_if<
    boost::is_arithmetic<                          // 启用条件
       typename boost::iterator_value<Iterator>::type
    >
  , typename                                       // 返回类型
      boost::iterator_value<Iterator>::type
>::type
sum(Iterator start, Iterator end)
{
    typename boost::iterator_value<Iterator>::type x(0);
    for (;start != end; ++start)
        x += *start;
    return x;
}
```

如果启用条件C的::value为true，enable_if<C,T>::type将为T，于是sum将会返回一个“Iterator的value_type”的对象。否则，sum将从重载决议过程中消失！我们马上就会解释为什么它会消失，然而，为了获得一个认知，请考虑：如果我们试图对非算术类型的迭代器调用sum，编译器将会报告没有函数可以匹配该调用。如果我们只是简单地用

```
std::iterator_traits<Iterator>::value_type
```

来代替enable_if<...>:: type，对value_type是std:: vector<int>的迭代器调用sum将会在sum内部失败，因为那里试图使用operator+=。如果迭代器的value_type是std::string，它将会干净地得到编译，但所得结果可能并不是我们想要的。

当有函数重载发挥作用时，这项技术就变得真正有趣起来。因为sum已经被限制为（接收）适当的参数，我们现在可以添加一个重载，它允许我们sum vector<vector<int> >的所有算术元素，以及其他算术类型的嵌套的容器。

```
// 给定一个指向某个container的Iterator，
// 获取该container的iterators的value_type。
template <class Iterator>
struct inner_value
  : boost::iterator_value<
      typename boost::iterator_value<Iterator>::type::iterator
    >
{};

template <class Iterator>
typename boost::lazy_disable_if<
    boost::is_arithmetic<                          // 禁用条件
        typename boost::iterator_value<Iterator>::type
    >
  , inner_value<Iterator>                          // 结果无函数
>::type
```

```
sum(Iterator start, Iterator end)
{
    typename inner_value<Iterator>::type x(0);

    for (;start != end; ++start)
        x += sum(start->begin(), start->end());

    return x;
}
```

lazy_disable_if中的“disable”指出当条件被满足时，该函数被从重载集中移走了，“lazy”则意味着函数的结果::type是以一个无参元函数调用第二个参数的结果[⊖]。

注意，只有当迭代器的值类型是另一个迭代器时，inner_value<Iterator>才能被调用。否则，当没有发现内部的（非）迭代器的值类型时，将会发生一个错误。如果我们试图贪婪地计算结果类型，那么，在重载决议期间，任何时候当迭代器的值类型被证明是算术类型而不是另一个迭代器时，就会发生错误。

现在，让我们看看这个魔法是如何工作的。这里是enable_if的定义：

```
template <bool, class T = void>
struct enable_if_c
{
    typedef T type;
};

template <class T>
struct enable_if_c<false, T>
{};

template <class Cond, class T = void>
struct enable_if
  : enable_if_c<Cond::value, T>
{};
```

注意，当C是false时，enable_if_c<C,T>::type不存在！C++标准的重载决议规则（14.8.3节）说，当一个函数模板的实参推导失败时，它对被考虑准备调用的候选函数集没有贡献，并且不会导致一个错误[⊜]。这个原则已经被David Vandevoorde和Nicolai Josuttis [VJ02]赋予“替换失败并非错误（Substitution Failure Is Not An Error，SFINAE）”的名号。

⊖ 出于完整性考虑，enable_if.hpp包含了平凡的disable_if和lazy_enable_if 模板，以及所有四个版本的_C后缀的版本，后者接收整型常量而不是外覆器作为第一个参数。

⊜ 你可能想知道为何需要inner_value和缓式评估，当enable_if自身不导致一个错误的时候。模板实参推导规则包括一个条款（14.8.2节第2段），它列出了一些条件，在这些条件下，函数模板签名中的一个无效的被推导的类型将会导致推导失败。enable_if所使用的形式在该条件清单之列，但实参推导期间的其他模板（例如iterator_value）的实例化过程中发生的错误则不在该清单之列。

9.10 sizeof技巧

尽管用做函数实参传进运行期世界的值是不变的，但我们仍可以使用sizeof运算符在编译期获得关于函数调用的结果类型的一些信息。这种技术已经成为“包括Boost Type Traits程序库中的很多组件”在内的众多低层模板元函数的基础。例如，给定：

```
typedef char yes;  // sizeof(yes) == 1
typedef char (&no)[2]; // sizeof(no)  == 2
```

我们可以编写一个trait，将类和联合体（unions）与其他类型区分开，如下：

```
template <class T>
struct is_class_or_union
{
    // 如果arg的类型无效，则SFINAE将排出掉这一个
    template <class U>
    static yes tester(int U::*arg);

    // 重载决议优先选择除“...”版本之外的任何其他候选者
    template <class U>
    static no tester(...);

    // 看看当U == T 时选择了哪一个重载
    static bool const value
        = sizeof(is_class_or_union::tester<T>(0)) == sizeof(yes);

    typedef mpl::bool_<value> type;
};

struct X{};
BOOST_STATIC_ASSERT(is_class_or_union<X>::value);
BOOST_STATIC_ASSERT(!is_class_or_union<int>::value);
```

这种特殊的SFINAE和sizeof联合使用技巧首先是由Paul Mensonides在2002年3月发现的。让人感到惭愧的是，在标准C++中，我们只能从运行期获取表达式的类型的大小，而不能从类型自身获得其大小。例如，如果能这么写的话就好了：

```
// 一般化的加法函数对象
struct add
{
    template <class T, class U>
    typeof(T+U) operator()(T const& t, U const& u)
    {
        return t+u;
    }
};
```

尽管标准不支持它，但很多编译器已经包含一个typeof运算符（有时采用一个保留的拼写法

之一：“__typeof”或“__typeof__”），而且C++委员会正在严肃地讨论如何将这个能力添加进标准语言。这个特性是很有用，过去的几年中，已经有好几个仅使用程序库实现的typeof被开发出来了，所有这些最终都依赖于能力较受限制的sizeof[Dew02]。这些程序库实现不完全是自动化的：用户定义的类型必须被手工关联上惟一的数字，通常通过一些traits类特化实现。你可以在本书的配书光碟（Boost预发行版）中找到一个这样的程序库的代码和测试，它是由Arkadiy Vertleyb开发的。

9.11 总结

本章展示的技术就像编程技巧大杂烩，其实它们有一个共同的特点：以威力强大的方式将纯粹的编译期元程序和运行期构造（constructs）连接起来。当然还有其他一些这样的机制潜伏在那里，但我们在这里讨论的这些技术应该为你提供了足够的工具，让你的元程序感受到运行期数据的现实世界。

9.12 练习

9-0. 很多编译器都包含一个“单继承”EBO。确切地说，它们在和一个数据成员相同的地址那里分配一个空基类，但它们永远不会将两个基类分配在相同的地址上。在这些编译器上，我们的storage实现对于F和G都是空的情况下并非最优化的。修复storage，以避免当预处理器中定义了NO_MI_EBO时发生这个缺陷。

9-1. 当F和G是相同的空类时，我们的组合模板将会发生什么？你如何修复这个问题？编写一个测试，它对于一致的空F和G情况会失败，然后修复compose_fg使得测试通过。

9.2. 我们可能无法使用compose_fg2来复合任意的函数对象，但我们可以使用它来组合静态初始化的函数对象（提示：请回顾9.6节开头“可作模板实参传递的”类型列表）。编译一个这样的小程序，如果你看得懂编译器的汇编语言（assembly language）输出，请分析结果代码的效率。

9-3*. 编写一个一般化的迭代器模板，它使用类型擦除（type erasure）来包装一个任意的迭代器类型，并为之提供一个运行期多态接口。该模板应该接收迭代器的value_type作为其第一个参数，且接收iterator_category作为第二个参数（提示1：使用Boost的iterator_facade模板使得编写该迭代器更容易一些。提示2：你可以使用结构选择（structure selection）来控制一个给定的成员函数是否为虚拟的）。

9-4. 修改第9.9节sum重载例子，使它可以加上任意的嵌套容器（例如std::list<std::list<std::vector<int>>>）的最内层算术元素。测试你的修改，确认它可以工作。

9-5. 回顾第3章中的量纲分析代码。不是使用BOOST_STATIC_ASSERT来侦测operator+和operator-内的量纲冲突，而是应用SFINAE来消除这些运算符的重载集中不适当的参数组合。比较在两种情况下当误用operator+和operator-时所获得的错误消息。

第10章　领域特定的嵌入式语言

如果语法糖没什么价值的话，那我们大家就都采用汇编语言编程了。

本章讨论我们认为的对元编程（一般而言）和C++ 元编程（尤其如此）最重要的应用领域：构建领域特定的嵌入式语言（domain-specific embedded languages, DSELs）。

我们今天使用的大多数模板元编程技术发明于实现DSEL的过程中。1995年的某一个时间，C++ 元程序第一次开始用于DSEL的创建，结果是令人印象深刻的。自那以后，人们对元编程的兴趣与日俱增，但是（可能因为每周都有人发明出利用模板的新方式），这种兴奋通常集中于实现技术，结果，我们往往忽视了发明这些实现技术所使用到的设计原则的威力和美感。在这一章中，我们将探究这些原则，并且描绘出方法论背后的大画面。

10.1　一个小型语言

至此你可能会感到好奇，“到底什么是领域特定的语言？”让我们从一个例子开始（稍后我们将讨论什么是“嵌入式（embedded）”）。

考虑在一个文本中搜索带有连字符的单词（例如“domain-specific”）首次出现的位置，如果你曾经用过正则表达式（regular expressions）[⊖]，我们坚信你不会考虑自己编写逐字符搜索的代码。实际上，如果你不是在考虑使用如下的一个正则表达式的话，我们会感到有些奇怪：

```
\w+(-\w+)+
```

如果你不熟悉正则表达式，那上面的东西看起来就像天书，但如果你熟悉的话，你也许会发现它简练且富有表达力。分解如下：

- \w表示“任何可以作为单词一个组成部分的字符”
- +（正闭包，positive closure）意思是“一个或多个重复（one or more repetitions）”
- -仅仅表示它自己，即连字符
- 括号组表达式就像在算术中的一样，因此最后的+更动的是整个子表达式-\w+

因此整个模式匹配任何采用单个连字符分隔的单词字符串。

正则表达式的语法被专门用来允许简短有效的文本模式表示。一旦你学会了它，你的军械库中就多了一种小型武器——实际上它是一种语言，有着自己的符号系统、规则和语义。正则表达式在它们的特定问题领域是如此有效，学习使用它是完全对得起付出的努力的。每当我们打算弃用它而去采用一个特殊的解决之前，我们总会三思而后行。不难想象我们为何会讨论到

⊖ 要想了解关于正则表达式的介绍，你可能需要暂时放下这本书半个小时，拿起关于该主题的一本好书，例如《Mastering Regular Expressions》（第2版，作者为Jeffrey E. F. Friedl）。如果你希望了解一些理论基础，你可以看一看Bernard Moret的大作：《The Theory of Computation》。它还讨论了有限状态机（finite state machines），我们将在下一章讨论这个主题。

这个地方，因为正则表达式是一个领域特定的语言（domain-specific language，DSL）的经典例子。

这里有一些具有区别性的属性，允许我们刻画DSL的特征。当然，首先它得是一门语言。然而可能有点让人惊讶的是，这项属性是很容易满足的——任何东西，只要满足以下特性，就构成了一门形式语言（formal language）：

1. 一个符号系统（或者说一套符号）。

2. 一套定义良好的规则，描述如何使用符号系统来构建形式良好的文法（well-formed compositions）。

3. 所有形式良好的文法的一个定义良好的子集（它们被赋予了具体的含义）。

注意，符号系统甚至可以不是文本形式的。摩尔斯电码（Morse code）和UML都是众所周知的使用图形符号系统的语言。它们俩不仅是有点儿非同寻常、但仍然完全有效的形式语言的例子，而且恰好还是有趣的DSLs。

现在，“领域特定（domain-specific）”部分的语言特征就变得更加有趣了，它们赋予了DSLs第二个与众不同的特性。

可能解释什么是“领域特定”最简单的方式就是“任何非通用的东西”。尽管无可否认该定义使得很容易对语言进行分类（“HTML是一种通用语言吗？不是，因此它是领域特定的”），但这个解释无法表达使得这些小型语言具有如此大吸引力的属性。例如，显然正则表达式语言不能称为是“通用的”，实际上，你可能根本不情愿称之为语言，至少在我们提出关于该术语的定义之前是这样。然而，正则表达式为我们带来了一些超出缺乏熟悉的编程构造（programming constructs）的东西，这些东西使得它值得被称为DSL。

特别是，通过使用正则表达式，我们拿“通用性”和“显著的更高层的抽象和表达力”做交易。专门的符号系统和记号允许我们在一个与心智模型相匹配的抽象层面来表达模式匹配（pattern-matching）。正则表达式的元素，包括字符、重复（repetitions）、可选项、子模式等，都与我们在口头描述一个模式时所使用的概念有着直接的映射。

允许根据接近问题领域的抽象的方式来编写代码，是所有DSLs的特征属性和背后的动机。在最佳场景中，代码中的抽象和那些心智模型中的一致：你只要使用语言的领域特定的记号（notation）写下关于问题自身的声明即可，语言的语义负责生成解决方案。

这听起来好像不大现实，但在实践中它并非像你想象的那样罕见。当FORTRAN编程语言被创建时，似乎一些人认为这意味着编程的终结。关于这门语言，在原始的IBM备忘录[IBM54]中这样写道：

由于FORTRAN应该在实质上消除编码和调试工作，因此与没有FORTRAN这样的系统（语言）的情形相比，前者应该能花费一小半的代价来解决问题。

按今天的标准来看，这个说法的确是事实：FORTRAN在“本质上”消除了编码和调试。因为当时大多数程序员面临的主要是“如何编写正确的循环和子例程调用”层面的问题，在FORTRAN中编程看上去不过是写下对问题的描述而已。显然，高级通用语言的出现已经提高了我们所考虑的何谓“编程”的门槛。

最成功的DSLs通常是申述式（declarative）语言，它为我们提供了用来描述“what”而不是

“how”的符号。正如你进一步要看到的那样，这种声明的特性是DSLs的魅力和能量的重要源泉。

10.2 路漫漫其修远兮

Jon Bentley在其关于DSLs的优秀文章中写道，“程序员每天都在和微型语言（microscopic languages）打交道”[Bent86]。既然你已经知道了它们的基本属性，那就很容易明白在我们的周围存在着许多小型语言（little languages）。

实际上，这方面的例子是如此之多，以至于本书不可能全部讨论它们——我们估计现在日常使用的DSLs有数千种，但是我们可以巡览其中的一些，以便为你展示较细致的观察。

10.2.1 Make工具语言

快速、可靠、可重复地构建软件，对于软件开发的日常实践来说，至关重要。可复用的软件的部署也是很重要的，而且在如今的开放源码时代，最终用户安装文件的制作也愈发重要。多年以来，已经涌现大量的工具来解决这个问题，但它们几乎都是一个极具威力的、构建描述的（build-description）语言Make的变种。作为一名C++程序员，你至少可能已经对Make有一点点熟悉，但是在此我们打算略微回顾一下，焦点放在它的“DSL性（DSL-ness）”方面，并且着眼于你自己的领域特定的语言的设计。

Make的首要领域抽象是围绕三个概念而构建的。

目标（Targets）

通常指需被构建的文件，或者作为构建过程组成部分的输入而被读入的源文件，也包括未关联文件的构建过程“伪”目标命名陈述。

依赖关系（Dependencies）

目标（target）之间的关系，当一个目标（target）不是最新时，允许Make决定需要重新构建该目标。

命令（Commands）

为了构建或更新一个目标（target）而采取的动作，通常为本机系统（native system）的外壳语言（shell language）中的命令。

Make语言的中枢构造（construct）称为规则（rule），在“Makefile”中采用如下的语法进行描述：

```
dependent-target : source-targets
        commands
```

例如，一个用于根据C++源文件构建程序的Makefile可能如下所示：

```
my_program: a.cpp b.cpp c.cpp d.cpp
        c++ -o my_program a.cpp b.cpp c.cpp d.cpp
```

其中“c++”是调用C++编译器的命令。这两行内容表明Make允许对其领域抽象进行简练地表示，包括目标（targets）（my_program和几个.cpp文件）、目标之间的依赖关系（dependency），以及用于根据依赖关系创建依赖目标的命令（command）。

Make的设计者意识到，这样的规则（rules）包含有一些关于文件名字的刻板的重复，因此

加入了对变量（variables）的支持，作为一种辅助能力。使用变量，上面的“程序”可被改写如下：

```
SOURCES = a.cpp b.cpp c.cpp d.cpp
my_program: $(SOURCES)
        c++ -o my_program $(SOURCES)
```

遗憾的是，对于大多数C/C++程序来说，这并不是一个非常现实的例子，大多数C/C++程序在头文件中包含依赖关系。一旦涉及头文件，为了确保最小限度且最快速度的重新构建(rebuild)，构建各个单独的目标文件（object files）并在头文件中表示它们之间的依赖关系，就变得很重要了。这里有一个例子（基于GNU Make手册中的一个例子）：

```
OBJECTS = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(OBJECTS)
        c++ -o edit $(OBJECTS)
main.o : main.cpp defs.h
        c++ -c main.cpp
kbd.o : kbd.cpp defs.h command.h
        c++ -c kbd.cpp
command.o : command.cpp defs.h command.h
        c++ -c command.cpp
display.o : display.cpp defs.h buffer.h
        c++ -c display.cpp
insert.o : insert.cpp defs.h buffer.h
        c++ -c insert.cpp
search.o : search.cpp defs.h buffer.h
        c++ -c search.cpp
files.o : files.cpp defs.h buffer.h command.h
        c++ -c files.cpp
utils.o : utils.cpp defs.h
        c++ -c utils.cpp
```

你可以在命令中再次看到一些用于构建每一个目标文件的重复的样板代码。这可以利用“隐式的模式规则（implicit pattern rules）”加以解决，该规则描述了如何根据一种目标（target）来构建另一个目标（target）：

```
%.o: %.cpp
        c++ -c $(CFLAGS) $< -o $@
```

这个规则使用模式匹配来描述如何从一个所依赖的.cpp文件构建一个.o文件，古怪的符号$<和$@表示匹配的结果。实际上，对这个特别的规则的需求是如此常见，以至于你的Make系统中很可能也有对它的支持。因此前述的Makefile就变成了：

```
OBJECTS = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit : $(OBJECTS)
```

```
        c++ -o edit $(OBJECTS)

main.o : main.cpp defs.h
kbd.o : kbd.cpp defs.h command.h
command.o : command.cpp defs.h command.h
display.o : display.cpp defs.h buffer.h
insert.o : insert.cpp defs.h buffer.h
search.o : search.cpp defs.h buffer.h
files.o : files.cpp defs.h buffer.h command.h
utils.o : utils.cpp defs.h
```

这个回顾已经足够了！探索Make的所有特性很容易就会充满整本书。这里的目的在于向你展示Make开始接近领域特定的语言的一个理想：允许一个问题仅仅通过描述就可以得到解决。在这个例子中，就是通过写下文件的名字和它们的关系。

实际上，形形色色的Make变种的其他大多数特性也都瞄准于更加接近这个理想。例如GNU Make可以自动地发现工作目录中符合条件的源文件，探索它们的头文件依赖关系，并合成规则来构建中间目标文件和最终可执行文件。在一个经典的混合[Veld04]的例子中，GNU Make蕴藏了如此多的特性，以至于它接近了通用语言的威力，虽然用起来很笨拙，但它仍然是领域特定的。

10.2.2 巴科斯-诺尔模式

毕竟这是关于元编程的讨论，接下来我们将介绍元语法（metasyntax）的思想。这其实就是巴科斯-诺尔模式（Backus Naur Form，BNF）所描述的东西：一个用于定义形式语言的语法的小型语言[⊖]。BNF主要的领域抽象称为“上下文无关的语法（context-free grammar）”，它是围绕两个概念进行构建的。

符号（Symbols）

即语法的抽象元素。在C++的语法中，符号包括标识符、一元运算符、字符串字面量、new表达式、语句以及声明等。前三种不能由语法中的其他符号所构成，因此被称为终端符号（terminal symbols）或记号（tokens）。其他的可以从零个或多个符号构建而成，因此被称为非终端符号（nonterminals）。

Productions（或“规则”）

指的是用于联合连续的符号以形成非终端符号的合法模式。例如，在C++中，一个new表达式可以通过结合new关键字（一个token）和一个new-type-id （一个非终端符号）来形成。

Productions通常依照如下语法进行编写：

```
nonterminal -> symbols...
```

其中箭头左边的nonterminal符号可以被任何“匹配右边的symbols序列的”输入序列所匹配。

这里有一个采用BNF编写的用于简单算术表达式的语法，其中的终端符号（terminals）采用粗体字显示，而非终端符号（nonterminals）则采用斜体字显示：

⊖ 实际上，BNF最初被用来指定编程语言Algol-60的语法。

```
expression -> term
expression -> expression + term
expression -> expression - term

term -> factor
term -> term * factor
term -> term / factor

factor -> integer
factor -> group

group -> ( expression )
```

即一个expression被一个term、或一个expression后跟一个+ token和一个term、或一个expression后跟一个- token和一个term所匹配。类似地，一个term被一个factor、或者一个term后跟一个* token和一个factor、或一个term后跟一个/ token和一个factor所匹配，等等。

这个文法不但编码了允许的表达式语法（最终不过是一个或多个以+、－、*、或/分隔的integers），通过将语法元素依照运算符通常的结合率和优先规则进行分组（grouping），还表达了一些重要的语义信息。例如，以下结构：

```
1 + 2 * 3 + 4
```

当根据上述文法进行解析时，可以描述成：

```
[1 + [2 * 3]] + 4
```

换句话说，子表达式（subexpression）2 * 3将被分组进单个term，然后和1相结合，形成一个新的（子）表达式（(sub-) expression）。绝不会将表达式解析为生成如下所示的错误分组：

```
[[1 + 2] * 3] + 4
```

你自己试试看，该文法不允许expression 1 + 2后面跟一个*。BNF对于编码形式语言（formal languages）的语法（syntax）和结构（structure）来说都是非常高效的。

还可以进行一些语言上的精化。例如，习惯上将所有生成一个给定的非终端符号（nonterminal）的productions进行分组，因此有时使用“|”符号分隔右侧不同的选项而无需重复“nonterminal ->”刻板的代码：

```
expression -> term
            | term + expression
            | term - expression
```

扩充的BNF（Extended BNF，EBNF），BNF的一个变种，加入了对用于分组的小括号、Kleene star（0个或多个）以及正闭包（positive closure）（一个或多个）运算符的使用（你可以从重复（repetition）的正则表达式中看出来）。例如，所有用于expression的规则都可被结合进如下的EBNF：

```
expression -> ( term + | term - )* term
```

也就是说，“一个expression被一个有着零个或多个重复的[一个term和一个+ token或者一个

term和一个- token]序列后跟一个term所匹配”。

只需几个简单的步骤，就可将用EBNF编写的文法转换为标准的BNF，因此不管使用哪种符号，基本的表达能力是相同的。这其实是一个强调的重点问题：EBNF可以清晰地阐明所允许的输入，但往往要付出“解析结构变得不那么明显”的代价。

10.2.3 YACC

正如我们在第一章提到的，YACC（Yet Another Compiler Compiler）是一个用于构建解析器、解释器和编译器的工具。YACC是一个翻译器，其输入语言是一个扩充的BNF形式，其输出是一个C/C++程序，该C/C++程序执行指定的解析和解释。在计算机语言中，解释某个已解析的输入过程称为语义评估（semantic evaluation）。YACC通过允许我们将一些数据（语义值(semantic value)）和每一个符号进行关联，以及将一些C/C++代码（语义动作（semantic action））和规则进行关联，来支持语义评估。被包围在大括号内的语义动作，从它的构成符号中计算规则左侧的非终端符号（nonterminal）语义值。一个用于解析和评估算术表达式的完整的YACC程序如下：

```
%{ // 将被插入生成的源文件中的C++代码
  #include <cstdio>
  typedef int YYSTYPE; // 所有语义值（semantic values）的类型

  int yylex();                              // 前置声明
  void yyerror(char const* msg);            // 前置声明
%}
%token INTEGER      /* 声明一个符号性的多字符标记 */
%start lines        /* lines是起始符号（start symbol） */

%% /* 语法规则（grammar rules）和动作（actions）*/
expression : term
           | expression '+' term { $$ = $1 + $3; }
           | expression '-' term { $$ = $1 - $3; }
           ;
term : factor
     | term '*' factor { $$ = $1 * $3; }
     | term '/' factor { $$ = $1 / $3; }
     ;

factor : INTEGER
       | group
       ;

group : '(' expression ')'    { $$ = $2; }
      ;

lines : lines expression
```

```
        {
          std::printf("= %d\n", $2);   // 在每一个expression后
          std::fflush(stdout);         // 打印其值
        }
        '\n'
      | /* 空 */
      ;

%% /* 将被插入生成的源文件中的C++代码 */
#include <cctype>

int yylex() // 执行标记化操作的函数（tokenizer function）
{
  int c;

  // 跳过空白字符（whitespace）
  do { c = std::getchar(); }
  while (c == ' ' || c == '\t' || c == '\r');

  if (c == EOF)
    return 0;
  if (std::isdigit (c))
  {
      std::ungetc(c, stdin);
      std::scanf("%d", &yylval); // 存储语义值（semantic value）
      return INTEGER;
  }
  return c;
}
// 标准错误处理器（error handler）
void yyerror(char const* msg) { std::fprintf(stderr,msg); }

int main() { int yyparse(); return yyparse(); }
```

正如你看到的那样，花括号里的一些C++程序片段并不完全是C++：它们包含一些古怪的$$和$n符号（其中n是一个整数）。当YACC将这些程序片段翻译成C++时，它用规则的左侧非终端符号的语义值的引用来替换$$，$n则用第n个右侧的符号的语义值进行替换。上面的语义动作在生成的C++代码中看上去如下：

```
yym = yylen[yyn];
yyval = yyvsp[1-yym];
switch (yyn)
{
case 1:
{ std::printf("= %d \n", yyvsp[0]); std::fflush(stdout); }
```

```
break;
case 8:
{ yyval = yyvsp[-2] * yyvsp[0]; }
break;
case 9:
{ yyval = yyvsp[-2] / yyvsp[0]; }
break;
case 11:
{ yyval = yyvsp[-2] + yyvsp[0]; }
break;
case 12:
{ yyval = yyvsp[-2] - yyvsp[0]; }
break;
}
yyssp -= yym;
...
```

这段代码只是一个充满类似丑陋难读的玩意儿的源文件片段，实际上，文法的BNF部分是采用大的整数数组（即解析表（parse tables））进行表达的：

```
const short yylhs[] = {
   -1,
    2,    0,    0,    3,    3,    4,    5,    5,    5,    1,
    1,    1,
};
const short yylen[] = {
    2,
    0,    4,    0,    1,    1,    3,    1,    3,    3,    1,
    3,    3,
};
const short yydefred[] = { ... };
const short yydgoto[] = { ... };
const short yysindex[] = { ... };
const short yyrindex[] = { ... };
const short yygindex[] = { ... };
```

你无需理解生成的代码是如何工作的：DSL负责保护我们免于涉及这些丑陋的细节，并允许我们采用高级术语来表达文法。

10.2.4 DSL摘要

至此，有一点应该很清晰了：DSLs可以使代码变得更简练且更易于编写。然而，使用小型语言的好处并非仅仅在于加快编码的速度。尽管有利的编程捷径常常使代码难于理解和维护，但一个领域特定的语言通常具有相反的效果，这要归因于它的高级抽象。让我们设想努力维护YACC为小型表达式解析器产生的低级解析器程序：除非我们有远见卓识去维护含有非常接近YACC程序自身的一些东西的注释，否则我们不得不从解析表反向工程（reverse engineer）BNF，

并使之与语义动作（semantic actions）相匹配。可维护性效果变得越极端，语言就越接近领域抽象。当我们接近理想的语言时，通常一眼就能看出一个程序是否能解决它被设计用来解决的问题。

稍息片刻。设想你正在为埃科美洁燃核反应堆（Acme Clean-Burning Nuclear Fusion Reactor）编写控制软件。以下公式来自一篇科学论文，描述如何从三个传感器电压等级计算出一个温度读数：

$$T = (a+3.1)(b+4.63)(c+2\text{x}10^8)$$

你需要将该计算实现为反应堆故障保护机制的一部分。理所当然，使用运算符记号（C++算术领域特定的子语言），你可以写：

```
T = ( a + 3.1 ) * ( b + 4.63 ) * ( c + 2E8 );
```

现在，将以上代码与假如C++不支持运算符时你所能编写的代码进行比较：

```
T = mul(mul(add(a, 3.1), add(b, 4.63)), add(c, 2E8));
```

你更信任哪一种记号可以帮助避免发生灾难？哪一种更容易和论文上的公式对应起来？我们认为答案是明摆着的。快速地瞥一眼使用运算符记号的代码，我们就可以发现它正确地实现了公式。这是一个真实的例子——很多人声称看过、甚至自己写过这样的代码，但是在现实中却很少碰到——自文档化的代码（self-documenting code）。

算术记号演化已进入今天我们使用的标准，因为它通过最低限度的额外语法清晰地表达了计算的意图和结构。由于算术对于编程的基础来说是如此重要，因此大多数计算机语言都提供了对“操作原始类型”的标准算术记号的内建支持。有不少语言还提供了对运算符重载的支持，允许用户以同样接近于原始领域抽象的语言，来表达对向量、矩阵这样的用户自定义类型的计算。

这样，由于系统了解问题领域，所以它可以在与程序员使用的同样的概念层生成错误报告。例如，YACC侦测并报告文法上的歧义，根据文法productions来描述它们，而不是转储（dumping）其解析表的细节。拥有领域知识甚至使得某些非同寻常的优化成为可能，等我们在本章后面讨论Blitz++程序库的时候，你就会看到这一点。

在继续前进之前，我们对DSLs做最后一次观察。可能并非巧合：Make和BNF都有一个“规则（rule）”的概念。这是因为DSLs往往是申述式（declarative）而非命令式（imperative）的语言。一个纯粹的申述式程序仅仅提及实体（entities）（例如符号、目标）以及它们之间的关系（例如解析规则、依赖性），而程序的处理或算法部分完全被编码于解释该语言的程序中。看待申述式程序的方式之一是，将其看做一种“将被语言的概念执行引擎所使用”的常性数据结构（immutable data structure）。

10.3 DSL

最初的Make程序只包含极弱的编程语言，在一开始应用时仅能满足基本的软件构建任务需求。此后，Make的变种对语言进行了扩充，但它们均未摆脱最初版本的束缚，没有一个达到可以称为通用语言的表达力。典型的大规模系统使用Make分派一些处理工作给Perl脚本或其他定

制的附加软件，这往往导致系统难于理解和修改。

另一方面，YACC的设计者认识到提供一个用于表达语义动作（semantic actions）的威力强大的语言的挑战，总比把工作留给其他工具做要好。从某种意义上说，YACC的输入语言实际上包含“任何一种用来处理其输出的语言”的所有能力。你正在编写一个编译器并且你需要一个符号表吗？好得很，将#include <map>添加到开头的%{...%}语句块即可，这样你就可以在语义动作中愉快地使用STL了。你正在解析XML并且希望将它发送给一个SAX（Simple API for XML）解释器吗？没问题，因为YACC输入语言嵌入了C/C++。

然而，YACC的方式并非没有缺点。首先，实现和维护一个新的“编译器”需要付出代价：在这个例子中，即是YACC程序自身。并且，一个尚不了解YACC的C++程序员不得不学习新语言的规则。在YACC的情况下，主要是语法，但通常可能有各种各样的新规则，比如说：不同的绑定、作用域以及名字查找等。如果你想看看这可能会变得多么糟糕，不妨考虑C++中所有不同种类的规则吧。如果不在工具开发方面付出额外的投资，就没有现成的用于测试或调试采用DSL编写的程序的设施（和DSL位于同一个抽象层），因此问题通常不得不在目标语言的低层（即在机器产生的代码中）进行调查研究。

最后，传统的DSLs对语言的互操作性强加了严重的约束。例如，YACC（几乎）不访问它所处理的C/C++程序片断的结构。它只是简单地查找未被引用的（nonquoted）$符号（在真正的C++程序中，这种符号是非法的）并且用相应的C++对象的名字来替换它们，这纯粹是一个文本替换。这种简单的途径对于YACC而言工作地很好，因为它不需要推导诸如C++类型、值或控制流程之类的东西的能力。在一个通用语言构造自身是领域抽象的一个组成部分的DSL中，平凡的文本操纵通常不符合要求。

这些互操作问题也阻止了DSLs彼此之间的协作。设想你对Make的语法和有限的内建语言特性感到不爽，你希望编写一个新的高级软件建置语言。看上去很自然就使用YACC来表达新语言的文法。Make在表达和解释低级构建系统概念（low-level build system concepts）（包括目标（targets）、依赖关系（dependencies）以及构建命令（build commands）等）方面仍然是相当有用的，因此，使用Make来表达语言的语义同样自然。然而，YACC动作（actions）是用C或C++写的。我们能够做的最好的事情就是编写“编写Makefiles”的C++程序片断，从而向该过程添加又一个编译阶段：首先YACC代码被编译成C++，然后生成的C++代码被编译并被执行以便生成Makefile，最后调用Make对其进行解释。现在看上去你需要我们的高级软件建置语言来整合在构建和使用语言自身过程中所涉及的不同阶段了。

解决所有这些缺点的方式之一是将YACC的途径里外颠倒：不是将通用语言嵌入到DSL中，而是将领域特定的语言嵌入到通用的宿主语言中。对你来说，在C++中做这一点看上去可能有点奇怪，因为你也许意识到C++不允许我们添加任意的语法扩充。那么我们如何在C++中嵌入另一种语言呢？的确，我们可以使用C++来编写一个解释器，并且在运行期解释程序，但是这种方式不会解决我们已经暗示的互操作问题。

呵呵，事情并没有那么神秘，希望你能原谅我们使它看上去有点神秘兮兮的。毕竟，每一个针对一个定义良好的特定领域的“传统的”程序库，例如几何、图形或矩阵乘法方面的程序库，都可被认为是一种小型语言：其接口定义语法，其实现定义语义。当然，事情不只这么多，但这就是基

本原理。我们似乎已经听到你在嘟囔着，“如果只要程序库就可以解决我们的问题，那我们为什么用一整章来讨论YACC和Make？”呃，事情不仅仅是关于程序库的问题。考虑如下对“Domain-Specific Languages for Software Engineering”（作者Ian Heering和Marjan Mernick [Heer02]）的引述：

与应用程序库结合使用，任何通用的编程语言都可担当DSL角色，那么为何首先开发DSLs呢？原因很简单：它们可以以更好的方式提供“领域特定能力”：

- 适当的或已建立的领域特定的记号，通常超出通用编程语言提供的有限制的、用户可自定义的运算符记号。一个DSL从一开始提供领域特定的记号。这种记号的重要性无论估计多高都不为过，因为它们和最终用户编程的适宜性相关，更一般地，与程序员生产力的提高有关。
- 适当的领域特定的构造和抽象，无法总是以简单的方式映射到可放入程序库中的函数或对象上。这意味着，一个使用应用程序库的通用语言只能间接地表达这些构造。再一次，DSL可以从一开始就加入领域特定的构造。

简而言之：

> **定义**
>
> 一个真正的DSL合并领域特定的记号、构造和抽象作为基础的设计考虑。一个领域特定的嵌入式语言（domain-specific embedded language，DSEL）不过是一个满足同样准则的程序库而已。

这种里外颠倒的方法解决了像YACC这样的翻译器和像Make这样的解释器的很多问题。设计、实现和维护DSL自身的工作被减少到生产一个程序库。然而，实现代价并非是最重要的考虑因素，因为DSLs和传统的程序库的实现都是一种长期的投资，这种投资是我们希望通过多次使用这些代码而得到回报的。真正的回报在于完全消除通常和跨语言边界有关的代价。

DSEL的核心语言规则由宿主语言规定的，因此，与学习独立的对应物相比，一门嵌入式语言的学习曲线是相当平坦的。程序员熟悉的所有用于编辑、测试和调试宿主语言的工具，都能用在DSEL中。通过定义，宿主语言编译器自身也被使用，因此额外的翻译阶段就被消除掉了，从而大幅降低了软件构建的复杂性。最后，尽管程序库互操作性在很多软件系统中呈现出偶然的问题，然而当和组成普通DSLs的问题相比时，整合多个DSELs需要花费的力气还是少得多。一个程序员可以在通用的宿主语言以及任何其他一些领域特定的嵌入式语言之间无缝切换，无需思维停顿。

10.4 C++用作宿主语言

对于我们来说，幸运的是，C++是一种适合实现DSELs的语言。它的多范型血统使它提供了许多工具，可以用来构建“结合了语法上的表达力和运行时效率”的程序库。尤其是C++提供了：

- 一个静态类型系统。
- 获得近于零抽象惩罚的能力[⊖]。

⊖ 对于当前的编译器来说，避免抽象所导致的效率惩罚有时需要程序员付出极大的注意力。Todd Veldhuizen讨论了一种称为“有保证的优化（guaranteed optimization）”的技术，利用它，你可以随意应用形形色色的抽象，而绝不会影响性能[Veld04]。

- 极具威力的优化器。
- 一个模板系统，可用于：
  - 生成新的类型和函数。
  - 在编译期执行任意的计算。
  - 剖析现有的程序组件（例如使用Boost Type Traits程序库中的类型归类元函数（type categorization metafunctions））。
- 一个宏预处理器（macro preprocessor），提供和模板所提供的代码生成能力正交的（文本的）代码生成能力（参见附录A）。
- 一套丰富的内建符号性的运算符（多达48种！），其中许多具有几种可能的写法，它们可以被重载，且重载的语义几乎没有任何限制。

表10.1列出了C++中运算符重载所提供的语法构造。表中具有多行的条目展示了同一个标记不大为人所知的替代拼写法。

表10.1 C++中可重载的运算符语法

+a	*−a*	*a + b*	*a−b*
++a	*−−a*	*a++*	*a−−*
*a * b*	*a / b*	*a % b*	*a, b*
a & b	*a | b*	*a ^ b*	*~a*
a bitand *b*	*a* bitor *b*	*a* ??' *b*	??−*a*
		a xor *b*	compl *a*
a & & b	*a | | b*	*a >> b*	*a << b*
a and *b*	*a* or *b*		
a > b	*a < b*	*a >= b*	*a <= b*
a == b	*a != b*	*! a*	*a = b*
	a not_eq *b*	not *a*	
a += b	*a −= b*	*a *= b*	*a /= b*
a %= b	*a &= b*	*a |= b*	*a ^= b*
	a and_eq *b*	*a* or_eq *b*	*a* xor_eq *b*
a >>= b	*a <<= b*	**a*	*&a*
*a−>*name	*a->**name	*a* [*b*]	*a* (*arguments*)
		a?? (*b*??)	
		a<:b:>	
new *ctor-expr*	delete *a*		

C++中的这些特性的独特结合，使得这样一个范畴的领域特定的程序库成为可能：它们既高效，语法又接近于从头构建起的语言⊖。而且，这些程序库可以使用纯粹的C++编写，这赋予它们与独立的DSLs相比以很大的优势，后者需要专门的编译器、编辑器以及其他一些工具。在接下来的几个小节中，我们将讨论一些例子，每一个例子都集中于DSL的设计而不是实现。

⊖ Haskell是另一门在构建DSELs的平台时值得一提的语言。Haskell对于DSEL构建的能力在相当大的程度上与C++有重叠，甚至在某些领域走的更远。例如，Haskell程序员可以定义新的运算符，这些新的运算符可以扩充内建语言的语法。然而，Haskell编译模型的局限性使其往往无法达到最佳性能。

命名空间名字

到目前为止，我们已经相当遵守纪律，总是对来自命名空间boost中的名字加上boost::前缀，对来自boost::mpl中的名字加上mpl::前缀，以避免混淆。但是，在这一章中，我们强调的是DSL语法的“甜蜜”的一面，因此我们打算忽略程序库“元素”的命名空间名字，并且相信你可以猜测到这些名字来自何处。

10.5 Blitz++和表达式模板

Blitz++ [Veld95a]是一个高性能数组算术程序库，开拓了本书中用到的很多技术和思想，它对C++ 元编程世界的影响无论评价多高都不过分。它是第一个使用显式元编程（explicit metaprogramming）的C++程序库[⊖]，并且是第一个实现了领域特定的嵌入式语言。我们不可能论及Blitz++的所有方面，因此我们将考察其最重要的创新：表达式模板（expression templates）[Veld95b]。

10.5.1 问题

如果我们将Blitz++所解决的问题归结为一句话，那就是：数组算术的幼稚实现，对于任何有意义的计算来说，是可怕的低效。为了弄明白我们所表达的意思，看看以下这条无趣的语句：

```
x = a + b + c;
```

其中x、a、b和c都是二维数组。数组加法运算符的规范的实现是：

```
Array operator+(Array const& a, Array const& b)
{
    std::size_t const n = a.size();
    Array result;

    for (std::size_t row = 0; row != n; ++row)
        for (std::size_t col = 0; col != n; ++col)
            result[row][col] = a[row][col] + b[row][col];

    return result;
}
```

为了使用这个运算符来计算表达式a + b + c，我们首先要计算a + b，这将会产生一个临时数组（称之为t），然后我们再计算t + c来生成最终的结果。

这里的问题在于那个临时对象t。执行这个计算的高效的方式是一次头计算三个数组，并将其和放入结果（result）中：

⊖ “显式元编程（explicit metaprogramming）”指的是将模板实例化看做一等编译期程序。显式元编程不仅仅是指大多数泛型编程所需要的平凡的类型操纵，例如通过std::iterator_traits访问一个迭代器的value_type。尽管从技术上说，它可以看做成一个元函数调用，但大多数泛型程序员并不这么认为。他们认为是因为代码的关系以及一些别的原因定义了元编程。

```
for (std::size_t row = 0; row != n; ++row)
   for (std::size_t col = 0; col != n; ++col)
      result[row][col] = a[row][col] + b[row][col] + c[row][col];
```

临时对象不但因其元素存储耗费一次额外的动态内存配置，而且导致CPU对该存储区进行两次完全的遍历：一次是写结果a + b，一次是为t + c读取输入。任何做过高性能数值计算的人都知道，这两次遍历是真正的杀手，因为它们会破坏缓存（cache）的局部性（locality）。如果所有四个具名的数组占满了缓存（cache），那么引入t将会使这四个数组之一被置换出去。

这里的问题在于上面的operator+的实现太贪婪：它企图一有可能就去计算a + b，而不是等到整个表达式结束（包括加上c）。

10.5.2 表达式模板

在表达式的解析树（parse tree）中，评估始于叶子并向上推进至根。这里需要的是某种方式——将计算延迟至直到程序库获得表达式的所有部分，换句话说，直到赋值运算符执行时。Blitz++采用的策略是为整个表达式构建编译器解析树的一个复制品，允许它自上到下管理计算（参见图10.1）。

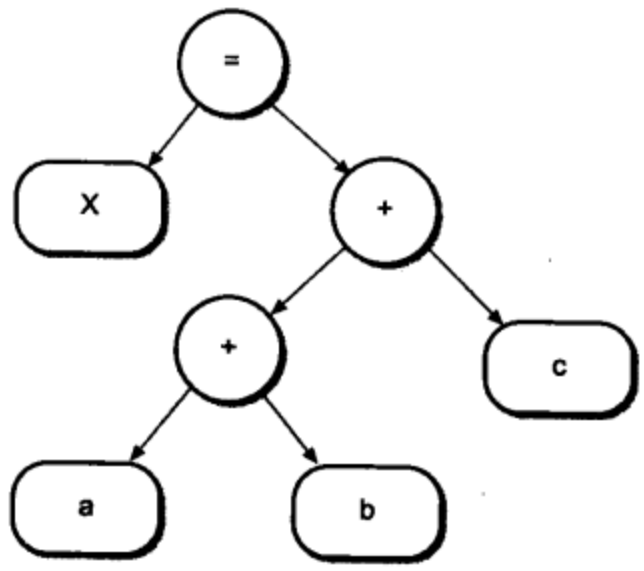


图10.1 x = a + b + c的解析树

然而，这不能是随随便便的一个普通的解析树：因为数组表达式可能涉及其他操作，比如乘法，这些操作需要它们自己的评估策略，而且，由于表达式可能是任意大并且是嵌套的，因此采用节点和指针构建的解析树就不得不在运行期通过Blitz++评估引擎进行遍历，以便探查其结构，这会对性能造成限制。此外，Blitz++不得不使用某种运行期分派机制来处理不同的操作类型的组合，这再一次限制了性能。

Blitz++的实际做法是采用表达式模板（expression templates）来构建编译期解析树。其工作原理可以简单地归结为：不是返回一个新计算出来的Array，运算符仅仅将实参的引用（references）打包在一个Expression实体中，并贴以操作（operation）标签：

```
// 操作标签（operation tags）
struct plus; struct minus;

// 表达式树节点
```

```
template <class L, class OpTag, class R>
struct Expression
{
    Expression(L const& l, R const& r)
      : l(l), r(r) {}

    float operator[](unsigned index) const;

    L const& l;
    R const& r;
};

// 加法运算符
template <class L, class R>
Expression<L,plus,R> operator+(L const& l, R const& r)
{
    return Expression<L,plus,R>(l, r);
}
```

注意，当我们写a + b时，仍然具有足够的信息执行计算，它被编码在类型Expression<Array, plus,Array>中，并且数据可通过表达式所存储的引用（references）进行访问。当写a + b + c时，我们得到一个如下类型的结果：

```
Expression<Expression<Array,plus,Array>,plus,Array>
```

并且数据仍然可通过嵌套的引用进行访问。表达式类模板的有趣之处在于，就像Array一样，它支持通过operator[]进行索引操作。且慢！我们刚才是否告诉过你operator+什么都不计算，并且Expression仅仅存储其参数的引用？对吧。如果操作的结果没有存储在Expression中，它就必须通过operator[]进行缓式计算。

为了弄明白这是怎么工作的，让我们检查如下简化了的一维float型数组实现。首先，为了将逐元素计算的算术逻辑和操作标签相关联，我们嵌入一些静态成员函数：

```
// 操作标签（operation tags）实现逐元素计算的算术
struct plus
{
  static float apply(float a, float b)
    { return a + b; }
};

struct minus
{
  static float apply(float a, float b)
    { return a - b; }
};
```

接下来，我们给Expression设计一个索引运算符，它调用其标签（tag）的apply函数来计算

适当的元素值：

```
// 表达式树节点
template <class L, class OpTag, class R>
struct Expression
{
    Expression(L const& l, R const& r)
      : l(l), r(r) {}

    float operator[](unsigned index) const
    {
      return OpTag::apply(l[index], r[index]);
    }
    L const& l;
    R const& r;
};
```

这看起来太简单，不是吗？令人惊讶的是，我们现在获得了完整的缓式表达式评估（lazy expression evaluation）功能。为了弄清它是如何运作的，让我们探察(a + b)[1]的评估。由于a + b的类型是Expression<Array,plus,Array>，因此：

```
(a + b)[1]
  == plus::apply(a[1], b[1])
  == a[1] + b[1]
```

现在，考虑采用贪婪策略对同样的表达式进行评估会发生些什么。仅仅为了获取一个元素，我们不得不计算一个临时数组（a+b），效率上反差之大，没有比这更惊人的了。

自然，(a + b + c)[1]也在不产生任何临时数组的情况下被计算出来：

```
(a + b + c)[1]
  == ((a + b) + c)[1]
  == plus::apply((a + b)[1], c[1])
  == plus::apply(plus::apply(a[1], b[1]), c[1])
  == plus::apply(a[1] + b[1], c[1])
  == (a[1] + b[1]) + c[1]
```

现在剩下的就是去实现Array的赋值运算符了。由于我们可以访问结果Expression的任何单个元素，而不必创建一个临时的Array，因此可以通过访问表达式中的每一个元素来计算整个结果：

```
template <class Expr>
Array& Array::operator=(Expr const& x)
{
    for (unsigned i = 0; i < this->size(); ++i)
       (*this)[i] = x[i];
    return *this;
}
```

就这么多！当然了，数组算术远不只是加法和减法，Blitz++必须要考虑这个简单例子所没有

处理的所有事情，从乘法到“铺展（tiling）”大型数组操作等，以便使它们待在缓存（cache）之内。然而，延迟表达式评估的基本技术，是允许程序库近乎最有效率地完成所有任务的工具[⊖]。

作为一个DSL，这一部分Blitz++在其平滑性方面具有欺骗性：其语法看上去和你幼稚的实现里完全一样，但是，你应该能看到在语法的背后，活动着一个“针对Blitz++领域调节过”的高度专门化的评估引擎。

中间结果

表达式模板的缺点之一是它们倾向于鼓励编写复杂的大型表达式，因为评估被延迟到了赋值运算符被调用时。如果程序员希望复用某个（些）中间结果，而早期又没有评估它，她就被迫声明一个类似如下（甚至更糟）的复杂类型：

```
Expression<
    Expression<Array,plus,Array>
  , plus
  , Expression<Array,minus,Array>
> intermediate = a + b + (c - d);
```

注意，这个类型不但精确且累赘地反映了计算的结构（因此当发生形式化的改变(formula changes）时需要对其进行维护），而且很打击人的信心。这个问题长期以来一直困扰着C++ DSELs。通常的迂回解决方式是使用类型擦除（见第9章）来“捕捉”表达式，但在这种情况下，要付出动态分派的代价。

在这方面最近已有很多讨论（Bjarne Stroustrup本人就是先锋），即关于复用已淡出的auto关键字以便在形形色色的声明中进行类型推导，从而上面的代码可以改写为：

```
auto intermediate = a + b + (c - d);
```

这个语言特性可以为C++ DSEL的作者和用户带来巨大的好处。

10.5.3 更多的Blitz++魔法

为了避免你感觉很难在我们已经讨论的东西里看见领域特定的语言，这里再多给出几个Blitz++语法创新的例子，我们认为你会发现它们更让人震撼。

数组初始化

由于Blitz++ Arrays并非C++标准称作的集合（参见标准8.5.1节），因此我们不能使用在大括号内列出初始化器的便利语法（就像可以对普通的内建数组所做的那样）。Blitz++重载了逗号运算符，从而使得类似的语法成为可能：

```
Array<float,2> A(3,3);
A = 1, 2, 3,
    4, 5, 6,
    7, 8, 9;
```

⊖ 如果你认为我们示范了一种滥用C++运算符重载的方式，是的，我们认罪！实际上，在本章中我们将花费大量的篇幅考察“滥用”运算符的创造性的方式。我们希望到本章结束时，你会将这些技术看成是合法的、理由充分的编程范型。

子数组（SubArray）语法

Blitz++具有一个Range类，它封装了一个索引序列。当一个Array使用一个Range进行索引时，就会产生一个惰性的SubArray视图而无需复制任何元素[⊖]：

```
// 将A的前两行和列加到B上
B += A(Range(0,2), Range(0,2))
```

Blitz++的Range对象令人兴奋的东西是，通过使用Math.h++程序库[KV89]开创的技术，你还可以对它们直接执行算术运算，结果产生极像多重嵌套的循环体的表达式。下面的例子摘自Blitz++在线手册：

```
// 一个三维模板（用于求解偏微分方程）
Range I(1,6), J(1,6), K(1,6);
B = (A(I,J,K) + A(I+1,J,K) + A(I-1,J,K)
 + A(I,J+1,K) + A(I,J-1,K) + A(I,J,K+1) + A(I,J,K-1)) / 7.0;
```

这种记号上的简化已被证明不仅仅是语法糖。类似的技术已经被用来减少对“复杂张量等式”的评估，从难以理解、易犯错误的代码（类似于FORTRAN（Scott Haney称之为“C++tran”））到类似于原始理论中的等式的单命令行程序（one-liners）[Land01]。使用DSEL，那些使用C++tran几乎不可能正确完成的项目，突然变得易于驾驭。

10.6 通用DSEL

DSEL最美好的特性之一是我们可以将它应用于通用编程惯用法领域中。换句话说，一个DSEL可以发挥一种通用宿主语言的扩充的作用。尽管当我们讨论同一个程序库时，看上去使用术语“通用”和“领域特定”有点矛盾，但是，当你考虑该领域为DSEL激活的具体编程惯用法时就很有意义了。

10.6.1 具名参数

具名参数（Named parameters）是很多语言都具有的一个特性，允许参数按名字而不是按位置传递。我们很希望看到它得到C++的直接支持。例如，在一个假想的支持具名参数的C++语言中，给定如下声明：

```
void f(int score = 0, char const* name = "x", float slew = .1);
```

我们可以这样调用f：

```
f(slew = .799, name = "z");
```

可以注意到，现在每一个实际参数的角色在调用点是完全清晰的，默认值可以用于任何参数，无论其位置在哪儿，另外也可用在其他正被传入的参数上。类似的原理当然也适应于模板参数。正如你可以想象的那样，具名参数确实在接收若干独立参数（每一个都具有非平凡的默认值）的接口中得到了回报。在Boost Graph库的算法中，你可以找到很多这样的函数。

⊖ 注意，就像大多数数组程序包（packages）一样，Blitz++使用operator()而不是operator[]来支持索引操作，因为operator()允许多个参数，而operator[]则不然。

Graph程序库的最初具名参数DSL使用一种称作“成员函数链”的技术，将（多个）参数值聚合进单个函数参数，本质上形成一个关于标签值的tuple。对于我们的例子而言，其用法看上去如下：

```
f(slew(.799).name("z"));
```

这里，表达式slew(.799)将构建一个named_params<slew_tag, double, nil_t>类的实例，空类nil_t作为它的惟一基类，并且包含值.799作为单个数据成员。这样，其name成员函数将以“z”进行调用，生成以下类型的一个实例：

```
named_params<
    name_tag, char const [2]       // .name("z")
  , named_params<
        slew_tag, double const     // slew(.799)
      , nil_t
    >
>
```

它拥有一份刚刚描述过的实例副本作为其惟一的基类，并包含一个“z”的引用作为其惟一的数据成员。我们可以深入探究每一个标签值是如何从这样的结构中抽取出来的，但在本书这里我们确信你的大脑已经开始思考如何解决这个问题了，因此，我们将它留作一个练习。让我们将精力放在DSL语法的选择以及使其工作所要满足的要求之上。

如果你思虑片刻，你会发现对于每一个参数名字，我们不但需要一个顶层的函数（为了生成一个链中初始的named_params实体），而且named_params还必须为我们希望跟踪的每一个参数名字包含一个成员函数。毕竟，我们也许是这么写的：

```
f(slew(.799).score(55));
```

由于当有很多可选参数时，具名参数接口具有很大的回报，并且由于在一个给定程序库中的不同函数所使用的参数名字之间可能存在一些交叠情况，因此可能会在设计中存在大量的耦合而告终。在使用具名参数接口的程序库中，将会存在一个用于所有函数的中央named_params定义。这样一来，向一个声明在某个头文件里的函数中添加一个新的参数名字，将意味着要回去修改named_params的定义，这继而又会导致每一个使用了我们的具名参数接口的翻译单元被重新编译。

在写作这本书时，我们重新考虑了用于具名函数参数支持的接口。在进行了一些试验后，我们发现，通过利用带有重载的赋值运算符的关键字对象（keyword objects），有可能提供理想的语法：

```
f(slew = .799, name = "z");
```

这种语法不但对于用户来说是良好的，而且对于包含f的程序库的作者来说，添加一个新的参数名字也很容易，而且不会导致任何耦合。在这里，我们不打算深究该具名参数程序库的实现细节。它并不复杂，我们建议你作为一个练习尝试实现一下。

在继续前进之前，我们还应该说一下，有可能为具名类模板参数引入类似的支持[AS01a,

AS01b]，尽管我们不知道如何创建这样优雅的语法。我们能够提出的最佳用法具有如下的形式：

```
some_class_template<
    slew_type_is<float>        // slew_type = float
  , name_type_is<char const*> // name_type = char const*
>
```

也许你可以探索一些我们尚未虑及的改进。

10.6.2 构建匿名函数

关于另一个“基于程序库的语言扩充”例子，让我们考虑为STL算法构建函数对象的问题。在第6章中，我们简短地查看了运行期lambda表达式。很多计算机语言已经加入了用于在运行中生成函数对象的特性，这一点是C++所缺乏的，并常常作为它的一个弱点而被引证。到写作本书时为止，已经有不下于4个主要针对函数对象构建的DSL实现。

Boost Bind程序库

其中最简单的一个是Boost Bind程序库[Dimov02]，它局限于三个特性，按照你的MPL lambda表达式的经验，其中有两个你应该很熟悉了。为了理解这个类似物，你需要知道，就像MPL的占位符类型（placeholder types）可以作为模板实参传递一样，Bind程序库具有可以作为函数实参传递的占位符对象（placeholder objects）。

Boost.Bind的第一个特性是偏函数（对象）应用（partial function (object) application），即将实参值绑定到函数（对象），生成一个具有较少参数的新函数对象。例如，为了生成一个将“hello,”赋予一个string的函数对象，我们可以这么写：

```
bind(std::plus<std::string>(), "hello, ", _1)
```

可以这样调用结果函数对象：

```
std:: cout << bind(                          // 打印“hello, world”
                  std::plus<std::string>()
                , "hello, ", _1
              )("world");
```

注意，在现实代码中不大看到外层函数实参（“world”）紧附在bind调用处。在现实代码中，我们通常将调用bind的结果传递给一些将继续调用其多次的算法。

Boost.Bind的第二个特性是函数复合（function composition）。例如，下面的表达式产生一个计算y = x(x- 0.5)的函数对象：

```
bind(
    std::multiplies<float>()
  , _1
  , bind(std::minus<float>(), _1, 0.5))
```

对于我们而言，bind的运作方式是如此自然，以至于我们很难想出替代的方案：如果内层bind表达式没有得到程序库的特别对待，那么它产生的函数对象将作为std::multiplies<float>实例的第二个参数进行传递，从而导致一个错误。

最后，Boost.Bind允许我们采用普通函数调用语法来调用成员函数。基本的思想在于：成员

函数可看做接收一个初始类实参（class argument）的自由函数，这在Dylan之类的语言中得到了支持，但再一次请注意，原生的C++并不支持这一点。然而，这并非仅仅是美学上的考虑：对于需要同时与自由函数和成员函数协作的泛型代码来说，调用语法上的不一致可能会变成一个严重的问题。

使用Bind程序库，我们可以将一个声明为

```
struct X { float foo(float) const; } obj;
```

的X::foo的成员函数转换为一个函数对象，并按如下方式进行调用：

```
bind(&X::foo, _1, _2)(obj, pi)
```

最流行的使用bind的方式之一是将一个成员函数部分地应用到一个类实例。例如，以下代码对[first, last)中的每一个元素x调用v.visit(x)：

```
std::for_each(first, last, bind(&Visitor::visit, v, _1));
```

这个对偏应用（partial application）的有限制的使用在基于事件的软件中是如此重要，以至于Borland实现了一个C++语言扩充closures（闭包），以便在他们的编译器中为之提供直接的支持。

在继续前进之前，让我们将上面使用的bind表达式的语法和我们在使用STL绑定器（binders）和复合器（composers）[⊖]时的语法进行简单地比较一下：

```
// 偏应用（partial application）
bind1st(std::plus<std::string>(), "hello, ")

// 函数复合（function composition）
compose2(
    std::multiplies<float>()
  , bind2nd(std::minus<float>(), 0.5)
  , identity<float>())

// 采用函数调用语法调用一个成员函数
mem_fun_ref(&X::foo)(obj, pi)

std::for_each(
    first
  , last
  , bind1st(mem_fun_ref(&Visitor::visit), v));
```

通过比较，我们认为有很好的论据表明，即便Boost.Bind提供的少量的语法糖，也已开始看上去像一个领域特定的语言了。

Boost Lambda程序库

Boost Lambda程序库（作者为Jaakko Järvi和Gary Powell），是Boost.Bind和MPL编译期lambda表达式设计的最初灵感的来源。Lambda程序库以甜美的语法扩充了Boost Bind的基础设

⊖ compose1、compose2以及identity包含在最初的STL设计中，但从未进入C++标准程序库。你在SGI STL、STLPort以及其他标准程序库实现中仍然可以找到以扩充形式存在的实现品。

施，使我们已经讨论过的一些例子变得更明晰。例如：

```
"hello, " + _1    // x -> "hello, " + x
_1 * (_1 - 0.5)   // x -> x * (x - 0.5)
```

这段代码有意思的地方在于operator*不表示乘法，operator+也不表示加法甚至是连接！相反，它们是用于构造器对象（可供以后调用）的运算符。"hello, " + _1的结果是一个函数对象，它接收一个参数（称之为x），并返回结果"hello, " + x。如果这东西听起来很熟悉，很好，在运行中构建的函数对象是Blitz++首先引入的表达式模板（expression templates）惯用法的又一个例子。

Lambda程序库的目标要比Boost.Bind更加雄心勃勃。即使你发现很难将Boost.Bind的语法看成一个DSL，我们还是认为Boost.Lambda语法自身很明显是一个小型语言。它的特性已经超出了对运算符的支持，它还实现了控制结构甚至异常处理！这里给出一些例子。

1. 将一个二维数组中的每一个元素值减半。

```
float a[5][10];
int i;
std::for_each(a, a+5,
  for_loop(var(i)=0, var(i)<10, ++var(i),
     _1[var(i)] /= 2
  )
);
```

2. 打印一个序列，采用句点替换掉偶数位元素。

```
std::for_each(a.begin(), a.end(),
    if_then_else(_1 % 2 != 0,
         std::cout << _1
       , std::cout << constant('.')
    )
);
```

3. 根据v中的每一个元素n，分别打印出对应的“zero”、“one”或“other: n”。

```
std::for_each(v.begin(), v.end(),
  (
    switch_statement(
      _1,
      case_statement<0>(std::cout << constant("zero")),
      case_statement<1>(std::cout << constant("one")),
      default_statement(std::cout << constant("other: ") << _1)
    ),
    std::cout << constant("\n")
  )
);
```

在上面的例子中，var和constant将其实参包装在一个特殊的类模板中，从而阻止它被贪婪地评估。例如，如果我们在刚才的例子中写std::cout << "\n"，它将会被评估一次（在for_each调用

的外部)。C++本来就是这样工作的。然而，constant("\n")的结果是一个返回"\n"的无参函数对象(nullary function object)。标准程序库没有为T（该函数对象的类型）提供一个流插入器(operator<<(ostream&, T))，但是Lambda程序库提供了一个重载的operator<<来处理T。不是执行流插入，Lambda程序库的operator<<只产生另一个无参函数对象，当它被调用时评估std::cout << "\n"。

需要使用var和constant以及使用for_loop这样的函数来代替C++内建的for，是对C++语言的局限性强加在我们头上的折中。尽管如此，Boost Lambda的表达力，以及它所构建的函数对象通常像手工编码的函数一样高效的事实，还是给人留下了深刻的印象。

Phoenix程序库

永不满足，C++程序库设计者继续寻求更有表达力的方式进行编程。在前进到其他领域之前，我们希望略微谈谈另两个函数式编程程序库的创新。第一个是Phoenix，它是作为Boost.Spirit解析器框架[Guz04]的一部分开发出来的，本章稍后会对Boost.Spirit进行讨论。除了加入了一些价值不菲的新功能外，Phoenix的作者为Boost.Lambda所支持的一些相同的控制结构发明了新的语法。例如，在Phoenix中，上面的if_then_else例子可以如下方式进行编写（注意在Phoenix程序库中，占位符被命名为“arg1”、“arg2”等)：

```
for_each(a.begin(), a.end(),
         if_(arg1 % 2 != 0)
         [
            std::cout << arg1
         ]
         .else_
         [
            std::cout << val('.')
         ]
);
```

Boost Lambda程序库的作者发现这种语法颇具吸引力，因此他们将其吸收进来作为if_then_else的一个候选品。正如你看到的那样，在这些程序库的设计中，存在着大量的“异花授粉”。

FC++程序库

FC++ [MS00b]（Functional C++的缩写）使得C++程序员可以使用Haskell这样的中坚函数式编程语言的惯用法，包括惰性序列（lazy sequences)、偏函数应用（partial function application)以及高阶多态函数（higher-order polymorphic functions）⊖等。这些范型是如此通用，并且如此不同于大多数C++程序员所习惯的东西，以至于使用FC++就像是在使用一门全新的编程语言。在这里，我们没有足够的篇幅充分地讨论FC++，但是我们可以展示一些例子，为你提供一个认知。

首先，我们来看看FC++ lambda表达式。就像在大多数传统的函数式编程语言中的一样（但

⊖ 当我们介绍元函数时，我们曾讨论过术语“高阶函数（higher-order function)”的含义，这是为“操作其他函数”的函数所起的一个别致的术语。在这里的上下文中，“多态”仅仅指的是可以操作不同类型的参数的函数，就像函数模板所做的那样。

和你到目前为止看到的C++ lambda表达式不同），FC++支持使用具名参数来改善lambda表达式的可读性。例如[⊖]：

```
lambda_var<1> X;
lambda_var<2> Fun;

g = lambda(Fun,X)[ Fun[Fun[X]] ]  // g(fun,x) -> fun(fun(x))
```

现在，这确实需要集中精力！Fun和X这两个名字不但在C++程序的层面具有一种意义，而且在lambda表达式生成的程序（函数对象）中也有一种意义。实际上，Boost的Bind和Lambda程序库处理占位符的方式并非特别的不同。占位符实现从输入实参位置到传递给被“绑定”函数的实参位置的映射。你可以将X认为是_1，Fun认为是_2。lambda(Fun,X)[...]做的全部事情就是添加另一个间接层，它交换占位符所表示的位置。

然而，FC++并未止步于具名Lambda参数。下面的例子展示了一个lambda表达式，它带有本质上是具名局部常量（named local constants）的东西：

```
// f(x) -> 2*(x+3)
lambda(X)[
    let[
        Fun == multiplies[2] // Fun = 2*_1
      , Y == X %plus% 3      // Y = X+3
    ].in[
      Fun[Y]                 // fun(Y), i.e. 2*(X+3)
    ]
]
```

上面的例子展示了FC++ DSL的其他一些特性。首先，你可以看到在表达式multiplies[2]中使用了偏应用（partial application）机制，它产生一个一元函数对象，为其参数x计算multiplies[2,x]。其次，%运算符被重载了，使得表达式x %f% y等价于f[x,y]，因而任何FC++二元函数对象（例如plus）都可以担当一种“具名中缀运算符（named infix operator）”。

FC++（领域特定的）语言设计者还做了另一个有趣的选择：他们不喜欢像Boost.Lambda这样的程序库的方式（在某些上下文中），即要求使用constant(...)或variable(...)来防止对任何未使用占位符的表达式的贪婪评估。他们推论，必须记住以下两个表达式中只有一个可以如预期工作，太容易犯错误了：

```
std::cout << _1 << "world"    // OK，构建一个函数对象
std::cout << "hello, " << _1 // 错误：立即流化（immediate streaming）
```

他们选择了一个简单规则：使用小括号的函数调用被立即评估，而使用方括号的函数调用则被缓式评估：

```
plus(2,x) // 立即
plus[2,X] // 缓式
```

类似地，对于立即的中缀评估（immediate infix evaluation），FC++提供了单独的语法：

⊖ FC++将lambda表达式内部的函数调用用方括号括起来，以便明确地延缓函数评估。

```
2 ^plus^ x // 立即
2 %plus% X // 缓式
```

结果，用于缓式评估的的语法立刻变得比Lambda和Phoenix程序库所使用的语法简洁了，且更加清晰。

看上去将%plus%用于命名旧式良好的中缀+运算符显得有点奇怪。实际上，这具有一些很明显的缺点，通过比较这两个大致等价的表达式，我们就可以看到这一点：

```
// Boost Lambda:
-(3 * _1) + _2

// FC++:
lambda(X,Y)[ negate[ 3 %multiplies% X ] %plus% Y ]
```

第一个更短小、更简单，对于任何工作于通常使用运算符符号的问题领域的人而言，它也更清晰。然而，在FC++语言设计的上下文中，有很好的理由去使用plus而不是+。为了理解它们，我们必须考虑plus所代表的C++实体的种类。什么东西允许我们既能编写plus[2,X]，又能编写plus(2,x)？不是函数、函数指针或数组，只有一个类实例才支持这么做：在FC++程序库中，plus必须是一个全局类实例（global class instance）。

现在，回想起FC++都是关于高阶函数式编程（higher order functional programming）的，这一点就很清晰了，+不是一个在所有上下文中都可用来表示加法的名字。你如何将+传递给一个函数？如果你把+运算符的意思理解为表示两个ints相加，好，你甚至无法命名它。如果你试图传递operator+的地址，并且它是被重载的，那么你的C++编译器就会问你到底你是什么意思。如果你的意思是指特别的模板化的operator+，那么再一次提醒你，没有任何将函数模板作为运行期函数参数进行传递的方法。再进一步回顾FC++支持高阶多态函数（higher-order polymorphic functions），很容易明白如果我们希望传递一个真正表示抽象+运算符的实体，它必须是一个类实例，如下：

```
struct plus
{
   template <class T, class U>
   typename plus_result<T,U>::type⊖
   operator()(T t, U u) const
   {
       return t + u;
   }
};
```

实际上，就像FC++的每一个特殊的特性一样，从隐式偏应用（implicit partial application）到显式惰性记号（explicit lazy notation），只有在支持函数对象的C++中才可以。为了满足其设

⊖ 至于如何实现plus_result则是一个很有趣的主题，几乎每一个C++ DSEL框架都有着不同的处理方式。在当前的C++语言中，你无法构建总是能够返回正确类型的元函数。在C++委员会中已经有大量的讨论，即添加一个运算符，从而使得情况大为简化：只要写decltype(t+u)即可。

计者的目标，FC++使用函数对象而不是数学表达式来使用运算符符号，这一点很重要。这一切的要点并非是说这些领域特定的语言中某一个要比另一个好，而是向你——DSEL设计者——说明你可以获得广泛的语法和语义选择。

10.7 Boost Spirit程序库

像YACC一样，Spirit是一个用于定义解析器的框架。主要区别在于，不是编译为中间C/C++代码，Spirit使用一种嵌入式领域特定的语言。这里是使用Boost.Spirit的嵌入式DSL语法，来描述前面使用YACC实现品的表达式文法例子：

```
group       = '(' >> expression >> ')';
factor      = integer | group;
term        = factor >> *(('*' >> factor) | ('/' >> factor));
expression  = term >> *(('+' >> term) | ('-' >> term));
```

你会注意到和传统的EBNF之间存在一些区别。最明显的可能是，因为像如下所示的连续值（consecutive values）的序列

```
'(' expression ')'
```

不符合C++文法，Spirit的作者不得不选择一些运算符来连接连续的文法符号。按照标准流抽取器的样子（毕竟它执行了粗糙的解析），他选择了operator>>。下一个值得一提的区别是*号和正闭包（positive closure）(+) 操作符，它们通常写在所要修改的表达式后面，现在必须写成前缀运算符，再一次是因为C++文法的限制。撇开这些小的让步不谈，Spirit文法非常接近于领域的常用符号。

Spirit实际上是一个很好的关于DSEL彼此互操作威力的例子，因为它其实由一组小型嵌入式语言组成。例如，下面的完整程序使用Phoenix函数式编程DSEL和另一个DSEL惯用法（Spirit称做闭包（closures）），将上面的文法和写在[...]之间的语义动作（semantic actions）集合在一起：

```
#include <boost/spirit/core.hpp>
#include <boost/spirit/attribute.hpp>
#include <iostream>
#include <string>

using namespace boost::spirit;
using namespace phoenix;
// 提供一个类型为int的具名变量……
struct vars : boost::spirit::closure<vars, int>    // CRTP
{
    member1 value; // 在惰性表达式中称做"value"
};

// calculator是一个带有名为"value"的附加int的文法（grammar）
struct calculator
```

```
    : public grammar<calculator, vars::context_t>    // CRTP
{
    template <class Tokenizer>
    struct definition
    {
      // 同样，所有rules都有一个名为“value”的附加int……
        rule<Tokenizer, vars::context_t>
          expression, term, factor, group, integer;

      // ……除了top rule外
        rule<Tokenizer> top;

      // 构建文法（grammar）
        definition(calculator const& self)
        {
            top = expression[self.value = arg1];

            group = '(' >> expression[group.value = arg1] >> ')';

            factor = integer[factor.value = arg1]
                | group[factor.value = arg1]
                ;

            term = factor[term.value = arg1]
                >> *(   ('*' >> factor[term.value *= arg1])
                      | ('/' >> factor[term.value /= arg1])
                    )
                ;

            expression = term[expression.value = arg1]
                >> *(   ('+' >> term[expression.value += arg1])
                      | ('-' >> term[expression.value -= arg1])
                    )
                ;

            integer = int_p[integer.value = arg1];
        }
            // 告诉Spirit以“top”开始解析
              rule<Tokenizer> const& start() const { return top; }
          };
};

int main()
{
    calculator calc;    // 待解析的文法
```

```
        std::string str;
        while (std::getline(std::cin, str))
        {
           int n = 0;
           parse(str.c_str(), calc[var(n) = arg1], space_p);
           std::cout << "result = " << n << std::endl;
        }
    }
```

10.7.1 闭包

我们将从两个层面来描述闭包（closure）：首先，我们从DSL用户的角度来检视它们，希望你先将对魔法是如何工作的考虑放在一边，然后再来检视实现的技术。

抽象

为了理解闭包的用法，知道这一点很重要：Spirit文法（grammars）和规则（rules）（正如你猜测的那样）都是函数对象。当用一对指向输入的适当的迭代器进行调用时，规则（rules）和文法（grammars）试图去解析它。这会导致自顶向下（top down）或递归向下（recursive descent）解析。例如，为了解析其第一个符号，expression rule依次调用term rule。

闭包（Closure）提供了一套和每一个rule调用相关联的变量，以rule自身的成员的方式进行访问。闭包的第一个成员的值变成rule的“返回值”（在我们的例子中只有一个成员，即value），当rule用在另一个rule的右侧时，可以通过使用Phoenix占位符arg1依附到rule的语义动作（semantic actions）进行访问。因此，例如在

```
term = factor[term.value = arg1]
       >> *(   ('*' >> factor[term.value *= arg1])
             | ('/' >> factor[term.value /= arg1])
           )
    ;
```

中，和第一个factor调用相关联的value首先被移入和当前term调用相关联的value。然后，当*号的每一个成员被解析时，和当前term调用相关联的value被相应地修改。

关于闭包，真正有趣的东西是它们使得另一种编程范型成为可能：动态作用域（dynamic scoping）。在C++中，未修饰名字（不带一个“::”前缀）通常指向它们在其中进行定义的、最内层封闭作用域：

```
#include <iostream>

namespace foo
{
  int x = 76;

  int g()
  {
     return x + 1;                    // 指的是foo::x
  }
}
```

```
int main()
{
   int x = 42;
   std::cout << foo::g();          // 打印77
}
```

然而，在动态作用域系统中，名字指向（refer to）它们被定义的调用堆栈上最接近的作用域。因而，在同样的代码中，foo::g将会看到建立于main()中的x的值，并且程序将打印出43。

闭包变量的完全限定名（rulename.membername）的作用域是动态的。这意味着，举个例子，我们文法中的任何语义动作（semantic action）都可以指向expression.value，并且在这种情况下，可以触及调用堆栈中与最近的expression rule调用相关联的value。

实现细节

看一看我们的闭包声明：

```
struct vars : closure<vars, int>
{
    member1 value;
};
```

你可能注意到的第一件事情是，该闭包使用了“奇特的递归模板模式（Curiously Recurring Template Pattern）”（第9章讨论过），因此，它可以访问vars的类型。然而更有趣的是，对value使用了特殊的类型member1。

显而易见，如果程序库允许你写rulename.closure-variable，那么rules必须包含和闭包变量具有相同名字的公有（public）数据成员。实际上，C++语言的限制在此给了我们很大的提示：自动允许闭包数据成员和rule数据成员具有相同名字的惟一方式就是使闭包成为rule类的公有基类（public base）。除此之外，绝无其他在rule中产生具有一致命名的public数据成员的方式。

同样明显的是，如果expression.value += 1之类的东西可按预期方式工作，那么expression.value不可以是类型int，否则，当其rules被调用时，将会导致立即（而不是稍后）进行一个整数加法（就像我们文法定义的那样）。这个问题听上去是否有些熟悉？实际上，它可以用一种熟悉的方式进行解决，即采用表达式模板（expression templates）。不是执行一个加法运算，expression.value += 1创建一个对象，当该对象被解析器适当地调用时，加1到int变量上（该int变量是为在最近的封闭expression调用的栈框架（stack frame）中的value而创建的）。

我们不打算深究动态作用域机制的本质实现细节，因为它与闭包的“DSEL性”没有直接的关系。如果你好奇的话，我们建议你看看Spirit和Phoenix源代码。再一次指出，重要的是要认识到这一点，表达式模板（expression templates）和缓式评估（delayed evaluation）允许我们使用原生C++中不直接支持的编程范型（programming paradigm）。

10.7.2 子规则

如果你仔细瞧瞧我们的calculator文法，你就会明白必然有类型擦除机制（type erasure）在发挥作用[⊖]。由于每一个rule赋值右边的表达式构建了一个高效的函数对象，因此我们可以预料

⊖ 参见第9章，了解关于类型擦除（type erasure）更多的信息。

这些函数对象的类型来表示表达式的结构。例如，不考虑语义动作（semantic actions）的效果（因为它使得事情进一步复杂化），factor rule右侧的类型类似于如下：

```
alternative<
    rule<Tokenizer, vars::context_t> // 针对integer rule
  , rule<Tokenizer, vars::context_t> // 针对group rule
>
```

而group rule的右侧则具有类似于如下的类型：

```
sequence<
    sequence<
        ch_p                                    // 针对'(' parser
      , rule<Tokenizer, vars::context_t> // expression rule
    >
  , ch_p                                        // 针对')' parser
>
```

然而，factor和group rules自身具有同样的类型。显然表达式模板（expression templates）产生的编译期多态正被转变为运行期多态：同样类型的rule对象必须包含某个函数指针或虚函数，或其他一些东西，允许它们中的一个去解析groups，而另一个则去解析factors。当然，在运行期进行行为选择伴有效率上的损失。在像这个简单的文法例子里，针对每一个rule调用的动态派发而付出的代价其实是合理的。

Spirit的主要作者Joel de Guzman曾这样写道[Guz03]：

……虚函数是一堵完全不透明的墙，它阻挡了所有元类型（meta type）信息通过。比如说，我们永远都无法获得RHS或一个rule的任何类型信息，去将表达式模板树（expression template tree）重构为别的什么东西（例如，进行自动提取左因子（automatic left factoring）、静态节点类型遍历（static node-type-traversal）、静态first-follow分析（static first-follow analysis）等）。

这些操作都是和解析器具体相关的问题，但要点仍然具有一般性：类型擦除（Type erasure）是一种“有损压缩（lossy compression）”技术，有价值的信息可能会永远消失。

Spirit设计者可能（可以）告诉我们去简单地写出每一个rule右侧的完整类型，但这个思想基本上是一个DSEL杀手。正如你可以想象的那样，为每一个rule写下一个复杂的类型，并且每当rules发生改变时重新写下这些类型，很快就会变得不可管理。更重要的是，我们不得不向我们的文法（grammars）填充关于rule类型的信息，而这从DSEL的角度来看纯属噪音，因为它与底层的领域抽象完全无关。

值得指出的是，甚至本章前面描述的auto语言扩充都不能为Spirit完全解决这个问题，因为文法规则（grammar rules）都彼此引用，这样，编译器永远无法知道第一个“auto初始化”的右侧的类型。

Spirit以一种我们熟悉的方式解决了这个在效率和表达力之间的紧张状态，即通过将工作延迟到尽可能晚的一刻。正像Blitz++通过“延迟矩阵（matrix）算术直到整个表达式可用”来获得效率一样，Spirit使用子规则（subrules）来延迟对静态类型信息的擦除工作，即直到整个文法（grammar）可用再进行擦除。下面对calculator的定义的改写版本使用了子规则（subrules）来获

得近乎最优化的解析效率：

```
template <class Tokenizer>
struct definition
{
    subrule<0, vars::context_t> expression;
    subrule<1, vars::context_t> group;
    subrule<2, vars::context_t> factor;
    subrule<3, vars::context_t> term;
    subrule<4, vars::context_t> integer;

    rule<Tokenizer> top;

    definition(calculator const& self)
    {
        top = (
            expression =
                term[expression.value = arg1]
                >> *(  ('+' >> term[expression.value += arg1])
                      |('-' >> term[expression.value -= arg1]) )

          , group =
                '(' >> expression[group.value = arg1] >> ')'

          , factor =
                integer[factor.value = arg1]
              | group  [factor.value = arg1]

          , term =
                factor[term.value = arg1]
                >> *(  ('*' >> factor[term.value *= arg1])
                      |('/' >> factor[term.value /= arg1]) )

          , integer =
                int_p[integer.value = arg1]

        )[ self.value = arg1 ];
    }

    // 告诉Spirit从“top”开始解析
    rule<Tokenizer> const& start() const { return top; }
};
```

在这个例子中有两件事情尤其值得注意。第一，为了获得这种延迟同时又避免强迫用户写出杂乱的类型，所有子规则（subrules）的定义都必须写在单个表达式中。直到对惟一的完整的rule——top进行赋值时，才会发生类型擦除（Type erasure）。在那一点，一个甚至比任何右侧类

型更混乱、并且包含所有子规则（subrules）的定义的类型，被捕捉于一个单个的虚函数中。一旦发生单个动态派发，对一个表达式的解析就只涉及正常的静态函数调用了，其中很多调用都可被内联。第二个值得注意的事情是，从动态派发的rules到静态派发的子规则（subrules）几乎根本不改变文法的表示。这是Spirit的一个特别美好的特性，它为我们提供了在编译期/运行期统一体中易于调整位置的能力，同时还如此接近基础的EBNF领域语言。

10.8 总结

我们希望这一章为你带来关于程序库设计的新观点。最有效的程序库为用户提供了新的表达层次，允许他们按照适合于问题领域的方式进行编程。尽管从历史上看，向任何编程环境引入新的惯用法和语法都被报以怀疑的目光（有时这种怀疑并非没有理由），然而实践也证明在简化编程和加速开发方面，这种方式的确具有极大的威力。

按照领域特定的语言的方式进行思考，为程序库设计选择提供了一个基础，并帮助我们判断哪一种新的编程惯用法和语法是有效的。通过依赖于那些最具号召力的既有领域抽象和符号，我们可以编写直接表达我们意图的程序。

对高级特性独一无二的联合运用（其中一些特性包括：灵活的运算符重载语法，只有一点或完全没有性能惩罚的高阶抽象，以及模板元编程的威力）为C++程序员构建富有表达力的高效DSEL提供了无与伦比的威力。此外，由于DSEL不过是程序库，所以用户可以在无需离开熟悉的编程环境的情况下，自由地将DSEL能力结合于同一个应用程序中。

占据了本书大部分篇幅的纯粹的编译期MPL构造，以及我们在第9章中讨论的用于桥接编译期和运行期代码的技术，都是构建DSEL的有效工具。在下一章中，我们将检视一个例子，看看是如何构建DSEL的。

10.9 练习

10-0. 考虑在Spirit中使用不同于>>的运算符来分隔连续的文法符号的可能性。还有其他更合适的运算符吗？为什么？提示：除了可读性之外，还要考虑C++文法规则（grammar rules）。

10-1. 你开始注意到在宿主语言的局限性驱动很多DSEL设计决策中存在的一个共通的主题了吗？考虑你可能如何设计一门“允许无限制的DSEL语法”的语言，并且进行成本/利益分析，比较使用假定的语言和使用你已经在C++中看到的东西。你也许可以翻翻历史，从LISP中的宏（macros）的用法获得灵感。

10-2. 使用本章讨论的任何一个Boost DSEL程序库来解决一个小问题。对你的用户体验进行评估：该程序库好在什么地方？又有何烦人之处？

10-3. 使用一个类似于本章中描述的草案，构建一个用来处理具名函数参数的小型DSEL。将你的设计与David Abrahams和Daniel Wallin设计的Boost具名参数程序库（Boost named parameters library）（配书CD上有这个程序库预发行版本）进行比较。

第11章 DSEL设计演练

在这一章中，我们将演练设计和实现一个领域特定的嵌入式语言以及一个操作元程序的过程。首先，我们来探究一个领域并识别其原则抽象，通过一个具体的例子，对其在现实世界中的含义获得一个认知。然后，我们设计一个DSEL来表达那些抽象，以该例作为对概念的实证。最后，我们将应用你已经在本书中学到的工具和技术来编写一个元程序，它处理该语言以生成有意义的、高效的运行期组件。

11.1 有限状态机

每一位软件工程师都应该熟悉有限状态机（finite state machine，FSM）。这个概念太有用了，以至于你会发现它无处不在，从硬件设备控制器到模式匹配和解析引擎（例如YACC所使用的）。这种形形色色的应用的开发者已经拥抱了FSM的使用，因为它们使紊乱的网状程序逻辑转换成了易于理解的表达形式。我们可以将这个威力归功于两样东西：对FSM抽象及其声明形式的根本上的简化。

很少有通用语言为构建FSM提供内建支持，C++也不例外。如果它不是每一个C++程序员技能的一个常规组成部分，那么它只能是那些希望有一个“可以使得构建FSM变得容易且有趣”（本该如此，不是吗）的工具的程序员所希望的。在这一章中，我们正是为了设计和构建这样的一个工具：一个有限状态机构建框架。

11.1.1 领域抽象

有限状态机的领域抽象（Domain Abstraction）由三个简单的元素构成：

状态（states）

一个FSM必须总是处于几个定义良好的状态之一。例如，一个简单的CD播放器的状态可能包括：Open、Empty、Stopped（机舱里有一张CD）、Paused以及Playing。与一个纯粹的FSM相关联的惟一持久的数据被编码于状态中，尽管FSM很少单独用于任何系统中。例如，YACC生成的解析器构建在一个状态机的栈（stack）上，整个系统的状态包括栈的状态和栈中的每一个FSM的状态。

事件（Events）

状态的改变是由事件所触发的。在我们CD播放器例子中，大多数事件都对应于面板上的按钮下压动作：play、stop、pause以及open/close（用于开、关机舱的按钮）。尽管如此，事件并不是非得从外部“推入”状态机。例如，在YACC解析器中，每一个事件表示一个不同的token，它被解析过程从输入流中“拉”出来。在一些系统中，事件包含相关联的数据。例如，在一个C++解析器中的一个标识符token（identifier token）可能携带有该标识符的文本，而一个整型字面量token（integer-literal token）则可能携带该整数的值。

转换（Transitions）

每一个状态可以有任意数量的到其他状态的转换。每一个转换被标记以一个事件。为了处理事件，FSM遵从从当前状态开始的转换，并被标记以该事件。例如，一个CD播放器具有一个从Playing到Stopped的转换，并被标以stop事件。通常，转换还具有一些关联动作（action），例如对于CD播放器例子而言的stop playback。对于YACC的情况，遵从转换意味着操纵FSMs的栈并且/或者执行用户的语义动作（semantic actions）。

11.1.2 符号

有好几种常见的方式用以书面的形式描述状态机，可能最友好的符号当属图形方式了。图11.1表示我们已经用做例子的CD播放器。在图形中，状态（states）以圆来表示，而转换（transitions）则以箭头表示，并标以触发它们的事件（events）。

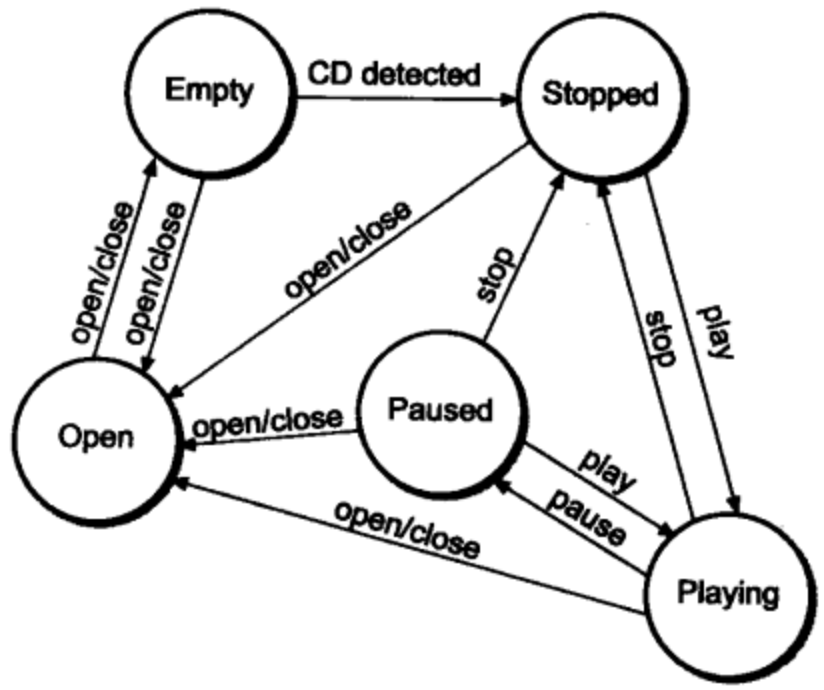


图11.1 以图形的方式描述CD播放器FSM

注意从Empty到Stopped的转换。还记得我们说过并非所有事件都需要从外界“推入”系统吗？为了模塑真实的CD播放器，当CD仓关闭时，FSM将开始一个CD侦测过程，如果它侦测到仓中有一个CD，系统就向自身发送一个cd-detected事件。为了使其工作，从Open到Empty的转换必须具有一个相关联的用于开始CD侦测过程的动作。当一个新的唱片被侦测到时，大多数CD播放器收集有关音轨的数目以及每一个音轨的总共播放时间信息，cd-detected事件应该包含该信息，以便转换动作在某处存储并在播放器的面板上显示出音轨的数目。

图形表示法直观地展示了在一个FSM中可以发生的任何东西，没有任何多余的语义元素。事实上，一个用于构建FSM的流行策略就是使用“带有一个代码产生后端（code-generating back end）”的图形用户界面来画出状态机。只要C++允许在它的输入语法中含有图形，那么它们就会成为完美的DSEL符号！

由于C++无法解析图形，我们打算用一个称为状态转换表（State Transition Table，STT）的不同的符号系统，它本质上不过是FSM的转换的垂直列表。表11.1展示了CD播放器的STT。

表11.1 CD播放器状态转换表

当前状态	事件	下一个状态	转换动作
Stopped	play	Playing	start playback
Stopped	open/close	Open	open drawer
Open	open/close	Empty	close drawer; collect CD information
Empty	open/close	Open	open drawer
Empty	cd-detected	Stopped	store CD information
Playing	stop	Stopped	stop playback
Playing	pause	Paused	pause playback
Playing	open/close	Open	stop playback; open drawer
Paused	play	Playing	resume playback
Paused	stop	Stopped	stop playback
Paused	open/close	Open	stop playback; open drawer

尽管这种FSM结构不如图形形式的明显，但仍然是相当容易理解的。为了处理一个事件，状态机找到这样的一行：第一列含有其当前的状态，第二列中包含有事件，该行的第三和第四列指示新状态和使得转换发生的动作。注意，我们在FSM的图形表示中没有考虑转换动作，目的是为了使得混乱最小化，而它们在STT中导致的干扰则很小或没有。

11.2 框架设计目标

那好，我们希望从状态机框架中获得些什么呢？

1. 互操作能力。状态机典型来说不过是一个用来描述（针对某些问题领域而非FSM构建的）系统逻辑的抽象。我们希望在FSM实现中能够使用为那些领域构建的程序库，因此我们希望确信能够与其他DSEL舒服地互操作。

2. 声明性。状态机作者应该具有描述FSM的结构而非实现其逻辑的经验。理想状况下，构建一个新的状态机不过是将其STT转换为C++程序而已。作为框架的提供者，我们应该能够无缝地改变状态机的逻辑的实现同时不对作者的描述造成影响。

3. 表达力。应该很容易表示和识别一个程序中的领域抽象。在我们的例子中，代码中的STT看上去应该酷似我们在纸上设计状态机时的模样。

4. 效率。像CD播放器这样简单的FSM应该完美地编译为极其紧凑的代码，它可被优化为甚至对于微型嵌入式系统而言都是适当的东西。可能更重要的是，关于我们的框架的效率的考虑，应该永远不要给程序员的这样的借口：在一个有限状态机的合理的抽象可能被采用别的方式应用的地方，使用特殊的逻辑。

5. 静态类型安全。在编译期捕获尽可能多的问题是很重要的。很多传统的FSM设计[LaFre00]的一个典型弱点是它们在运行期进行大部分的检查。尤其是，不应该用不安全的向下转型（downcasts）来访问不同的事件（events）中包含的不同数据类型。

6. 可维护性。对状态机设计的简单改变应该只导致对其实现的简单的改变。这看上去像是一个显而易见的目标，但要做到这一点其实很难——一些专家们为此付出了努力并以失败告终。例如，当使用State设计模式[Mart98]时，仅仅一个改变，例如添加一个转换（transition），就可

能导致要重构多个类（class）。

7. 可伸缩性。FSM可能会变得比上面例子复杂得多，比如加入每状态进入和退出动作（per-state entry and exit actions）、有条件的转换哨位（conditional transition guards）、缺省的和非触发的转换（default and triggerless transitions），甚至还有子状态（sub-states）等特性。如果现在框架不支持这些特性，那它应该具有相当好的可扩充性，以便以后支持这些特性。

11.3 框架接口基础

我们可以立刻做出一些容易的选择。因为上面的目标1，当然，因为这一章的名字——我们将设计一个嵌入式DSL。这意味着采用程序库的方式而不是构建一个翻译器（像YACC那样）或一个解释器（像Make那样）。

我们通常希望创建一个有限状态机的多个实例，每一个都具有单独的状态——尽管就我们的CD播放器例子而言这一点并不明显。因此，将FSM的逻辑封装在一个可复用的包里是很有意义的。由于C++的数据封装基本单元是类，我们的框架将帮助我们构建FSM类。因为整个FSM会被表示为一个类，看上去将start playback这样的转换动作表示为该类的成员函数是合情合理的。

我们希望能够使用像Playing和open_close这样的可读性名字来指示状态和事件。在这一点，关于使用什么种类的C++实体（例如类型、整数值、函数对象等）来指示诸如Playing之类的状态名字，我们说不了太多。然而事件就不同了，在CD播放器中，只有cd-detected事件包含数据，但通常每一个不同种类的事件可能需要传输一个不同的数据类型。因此，事件名字应该表示类型。对于在一个事件中嵌入任意数据而言，FSM作者声明一个具有相应的数据成员的事件类即可。

鉴于有限状态机是类，并且那些事件将采用types进行实现，因此可以设想一个使用我们框架构建的状态机可能会用如下方式使用：

```
int main()
{
    player p;                          // 一个FSM实例

    p.process_event(open_close()); // 用户打开CD播放器
    p.process_event(open_close()); // 放入CD并关仓
    p.process_event(                   // 侦测到CD
        cd_detected(
             "Louie, Louie"
           , std::vector<std::clock_t>( /* 音轨长度 */ )
         )
    );
    p.process_event(play());           // 等等……
    p.process_event(pause());
    p.process_event(play());
    p.process_event(stop());
    return 0;
}
```

11.4 选择一个DSL

我们的下一个挑战是设计一个领域特定的语言，它允许程序员描述一个有限状态机，就像player所实现的有限状态机一样。稍后，我们将编写元程序代码，它处理FSM描述，以便产生一个像player的类。正如早先所暗示的，状态机作者打算采用状态转换表的方式来描述，因此，让我们来考察用于表示它的可能的语法。

11.4.1 转换表

“为转换（transitions，即STT中的行）决定一个表示法”使得事情真正开始变得有趣起来。我们有很多选项！让我们考察一些可能的“编写表中的前两行”的方式，并分别对其进行分析，以便获得对可支配选项的范围认知。眼下，我们不打算对如何使用这些语法来构建FSMs操太多的心。要点仅在于考虑STTs如何能映射到C++语法的：

```
// 当前       事件          下一       动作
// 状态                     状态
 [ Stopped, play,       Playing, &fsm::start_playback ]
 [ Stopped, open_close, Open,    &fsm::open_drawer    ]
```

我们第一个尝试是采用酷似STT的语法使得状态机的结构保持清晰。怎样才能使这个语法工作呢？为了使得方括号合法，必须有一个类，例如transition_table，它带有重载的operator[]。因为C++编译器不允许我们孤立地编写方括号表达式，所以用户必须在表的前面加上该类的一个实例名字，如下：

```
transition_table STT; // 由FSM框架提供
...

  // 当前       事件          下一       动作
  // 状态                     状态
STT[ Stopped, play,       Playing, &fsm::start_playback ]
   [ Stopped, open_close, Open,    &fsm::open_drawer    ]
```

接下来，因为operator[]只允许有一个参数，因此至少要有一个重载的逗号运算符将方括号内的各项联合起来。注意到这一点，我们甚至可以使语法变得更像表，这是通过将逗号运算符替换为operator|而达成的：

```
  // 当前       事件          下一       动作
  // 状态                     状态
STT[ Stopped  | play        | Playing  | &fsm::start_playback ]
   [ Stopped  | open_close  | Open     | &fsm::open_drawer    ]
```

然而，考虑到事件名字将指示类型，我们在上面探讨的两种语法都带来一个问题：无法在一个运行期表达式中使用一个类型（好像它是一个对象那样）。我们必须传递该类型的一个实例，因此我们的表最终看起来可能更像这样：

```
  // 当前       事件          下一       动作
  // 状态                     状态
```

```
STT[ Stopped  | play()       | Playing  | &fsm::start_playback ]
   [ Stopped  | open_close() | Open     | &fsm::open_drawer    ]
```

每一行多两个小括号并不会把语法搞得太糟糕，但我们需要事件能被默认构造（default-constructed）。对于任何有经验的程序库设计者来说，对默认构造能力的需要都是一个危险的信号。在这个例子中，事件（events）并非都是轻量级的类型，而且，只是为了可以构建转换表而构建实例的做法可能并不合适。

应用软件工程的基础定理（Fundamental Theorem of Software Engineering）[⊖]，我们可以要求用户将事件的类型信息间接地传输给框架来解决这个问题，并可采用一个小型外覆器模板（wrapper template）做到这一点：

```
// 由FSM框架提供
template <class Event>
struct on
{
    typedef Event type;
};
...

  // 当前       事件                  下一       动作
  // 状态                             状态
STT[ Stopped |on<play>()       | Playing |&fsm::start_playback ]
   [ Stopped |on<open_close>()| Open    |&fsm::open_drawer     ]
```

这在原理上是可以工作的，但语法开始变得有点笨拙，使得事件的名字因为夹杂语法“噪音”而不明显。我们可以通过以下方式来恢复一定程度的可读性：

```
on<play>       play_;
on<stop>       stop_;
on<open_close> open_close_;
...

  // 当前       事件           下一       动作
  // 状态                      状态
STT[ Stopped | play_       | Playing | &fsm::start_playback ]
   [ Stopped | open_close_ | Open    | &fsm::open_drawer    ]
```

这从根本上来说并不赖。然而不幸的是，有一个问题开始来扼杀这种可爱的模式。还记得我们第4个设计目标“效率”么？到目前为止我们看到的所有这些设计的问题在于，它们倾向于从两种方式来伤害状态机的效率：

1. 我们将指针传递给转换动作，作为一些operator ()函数的参数。这意味着我们将必须把它们以数据成员的方式存储在某个地方，以后当FSM实际执行转换时通过存储的指针来调用。结果，即使是最简单的转换函数也将不会得到内联。这些代价并不是在所有设计中都很显著。例

⊖ “我们可以引入一个额外的间接层来解决任何问题”。参见2.1.2节以了解该思想的起源。

如，当Boost.Bind[⊖]用于构建一个“比较”函数对象（用于与std::sort协同使用）时，在排序过程中移动序列元素的代价通常会“淹没”通过单个函数指针进行重复调用的开销。然而，对于我们的有限状态机的情形来说，为改变状态而执行的代码通常是不值得提的，以至于所添加的间接层是必须要考虑的。

2. 我们还在运行期构建了整个转换表。这在本质上并不是一个效率问题：表的构建可以被内联，并且，正如我们已经在Blitz++中看到的那样，完全可以在运行期使用表达式模板（expression templates）构建一个超级高效的计算引擎。不幸的是，用于构建这样的引擎的运算符数量与其类型结构的复杂度具有直接的正比关系。在构建起一个足够复杂的类型来表示一个相当大的STT后，我们无法让用户写下该类型。如果我们希望将状态机保存在某个变量中，那么表就需要被立即传递给一个函数模板，或者我们不得不求助于某种类型擦除机制（type erasure）[⊖]，而这总是会导致另一层的函数指针间接调用。

我们可以避免通过一个转换动作函数指针而导致的间接代价，这是通过将动作成员指针作为模板实参传递而做到这一点的，参见第9章的描述：

```
//          当前          事件              下一          动作
//          状态                            状态
transition< Stopped,  play,          Playing, &fsm::start_playback >,
transition< Stopped,  open_close,    Open,    &fsm::open_drawer    >,
```

这个语法并非多么甜美，但我们仍然认为它足够表格化。在这个方向上我们可以更进一步，只要将逗号的位置挪一挪，并加以必要的注释即可：

```
//    当前          事件              下一          动作
//    状态                            状态
//  +---------+------------+---------+----------------------+
row < Stopped , play       , Playing , &fsm::start_playback  >,
row < Stopped , open_close , Open    , &fsm::open_drawer     >,
//  +---------+------------+---------+----------------------+
row < Paused  , play       , Playing , &fsm::resume_playback >,
row < Paused  , stop       , Stopped , &fsm::stop_playback   >,
row < Paused  , open_close , Open    , &fsm::stop_and_open   >,
//  +---------+------------+---------+----------------------+
```

尽管我们不得不将transition替代为不那么有意义的标识符row（从而使得例子可以适合页宽），然而新的格式看上去可读性更好。

与我们前面的那些尝试相比，这种途径具有两个重要的实际优势，不管你采用什么样的代码布局。首先，它可以只使用类型表达式进行实现，从而就不会因为过早的跨越编译期/运行期边界而导致的效率损失。这是因为动作（action）函数指针是一个模板参数，它在编译期是已知的，并且可被完全内联（inlined）。其次，因为每一个row<...>实例化都是一个类型，所以我们可以将一个有关它们的逗号分隔的列表作为参数传递给一个MPL序列，并且用于操纵类型序列

⊖ 参见第10章以便了解更多关于Boost Bind程序库的知识。

⊖ 参见第9章以便了解关于类型擦除的更多知识。

的所有MPL工具都能任由我们使用。

既然已经知道了将要使用的转换表格式，我们现在就可以选择状态（state）名字所指示的C++实体的种类。呃，状态机是有状态的。换句话说，它们并非很好地符合纯粹模板元编程的编译期世界。我们需要将状态的名字作为模板参数传递给row<...>，但还需要能够在单个数据成员中存储“表示任一个不同的FSM状态”的东西。整型常量满足这两个约束。幸运的是，C++为我们提供了定义带有惟一值的具名整型常量集合的便利方式：

```
enum states {
   Stopped, Open, Empty, Playing, Paused
 , initial_state = Empty
};
```

正如你看到的那样，我们还定义了一个额外的常量initial_state。我们赋予该特别的标志符以特别的含义：框架将使用它来决定默认构造的FSM实例的初始状态。

11.4.2 组装成一个整体

除了一些小细节外，我们现在已经探索了DSEL的所有语法方面，接下来是一个完整的例子，展示如何用DSEL来描述一个FSM：

```
struct play; struct open_close;struct cd_detected;                    // 事件
struct pause; struct stop;

class player : public state_machine<player>
{
private:
    // FSM 状态列表
    enum states {
        Empty, Open, Stopped, Playing, Paused
      , initial_state = Empty
    };

    // 转换动作 (transition actions)
    void start_playback(play const&);
    void open_drawer(open_close const&);
    void close_drawer(open_close const&);
    void store_cd_info(cd_detected const&);
    void stop_playback(stop const&);
    void pause_playback(pause const&);
    void resume_playback(play const&);
    void stop_and_open(open_close const&);

    friend class state_machine<player>;
    typedef player p; // 使转换表更加整洁

    // 转换表 (transition table)
```

```
    struct transition_table : mpl::vector11<

    //    开始状态    事件          下一状态    动作
    //  +---------+------------+---------+--------------------+
    row < Stopped , play        , Playing , &p::start_playback  >,
    row < Stopped , open_close  , Open    , &p::open_drawer     >,
    //  +---------+------------+---------+--------------------+
    row < Open    , open_close  , Empty   , &p::close_drawer    >,
    //  +---------+------------+---------+--------------------+
    row < Empty   , open_close  , Open    , &p::open_drawer     >,
    row < Empty   , cd_detected , Stopped , &p::store_cd_info   >,
    //  +---------+------------+---------+--------------------+
    row < Playing , stop        , Stopped , &p::stop_playback   >,
    row < Playing , pause       , Paused  , &p::pause_playback  >,
    row < Playing , open_close  , Open    , &p::stop_and_open   >,
    //  +---------+------------+---------+--------------------+
    row < Paused  , play        , Playing , &p::resume_playback >,
    row < Paused  , stop        , Stopped , &p::stop_playback   >,
    row < Paused  , open_close  , Open    , &p::stop_and_open   >
    //  +---------+------------+---------+--------------------+
    > {};
};
```

上面代码中的一个新细节是player的基类被授予了友元关系。因为player的所有嵌套的定义，包括其状态（states）、转换表（transition table）以及动作（actions），都是状态机框架专用的，而不是给公众使用的，因此将它们标为private。除了默认构造器外，状态机的公开接口仅由process_event成员函数（由基类提供）构成。

我们应该指出的另一个细节是对奇特的递归模板模式（Curiously Recurring Template Pattern，CRTP）的使用，player派生于state_machine<player>[⊖]。就像其他很多DSEL设计选择那样，它受到C++语言约束的驱动。考虑row可能如何进行声明，从而可以接收一个指向player的成员函数的指针作为一个模板参数。row应该是类似于如下的东西：

```
template <
    int CurrentState
  , class Event
  , int NextState
  , void (player::*) (Event const&)
>
struct row
{ ... };
```

换句话说，row需要知道player的类型。我们可以叫row的作者自己去供应FSM类型（作为一

⊖ 参见第9.8节以便了解关于CRTP更多的知识。

个初始的模板参数)：

```
template<
    class Fsm     // 显式的FSM说明
  , int CurrentState
  , class Event
  , int NextState
  , void (Fsm::*action)(Event const&)
>
struct row
{ ... };
```

该途径将会向转换表中添加额外的一栏，这栏东西不是别的，而是相同类名字的冗余复制而已。由于C++已经要求状态机的作者采用FSM类名字来修饰所有的成员函数指针，因此这种做法无异于“火上浇油”。然而，使用CRTP，FSM作者将类名字仅传递给基类模板state_machine一次。由于state_machine具有对派生类名字的完全访问权，因此可以为特定的Derived状态机供应一个便利的嵌套的row模板：

```
template<class Derived>
class state_machine
{
    ...
  protected:
    template<
        int CurrentState
      , class Event
      , int NextState
      , void (Derived::*action)(Event const&)
    >
    struct row
    {
        // 供我们的元程序以后使用
        static int const current_state = CurrentState;
        static int const next_state = NextState;
        typedef Event event;
        typedef Derived fsm_t;

        // 执行转换动作 (transition action)
        static void execute(Derived& fsm, Event const& e)
        {
            (fsm.*action) (e);
        }
    };
};
```

注意，在上面的代码中，我们已经将row填充在state_machine本体内。嵌套的定义仅仅充当

一个用途：允许方便地访问row的模板参数的值。action参数通过一个调用它的execute函数进行访问，这看上去似乎有点奇怪。不幸的是，在C++中，一个嵌套的常量成员指针永远不可以成为一个编译期常量。如果action被赋值给一个static const成员（就像其他模板参数一样），那么通过它进行调用将不会被内联（inlined）。

11.5 实现

现在我们开始接触state_machine的实现细节，我们同样可以冒冒失失地一头扎进去。你也许会问自己，“到底要付出多少劳动才能心想事成？”很大程度上答案可以归结为实现process_event，它毕竟对FSM的运行期行为负全责（当然转换动作除外，它们是由派生的FSM的作者供应的）。

process_event成员函数呈现了一个经典的“双分派（double dispatch）”问题：给定一个起始状态（start state）和一个事件（event），我们需要选择一个目标状态（target state）和要执行的一个转换动作（transition action）。通常而言，双分派的实现可能相当具有挑战性，但在这个例子中，我们具有独特的优势：我们在编译期知道事件的类型，这就允许利用编译器的重载决议能力。如果打算手工编写一个FSM实现品而不是让程序库生成一个，我们应该为每一个事件类型（event type）提供单独的process_event重载实现，如下：

```
// “play”事件处理器
void process_event(play const& e)
{
  switch (this->state)
  {
   case Stopped:
      this->start_playback(e);
      this->state = Playing;
      break;
   case Paused:
     this->resume_playback(e);
     this->state = Playing;
     break;
   default:
      this->state = no_transition(this->state, e);
  }
}
// “stop”事件处理器
void process_event(stop const& e)
{
    ...
}
// 等等……
```

理想上说，为了自动地做这件事情，我们只要实例化一些“针对当前状态（current states）、

动作（actions）和目标状态（target states）进行参数化”的模板，并包含switch语句就可以了。然而，只要看看play事件处理器，我们就能可以发现一个问题。在switch语句中可能会有任意数量的cases，每一case针对一个事件（event）的所有转换（transition），而且C++没有为我们提供这样一种直接生成任意大小switch语句的方法。为了根据转换表中的信息（将被逐行（row）处理）创建一个case语句，我们需要从有着类似外观的小片代码构建switch语义。使用C++ 模板我们能够生成的最小代码单元是一个函数调用，因此这些片断应该是函数。将每一个case分解为单独的函数会产生类似如下的一些内容（针对play事件处理器的情形）：

```
// "play"事件处理器
void process_event(play const& e)
{
    this->state = case_Stopped(e);
}

int case_Stopped(play const& e)
{
    if (this->state == Stopped)
    {
       this->start_playback(e);
       return Playing;
    }
    else return this->case_Paused(e);
}

int case_Paused(play const& e)
{
    if (this->state == Paused)
    {
       this->resume_playback(e);
       return Playing;
    }
    else return this->case_default(e);
}
int case_default(play const& e)
{
    return this->no_transition(this->state, e);
}
```

在这里，process_event(play const&)将其实现转发给case_Stopped。case_Stopped首先检查当前的state是否为Stopped，如果是，即采取相应的转换动作（start_playback）并返回Playing作为新的状态。否则，case_Paused检查state是否为Paused，如果是，则采取resume_playback()动作并返回Playing。否则，case_default调用no_transition来处理对一个play事件没有向外转换的状态[⊖]。

⊖ 这个版本的process_event不会招致四次函数调用开销，我们依赖编译器的内联和优化能力使其高效。

正如你看到的那样，这些语义和上面的switch语句的语义一致。如果我们可以为一个给定的事件（event）上的所有转换（transition）生成一个case_State函数，通过遍历转换表的每一行（row），我们就可以逐步构建起正确的行为。当然，我们尚未稳操胜券，因为我们无法生成“case_State”函数，问题出在“State”所表示的名字可变部分上了——模板元程序无法生成新的标识符。然而，我们可以用如下方式将每一个状态（state）与一个单独的函数进行关联：

```
template <int State>
struct case_
{
    static int dispatch(player& fsm, int state, play const& e)
    { ... }
};
```

假如我们能够适当地填充每一个大括号里的代码，case_<Stopped>::dispatch就等价于case_Stopped，而case<Paused>::dispatch则等价于case_Paused。为了给这些函数生成函数体，我们需要State（供检查使用），一个转换动作（transition action）（供执行使用），和一个下一个状态（next state）（供切换使用）。我们可以分别在单独的模板参数中传递它们，但传递转换表整个一行（row）可能更简单，因为row的成员提供了对于那些信息（甚至更多信息）的访问。然而，如果case_不是携带一个状态值（state value）作为其单独的模板实参，那么它的命名看上去就很糟糕。让我们转而称其为event_dispatcher：

```
template<class Transition> // 转换表中的一行（row）
struct event_dispatcher
{
    typedef typename Transition::fsm_t fsm_t;
    typedef typename Transition::event event;

    static int dispatch(
        fsm_t& fsm, int state, event const& e)
    {
        if (state == Transition::current_state)
        {
            Transition::execute(fsm, e);
            return Transition::next_state;
        }
        else { ... }
    }
};
```

很方便，每一个row都为我们提供了状态机的身份（::fsm_t）和正被分派的事件（::event）。从而允许event_dispatcher比case_更具有一般性，后者会被绑定到一个具体的状态机和事件上。

为了完成event_dispatcher，我们必须填充else子句，通常这只需要调用下一个case的dispatch函数即可。如果针对下一个case的event_dispatcher是一个模板参数的话，事情就相当简单了：

```
template<
    class Transition
  , class Next
>
struct event_dispatcher
{
    typedef typename Transition::fsm_t fsm_t;
    typedef typename Transition::event event;

    static int dispatch(
        fsm_t& fsm, int state, event const& e)
    {
        if (state == Transition::current_state)
        {
            Transition::execute(fsm, e);
            return Transition::next_state;
        }
        else // 前进到链中的下一个节点
        {
            return Next::dispatch(fsm, state, e);
        }
    }
};
```

为了处理default的情形，我们将引入一个default_event_dispatcher，它带有一个dispatch函数，能调用FSM的no_transition处理器（handler）。因为派生的FSM类只给state_machine<FSM>授权了友元关系，而没有给default_event_dispatcher授权友元关系，因此，必须通过state_machine的一个成员间接地调用该处理器：

```
struct default_event_dispatcher
{
    template<class FSM, class Event>
    static int dispatch(
        state_machine<FSM>& m, int state, Event const& e)
    {
        return m.call_no_transition(state, e);
    }
};

template <class Derived>
class state_machine
{
    ...
    template <class Event>
    int call_no_transition(int state, Event const& e)
    {
```

```
            return static_cast<Derived*>(this)  // CRTP向下转型（downcast）
                        ->no_transition(state, e);
        }
        ...
};
```

现在，为了处理play事件，我们只要装配如下类型并调用其dispatch函数即可：

```
event_dispatcher<
    row<Stopped, play, Playing, &player::start_playback>
  , event_dispatcher<
    row<Paused, play, Playing, &player::resume_playback>
  , default_event_dispatcher
  >
>
```

如果你仔细地端详该类型的结构，就会发现它反映了fold算法的执行模式，以default_event_dispatcher开始，并将其“折叠（folding）”进连续的event_dispatcher特化。为了产生它，我们只要对表中的行（rows）（包含有正在分派的事件）执行fold即可：

```
// 获取transition 所关联的Event
template <class Transition>
struct transition_event
{
    typedef typename Transition::event type;
};

template<class Table, class Event>
struct generate_dispatcher
  : mpl::fold<
        mpl::filter_view<   // 选择Event触发的rows
            Table
          , boost::is_same<Event, transition_event<_1< >
        >
      , default_event_dispatcher
      , event_dispatcher<_2,_1>
    >
{};
```

最后，我们准备编写state_machine的process_event函数了！不是为每一个事件（event）类型编写重载，我们将使用一个（针对每一个事件类型模板化的）成员函数，它不过是生成dispatcher并调用其::dispatch成员而已：

```
template<class Event>
int process_event(Event const& evt)
{
    // 生成dispatcher类型
    typedef typename generate_dispatcher<
```

```
            typename Derived::transition_table, Event
        >::type dispatcher;
        // 分派事件（event）
        this->state = dispatcher::dispatch(
            *static_cast<Derived*>(this)        // CRTP向下转型（downcast）
          , this->state
          , evt
        );

        // 返回新状态（state）
        return this->state;
    }
```

再一次请注意，我们利用了奇特的递归模板模式（Curiously Recurring Template Pattern）来提供功能——该功能依赖于在基类的成员函数中知道派生类的完整类型信息。上面的static_cast允许dispatcher将transition_table中的Derived成员函数指针应用到*this上。

至此，state_machine就只剩下一丁点儿东西未实现了。我们需要一个state成员，以及一个对该成员进行初始化的构造器：

```
...
 protected:
   state_machine()
     : state(Derived::initial_state)
   {}

 private:
   int state;
...
```

提供一个默认的no_transition处理器是好事，毕竟，一个需要不同行为的用户总是可以在他的派生类中编写一个no_transition函数：

```
...
 protected:
    template <class Event>
    int no_transition(int state, Event const& e)
    {
        assert(false);
        return state;
    }
...
```

在这本书的配书光碟中，examples/chapter11/example16.cpp包含有我们所探索的方案的完整实现品。

11.6 分析

至此，你对DSEL的开发过程应该有一个很好的认知了。这些步骤反映了我们在第10章中用

于分析每一个领域特定的语言的那些步骤。

1. 识别领域抽象。

2. 试验以代码来表示这些抽象。

3. 构建一个原型。

4. 迭代（精化，反复重构）。

那么，这一次设计的方案如何呢？我们达到了预期设计目标了吗？

1. 互操作能力。获得了与其他DSL的互操作能力，因为我们是以程序库的形式创建DSL的，换句话说，它是一个领域特定的嵌入式语言。我们可以在事件（event）中嵌入来自其他领域程序库的类型，在转换动作（transition actions）的内部调用来自其他领域的任意函数，或者从使用其他DSEL编写的代码中操作FSM。

2. 声明性。回顾我们的player实现，作者编写的大多数代码都是位于转换表自身，并且在player声明中的几乎每一样东西对于其含义来说都是必不可少的。它看上去像是从领域语言到C++的直接转换。此外，可以在不改动状态机声明的前提下完全替代框架的实现。你可以到http://boost-consulting.com/mplbook/examples/player2.cpp下载一个state_machine实现品，它以O(1)时间复杂度在一个函数指针的静态表进行查找，从而实现分派（dispatches）。

3. 表达力。在player中声明的STT酷似我们期望中的表，在格式化习惯方面与我们一般使用的极为相似。

4. 效率。为process_event生成的代码避免了所有运行期分派（除了切换当前的状态外），并且不需要任何内存访问或表查找（table lookups），因为event_dispatcher使用编译期常量来进行比较。这个设计是高效的，因为我们只要有可能就“不留情面地”使得每一样东西都位于编译期元数据世界。

作者采用两款不同的编译器分析了这个例子的的汇编语言输出，生成的代码看来可以与手工编码的状态机媲美。如果你打算在一个各个周期都要考虑的系统中使用这个框架，你可能希望将这个例子扔到你的目标编译器中并观察结果。你可能还希望扩充STT，加入更多的事件（events）和转换（transitions），来看看代码的效率是否与状态机的大小有着良好的比例关系。

5. 静态类型安全。该框架是相当类型安全的。在整个系统中，只有两个static_casts（process_event和call_no_transition之一），并且潜在的损害是受到限制的，因为只有当Derived真正派生于state_machine时才能通过编译。

6. 可维护性。只要创建新的类型，就可以向系统添加新事件（events）。新状态（states）可以用类似的方式添加进来，并通过扩充states enum即可。该enum甚至还可以定义在player的外面，如果我们够小心的话。但这种方式对降低耦合没什么大的帮助，因为转换表必须包含状态（state）的名字，并且在FSM的声明中必须是可见的。转换（Transitions）也易于添加进来，只要在transition_table中写入新的rows即可。

有一点值得一提，就是当FSM演化时维护表的可视化布局的代价。如果我们希望紧密地匹配领域抽象的话，这种代价看来是不可避免的。尽管如果维护代价变得太高我们有“置严格的布局于不顾”的灵活性，然而试验表明，表的表示法对于其易理解性有着极大的影响，因此我

们不愿轻率地迈出这一步。

7. 可扩充性。从我们已经看到的东西来评价框架的可扩充性有点儿难。但是在这里我们可以说的是，该设计看上去具有足够的模块性，从而使添加新特性相当容易。如果你打算做本章的一些练习的话，你将有机会体会这到底有多么容易。归因于DSL的声明性，我们至少可以肯定能在不破坏现有用户代码的前提下添加新特性。

11.7 语言方向

尽管迄今为止C++是构建高效DSEL最适合的语言，然而仍然有许多改善的余地。正如你已经看到的那样，在当前的语言中你可以实现的DSEL是令人惊奇的，但它们同时也需要付出大量的工作，而这样的工作量只有当你确实计划花费大量的时间工作于DSEL的问题领域时才无可厚非。此外，尽管C++的运算符重载规则丰富且灵活，然而我们常常不得不勉强接受不那么完美的语法。在通用编程领域中看上去自由的东西，对于方便地表达任意的专用领域的语法而言，就显得不那么自由。

当然，事情并非一定要是这样。尽管C++可能永远都不允许任意的语法扩充，然而对语言进行一些小的改变将会大幅改善DSEL的编写。我们在这一章看到的问题之一是，尽管C++的运行期语法丰富得难以置信，然而一旦我们跨越边界进入运行期——通过传递一个常量（在我们的例子中，即一个成员函数指针）给一个函数，就不可能再在编译期世界把那个常量当元数据使用了。比如说，如果我们能够扩充语言的能力以便进行“常量折叠（constant folding）”，那就有可能在需要纯元数据的上下文中利用C++丰富的运行期语法了[n 1521]。

11.8 练习

11-0. 能够根据转换（transitions）的起始状态（start state）对其进行分组是有好处的，这样每一个起始状态都只需编写一次就可以了。设计这样的一个分组化的表示，并且修改FSM框架的设计来支持它。评估你的改动在减少冗余、刻板的编码方面所取得的成功。

11-1. 我们并非马上就放弃基于表达式模板（expression template-based）的设计。如何重新夺回因通过传递函数指针而导致的效率损失？（提示：它们必须作为模板实参进行传递）使用这项技术改写你喜爱的表达式模板DSEL语法，并评估其作为一个DSEL的成功之处。

11-2. 实现并测试我们在11.4.1节早期探索但很快又抛弃的基于表达式模板的（expression-template-based）FSM DSEL。评估其易用性和效率的折中。

11-3. 评估实现下列基于表达式模板的（expression-template-based）FSM DSEL的可能性：

```
player()
{
        Stopped[
            play        => Playing | &player::start_playback
          , open_close  => Open    | &player::open_drawer
          ]
      ,
```

```
        Open[
            open_close => Empty    | &player::close_drawer
          ]

      // ...
      ;
  }
```

基于你的评估，解释为何这种语法不可行，或如果可行，实现一个原型来示范它。

11-4. 扩充FSM实现以支持可选的每状态进入（entry）和退出（exit）动作。

11-5. 转换哨位（Transition guards）是你可以赋给某些转换（transitions）的附加判断式，从而可以根据一些条件来禁用或启用该转换（transitions）。尽管形式上有点累赘[⊖]，但它们有助于减少FSM的大小和复杂性，有时效果还很明显，因此通常还是需要的。为本章的例子设计并实现可选的转换哨位（Transition guards）支持。

11-6. 扩充FSM实现来支持“catch-all转换”，使以下任何一个语句都是成立的：

```
// 无论当前状态是什么，都允许“reset_event”
// 触发一个到“initial_state”的转换
row< _, reset_event, initial_state, &self::do_reset >

// 任何接收到“error”状态的事件
// 触发一个到“finished”的转换
row< error, _, finished, &self::do_finish >

// 任何接收到任何状态的事件
// 触发一个到“done”的转换
row< _, _, done, &self::do_nothing >
```

选择并实现一个确定性的模式，用来处理带有重叠条件的转换。

11-7*. 扩充FSM实现以支持嵌套的（复合）状态（states），一个可能的设计草案如下所示：

```
class my_fsm
    : fsm::state_machine< my_fsm >
{
    // ...
    struct ready_to_start_;
    typedef submachine<ready_to_start_> ready_to_start;

    struct transition_table : mpl::vector<
          row< ready_to_start, event1, running, &self::start >
        , row< running,        event2, Stopped, &self::stop >
        // ...
        > {};
};
```

⊖ 任何使用转换哨位的有限状态机总可以被转换为一个等价的不使用转换哨位的“纯”FSM。

```
// 该翻译单元（translation unit）中的另外某个地方

template<>
struct my_fsm::submachine<ready_to_start_>
  : state_machine< submachine<ready_to_start_> >
{
    // 状态（states）
    struct ready;
    struct closed;
    struct recently_closed;

    struct transition_table : mpl::vector<
          row< ready,  event3, closed,        &self::close >
        , row< closed, event4, recently_closed >
        // ...
        > {};
};
```

11-8. 对于一个给定的事件，我们的分派代码线性地查找状态表找出外向转换（outgoing transitions）。在最坏情况下，时间复杂度为O(S)，其中S为FSM中的状态（states）的总数。在配书光碟的代码例子目录中，player2.cpp示范了一个state_machine，它以O(1)时间复杂度查找一个静态的函数指针表。然而，这会导致运行期内存访问和函数指针间接调用开销。实现并测试第三种分派模式，它通过产生$O(\log_2 S)$时间复杂度的二分查找（binary search）从而避免所有这些缺点。

附录A　预处理元编程简介

A.1　动机

即便有模板元编程和Boost 元编程库的强大能力可供我们自由支配，一些C++编程任务仍然需要大量地重复样板式的代码。在第5章中，我们在实现tiny_size时就看过一个例子：

```
template <class T0, class T1, class T2>
struct tiny_size
  : mpl::int_<3> {};
```

姑且不谈上面的主模板（primary template）的参数列表中的重复模式（repeated pattern），下面还有三个局部特化版本（partial specializations），它们更是遵循一个可预知的模式：

```
template <class T0, class T1>
struct tiny_size<T0,T1,none>
  : mpl::int_<2> {};

template <class T0>
struct tiny_size<T0,none,none>
  : mpl::int_<1> {};

template <>
struct tiny_size<none,none,none>
  : mpl::int_<0> {};
```

在这个例子中，只有少量代码具有这种“机械味”，然而，如果我们实现的是一个“large”而非“tiny”，那么具有“机械味”的代码量就可能相当多。当一个模式的实体数目超过两、三个时，手工书写很容易出错，可能更重要的是，代码会变得难以阅读，因为代码中重要的抽象部分其实正是那个模式，而非该模式的各个体实体。

A.1.1　代码生成

不是采用手工编写，“机械模样”的代码确实应该被“机械地”生成。对于程序库的作者，要想编写一个可以生成遵循特定模式的代码片段的程序，他面临两种选择：一是直接将预生成的源代码文件随程序库发布，二是发布生成器（generator）本身。两种做法都有缺点。如果客户只得到了预生成的源代码，那么他们就被限制在程序库作者所生成的东西上了。经验表明，可能今天一个模式有三个实体就够他们用的了，但明天可能就有人需要四个实体！另一方面，如果客户得到了生成器程序（generator program），那么他们还需要一个可以用来执行该生成器的程序（例如解释器（interpreters）），并且他们还必须将生成器整合到生成（build）过程中，除非……

A.1.2 进入预处理器

...除非生成器是一个预处理元程序。C/C++预处理器（preprocessors）可以在编译的预处理阶段执行复杂的程序[⊖]，尽管它们并非为此目的而设计。用户可以通过#define（在代码中）或-D选项（编译器的命令行中）来控制代码的生成过程，从而使前述的生成（build）整合过程不费吹灰之力[⊜]。例如，我们可以将上面的tine_size主模板参数化如下：

```
#include <boost/preprocessor/repetition/enum_params.hpp>

#ifndef TINY_MAX_SIZE
#  define TINY_MAX_SIZE 3  // maximum size的默认值为3
#endif

template <BOOST_PP_ENUM_PARAMS(TINY_MAX_SIZE, class T)>
struct tiny_size
  : mpl::int_<TINY_MAX_SIZE>
{};
```

要测试这个元程序，你可以以“预处理（preprocessing）”模式（通常使用-E选项）运行编译器，同时确保Boost的根目录已放在你的编译器的#include路径中。例如[⊝]：

```
g++ -P -E -Ipath/to/boost_1_32_0 -I. test.cpp
```

有了适当的元程序，我们不但可以调整tiny_size的参数的个数，还可以调整整个tiny实现品的最大尺寸——只要将TINY_MAX_SIZE #define为适当的值即可。

Boost Preprocessor程序库[MK04]在预处理元编程中充当的角色与MPL在模板元编程中充当的角色类似。它提供了一个高阶组件框架（framework）（例如BOOST_PP_ENUM_PARAMS），使元编程任务变得容易完成，否则元编程可能会令人很痛苦。在本附录中，我们并不去深究预处理器工作的本质细节，或是预处理器元编程的一般原则，或是BPL程序库如何工作的诸多细节，而是在一个较高层次上为你展示足够的内容，使你能够有效地使用它并且使自己有能力探索其余的部分。

A.2 预处理之基础抽象

我们从第2章开始讨论模板元编程，描述了它的元数据（可能的模板实参）和元函数（类模板）。在这两个基础抽象之上，我们建立起本书其余部分讨论的编译期计算的大局观。在这一节，我们将为预处理器元编程铺设一个类似的基础。这里介绍的某些东西对于你可能只是一个回顾，但在深入细节之前识别这些基本概念还是很重要的。

A.2.1 预处理标记（Preprocessing Tokens）

预处理器中的基本数据单元是预处理标记（preprocessing token）。预处理标记与你在C++中

⊖ 如预处理元程序。

⊜ 因为预处理元程序的解释器就是预处理器。

⊝ GCC的-P选项禁止在预处理输出中包含源文件和行号标记。

使用的标识符（identifier）、运算符符号（operator symbol）以及字面常量（literal）等标记大致对应。从技术上说，预处理标记（preprocessing tokens）和正规的标记（regular tokens）是有些区别的（详见C++标准第2节），但就目前我们的讨论来说可以暂且忽略。事实上，下文不区分这两个术语的使用。

A.2.2 宏

预处理宏有两种风格。类似对象的宏可以如下方式进行定义：

```
#define identifier replacement-list
```

这里identifier是宏的名字，replacement-list是零个或多个标记的序列。后继的程序文本中所有出现identifier的地方都会被预处理器展开为相应的replacement-list。

另一种是类似函数的宏，它好比“预处理阶段的元函数”，定义方式如下：

```
#define identifier(a1, a2, ... an) replacement-list
```

其中每一个ai 标识符都命名了一个宏形参（macro parameter）。当宏名字出现在后继的程序文本中并且后跟适当的实参列表时（argument list），它将被扩充为replacement-list，而且其中每个出现参数的地方都会被替换为用户给出的实参[⊖]。

A.2.3 宏实参

定义

宏实参是以下两种预处理标记（Preprocessing tokens）的非空序列：

- 除逗号或小括号之外的预处理标记。
- 由一对小括号包围的一组预处理标记。

这个定义对预处理器元编程的重要性不可低估。首先注意，以下两种tokens地位特殊：

```
, ( )
```

因此，一个宏实参不能包含没有配对的小括号，或者没有被小括号包围的逗号。例如，以下跟在FOO定义后面的两行代码都是形式不良的（ill-formed）：

```
#define FOO(X) X // 一元标识符宏
FOO(,)           // 未加小括号的逗号，或两个空实参
FOO())           // 未配对的小括号，或缺少实参
```

同时也要注意，下面几种标记都不是处于特殊的地位，预处理器对大括号、方括号以及尖括号的配对情况一无所知：

```
{ } [ ] < >
```

所以，下面两行代码也是形式不良的：

```
FOO(std::pair<int, long>)                    // 被解释为以“,”分隔的两个实参
```

⊖ 关于宏展开是如何工作的，我们忽略了很多细节，建议你花几分钟看看C++标准16.3节，那里以直观的术语描述了它的工作过程。

```
FOO({ int x = 1, y = 2; return x+y; })     // 被解释为以“,”分隔的两个实参
```

如果加上一对冗余的小括号将传递的参数包围起来，代码就正确了：

```
FOO((std::pair<int,int>))                  // 一个实参
FOO(({ int x = 1, y = 2; return x+y; }))   // 一个实参
```

然而，鉴于逗号的地位的特殊性，在不了解一个宏实参包含多少以逗号分隔的标记序列的情况下，是不可以随便去掉小括号的[⊖]。如果你写了一个宏，希望它能够接收包含任意多个逗号的实参，那么对于使用该宏的用户来说，可以将实参用小括号括起来，并将其中以逗号分隔的标记序列数目作为一个附加参数传入，或将同样的信息编码到一种预处理器数据结构中（本附录后面对此有讨论）。

A.3 Preprocessor程序库的结构

深入考察Boost Preprocessor程序库已经超出了本书的范围，这里我们将给你深入了解该程序库的“工具”。为此，你需要使用Preprocessor程序库的电子文档，参见你的Boost安装的libs/preprocessor子目录中的index.html文件。

打开后，在浏览器视窗的左边你会看到索引，点击其中的“Headers”链接，你会看到boost/preprocessor/目录的结构。程序库的大多数头文件都根据有关功能被组织在不同的子目录中。顶层的目录仅仅包含一些通用的宏的头文件，以及对应每个子目录的头文件（这种头文件仅仅把相应子目录中的头文件都包含进去。例如，boost/preprocessor/selection.hpp包含了“selection”子目录中的两个头文件min.hpp和max.hpp）。不对应任何子目录的头文件一般声明了一个与文件名同名的宏（无文件扩充名，有BOOST_PP前缀）。例如，boost/preprocessor/selection/max.hpp声明了BOOST_PP_MAX。

你还会注意到，通常一个头文件会声明一个附加的宏，它以_D、_R或_Z为后缀[⊖]。例如，boost/preprocessor/selection/max.hpp中也声明了BOOST_PP_MAX_D。在本附录中我们会忽略这些宏。倘若有朝一日你想弄明白它们是如何优化预处理速度的，可以参考电子文档“reentrancy”部分的Topics小节。

A.4 Preprocessor程序库的基础抽象

在本节中，我们将讨论Preprocessor程序库的基本抽象，并分别给出一些简单的例子。

A.4.1 重复

我们可以使用BOOST_PP_ENUM_PARAMS来重复性地生成class T0、class T1……class Tn，这就是一般概念“横向重复（horizontal repetition）”的一个具体例子。该程序库中还有一个纵向重复（vertical repetition）概念，我们等一会儿介绍。横向重复宏可以在程序库的repetition/子

⊖ C99的预处理器可以并且能做得更多（借助于变长宏（variadic macros））。C++标准委员会在下一个标准版本中倾向于采纳C99中的预处理器扩充功能。

⊖ 如果出现后缀为_1ST、_2ND或_3RD的宏，它们也应该被忽略，但是原因不同：它们已废弃，不久将从程序库中移除。

目录中找到。

A.4.1.1 横向重复

要使用横向重复生成tiny_size的特化，我们可以这样写：

```
#include <boost/preprocessor/repetition.hpp>
#include <boost/preprocessor/arithmetic/sub.hpp>
#include <boost/preprocessor/punctuation/comma_if.hpp>

#define TINY_print(z, n, data) data

#define TINY_size(z, n, unused)                                    \
  template <BOOST_PP_ENUM_PARAMS(n, class T)>                      \
  struct tiny_size<                                                \
      BOOST_PP_ENUM_PARAMS(n,T)                                    \
      BOOST_PP_COMMA_IF(n)                                         \
      BOOST_PP_ENUM(                                               \
          BOOST_PP_SUB(TINY_MAX_SIZE,n), TINY_print, none)         \
  >                                                                \
    : mpl::int_<n> {};
BOOST_PP_REPEAT(TINY_MAX_SIZE, TINY_size, ~)

#undef TINY_size
#undef TINY_print
```

代码生成过程从BOOST_PP_REPEAT开始，BOOST_PP_REPEAT是一个高阶宏（higher-order macro），它会重复地调用它的第二个参数所指定的宏（也就是TINY_size）。它的第一个参数指定了重复调用的次数。第三个参数可以是任意数据，它会原封不动地传给正被调用的宏。在这里，TINY_size并没有使用该数据，所以选择传递“~”具有任意性[⊖]。

TINY_size宏每次被BOOST_PP_REPEAT调用时都会生成一个tiny_size的不同特化。TINY_size接收三个参数：

- z与前面提到的_Z后缀宏有关。若非出于优化的目的，我们永远都不需使用它。眼下我们大可以忽略它。
- n表示重复的索引（即当前为第几次重复）。在对TINY_size的重复调用过程中，n依次为0、1、2……
- unused，对于这个例子，在每次重复中都是“~”。通常，BOOST_PP_REPEAT会将用户传给它的实参原封不动地转发给被调用的宏（例如TINY_size）。

因为像TINY_size这样的宏的replacement-list有好几行，所以除了最后一行，其他行都以反斜杠“\”结尾。其开头的几行调用BOOST_PP_ENUM（我们在主模板中已经使用了它）来生成

⊖ 其实“~”并不是一个完全任意的选择，“@”和“$”本该是不错的选择，只可惜从技术上而言，它们并不属于C++实现品必须支持的基本字符集。而像“ignored”这样的标识符则可能本身就是宏，会被展开，从而导致意想不到的结果。

以逗号分隔的模板参数列表，所以，对TINY_size的每次调用都会生成类似如下的代码[⊖]：

```
template <class T0, class T1, ... class Tn-1>
struct tiny_size<
    T0, T1, ... Tn-1
    更多……
>
  : mpl::int_<n> {};
```

如果BOOST_PP_COMMA_IF接收的数值型实参不为零，就会生成一个逗号，否则为空。当n为0时，BOOST_PP_ENUM_PARAMS(n,T)什么也不会生成，而紧跟在它后面的BOOST_PP_COMMA_IF(n)也为空，因为“<”后面直接跟“,”是非法的。

接下来的一行代码使用BOOST_PP_ENUM生成TINY_MAX_SIZE-n个以逗号分隔的“none”。除了每次重复都以逗号分隔之外，BOOST_PP_ENUM和BOOST_PP_REPEAT没什么区别，所以它的第二个参数（这里是TINY_print）必须和TINY_size具有相同的签名（signature）。在本例中，TINY_print忽略当前重复次数（索引）n，总是简单地将自身替换为第三个参数，也就是“none”。

BOOST_PP_SUB实现了标记的减法。虽然预处理器本身可以对普通的算术表达式进行评估：

```
#define X 3
...
#if X - 1 > 0  // OK
  任何东西
#endif
```

然而预处理元程序却只能操作标记，认识这一点非常重要。通常，当Preprocessor程序库中的某个宏需要接收一个数值参数时，该数值参数必须以单个标记的形式进行传递。如果在上面的例子中，我们写的是TINY_MAX_SIZE-n，而不是BOOST_PP_SUB(TINY_MAX_ SIZE,n)，那么BOOST_PP_ENUM在每一次调用中将包含三个标记：一是3-0，然后是3-1，最后是3-2。然而，BOOST_PP_SUB则能够生成单个的标记结果，对于BOOST_PP_SUB(3,0)，其生成的是3，BOOST_PP_SUB(3,1)是2，再后来是1——在连续的重复中。

命名约定（Naming Conventions）

注意，TINY_size和TINY_print在使用后立即被#undef了，其间没有任何被#include的头文件。所以它们可以看做“local（局部的）”宏定义。因为预处理器无视作用域的存在，所以为了防止名字冲突，仔细地选择名字是非常必要的。我们建议使用PREFIXED_lower_ case这种形式作为局部宏的名字，而PREFIXED_UPPER_CASE作为全局宏的名字。惟一例外的是仅有一个小写字母的名字，你可以用它作为局部宏的名字：没有任何其他头文件会#define一个全局的（global）单字母小写的宏——那是非常糟糕的风格。

⊖ 注意，续行符“\”及其后面的换行符会被预处理器移除，所以在预处理后的输出中，结果代码实际上只有一行。

A.4.1.2 纵向重复

如果你将前面的例子进行预处理，结果将是很长的一行，包含如下内容：

```
template <> struct tiny_size< none , none , none > : mpl::int_<0>
 {}; template < class T0> struct tiny_size< T0 , none , none > :
mpl::int_<1> {}; template < class T0 , class T1> struct tiny_size
< T0 , T1 , none > : mpl::int_<2> {};
```

所有重复模式的实体都生成于预处理输出结果中的同一行，这就是横向重复的特点。对于某些任务，比如生成tiny_size主模板，这是相当合适的。然而，在这种情况下，这种做法至少存在两点不足：

1. 如果不将结果代码手工重新编排，则很难验证我们的元程序做了正确的事情。

2. 嵌套的横向重复在不同的预处理器下的效率差异较大。对于tiny_size，横向重复生成的每个特化都包含另外三个横向重复：两个对BOOST_PP_ENUM_PARAMS的调用，一个对BOOST_PP_ENUM的调用。当TINY_MAX_SIZE为3时，你也许不会关心这个问题。但是，在目前仍在使用的预处理器中，至少有一个在TINY_MAX_SIZE达到8时编译速度会显著地变慢[⊖]。

解决这些问题的方案自然是纵向重复（vertical repetition）。纵向重复可以跨越多行生成具有特定模式的实体。Preprocessor程序库提供了两种方式的纵向重复：局部迭代（local iteration）和文件迭代（file iteration）。

局部迭代

在我们的例子中示范局部迭代最直接的办法就是将对BOOST_PP_REPEAT的调用替换为以下调用：

```
#include <boost/preprocessor/iteration/local.hpp>

#define BOOST_PP_LOCAL_MACRO(n)    TINY_size(~, n, ~)
#define BOOST_PP_LOCAL_LIMITS      (0, TINY_MAX_SIZE - 1)
#include BOOST_PP_LOCAL_ITERATE()
```

局部迭代会重复调用用户自定义的带有特殊名字的宏：BOOST_PP_LOCAL_MACRO，它的参数是迭代索引（iteration index）。因为我们已经定义了TINY_size，所以只需定义BOOST_PP_LOCAL_MACRO来调用它即可。迭代索引区间则由另一个用户自定义的宏BOOST_PP_LOCAL_LIMITS给出，该宏必须展开为以括号包围的一对整数值，表示传递给BOOST_PP_LOCAL_MACRO的索引值闭区间[⊜]。注意，这里BOOST_PP_LOCAL_LIMITS的数值实参可以包含由多个标记组成的整型表达式[⊝]，这在Preprocessor程序库中是比较罕见的，这里是其一。

最后指出，重复过程是通过包含（#include）调用BOOST_PP_LOCAL_ITERATE的结果而发起的，该宏最终被替换为Preprocessor程序库中的一个文件。你会惊讶地发现，与嵌套的横向

⊖ 这句话的潜台词是：其他预处理器可以轻松地处理很多（如256*256）个嵌套重复，而不存在任何速度方面的问题。

⊜ 即该闭区间内的每个整数被依次传给BOOST_PP_LOCAL_MACRO。

⊝ 例如，TINY_MAX_SIZE —1 就包含了三个标记。

重复相比，许多预处理器处理重复的文件包含要更快，然而这的确是事实。

如果我们将新的例子扔给预处理器，就会得到如下结果，分别位于不同的三行：

```
template <> struct tiny_size< none , none , none > : mpl::int_<0>
 {};

template < class T0> struct tiny_size< T0 , none , none > : mpl::
int_<1> {};

template < class T0 , class T1> struct tiny_size< T0 , T1 , none
> : mpl::int_<2> {};
```

在可验证性方面，这是一个很大的改善，但仍不够理想。随着TINY_MAX_SIZE的增大，验证模式生成的实体（代码）是否符合我们的意思会变得愈来愈困难。如果在输出中可以分解出更多的行那我们将会得到识别性更好的形式。

到目前为止我们使用的两种重复方法都还有另外一个缺点，虽然在这个例子中没有得到体现。如果tiny_size有一个我们希望对之进行调试（debug）的成员函数会发生些什么。如果你曾试图使用调试器（debugger）单步跟踪一个由预处理宏生成的函数，你就会明白那是一种多么让人沮丧的经历，最好的情形是：调试器停在宏最终被调用的那一行代码上[㊀]，所以你无法知道那里到底生成了什么代码。更糟糕的是，一旦体现在调试器上，生成的函数中每一条语句都挤在同一行上！

文件迭代

显而易见，调试能力依赖于保持生成的代码和描述代码模式的源文件中的代码[㊁]之间的关联性。文件迭代方式通过重复包含（#include）相同的源文件[㊂]来生成符合某个模式的代码实体。文件迭代对调试能力的影响类似于模板：尽管在调试器中“样板”代码的不同实体出现在同一行上，但我们总算能够单步跟踪函数的源代码了。

要在我们的例子中应用文件迭代，我们可以将前面的局部迭代的代码以及TINY_size的定义替换为如下代码：

```
#include <boost/preprocessor/iteration/iterate.hpp>
#define BOOST_PP_ITERATION_LIMITS (0, TINY_MAX_SIZE - 1)
#define BOOST_PP_FILENAME_1       "tiny_size_spec.hpp"
#include BOOST_PP_ITERATE()
```

BOOST_PP_ITERATION_LIMITS遵从与BOOST_PP_LOCAL_LIMITS所采取的相同的模式，都允许我们指定一个迭代索引闭区间。BOOST_PP_FILENAME_1指定了被重复包含（#include）的文件名（等一会儿我们就会向你展示该文件）。后缀_1指出这是文件迭代的第一个嵌套层[㊃]。如果我们需要在tiny_size_spec.hpp中再一次调用文件迭代，我们就得使用

㊀ 而不是该宏定义的地方。——译者注

㊁ 即“样板”代码。——译者注

㊂ “样板”代码源文件，每次包含生成不同的代码实体。——译者注

㊃ 即第一个“样板”文件。——译者注

BOOST_PP_FILENAME_2了。

tiny_size_spec.hpp的内容你应该熟悉，其绝大部分和TINY_size的定义是一样的，只是去掉了每行末尾的“\”：

```
#define n BOOST_PP_ITERATION()

template <BOOST_PP_ENUM_PARAMS(n, class T)>
struct tiny_size<
    BOOST_PP_ENUM_PARAMS(n,T)
    BOOST_PP_COMMA_IF(n)
    BOOST_PP_ENUM(BOOST_PP_SUB(TINY_MAX_SIZE,n), TINY_print, none)
>
  : mpl::int_<n> {};

#undef n
```

程序库将当前迭代索引（iteration index）通过BOOST_PP_ITERATION()的结果传递给我们。n不过是一个用于减少语法噪音的便利的局部宏[⊖]。注意我们在tiny_size_spec.hpp里面并没有使用“包含哨卫（#include guards）”，这是因为我们需要多次处理该头文件[⊖]。

现在，预处理结果应该能够保留模式的行结构（即“样板”代码的结构）了，对于较大的TINY_MAX_SIZE值也更容易验证生成的代码实体的正确性了。例如，当TINY_MAX_SIZE为8时，下面的代码摘自GCC预处理阶段的输出：

```
...
template < class T0 , class T1 , class T2 , class T3>
struct tiny_size<
    T0 , T1 , T2 , T3
    ,
    none , none , none , none
>
  : mpl::int_<4> {};

template < class T0 , class T1 , class T2 , class T3 , class T4>
struct tiny_size<
    T0 , T1 , T2 , T3 , T4
    ,
    none , none , none
>
  : mpl::int_<5> {};
.....
```

⊖ 因为用完后就被#undef了。——译者注

⊖ 因为我们希望每次都由其中的“样板”代码生成一份不同的代码实体。——译者注

自迭代（Self-Iteration）

对于文件重复（文件迭代），每当我们表达一个哪怕是非常简单的新代码模式（即“样板”代码）时，也必须创建一个全新的文件（例如tiny_size_spec.hpp），这不够方便。幸运的是，BPL提供了一个BOOST_PP_IS_ITERATING宏，允许我们将模式（“样板”代码）直接放在引发迭代的文件中[⊖]。如果我们正处于迭代中，BOOST_PP_IS_ITERATING就会扩充为一个非零的值，我们可以利用该值来选择引发迭代的部分或提供了重复模式的部分（即“样板”代码部分）。下面为示范自迭代的完整的tiny_size.hpp文件。特别要注意TINY_SIZE_HPP_INCLUDED“#include哨卫”的使用以及使用的位置。

```
#ifndef BOOST_PP_IS_ITERATING

#  ifndef TINY_SIZE_HPP_INCLUDED
#    define TINY_SIZE_HPP_INCLUDED

#    include <boost/preprocessor/repetition.hpp>
#    include <boost/preprocessor/arithmetic/sub.hpp>
#    include <boost/preprocessor/punctuation/comma_if.hpp>
#    include <boost/preprocessor/iteration/iterate.hpp>

#    ifndef TINY_MAX_SIZE
#      define TINY_MAX_SIZE 3 // default maximum size is 3
#    endif

// 主模板
template <BOOST_PP_ENUM_PARAMS(TINY_MAX_SIZE, class T)>
struct tiny_size
  : mpl::int_<TINY_MAX_SIZE>
{};

// 生成特化
#    define BOOST_PP_ITERATION_LIMITS (0, TINY_MAX_SIZE - 1)
#    define BOOST_PP_FILENAME_1    "tiny_size.hpp" // 该文件
#    include BOOST_PP_ITERATE()

#  endif // TINY_SIZE_HPP_INCLUDED

#else // BOOST_PP_IS_ITERATING

#  define n BOOST_PP_ITERATION()

#  define TINY_print(z, n, data) data
```

⊖ 即直接在“样板”代码文件里引发迭代。——译者注

```
// 特化模式
template <BOOST_PP_ENUM_PARAMS(n, class T)>
struct tiny_size<
    BOOST_PP_ENUM_PARAMS(n,T)
    BOOST_PP_COMMA_IF(n)
    BOOST_PP_ENUM(BOOST_PP_SUB(TINY_MAX_SIZE,n), TINY_print, none)
>
  : mpl::int_<n> {};

# undef TINY_print
# undef n

#endif // BOOST_PP_IS_ITERATING
```

更多的信息

限于篇幅，这里我们只能讲一小部分文件迭代的内容。要想了解更多的细节，我们建议你深入阅读Preprocessor程序库中BOOST_PP_ITERATE以及与之相关的宏电子文档。还有一点要特别注意，对于“代码重复”，没有哪种技术是“绝对”好的：你的选择将取决于便利性、可验证性、可调试性、编译速度以及你自己对“逻辑一致性”的感觉。

A.4.2 算术操作、逻辑操作以及比较操作

正如我们前面提到的Preprocessor程序库中许多（宏的）接口都要求单标记数值参数（single-token numeric arguments），并且当这些数字用于数学计算时，直截了当地使用算术表达式是不行的[⊖]。在tiny_size例子中，我们使用BOOST_PP_SUB对两个数字标记（numeric tokens）进行减法运算。Preprocessor程序库的arithmetics/子目录中包含有一套用于非负整型标记（integral token）算术运算的宏，如表A.1所示。

表A.1 Preprocessor程序库中的算术操作

表达式	单标记结果的值（Value of Single Token Result）
BOOST_PP_ADD(x,y)	x + y
BOOST_PP_DEC(x)	x - 1
BOOST_PP_DIV(x,y)	x / y
BOOST_PP_INC(x)	x + 1
BOOST_PP_MOD(x,y)	x % y
BOOST_PP_MUL(x,y)	x * y
BOOST_PP_SUB(x,y)	x - y

logical/子目录则包含方便的布尔逻辑运算，如表A.2所示。表A.3是更有效率的操作，但它们的操作数必须是0或1（即一个二进制位（Bit））。

⊖ 比如“A+B*C”是不行的。——译者注

表A.2 Preprocessor程序库中的整型逻辑运算

表达式	单标记结果的值（Value of Single Token Result）
`BOOST_PP_AND(x,y)`	`x && y`
`BOOST_PP_NOR(x,y)`	`!(x || y)`
`BOOST_PP_OR(x,y)`	`x || y`
`BOOST_PP_XOR(x,y)`	`(bool)x != (bool)y ? 1 : 0`
`BOOST_PP_NOT(x)`	`x ? 0 : 1`
`BOOST_PP_BOOL(x)`	`x ? 1 : 0`

表A.3 Preprocessor程序库中的位逻辑运算

表达式	单标记结果的值（Value of Single Token Result）
`BOOST_PP_BITAND(x,y)`	`x && y`
`BOOST_PP_BITNOR(x,y)`	`!(x || y)`
`BOOST_PP_BITOR(x,y)`	`x || y`
`BOOST_PP_BITXOR(x,y)`	`(bool)x != (bool)y ? 1 : 0`
`BOOST_PP_COMPL(x)`	`x ? 0 : 1`

最后，comparison/子目录提供整型标记比较运算，如表A.4所示：

表A.4 Preprocessor程序库中的比较运算

表达式	单标记结果的值（Value of Single Token Result）
`BOOST_PP_EQUAL(x,y)`	`x == y ? 1 : 0`
`BOOST_PP_NOT_EQUAL(x,y)`	`x != y ? 1 : 0`
`BOOST_PP_LESS(x,y)`	`x < y ? 1 : 0`
`BOOST_PP_LESS_EQUAL(x,y)`	`x <= y ? 1 : 0`
`BOOST_PP_GREATER(x,y)`	`x > y ? 1 : 0`
`BOOST_PP_GREATER_EQUAL(x,y)`	`x >= y ? 1 : 0`

由于通常我们需要在若干可用的比较操作中选择一种，所以了解这一点也许很有意义：BOOST_PP_EQUAL和BOOST_PP_NOT_EQUAL的时间复杂度为O(1)，其他比较操作通常相对慢一些。

A.4.3 控制结构

在control/子目录中，Preprocessor程序库提供了一个BOOST_PP_IF(c,t,f)宏，它所扮演的角色和mpl::if_类似。为了探究“控制”的含义，我们将为一个泛型函数对象框架Boost Function程序库[⊖]生成代码。boost::function为每种不同参数个数（它所支持的参数个数的上限由一个宏控制）的函数类型都准备了一份局部特化版本：

```
template <class Signature> struct function;    // 主模板
```

⊖ 第9章中我们讨论类型擦除（type erasure）时，曾略微谈到过Boost Function的设计。参见配书光碟中boost_1_32_0/libs/function/index.html处的Function library文档，以便了解更多的信息。

```
template <class R>                                          // arity = 0
struct function<R()>
  定义略……

template <class R, class A0>                                // arity = 1
struct function<R(A0)>
  定义略……

template <class R, class A0, class A1>                      // arity = 2
struct function<R(A0,A1)>
  定义略……

template <class R, class A0, class A1, class A2>            // arity = 3
struct function<R(A0,A1,A2)>
  定义略……

等等
```

前面我们已经讲过一些策略可以用来生成上面的代码模式，所以这里对该问题就不做冗长地复述了。我们可以使用tiny_size所采用的文件迭代方式：

```
#ifndef BOOST_PP_IS_ITERATING

#  ifndef BOOST_FUNCTION_HPP_INCLUDED
#    define BOOST_FUNCTION_HPP_INCLUDED

#    include <boost/preprocessor/repetition.hpp>
#    include <boost/preprocessor/iteration/iterate.hpp>

#    ifndef FUNCTION_MAX_ARITY
#      define FUNCTION_MAX_ARITY 15
#    endif

template <class Signature> struct function;     // 主模板

// generate specializations
#    define BOOST_PP_ITERATION_LIMITS (0, FUNCTION_MAX_ARITY)
#    define BOOST_PP_FILENAME_1  "boost/function.hpp" // 该文件
#    include BOOST_PP_ITERATE()

#  endif // BOOST_FUNCTION_HPP_INCLUDED

#else // BOOST_PP_IS_ITERATING

#  define n BOOST_PP_ITERATION()

// specialization pattern
```

```
template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
  定义略……

# undef n

#endif // BOOST_PP_IS_ITERATING
```

上面使用的BOOST_PP_ENUM_TRAILING_PARAMS在其第一个参数不为0时会生成一个前导“,”，其他方面则跟BOOST_PP_ENUM_PARAMS类似。

A.4.3.1 实参选择

出于和C++标准程序库算法互操作的考虑，如果接收一个参数或两个参数的function可以分别派生于适当的std::unary_function或std::binary_function特化，那将会很好[⊖]。在处理这些特殊情况时，BOOST_PP_IF是一个极好的工具：

```
# include <boost/preprocessor/control/if.hpp>
# include <boost/preprocessor/comparison/equal.hpp>
// specialization pattern
template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
  BOOST_PP_IF(
      BOOST_PP_EQUAL(n,2), : std::binary_function<A0, A1, R>
    , BOOST_PP_IF(
        BOOST_PP_EQUAL(n,1), : std::unary_function<A0, R>
      , 空参数……
      )
  )
{ 类体省略…… };
```

不幸的是，我们第一个尝试就碰到了几个问题：首先，你不能把一个空参数（empty argument）传给预处理器（参见285页脚注3）。其次，因为尖括号并没有被预处理器特殊对待，所以上面std::unary_function和std::binary_function特化版本中的逗号被看做是分隔宏参数的标志，从而预处理器会在两处抱怨我们传递给BOOST_PP_IF的实参的个数有误（即实参过多）。

因为它捕获了所有问题，所以让我们将精力放在内层的BOOST_PP_IF调用上一会儿。mpl::eval_if所使用的策略（选择一个无参函数（nullary function）进行调用）可以很好地运用到这里。Preprocessor程序库中并没有与mpl::eval_if直接对应的宏，但其实这里并不需要类似mpl::eval_if的东西，因为我们只要在BOOST_PP_IF后面多添加一对小括号就可达到目的：

```
#define BOOST_FUNCTION_unary() : std::unary_function<A0,R>
```

⊖ 虽然继承自std::unary_function或std::binary_function对于和一些老旧的程序库之间的互动可能是必需的，但这样做可能会阻止“空基类优化（Empty Base Optimization）”——考虑当这样两个派生类的对象是同一个对象的一部分（子对象）的情况。关于这方面的更多信息参见第9章关于结构选择的小节。通常，直接暴露first_argument_type和second_argument_type以及result_type typedefs是较好的选择。

```
#define BOOST_FUNCTION_empty()    // 空

...

    , BOOST_PP_IF(
          BOOST_PP_EQUAL(n,1), BOOST_FUNCTION_unary
        , BOOST_FUNCTION_empty
      )()

#undef BOOST_FUNCTION_empty
#undef BOOST_FUNCTION_unary
```

一个“什么也不生成的无参宏（nullary macro）”通常在很多地方都是有用的，以至于Preprocessor程序库的“facilities”分组中提供了一个现成的：BOOST_PP_EMPTY。现在我们可以纠正上面的错误了，我们可以将BOOST_FUNCTION_binary()、BOOST_FUNCTION_unary()以及BOOST_PP_EMPTY()的评估一直延迟到外围的BOOST_PP_IF调用（展开）结束后，因为std::binary_function<A0,A1,R>也存在“逗号问题”：

```
#  include <boost/preprocessor/facilities/empty.hpp>

#  define BOOST_FUNCTION_binary() : std::binary_function<A0,A1,R>
#  define BOOST_FUNCTION_unary() : std::unary_function<A0,R>

// 特化模式
template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
  BOOST_PP_IF(
      BOOST_PP_EQUAL(n,2), BOOST_FUNCTION_binary
    , BOOST_PP_IF(
        BOOST_PP_EQUAL(n,1), BOOST_FUNCTION_unary
      , BOOST_PP_EMPTY
      )
  )()
{
    ……类体被省略
};

# undef BOOST_FUNCTION_unary
# undef BOOST_FUNCTION_binary
# undef n
```

注意，由于我们碰巧使用了文件迭代，所以我们也可以直接对n的值使用#if：

```
  template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
  struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
#if n == 2
    : std::binary_function<A0, A1, R>
```

```
#elif n == 1
    : std::unary_function<A0, R>
#endif
```

BOOST_PP_IF具有一个优点，允许我们将逻辑封装到一个可复用的宏中（以n为参数），它与所有的重复构造（repetition constructs）都是兼容的：

```
#define BOOST_FUNCTION_BASE(n)                                          \
    BOOST_PP_IF(BOOST_PP_EQUAL(n,2), BOOST_FUNCTION_binary              \
      , BOOST_PP_IF(BOOST_PP_EQUAL(n,1), BOOST_FUNCTION_unary           \
          , BOOST_PP_EMPTY                                              \
        )                                                               \
    )()
```

A.4.3.2 其他选择构造

BOOST_PP_IDENTITY也位于“facilities”分组中，它是BOOST_PP_EMPTY的一个有趣的堂兄弟：

```
#define BOOST_PP_IDENTITY(tokens) tokens BOOST_PP_EMPTY
```

你可以认为它创建了一个返回标记的无参宏（nullary macro）：当在后面加一对空括号时，末尾的BOOST_PP_EMPTY展开为空，从而只剩下标记。如果我们想在function是三元以上时让它继承自mpl::empty_base的话，就可以使用BOOST_PP_IDENTITY：

```
// specialization pattern
template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
  BOOST_PP_IF(
      BOOST_PP_EQUAL(n,2), BOOST_FUNCTION_binary
    , BOOST_PP_IF(
        BOOST_PP_EQUAL(n,1), BOOST_FUNCTION_unary
      , BOOST_PP_IDENTITY(: mpl::empty_base)
      )
  )()
{
    类体被省略……
};
```

还有一个要注意的宏是BOOST_PP_EXPR_IF，它根据第一个参数（一个布尔值）决定是否展开为第二个参数：

```
#define BOOST_PP_EXPR_IF(c,tokens)                                      \
   BOOST_PP_IF(c,BOOST_PP_IDENTITY(tokens),BOOST_PP_EMPTY)()
```

例如，BOOST_PP_EXPR_IF(1,foo)展开为foo，而BOOST_PP_EXPR_IF(0,foo)则展开为空。

A.4.4 Token粘贴

如果能有一个泛化的方式来访问所有函数对象的参数类型和返回类型（而不是仅限于一元或二元函数）该多好啊！使用一个元函数将函数签名（Signature）“编码”为一个MPL序列是一

个可行的方案。我们只需为每个不同“元”的function给出一个signature的特化版本：

```
template <class F> struct signature; // 主模板

// 针对boost::function的局部特化
template <class R>
struct signature<function<R()> >
  : mpl::vector1<R> {};

template <class R, class A0>
struct signature<function<R(A0)> >
  : mpl::vector2<R,A0> {};

template <class R, class A0, class A1>
struct signature<function<R(A0,A1)> >
  : mpl::vector3<R,A0,A1> {};

...
```

要生成这些特化版本，只需向我们的模式中添加如下“样板”代码：

```
template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct signature<function<R( BOOST_PP_ENUM_PARAMS(n,A) )> >
  : mpl::BOOST_PP_CAT(vector,n)<
      R BOOST_PP_ENUM_TRAILING_PARAMS(n,A)
    > {};
```

BOOST_PP_CAT实现了标记粘贴（token pasting）——它将两个参数“粘”成单个标记。因为这是一个通用的宏，所以它位于程序库目录树最顶层的cat.hpp中。

尽管预处理器有一个内建的标记粘贴运算符（token-pasting operator）“##”，但是它只能用在宏定义中。如果我们在这里使用它，则根本不起任何作用：

```
template <class R>
struct signature<function<R()> >
  : mpl::vector##1<R> {};

template <class R, class A0>
struct signature<function<R(A0)> >
  : mpl::vector##2<R,A0> {};

template <class R, class A0, class A1>
struct signature<function<R(A0,A1)> >
  : mpl::vector##3<R,A0,A1> {};

...
```

此外，##经常会产生令人惊讶的结果，因为它在其实参展开之前就会起作用：

```
#define N          10
```

```
#define VEC(i)    vector##i

VEC(N)          // vectorN
```

与之形成对比的是，BOOST_PP_CAT将“拼接”操作延迟到其所有实参被完全评估（展开）之后：

```
#define N         10
#define VEC(i)    BOOST_PP_CAT(vector,i)

VEC(N)          // vector10
```

A.4.5 数据类型

BPL还提供了数据类型（data types），你可以认为它类似于MPL类型序列（type sequence）。Preprocessor程序库的数据类型存储的是宏实参而不是C++类型。

A.4.5.1 序列

序列（sequence）（简写为seq）是非空的宏实参字符串（其中每个宏实参都以小括号括住）。例如，下面是一个含有三个元素的序列：

```
#define MY_SEQ    (f(12))(a + 1)(foo)
```

下面展示如何使用序列来生成来自Boost Type程序库的is_integral模板（参见第2章）特化的过程：

```
# include <boost/preprocessor/seq.hpp>
template <class T>

struct is_integral : mpl::false_ {};

// 带有无符号对应物的整型序列
#define BOOST_TT_basic_ints            (char)(short)(int)(long)

// 生成一个包含有“signed t”和“unsigned t”的序列
#define BOOST_TT_int_pair(r,data,t)      (signed t)(unsigned t)

// 一个包含所有整型的序列
#define BOOST_TT_ints                                          \
   (bool)(char)                                                \
   BOOST_PP_SEQ_FOR_EACH(BOOST_TT_int_pair, ~, BOOST_TT_basic_ints)

// 生成一个针对类型t的is_integral特化
#define BOOST_TT_is_integral_spec(r,data,t) \
   template <>                              \
   struct is_integral<t> : mpl::true_ {};

BOOST_PP_SEQ_FOR_EACH(BOOST_TT_is_integral_spec, ~, BOOST_TT_ints)
```

```
#undef BOOST_TT_is_integral_spec
#undef BOOST_TT_ints
#undef BOOST_TT_int_pair
#undef BOOST_TT_basic_ints
```

BOOST_PP_SEQ_FOR_EACH是一个高阶宏（higher-order macro），与BOOST_PP_REPEAT类似。它以其第一个参数为“函数”，以第三个参数所表达的序列中的各元素为实参，逐个进行调用。

序列是最具效率、最灵活也是最容易使用的Preprocessor程序库数据类型——前提是你不需要一个空序列：一个空序列不包含任何标记，因此不能作为宏实参进行传递。这里讲到的其他数据结构都可以为空。

操纵序列的设施都位于程序库的seq/子目录中，见表A.5。其中t代表序列$(t_0)(t_1)...(t_k)$。s、r和d与我们前面提到的z参数的用途类似（目前建议你不予理睬）。

表A.5 Preprocessor序列操作

表达式	结果
BOOST_PP_SEQ_CAT(t)	$t_0t_1...t_k$
BOOST_PP_SEQ_ELEM(n,t)	t_n
BOOST_PP_SEQ_ENUM(t)	$t_0,t_1,...t_k$
BOOST_PP_SEQ_FILTER(pred,data,t)	t，其中没有不满足pred的元素
BOOST_PP_SEQ_FIRST_N(n,t)	$(t_0)(t_1)...(t_{n-1})$
BOOST_PP_SEQ_FOLD_LEFT(op, x, t)	$...op(s,op(s,op(s,x,t_0),t_1),t_2)...$
BOOST_PP_SEQ_FOLD_RIGHT(op, x, t)	$...op(s,op(s,op(s,x,t_k),t_{k-1}),t_{k-2})...$
BOOST_PP_SEQ_FOR_EACH(f, x, t)	$f(r,x,t_0)f(r,x,t_1)...f(r,x,t_k)$
BOOST_PP_SEQ_FOR_EACH_I(g, x, t)	$g(r,x,0,t_0)g(r,x,1,t_1)...g(r,x,k,t_k)$
BOOST_PP_SEQ_FOR_EACH_PRODUCT(h, x, t)	笛卡尔乘积，参见在线文档
BOOST_PP_SEQ_INSERT(t,i,tokens)	$(t_0)(t_1)...(t_{i-1})$ (tokens) $(t_i)(t_{i+1})...(t_k)$
BOOST_PP_SEQ_POP_BACK(t)	$(t_0)(t_1)...(t_{k-1})$
BOOST_PP_SEQ_POP_FRONT(t)	$(t_1)(t_2)...(t_k)$
BOOST_PP_SEQ_PUSH_BACK(t,tokens)	$(t_0)(t_1)...(t_k)$(tokens)
BOOST_PP_SEQ_PUSH_FRONT(t,tokens)	(tokens)$(t_0)(t_1)...(t_k)$
BOOST_PP_SEQ_REMOVE(t,i)	$(t_0)(t_1)...(t_{i-1})(t_{i+1})...(t_k)$
BOOST_PP_SEQ_REPLACE(t,i,tokens)	$(t_0)(t_1)...(t_{i-1})$(tokens)$(t_{i+1})...(t_k)$
OOST_PP_SEQ_REST_N(n,t)	$(t_n)(t_{n+1})...(t_k)$
BOOST_PP_SEQ_REVERSE(t)	$(t_k)(t_{k-1})...(t_0)$
BOOST_PP_SEQ_HEAD(t)	t_0
BOOST_PP_SEQ_TAIL(t)	$(t_1)(t_2)...(t_k)$
BOOST_PP_SEQ_SIZE(t)	k+1
BOOST_PP_SEQ_SUBSEQ(t,i,m)	$(t_i)(t_{i+1})...(t_{i+m-1})$
BOOST_PP_SEQ_TO_ARRAY(t)	$(k+1,(t_0,t_1,...t_k))$
BOOST_PP_SEQ_TO_TUPLE(t)	$(t_0,t_1,...t_k)$
BOOST_PP_SEQ_TRANSFORM(f, x, t)	$(f(r,x,t_0))$ $(f(r,x,t_1))...(f(r,x,t_k))$

值得注意的是：虽然序列的长度没有上限，但是像BOOST_PP_SEQ_ELEM这样接收数值参数的操作最多只能处理256个数值。

A.4.5.2 Tuples

tuple是一种非常简单的数据结构，程序库为它提供了随机访问以及其他一些基本操作。tuple的形式为：以小括号括起来的、以逗号分隔的宏实参列表。例如，下面是一个含有三个元素的tuple：

```
#define TUPLE3    (f(12), a + 1, foo)
```

程序库的tuple/子目录下包含对tuple的操作，这些操作支持最多含有25个元素的tuple。例如，一个tuple的第N个元素可以通过BOOST_PP_TUPLE_ELEM来访问，如下：

```
// 长度索引tuple
BOOST_PP_TUPLE_ELEM( 3 , 1 , TUPLE3)    // a + 1
```

注意：我们必须将tuple的长度作为第一个参数传递给BOOST_PP_TUPLE_ELEM。事实上，所有tuple操作都要求显式指定tuple的长度。“tuple”分组中的另外四个操作这里就不作介绍了，你可以参考Preprocessor程序库的电子文档来了解更多的细节。注意，利用BOOST_PP_SEQ_TO_TUPLE，可将序列转换为tuples，利用BOOST_PP_TUPLE_TO_SEQ，则可将非空的tuples转换回序列。

tuples最强大的功能是：它们可以方便地带有与一个宏实参列表相同的表示：

```
#define FIRST_OF_THREE(a1,a2,a3)   a1
#define SECOND_OF_THREE(a1,a2,a3)  a2
#define THIRD_OF_THREE(a1,a2,a3)   a3

// 使用tuple作为实参列表
# define SELECT(selector, tuple)   selector tuple

SELECT(THIRD_OF_THREE, TUPLE3)  // foo
```

A.4.5.3 Arrays

array其实就是一个tuple，但它的第一个元素是tuple的长度（非负数）：

```
#define ARRAY3    ( 3, TUPLE3 )
```

因为array“随身携带”着自身的长度信息，所以程序库中用来操作arrays的接口要比操作tuples接口方便多了：

```
BOOST_PP_ARRAY_ELEM(1, ARRAY3)    // a + 1
```

程序库中的array/子目录中包含操作arrays的设施，这些操作支持最多25个元素的arrays，它们总结在表A.6中，其中a是这样的一个arrary：(k,(a0,a1,...,ak-1)。

表A.6 Preprocessor Array操作

表达式	结果
`BOOST_PP_ARRAY_DATA(a)`	$(a_0,a_1, \ldots a_{k-1})$
`BOOST_PP_ARRAY_ELEM(i,a)`	a_i
`BOOST_PP_ARRAY_INSERT(a, i, tokens)`	$(k+1,(a_0,a_1, \ldots a_{i-1},\text{tokens},a_i,a_{i+1}, \ldots a_{k-1}))$
`BOOST_PP_ARRAY_POP_BACK(a)`	$(k-1,(a_0,a_1, \ldots a_{k-2}))$
`BOOST_PP_ARRAY_POP_FRONT(a)`	$(k-1,(a_1,a_2, \ldots a_{k-1}))$
`BOOST_PP_ARRAY_PUSH_BACK(a, tokens)`	$(k+1,(a_0,a_1, \ldots a_{k-1},\text{tokens}))$
`BOOST_PP_ARRAY_PUSH_FRONT(a, tokens)`	$(k+1,(\text{tokens}, a_1,a_2, \ldots a_{k-1}))$
`BOOST_PP_ARRAY_REMOVE(a, i)`	$(k-1,(a_0,a_1, \ldots a_{i-1},a_{i+1}, \ldots a_{k-1}))$
`BOOST_PP_ARRAY_REPLACE(a, i, tokens)`	$(k,(a_0,a_1, \ldots a_{i-1}, \text{tokens},a_{i+1}, \ldots a_{k-1}))$
`BOOST_PP_ARRAY_REVERSE(a)`	$(k,(a_{k-1},a_{k-2}, \ldots a_1,a_0))$
`BOOST_PP_ARRAY_SIZE(a)`	k

A.4.5.4 Lists

list是一个含有两个元素的tuple，其第一个元素是list的第一个元素，第二个元素则是其余元素的一个list或BOOST_PP_NIL（若无其余元素的话）[⊖]。list的访问方式和运行期链表很相似。下面是一个含有三个元素的list：

```
#define LIST3    (f(12), (a + 1, (foo, BOOST_PP_NIL)))
```

用于操纵list的设施全部位于程序库的list/子目录中。由于这些操作是序列操作的一个子集，所以在此我们就不一一罗列了。有了对序列操作的讨论的基础，再阅读有关文档，不难理解list操作的含义。

和序列类似，list没有固定的长度上限。但是和序列不同的是，list可以为空。通常，在一个预处理数据结构中，你很少需要大于25个元素，而且与其他数据结构相比，list的操纵速度通常较慢，且可读性差，所以不到万不得已不要使用list。

A.5 练习

A-0. 对第5章中实现的类型序列进行完全地预处理化（preprocessor-ize），以便消除所有刻板的代码，并且可以通过改变TINY_MAX_SIZE来调整tiny序列的最大尺寸。

⊖ 显然这是一个递归定义。——译者注

附录B　typename和template关键字

template关键字用来引入模板声明和定义：

```
template <class T>
class vector;
```

typename关键字通常用来取代class来声明模板类型的参数[⊖]：

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

在语言中，这两个关键字都拥有第二个角色。本附录将描述该角色，为何需要它，以及如何正确地应用typename和template来满足这个需要。由于这些规则相当微妙，所以许多人等到编译器发出抱怨才去思考typename或template的使用问题，但学习这些技术细节是值得的，因为：

- 你将会花费更少的时间来修复琐细的语法错误。
- 当编译器发出抱怨时，你会理解你做错了什么。
- 你的代码会更具有移植性。很多编译器并不严格地遵从标准发出足够的抱怨，当你遗漏了typename或template时，它们不会告诉你。
- 你的代码将更加按你的意图工作。编译器并不能侦测到所有的误用，遗漏这两个关键字之一可能会导致你的程序悄悄地行为不端。

B.1　议题

模板编译分为两个阶段：第一阶段发生在模板的定义点（point of definition），第二个阶段发生在它的每一个实例化点（points of instantiation）。根据C++标准的描述，一个模板在其定义点必须对其语法的正确性进行彻底地检查[⊖]，这样它的作者就可以在该模板被实例化之前很早就知道它是“形式良好的（well-formed）”：

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 i1, ForwardIterator2 i2)
{
    T tmp = *i1;   // 错误：未知的标识符T
    *i1 = *i2;
    *i2 = tmp;
}
```

B.1.1　问题1

在C++标准化过程中，委员会发现有一些情形是不可能在模板的定义点进行完全的语法检查

⊖　我们马上就会讨论本书为何使用class而不是typename。

⊖　在这方面，并非所有编译器都遵从标准，有不少编译器将一些或所有检查延迟到实例化点。

的。例如，考虑如下包含有一个iter_swap定义的翻译单元（translation unit）：

```
double const pi = 3.14159265359;

template <class T> struct iterator_traits; // 仅声明

template <class FwdIterator1, class FwdIterator2>
void iter_swap(FwdIterator1 i, FwdIterator2 j)
{
    iterator_traits<FwdIterator1>::value_type* pi = &*i;
    其他代码……
}
```

编译器必须检查iter_swap是否存在语法错误，但它现在还没有看到一个iterator_traits定义。iterator_traits的::value_type可以是一个类型，如果是这样，突出显示的那一行代码就是合法的声明。然而，它也可能是一个枚举值：

```
template <class T>
struct iterator_traits
{
    enum { value_type = 0 };
};
```

在这种情况下，iter_swap的第一行代码就变成了胡扯。好像我们可以认为编译器应该推导出value_type必须是一个类型，因为在其他情况下iter_swap的第一行代码都是无效的。然而，请考虑如下的反例：

```
class number
{
 public:
    template <class U>
    number& operator=(U const&);
    int& operator*() const;
};

number operator*(number, double);

template <class T>
struct iterator_traits
{
    static number value_type;
};
```

在这种情形下，iter_swap可能仍是有效的，其第一行代码会变成将一个数值乘以pi，然后再对其进行赋值：

```
(iterator_traits<FwdIterator1>::value_type * pi) = &*i;
```

但如果是这样的话，iter_swap的语法结构将变成一个完全不同的东西。

我们还可能倾向于认为编译器对iter_swap进行语法检查——如果它已经看到了iterator_traits的定义的话，然而特化（specializations）破坏了这种可能性：任何给定的iterator_traits实体都可能采用完全不同的方式进行定义：

```
template <>
struct iterator_traits<int*>
{
    static void* value_type;
};
```

问题在于iterator_traits<FwdIterator1>::value_type是一个依赖性的名字（dependent name），它所扮演的语法角色取决于FwdIterator1到底是什么，然而在iter_swap的定义点，是绝不可能知道这一点的。

B.1.2 消除类型歧义

typename关键字告诉编译器，一个依赖名表示的是一个依赖性的类型（dependent type）：

```
template <class FwdIterator1, class FwdIterator2>
void iter_swap(FwdIterator1 i, FwdIterator2 j)
{
    typename iterator_traits<FwdIterator1>::value_type* pi = &*i;
    其他代码……
}
```

现在iterator_traits<FwdIterator1>::value_type的语法角色就很清晰了，编译器知道pi指的是一个指向iter_swap本体其他部分的指针。如果我们不写typename，编译器就会假定value_type是一个非类型的值，pi则为iter_swap中的一个const double。

B.1.3 使用class vs. 使用typename

正如前面所言，以下两个声明是等价的：

```
template <class T>
class vector;

template <typename T>
class vector;
```

赞同使用typename的论点认为，这在概念上是精确的：类看上去指示参数必须是一个类类型，而实际上任何类型都可以，比如vector<int>就完全没有问题。

为了理解赞同使用类关键字的论点，考虑如下声明中对typename的使用：

```
template <typename T, typename T::value_type>
struct sqrt_impl;
```

你也许没有理解上面的代码，但只有第一个对typename的使用是在声明一个type参数，第二个typename是在声明T::value_type是一个类型。因此，传给sqrt的第二个参数是一个类型为T::value_type的值。

如果上例看起来有点迷糊，我们没理由责备你。也许以下这个等价的声明有助于澄清这一点：

```
template <class T, typename T::value_type n>
struct sqrt_impl;
```

如果是这样，你就理解了使用class来声明模板类型参数的论点：如果typename仅被用于在模板参数列表中表示一件东西的含义（即消除语法歧义），就不会那么令人感到迷惑了。

我们无意告诉你应该选择哪一种编程实践，善良的人们可能不赞成概念上的精确比避免在罕见情况下（即将typename用于非类型参数声明）的混淆更重要。事实上，本书的作者就不同意，这也是为何你在本书中看到使用的是类，但在MPL参考手册中则使用typename的原因。

B.1.4 问题2

对于模板成员而言存在同样的问题：

```
double const pi = 3.14159265359;

template <class T>
int f(T& x)
{
    return x.convert<3>(pi);
}
```

T::convert可能是一个成员函数模板，在这种情况下突出显示的代码将pi传递给convert<3>的一个特化。另外，它实际上也可能是一个数据成员，在这种情况下，f返回(x.convert < 3) > pi。虽然这并非一个特别有意义的计算，但编译器并不知道这一点。

B.1.5 采用template消除歧义

template关键字告诉编译器，一个依赖名是一个成员模板：

```
template <class T>
int f(T& x)
{
    return x.template convert<3>(pi);
}
```

如果我们遗漏了template关键字，编译器就会假定x.convert不是一个模板的名字，并且跟在它后面的<被解析为“小于”运算符。

B.2 规则

在这一节中，我们将探讨C++标准关于template和typename关键字的使用规则，并走马观花，看看一些说明性的例子。

B.2.1 typename

以下相关的标准描述摘自C++98标准14.6节[temp.res]第5段：

关键字typename只能用于模板声明和定义中，包括函数模板或成员函数模板的返回类型中，

类模板的成员函数定义的返回类型中，或类模板中的嵌套类的成员函数定义的返回类型中，类模板的静态成员定义的类型修饰符中，或类模板中的嵌套类的静态成员定义的类型修饰符中。关键字typename只能用于限定性名字，但那些名字不必是依赖性的（dependent）。关键字typename不允许用在基修饰符或成员初始化列表中，在这些上下文（contexts）中，一个依赖于模板参数的限定性名字（14.6.2）被隐式地假定为一个类型名字。

B.2.1.1 必须使用typename的场合

在模板中，任何指示为类型的qualified dependent names的前面都需要使用typename关键字。

标明依赖类型名

在下面的例子中，类型C::value_type依赖于模板参数C。

```
// 成员数据声明
template <class C>
struct something
{
    typename C::value_type x;
};
```

依赖类型的属性之一是传递性（transitive）。在下面的例子中，C::value_type依赖于C，value_type::is_const依赖于value_type（因此也依赖于C）。

```
// member type声明
template <class C>
struct something
{
    typedef typename C::value_type value_type;
    typedef typename value_type::is_const is_const;
};
```

在下面的例子中，add_const 元函数的::type成员依赖于模板参数T。

```
template <class T>
struct input_iterator_part_impl
{
    typedef typename boost::add_const<T>::type const_T;
};
```

应用上下文

你已经看到typename是如何应用于类模板本体之内的。在参数列表中也需要使用它，包括在默认实参表达式中：

```
template <
       class T
    , typename non_type_parameter<T>::type value
        = typename non_type_parameter<T>::type()
>
```

```
struct initialized
{};
```

以及在函数模板中，包括用在它们的本体中：

```
template <class Sequence>
typename Sequence::iterator          // 在返回类型中
find(
   Sequence seq
 , typename Sequence::value_type x   // 在参数类型中
)
{
    typename Sequence::iterator it   // 在函数本体内
      = seq.begin();
    等等……。
}
```

由于规则是“每一个依赖名都要使用一个typename”，因此在一个声明内可能需要用到好几个typename关键字。

```
template <class Sequence>
struct key_iterator_generator
{
    typedef typename projection_iterator_gen<
        select1st<typename Sequence::value_type>
      , typename Sequence::const_iterator
    >::type type;
   };
```

微妙之处

一个类型可能会因为一些微妙的原因而变成依赖性的。在下面的例子中，index<1>::type是依赖性的，因为我们可以针对一个给定的迭代器类型来特化索引成员模板。

```
template <class Iterator>
struct category_index
{
    template <long N> struct index
   {
       typedef char(&type)[N];
   };

    int category(std::input_iterator_tag)
    {
        return sizeof(typename index<1>::type);
    }
    int category(std::forward_iterator_tag)
        {
        return sizeof (typename indx<2>::type);
```

```
    }

};
template <>
template <long N>
struct category_index<int*>::index
{
    typedef char(&type)[N + 1];
};
```

换句话说，为了消除语法上的歧义，category_index主模板等价于：

```
template <class Iterator, long N> struct index
{
    typedef char(&type)[N];
};

template <class Iterator>
struct category_index
{
};
```

B.2.1.2 允许（但非必须）使用typename的场合

对于一个模板内部的非依赖名而言，是否使用typename关键字对其进行限定完全是可选的。在下面的例子中，std::unary_function<T,T*>不是依赖性的，因为它总是一个类，不管T最终到底是什么。

```
template <class T>
struct something
{
    // 没问题
    std::unary_function<T,T*> f2;
    std::unary_function<int,int>::result_type x2;
    // 也没问题
    typename std::unary_function<T,T*> f1;
    typename std::unary_function<int,int>::result_type x1;
};
```

B.2.1.3 禁止使用typename的场合

typename关键字不能使用在模板之外的任何地方：

```
struct int_iterator
{
    typedef typename int value_type; // 错误
};
```

typename也不允许用在非限定性名字（也就是不带::前缀的）上，即便它们是依赖性的也不行。

```
template <class T>
struct vector
{
    typedef typename int value_type;          // 错误
    typedef typename pair<int,T> pair_type; // 错误
    typedef typename T* pointer;              // 错误
};
```

typename不允许用于基类的名字上，即便它是依赖性的也不行：

```
template <class T> struct base_gen;

template <class T>
struct derived
  : typename base_gen<T>::type // 错误
{};
```

但在下面的例子中，typename是必不可少的，因为T::value_type并不是一个基类的名字：

```
template <class T>
struct get_value
  : std::unary_function<T, typename T::value_type> // OK
{};
```

由于显式（完全）特化（explicit (full) specialization）并不是一个模板声明，因此以下代码中的做法当前是不允许的，但是核心语言问题#183论点赞同在标准将来的修订版中允许这种做法[⊖]。

```
template <class T> struct vector;

template <class T> struct vector_iterator
  : mpl::identity<T> {};

template <>
struct vector<void*>
{
    typedef typename                                  // 错误
      vector_iterator<void*>::type iterator;
};
```

B.2.1.4 杂项

- C++标准（14.6.1节）允许我们使用类模板自身的名字（不带实参）作为正被实例化的特化的同义词（synonym），这意味着我们可以使用一个模板的名字来限定依赖的基类的成员。例如，不是写：

⊖ 参见http://www.open-std.org/jtcl/sc22/wg21/docs/cwg_defects.html#183.

```
template <class T> class base;
template <class T>
struct derived
  : base<typename whatever<T>::type> // 下面又重复了一遍
{
    typedef base<typename whatever<T>::type> base_;
    typedef typename base_::value_type value_type;
};
```

我们可以简单地写成：

```
template <class T> struct base;

template <class T>
struct derived
  : base<typename whatever<T>::type> // 不再重复
{
    typedef typename derived::value_type value_type;
};
```

- 随着核心语言问题#11被接受[⊖]，

```
template <class T> struct base;

template <class T>
struct derived
  : base<T>
{
    using typename base<T>::value_type;
};
```

等价于

```
template <class T> struct base;

template <class T>
struct derived
  : base<T>
{
    typedef typename base<T>::value_type value_type;
};
```

- 核心语言问题#180澄清typename不允许用于友员声明中[⊜]，例如：

```
template <class T>
class X
{
```

⊖ 参见http://www.open-std.org/jtcl/sc22/wg21/docs/cwg_defects.html#11。

⊜ 参见http://www.open-std.org/jtcl/sc22/wg21/docs/cwg_defects.html#180。

```
    friend class typename T::nested; // 错误
};
```

B.2.2 template

以下相关的标准描述摘自C++标准14.2节[temp.names]第4段：

当成员模板特化的名字出现在一个后缀表达式（postfix-expression）中的.或->之后，或者出现于一个限定标识（qualified-id）中的嵌套的名字修饰符之后，并且后缀表达式或限定标识显式依赖于一个模板参数（14.6.2）时，那么，成员模板名字必须施以template关键字作为前缀，否则该名字就被假定为一个非模板的名字。

以及第5段

如果一个名字被施以template关键字作为前缀，而它其实又不是一个成员模板的名字，那么这个程序就是形式不良的（注意：关键字template不允许应用在类模板的非模板成员）。

核心语言问题#30补充道[⊖]：

此外，如果后缀表达式或者限定标识不是出现在一个模板的作用域时，成员模板的名字就不应该加上template关键字作为前缀。（注意：和typename的情形一样，template关键字作为前缀在一些不是严格需要的场合也是可以的，即，当->或.的左边的表达式或嵌套的名字修饰符不依赖于一个模板参数时。）

B.2.2.1 必需使用template的场合

在通过“.”、“->”、或“:”限定的依赖名访问成员模板之前，template关键字是必不可少的。下面的例子中，onvert和base依赖于T：

```
template <class T> void f(T& x, T* y)
{
    int n = x.template convert<int>();
    int m = y->template convert<int>();
}

template <class T> struct other;
template <class T>
struct derived
  : other <T>::template base<int>
{};
```

注意，有别于typename关键字，甚至对表示基类的类模板名字而言，template关键字也是必需的。

B.2.2.2 允许（但非必须）使用template的场合

只要它实际上是在一个成员模板id之前，在一个模板中的任何地方，template关键字都是可选的。例如：

⊖ 参见http://www.open-std.org/jtcl/sc22/wg21/docs/cwg_defects.html#30。

```
template <class T>
struct other
{
   template <class U> struct base;
};

template <class T>
struct derived1
  : other<int>::base<T>                    // OK
{};

template <class T>
struct derived2
  : other <int>::template base<T>     // 也没问题
{};
```

B.2.2.3 禁止使用template的场合

template关键字禁止用在模板之外的任何地方，包括显式（完全）模板特化（按照先前引述的核心语言问题#30）：

```
template <> struct derived<int>
  : other<int>::template base<int> // 错误
{};
```

该关键字还禁止用在using声明中：

```
template <class T>
struct derived
  : base<T>
{
    using base<T>::template apply; // 错误
};
```

核心语言问题#109澄清了这条禁令（这不是一个缺陷（Not a Defect，NAD））[⊖]。

⊖ 参见http://www.open-std.org/jtcl/sc22/wg21/docs/cwg_closed.html#109。

附录C　编译期性能

模板元程序的解释（Interpretation）天生是效率低下的。当一个类模板被实例化时，C++编译器必须满足所有标准的要求，包括匹配局部特化，构建类的内部表示，并记录模板的命名空间中的特化。另外，可能还要满足编译器自身设计或环境所强加的要求，例如为连接器（linker）生成重整符号名字（mangled symbol names），或为调试器（debugger）记录信息等。这些活动都不是直接和元程序的有目的的计算直接相关的。

这个低效表现在程序在编译方面所花费的时间和编译器所使用的资源上。没有很好地理解其代价就广泛地使用元编程将会加剧这种效果。由于你的元程序通常被其他程序员所使用，这些程序员更多地关心快速的编译/编辑/调试周期而不是你的程序库是怎么实现的，因此，如果编译变得非常慢或因为资源耗用已经超出限制而停止编译的话，他们不大可能对此表示理解。

幸运的是，问题并非不可避免，如果你知道如何控制情形的话，这是可以避免的。附录C为你提供了分析和控制元程序效率的工具。

C.1　计算模型

在不检视每一款编译器的实现的前提下，我们真能讨论一些关于程序编译期开销的有意义的东西吗？

使用用于分析运行期复杂度的标准技术，我们可以做到这一点[⊖]。当描述运行期程序的复杂度时，我们计算它在一个抽象机（abstract machine）上执行原语操作（primitive operations）的次数。抽象机是真实硬件（actual hardware）的模型，它隐藏了指令周期时间（instruction cycle times）、高速缓存的位置（cache locality）以及寄存器的使用（register usage）之类的问题。对于模板元程序的情形，抽象机是一个编译器实现模型，它隐藏了诸如其内部数据结构（internal data structures）、符号表查找效率（symbol table lookup efficiency）以及解析算法（parsing algorithm）之类的问题。

我们根据所必需的模板实例化的数目来度量元程序的复杂性。这并不完全是一个任意的选择：编译时间往往和所执行的模板实例化的数目有着紧密的关系。当然，这并不是完美的选择，但是，只有将那些“有时”相关的因素排除掉，我们才能充分地简化抽象机，从而对程序的性能作出推论。

C.1.1　标记缓存（Memoization）

即使我们忽略其他因素，只根据模板实例化来考虑复杂度也是有点奇怪的，因为在一个翻译单元中一个特定的模板特化只实例化一次：

⊖　参见http://en.wikipedia.org/wiki/Computational_complexity_theory。

```
typedef foo<char>::type t1; // foo<char>在此实例化
...
typedef foo<char>::type t2; // foo<char>仅被查找
```

和常规函数调用工作方式不同，当一个元函数被同样的实参再次调用时，编译器并不再次履行全部的计算工作。如果你熟悉标记缓存（memoization）的思想[⊖]，你可以想象到所有元函数结果都被标记缓存的了。在第一次调用的地方，实例化的类存储在一个查找表（lookup table）中，以模板的实参进行索引（indexed）。对于以同样的实参发生的后继调用，编译器仅到表中查找模板具现体即可。

C.1.2 一个例子

考虑如下经典的fibonacci递归函数，它具有指数级的复杂度：

```
unsigned fibonacci(unsigned n)
{
    if (n < 2)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

调用fibonacci(3)会导致以下一系列的调用：

```
fibonacci(3)
  fibonacci(2)
    fibonacci(1)
    fibonacci(0)
  fibonacci(1)
fibonacci(2)
  fibonacci(1)
  fibonacci(0)
```

现在，让我们将其直接翻译成模板：

```
template<unsigned n, bool done = (n < 2)>
struct fibonacci
{
    static unsigned const value
      = fibonacci<n-1>::value + fibonacci<n-2>::value;
};

template<unsigned n>
struct fibonacci<n,true>
{
    static unsigned const value = n;
};
```

⊖ 参见http://en.wikipedia.org/wiki/Memoization。——译者注

在这种情况下，fibonacci<3>::value可能会导致如下的实例化和查找序列，其中实例化被以粗体字显示：

```
fibonacci<3>
  fibonacci<2>
    fibonacci<1>
    fibonacci<0>
  fibonacci<1>
fibonacci<2>
```

编译期fibonacci函数的复杂度不是O(exp(n))，而是O(n)。即使你把查找也算上，复杂度仍然是O(n)，因为对于每一个n来说，至多发生一次实例化和一次查找。

C.1.3 我们隐藏了什么？

采用这种方式描述抽象机，有什么东西被隐藏起来了呢？没有看到编译器的源代码，我们不能确定。为了充分揭示这一点，我们将讨论所知道的被清扫掉的幕后的东西。

正如先前所言，我们在隐藏编译器的实现细节。我们马上就会讨论从抽象中泄露这些细节并使其可见的方式。我们还会隐藏关于元程序实现的一些细节。例如，一些关联式序列（associative sequences）使用函数重载决议来实现它们的查找策略[⊖]。重载决议在编译器中可能具有不可忽略的开销，但我们不打算考虑它[⊖]。

C.2 管控编译时间

为了改善元程序的执行（编译）时间，你可以做的头等重要的事情是：减少它们计算的复杂度。如果你希望访问任意的元素，可以使用mpl::vector，因为这种访问的复杂度是O(1)而不是O(N)，使用mpl::list的复杂度则是O(N)。如果你可以使用mpl::lower_bound的话，就不要在一个序列中线性地查找一个元素，等等。没有什么措施可以取代选择得当的算法和数据结构。

遗憾的是，大多数编译器并非有意识地用来模板元编程，很多编译器对此采用了糟糕的实现策略。例如，一个理想的编译器应该将所有标记缓存的模板特化存储于一个哈希表中，这样可以保证对每一个标记缓存的模板特化的查找的时间复杂度为O(1)。然而，到本书写作时为止，大多数编译器却采用了一个链表（linked list）来存储特定类模板的实例。这样一来，从技术上而言查找就是按照已经被标记缓存的模板实例的数目线性递增的了。通常这个O(N)复杂度的效果会被实例化所导致的开销所掩盖，但正如我们将要看到的那样，它还是能被观察到的。

我们碰巧知道已经测试的编译器的这个实现细节，但对于其他具体的编译器而言，还有许多五花八门的怪癖是我们所不知道的。通过使用专用的测试程序，可以对我们的元程序设计决策的在现实世界中的效果，以及当我们关心元程序的执行（编译）速度时应该选用哪一种编译器，获得一个认知。在本附录中，我们将讨论这些黑盒测试（black-box tests）的实验性结果，

⊖ 这儿不涉及运行期执行，函数调用被封装于sizeof或typeof中，参见第9章的描述。

⊖ 作者在勘误页面上说，在实践中，重载决议看上去比模板实例化快很多，如果可以选择，那么采用重载决议要好得多。——译者注

还将揭示一些技术，你可以使用它们来避免我们已经发现的一些可能会出问题的地方。

请注意，你可以在配书光碟中找到我们在这里描述的测试的完整细节。

C.3 测试

C.3.1 标记缓存（Memoization）的效力

模板实例是否真的被标记缓存的了？如果是，标记缓存保存多少东西？为了揭示这一点，我们可以对fibonacci模板做一些微小的改变[⊖]。

```
template<unsigned n, unsigned m = n, bool done = (n < 2)>
struct fibonacci
{
    static unsigned const v1
      = fibonacci<n-1,m-1>::value;

    static unsigned const value
      = v1 + fibonacci<n-2,m-STEP>::value;
};

template<unsigned n, unsigned m>
struct fibonacci<n,m,true>
{
    static unsigned const value = n;
};
```

当STEP == 2时，调用fibonacci<N>仍然导致同样数量的模板实例化。然而，对于STEP == 1，我们可以确保fibonacci<N-2,...>::value的计算永远不会以以前使用过的一套实参来调用fibonacci。新的参数m为fibonacci特化提供了一个“额外的元（additional dimension）”，我们利用它来避免标记缓存。

通过从“当STEP == 1时，计算fibonacci<N>::value所占用的时间”中减去“当STEP == 2时，计算fibonacci<N>::value所占用的时间”，我们可以看到随着N的增大标记缓存所带来的性能改善（参见图C.1）。

对于我们测试的所有编译器而言，两个计算之间的开销差别以N^3级别上升，因此标记缓存确实带来了很大的收益。

C.3.2 标记缓存查找的开销

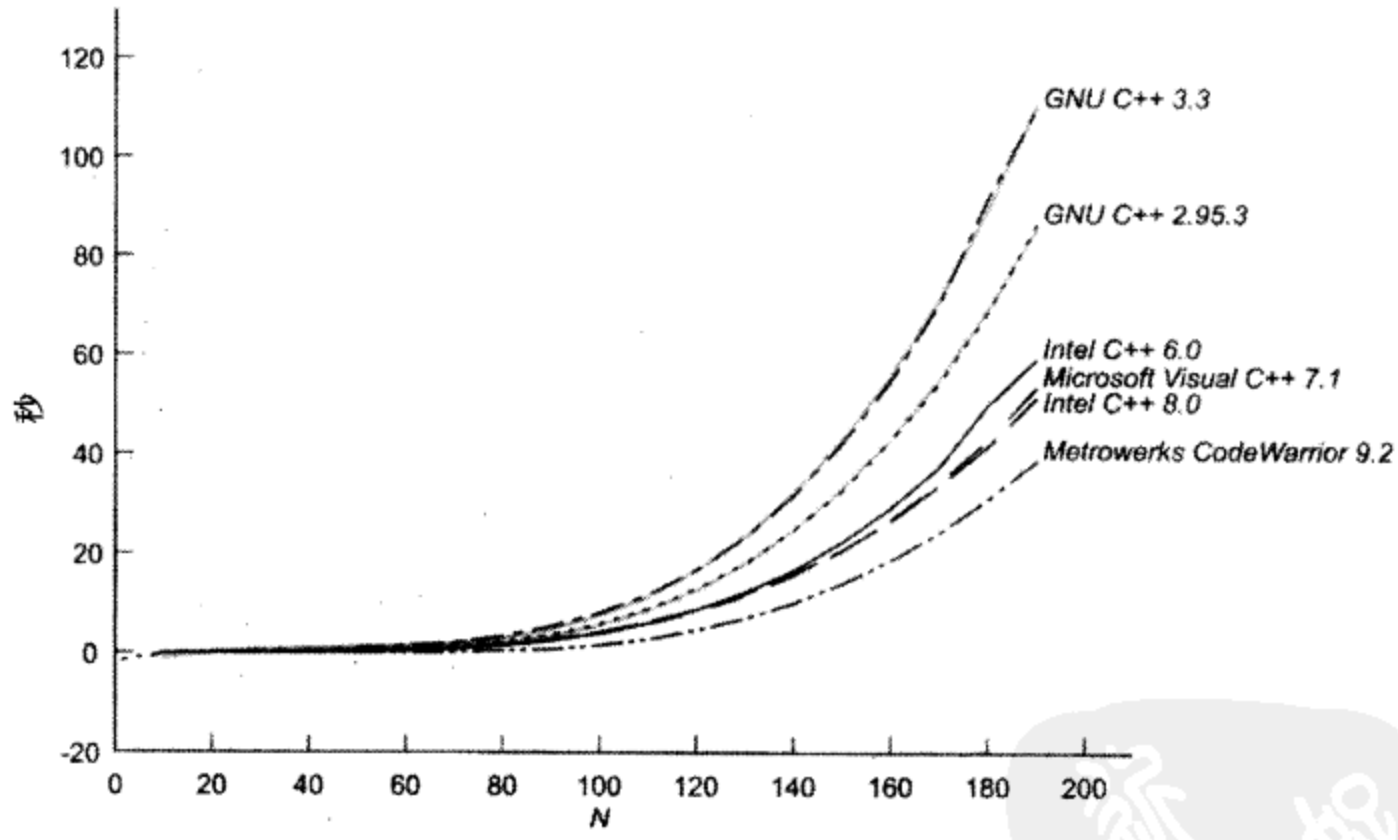
查找一个先前标记的（mentioned）[⊖]模板实例的开销有多大？为了度量这一点，我们可以使用另一个Fibonacci测试变种：

```
template<unsigned n, bool done = (n < 2)>
struct lookup
```

⊖ 这里展示的代码是对生成曲线图的实际代码的轻微简化版，请参考配书光碟以了解细节。

⊖ 同标记缓存（memoized）。——译者注

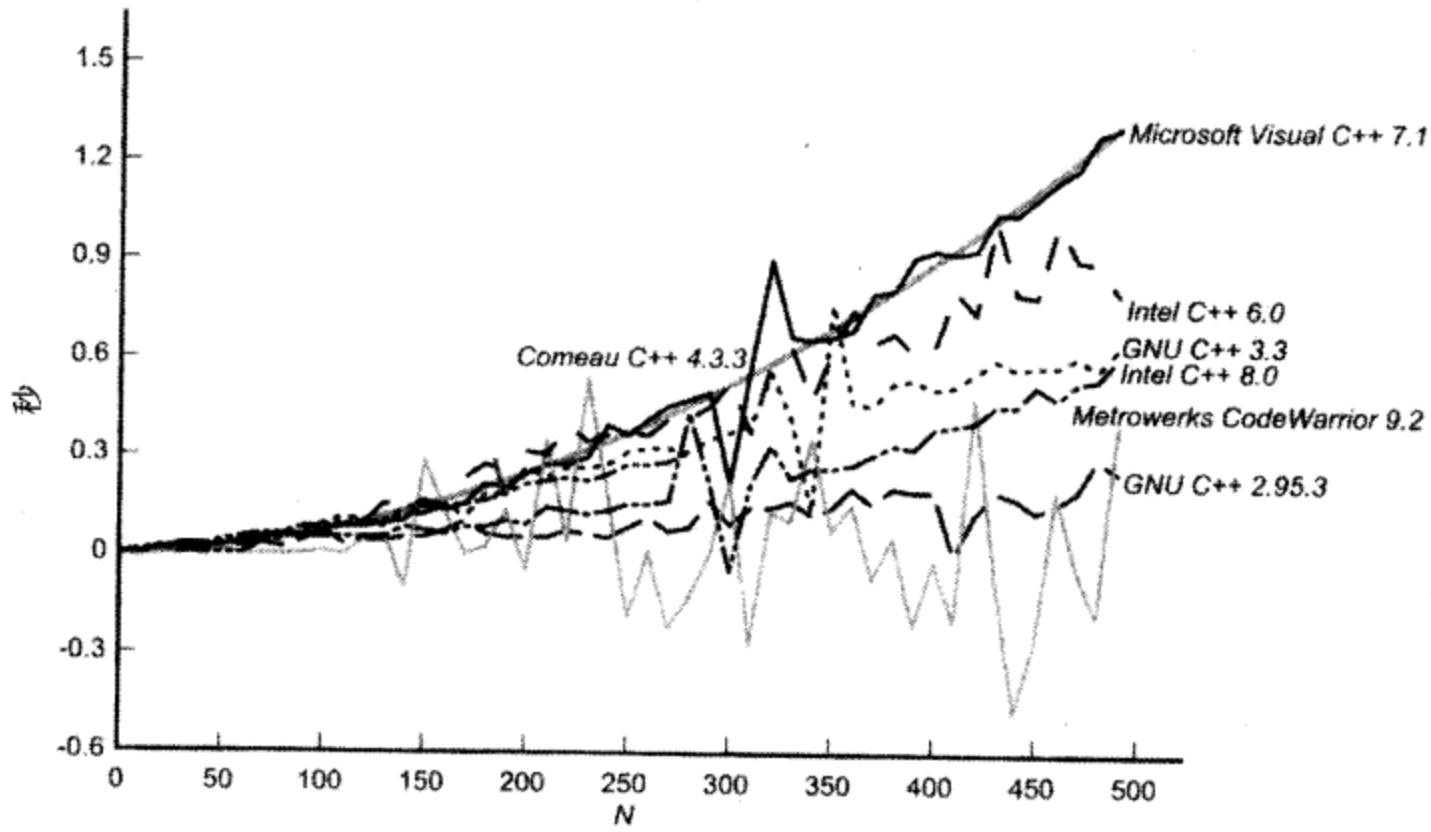
```
{
    static unsigned const v1
      = lookup<n-1>::value;

    static unsigned const value = v1

#ifndef BASELINE // 进行标记缓存查找
      + lookup<((n%2) ? 0 : n-2)>::value
#endif
      ;
};

template<unsigned n>
struct lookup<n,true>
{
    static unsigned const value = n;
};
```



图C.1 标记缓存带来的性能改善

定义了BASELINE而计算lookup<N>::value的开销，和未定义BASELINE而计算lookup<N>::value的开销之间的差别，显示了当N个模板特化已经被标记（mentioned）时标记缓存查找的开销。注意，这已经不再是Fibonacci计算了，尽管它遵从同样的实例化/查找模式。我们常常为标记缓存查找选择lookup<0>而非lookup<n-2>，因为特化往往被存储在链接表（linked lists）之中。

那种“从最近的标记的（mentioned）特化存储处的尾部开始查找”的编译器，在严格的Fibonacci计算方面总是具有优势。图C.2展示了结果。



图C.2 一个给定模板的查找开销 vs. 特化开销

首先要注意的是，对于所有编译器而言，数字都比较小。Microsoft Visual C++ 7.1的所耗费的时间（差）最大，该时间以N^2级别上升，这正如你预期——如果一个给定模板的所有特化都存储在一个单链表（single linked list）中的话。对于Metrowerks在查找方面的古怪的性能表现我们无法解释，但我们可以说，对于每一个标记缓存查找，至少能平均接近零开销。

C.3.3 标记（Mentioning）一个特化

一旦一个平凡的类模板特化已经被标记（mentioned），那么对它进行实例化看上去就无需付出代价了。例如：

```
template <class T> struct trivial { typedef T type; };
typedef mpl::vector<trivial<int> > v; // 只是一个“标记”
trivial<int>::type x;                 // 无开销的实例化
```

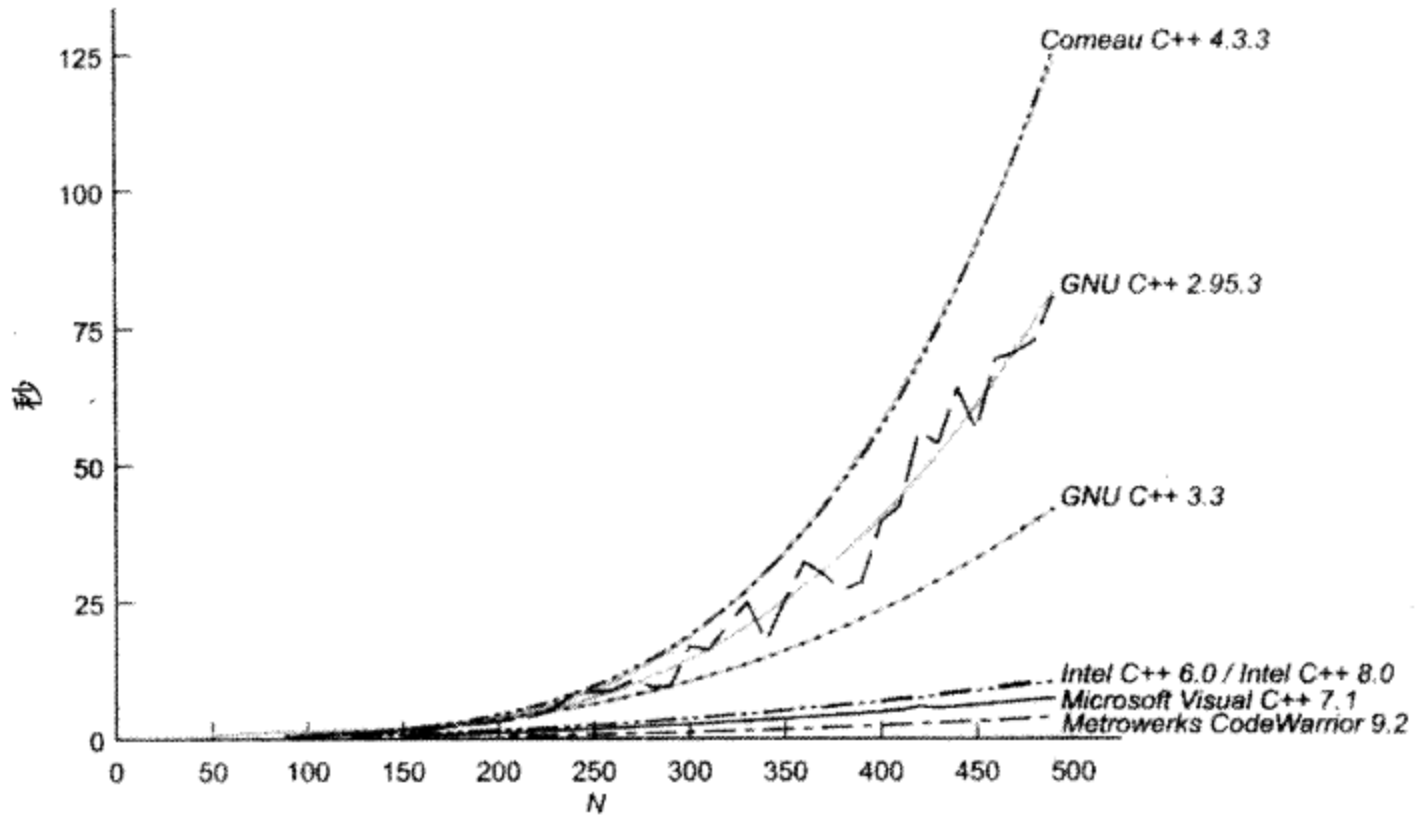
考虑到我们在根据模板实例化来度量效率，这个结果可能会让你感到惊讶。一开始这也让我们感到惊讶，但我们认为现在已经搞明白其中的缘由了。

如你所知，trivial<int>是一个完全合法的类型，即使在它被实例化之前也是如此。编译器需要为该类型标以某种惟一的标识符，例如，它们借此可以识别出出现的两个mpl::vector <trivial<int> >是同一个类型。编译器当然可以采用人类可读的名字来识别（标记）trivial<int>，但随着类型变复杂，匹配长名字可能会带来很大的开销。我们相信大多数编译器在第一次标记（mention）一个模板特化时会分配一个“标记缓存记录（memoization record）”，并且使之保持“empty”状态，直到该特化被实例化的那一刻为止。这样，特化的标记缓存记录的地址就可被用作惟一标识符。一个类模板的定义越简单，标记缓存记录的“empty”和“full”状态彼此之

间就越接近，从而实例化所需的时间就越少。

你也许会问自己通过计数模板特化来度量元程序的效率是否真的有意义——如果某些实例化是被有效地即刻完成的话。答案是“是的，有意义”。因为这里不存在循环，所以一个元函数的实现品可以直接标记（mention）一个模板特化的常量成员。所见即所得，换句话说，直到元函数实例化它所标记（mention）的那些模板之一。这样，每一个元函数调用只能直接创建一个常量数目的新标记缓存记录。“避开”这种常量因素的限制的惟一方式是，该元函数去实例化另一个模板。

图C.3中的曲线展示了简单标记（mentioning）（而不实例化）同一个模板的N个不同特化的开销。正如你看到的那样，不同的编译器的表现差别很大，Comeau和两个版本的GCCs在这方面表现出来的时间复杂度均为$O(N^3)$。



图C.3 标记同一个模板的N个特化的开销

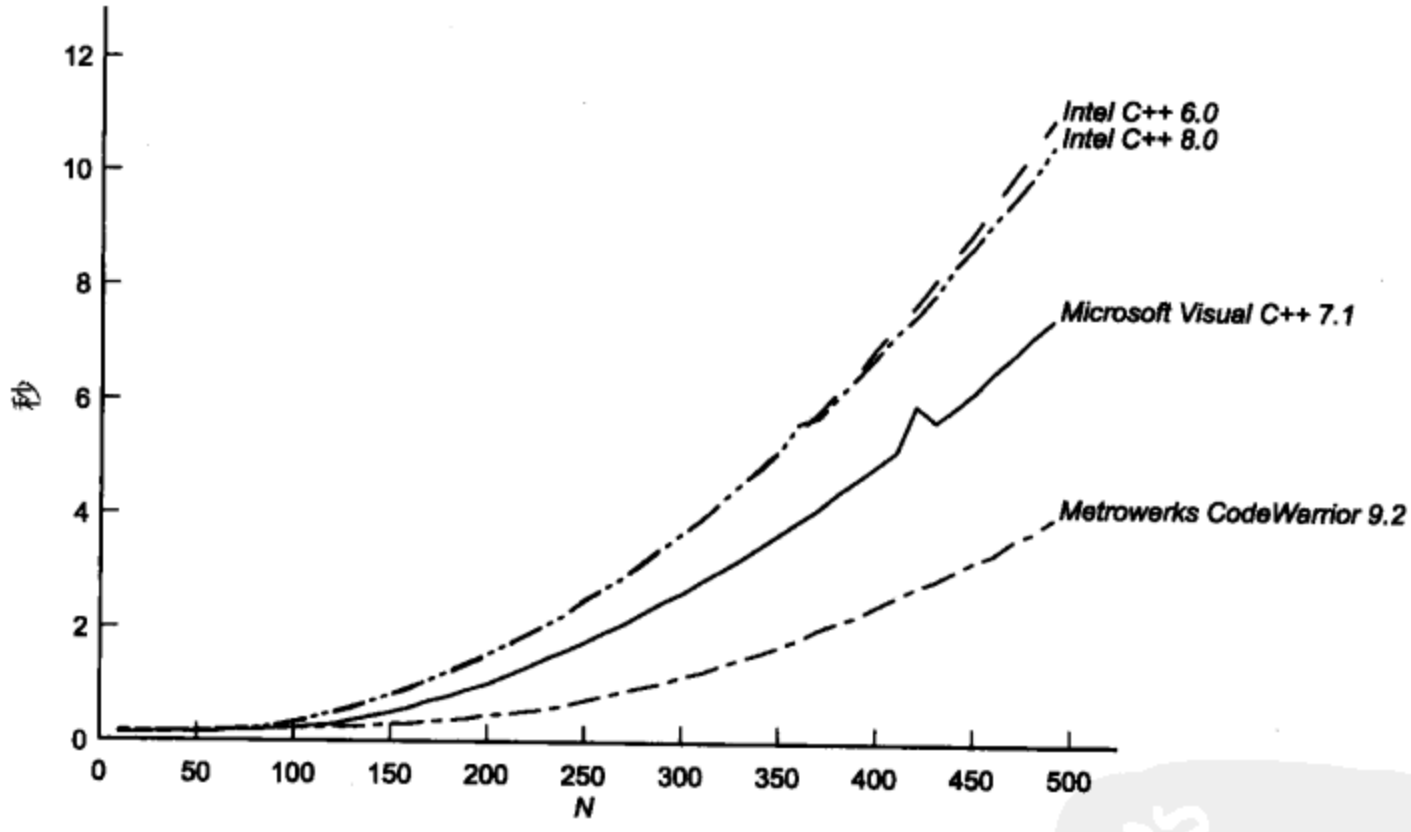
通过将复杂度为$O(N^3)$的数据从图中排除掉，我们可以看到在其他编译器上标记（mentioning）同一个模板的N个特化的开销的复杂度为$O(N^2)$（参见图C.4）。

C.3.4 嵌套的模板实例化

对于编译期编程来说，嵌套的模板实例化是不可或缺的东西。即使你可以使用MPL算法来避免看到递归，它们仍然隐藏在底层，悄悄地进行着。为了测试递归模板实例化的效果，我们编译以下简单的程序，其间不断增大N的值：

```
template< int i, int test > struct deep
  : deep<i-1,test>
{};

template< int test> struct deep<0,test>
```

```
{
    enum { value = 0 };
};
template< int n > struct test
{
    enum { value = deep<N,n>::value };
};

int main()
{
    return test<0>::value + test<1>::value + test<2>::value
        + test<3>::value + test<4>::value + test<5>::value
        + test<6>::value + test<7>::value + test<8>::value
        + test<9>::value;
}
```

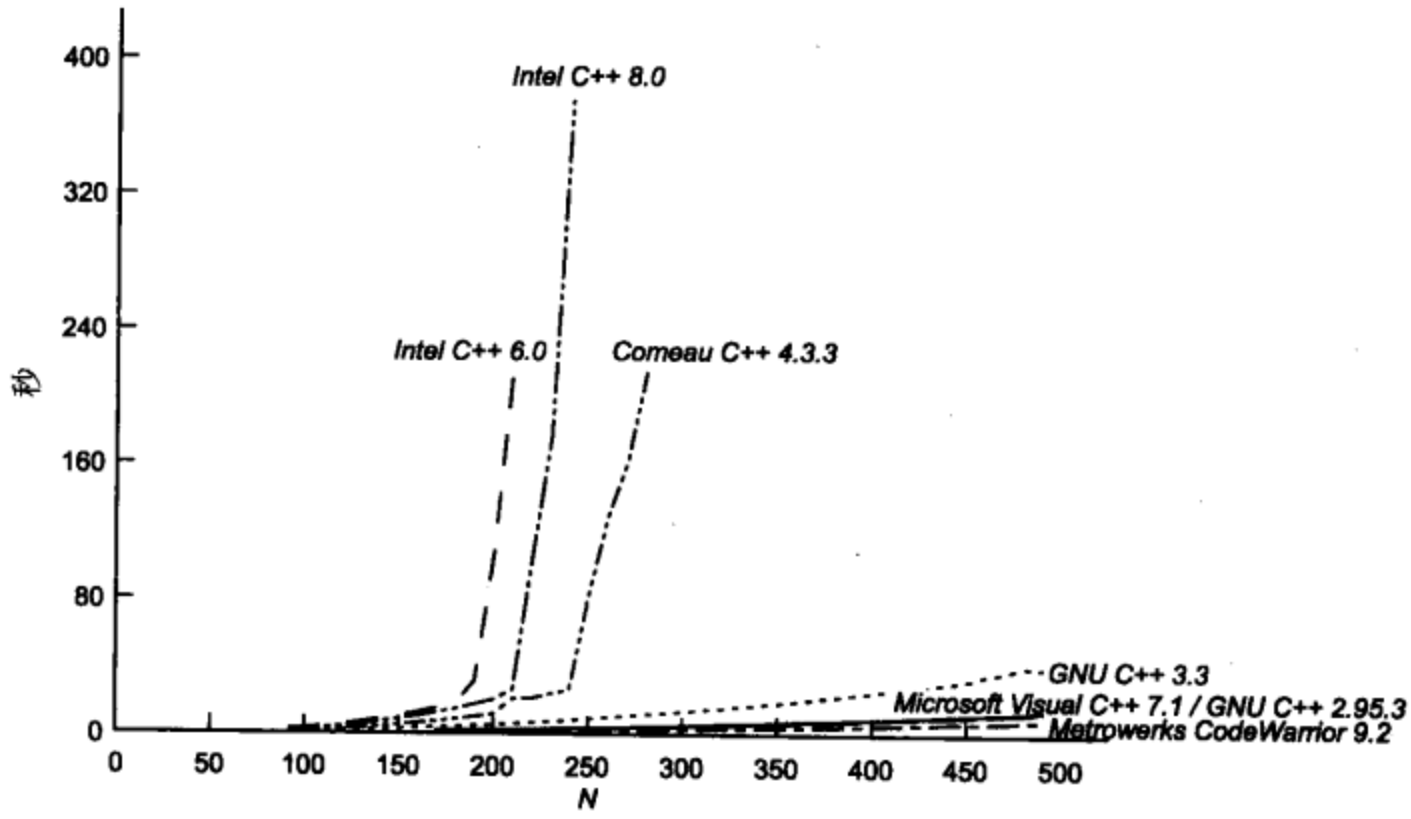


图C.4 标记（Mentioning）同一个模板的N个特化的开销

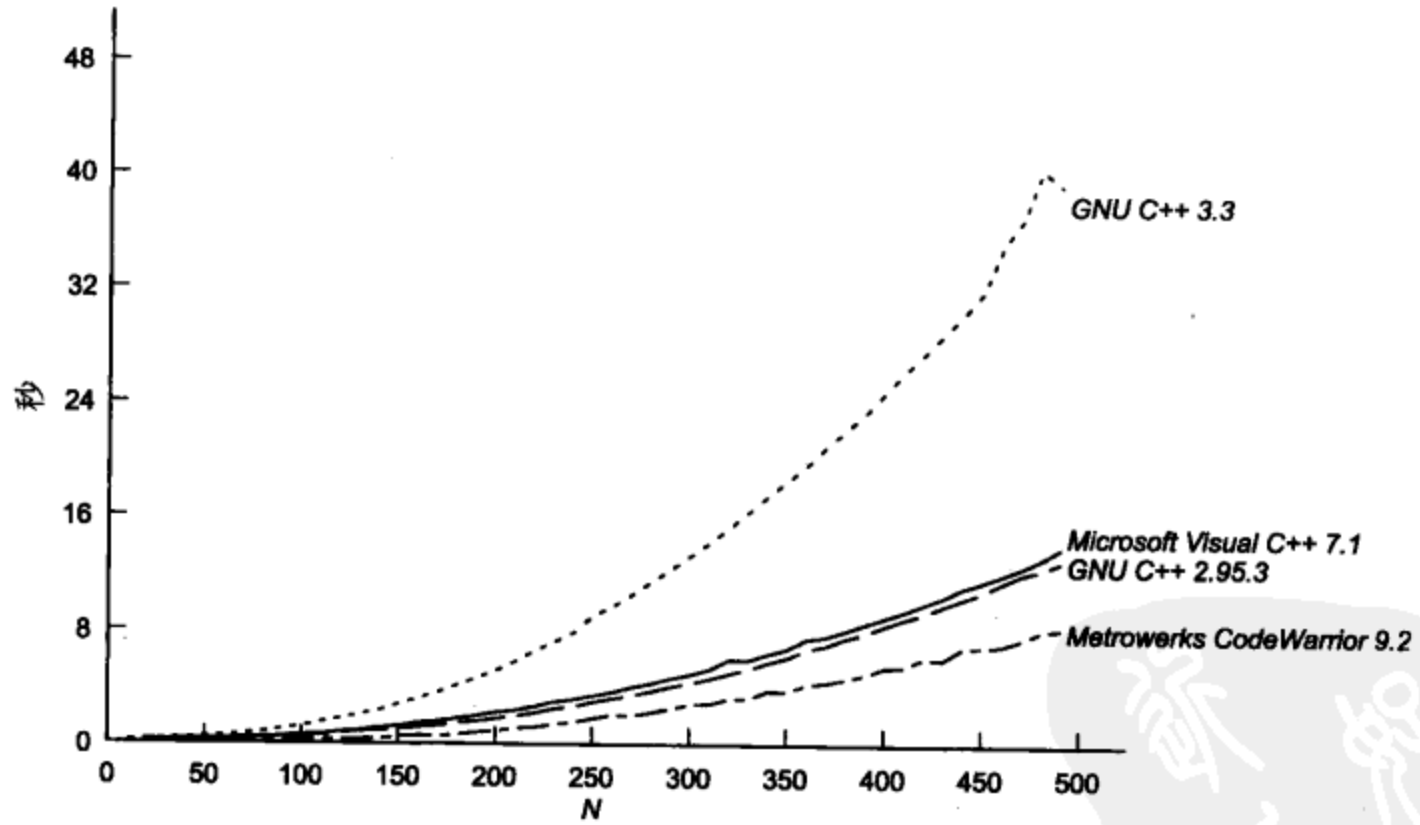
正如你可以从图C.5中看到的那样，当N值较小时，所有编译器表现的行为是类似的，但是当N达到100左右时，基于EDG的编译器开始消耗更多的时间，当N大约为200时，它们的编译时间呈爆炸性增长[⊖]。

如果从图C.5中移除基于EDG的编译器的数据曲线，我们可以看清楚其他编译器表现出的行为，它们均表现出$O(N^2)$时间复杂度（参见图C.6）。

⊖ 最终会导致编译资源耗尽。——译者注



图C.5 时间 vs. 嵌套深度



图C.6 时间 vs. 嵌套深度

C.3.4.1 无转发的嵌套实例化

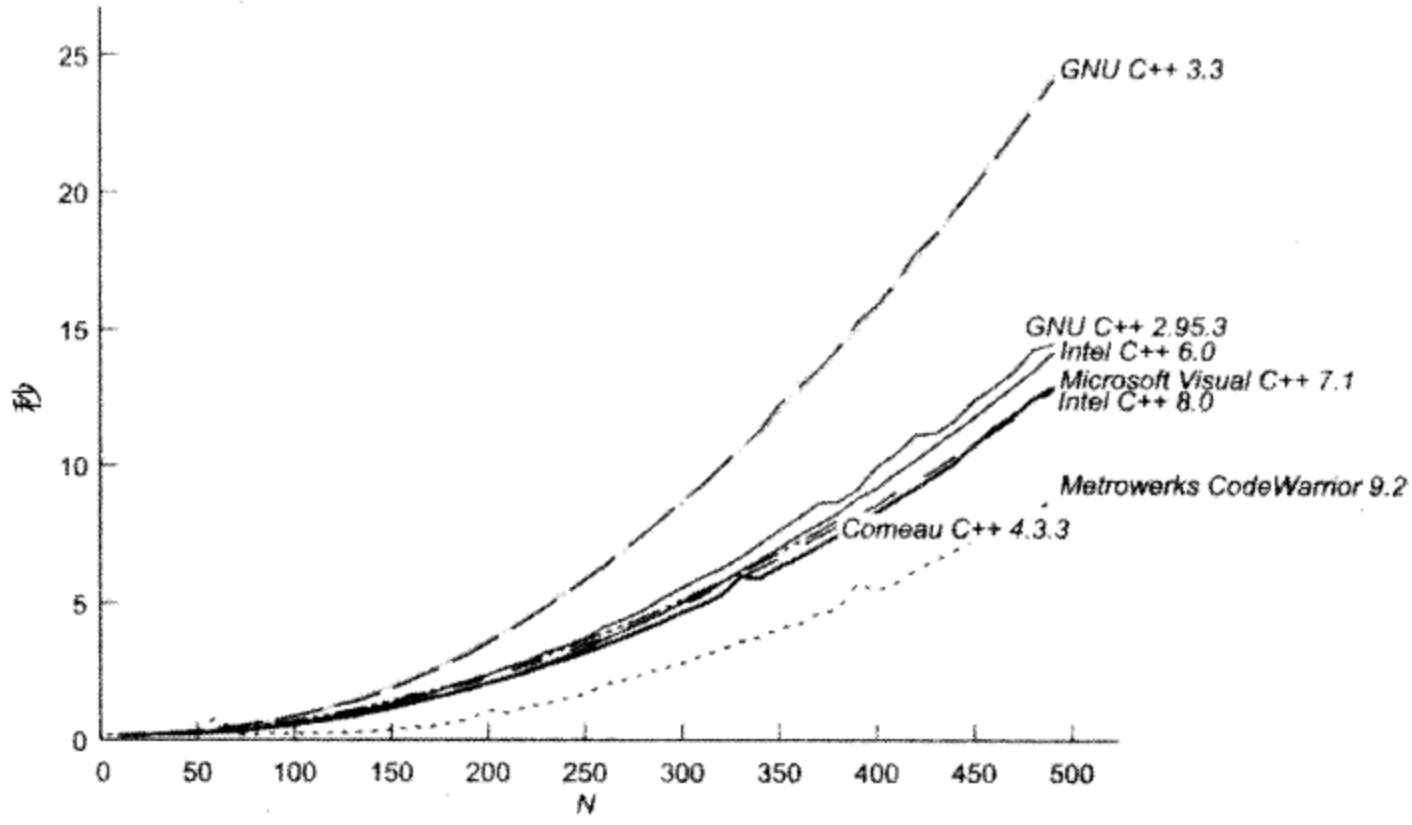
以下版本的deep模板，它在每一层都再次声明它的::value，而不是继承它[⊖]，揭示了时间爆炸是由深度嵌套的元函数转发所触发的：

```
template< int i, int test > struct deep
{
```

⊖ 参见C.3.4第一行代码。

```
    enum { value = deep<i-1,test>::value }; // no forwarding
};

template< int test> struct deep<0,test>
{
    enum { value = 0 };
};
```

正如你在图C.7中看到的那样，EDG病态的行为不见了，基于EDG的编译器[⊖]表现出的时间介于其他大多数编译器之间。GNU C++ (GCC) 3.3的性能也有稍许提高，但其时间复杂度没有改变，和其余编译器一样，仍然是$O(N^2)$。



图C.7 时间 vs. 嵌套深度（无转发）

元函数转发对于简化程序的价值巨大，因而我们不愿反对它，即使对于EDG用户来说也是如此。据我们所知，在现实代码中我们从未撞到“EDG之墙”[⊖]。事实上，我们只是在写作本书过程中，编写测试例子时才发现这一情况。也就是说，如果你发现你的元程序在基于EDG的编译器上执行起来比在其他编译器上慢，你也许需要复审代码是不是使用了深层转发（deep forwarding）。

C.3.4.2 使用递归开解（Recursion Unrolling）限制嵌套深度

对于那些复杂度应该是O(N)的操作而言，即使是$O(N^2)$的行为也是在没什么吸引力的。由于无法进入编译器实现品内部并对其进行修复，所以我们只能减少嵌套实例化的深度。由于时间是深度的平方，因此能够减少深度两倍的因素就可以减少四倍的时间，能够减少四倍时间的因

⊖ 即Intel C++ 6.0/8.0和Comeau C++ 4.3.3。

⊖ 我们曾偶然看到一些在基于EDG的编译器上编译速度变慢的元程序，但当时不知道所谓的深层转发的效果，而且其行为看上去也不像我们曲线针对深层转发所指示的那般陡峭。

素就能够使一个递归元函数快十六倍，等等。进一步而言，如果一个编译器具有像“EDG之墙”这样的病态行为，或明确地将其深度限制为某一个数值（有些编译器就是这么做的），那么，减少深度对你而言就有天壤之别了。

考虑这个mpl::find的内脏的实现：

```
namespace boost { namespace mpl {

  template <class First, class Last, class T>

  struct find_impl;

  // 查找序列尾部的"find_impl"
  template <class First, class Last, class T>
  struct find_impl_tail
    : find_impl<
          typename next<First>::type
        , Last
        , T
      >
  {};

  // 如果First指示T即为true
  template <class First, class T>
  struct deref_is_same
    : is_same<typename deref<First>::type,T>
  {};

  template <class First, class Last, class T>
  struct find_impl
    : eval_if<
          deref_is_same<First,T>  // 找到了?
        , identity<First>
        , find_impl_tail<First,Last,T>  // 查找尾部
      >
  {};

  // 终结条件
  template <class Last, class T>
  struct find_impl<Last, Last, T>
  {
      typedef Last type;
  };
}}
```

find_impl立刻导致了这样的后果：对其遍历的序列的每一个元素，都有一层递归。现在我们使用递归开解（recursion unrolling）来改写它：

```
// 查找算法的单个步骤
template <
    class First, class Last, class T
  , class EvalIfUnmatched = next<First>
>
struct find_step
  : eval_if<
        or_<
            is_same<First,Last>                 // 序列为空
          , deref_is_same<First,T>              // 或找到了T
        >
      , identity<First>
      , EvalIfUnmatched
    >
{};

template <class First, class Last, class T>
struct find_impl
{
    typedef typename find_step<First,Last,T>::type step1;
    typedef typename find_step<step1,Last,T>::type step2;
    typedef typename find_step<step2,Last,T>::type step3;
    typedef typename find_step<
        step3,Last,T, find_impl_tail<step3,Last,T>
    >::type type;
};
```

现在对find_impl的每一个调用都等于调用四步find_step，从而减少了四倍的实例化深度。当一个正被查找的序列支持随机访问时，通过针对小于开解因子（unrolling factor）的长度对算法进行特化，还可以使这种手法更加高效，从而可以避免在每一步中对迭代器的身份进行检查。任何时候只要适合于目标编译器，MPL算法就会使用这些技术。

C.3.5 局部特化的数目

图C.8中的曲线展示了在增加局部特化数目的情况下，实例化一个类模板在时间上的效果。

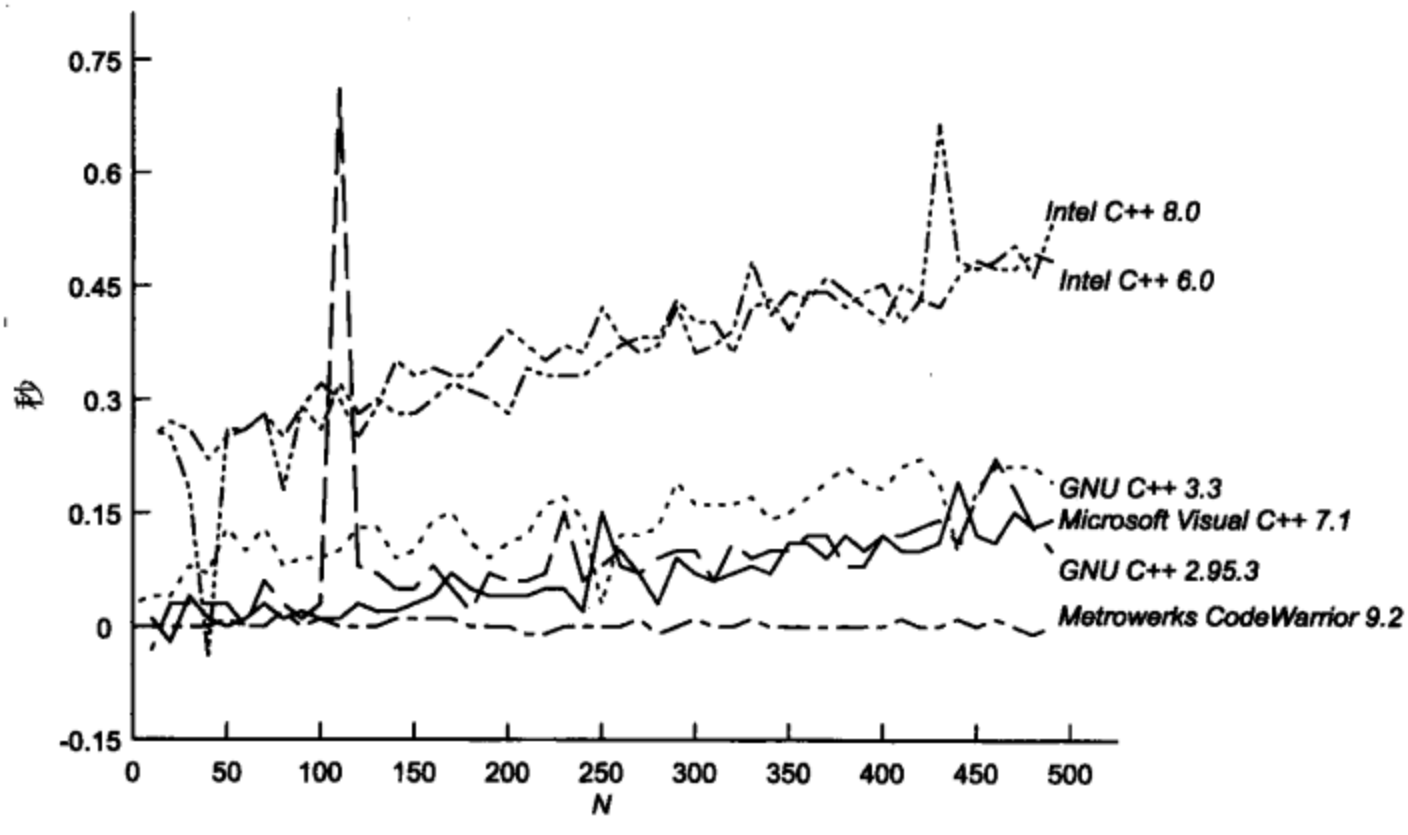
Comeau C++从这个曲线中省略掉了，因为它是病态得慢，即使N==0也是如此，应该是因为一些看上去与局部特化无关的原因。大多数其他编译器表现出了一些效果，但效果是如此小，你完全可以忽略它。

C.3.6 长符号

符号名的长度看上去对编译时间没有任何影响[⊖]。例如，在所有其他方面都相同的情况下，

⊖ 值得指出的是，我们没有测试长函数名字的效果，在这种情况下可能会带来完全不同的影响，因为编译器通常要对函数名字进行重整（mangle）以供链结器（linker）使用。

实例化以下两个模板所带来的时间开销是一样的：



图C.8 实例化时间 vs. 局部特化的数目

```
wee<int>
a_ridiculously_looooooooooooooooong_class_template_name<int>
```

还有，将长符号名字传递给模板对编译时间也没有可测量的效果影响，因此，以下两个例子的实例化的时间开销相同：

```
mpl::vector<a_ridiculously_looooooooooooooooong_class_name>
mpl::vector<int>
```

C.3.7 元函数实参的结构复杂性

我们考虑了三种描述“可能被当作元函数实参传递的template特化”的复杂性的方式：

1. 它的元数（arity）（即模板参数的数目）
2. 它的节点（nodes）数目
3. 它的嵌套深度

例如，mpl::vector30的arity是30，我们对传递具有高的元数的模板给元函数的开销具有很大的兴趣。另一个例子，如果你将每一个惟一的模板特化看成一个节点，以下的“列表（list）”类型就具有四个节点，其深度（depth）也为4：

```
node<int, node<long, node<char, node<void, void> > > >
```

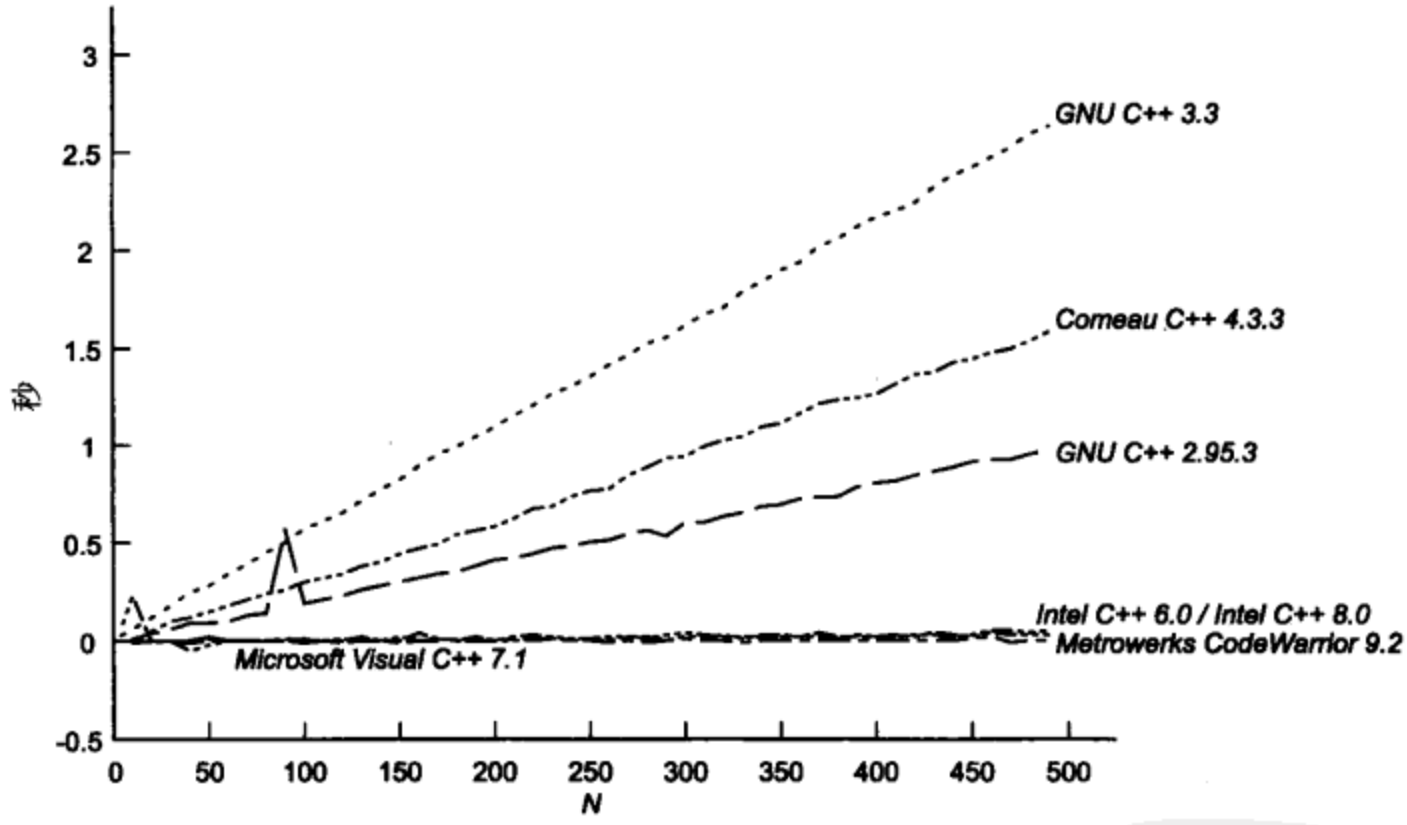
而下面这个“DAG[⊖]”类型则具有四个节点，深度为3：

⊖ Directed Acyclic Graph，有向无环图。

```
// 1    2    3 <== 深度              节点
   node<                            // #1
       node<                        // #2
           node<void,void>          // #3
         , node<int,void>           // #4
       >
     , node<void, void>             // #3再次出现
   >
```

C.3.7.1 结构复杂性测试

除了测量传递像int这样的简单类型的开销外，我们还测量了通过一连串元函数调用递归传递五花八门的复杂结构的开销。如你在图C.9中所见，实参的元数不会导致有关开销，但GCC和Comeau除外，后两者所表现出来的开销和N呈线性增长[⊖]。



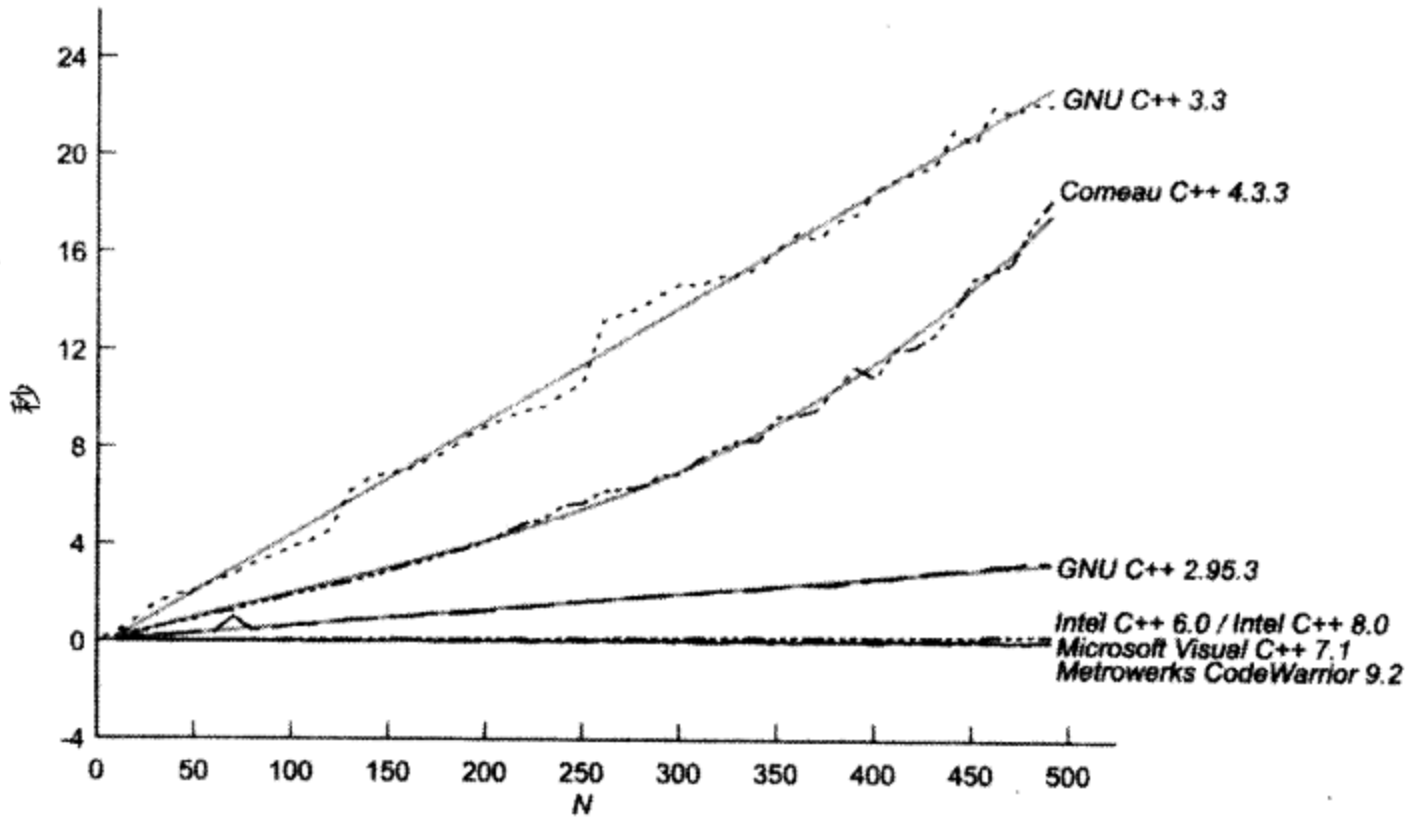
图C.9 递归实例化时间与实参的元数

图C.10显示了当我们传递一个具有N个惟一节点的二叉平衡树（balanced binary tree）（深度为$\log_2 N$）时的情形，从图上可见，曲线相当发散：Comeau的时间复杂度为$O(N^2)$，GCC的新版本反而比2.95.3版更糟糕，前者和N呈线性增长，其余编译器则表现出没有任何开销[⊖]。

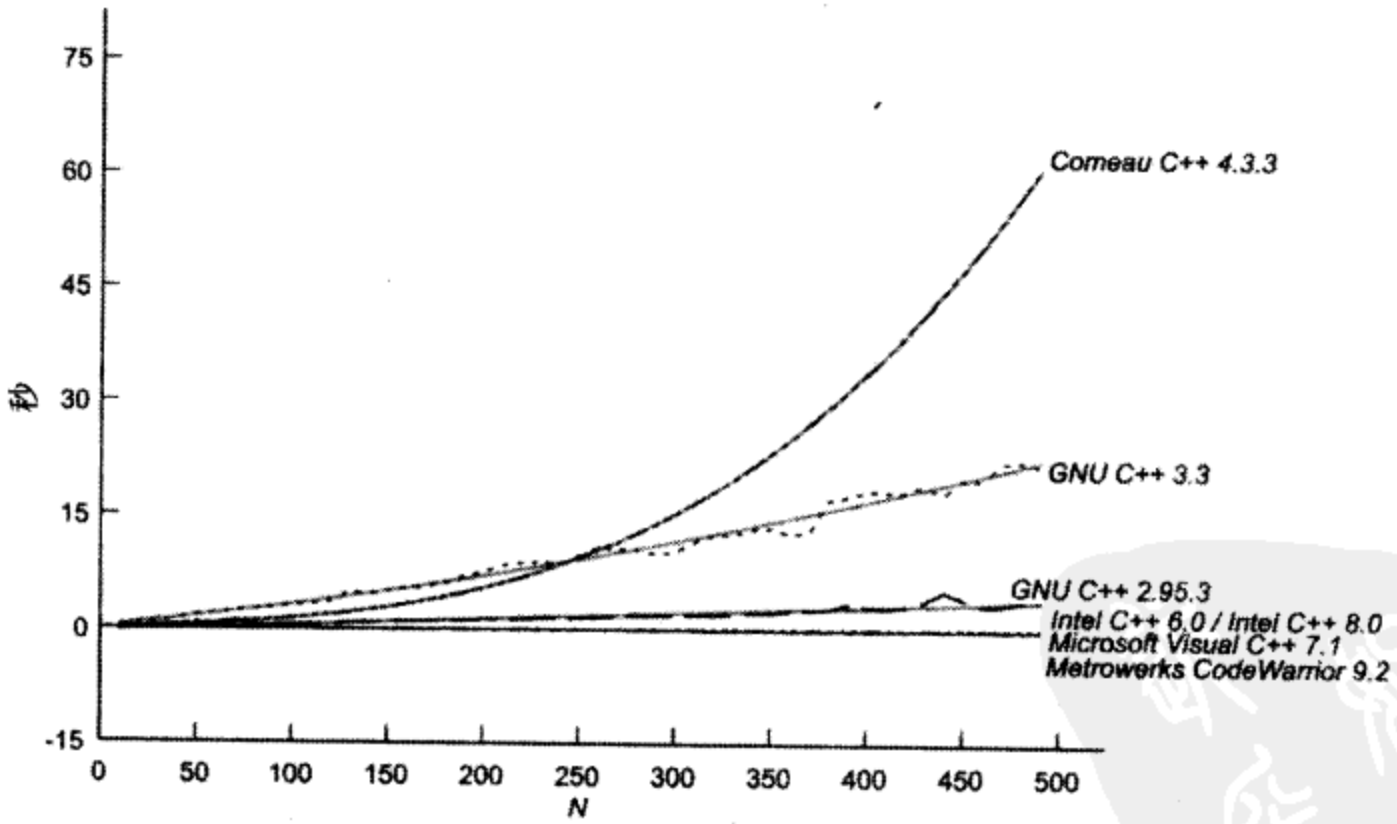
最后，传递一个具有N个不同节点的列表（深度为N）所产生的曲线如图C.11所示。

⊖ Microsoft Visual C++ 7.1看上去对模板参数的数目有个硬编码的限制：63个。如果参数数目超过63，则编译失败，并报以“fatal error C1111: too many template parameters.”

⊖ 事实上，GCC 2.95.3也许在这方面展示出$O(N^2)$的复杂度，但由于系数（coefficient）是如此低，让我们很难确定的确如此。



图C.10 递归实例化时间vs.实参树节点（Argument Tree Nodes）数目



图C.11 递归实例化时间vs.实参嵌套深度（Argument Nesting Depth）

这个变动略微改变了GCC的结果，对此我们茫然不解，给不出合理的解释。其他编译器的效果都没变（Comeau除外）。Comeau测试结果变得突出得糟糕，因此显而易见，它不仅对节点的数目有反应，对嵌套深度也有反应。对于任何其他编译器而言，big-O复杂度都没有改变。

C.3.7.2 使用序列派生（Sequence Derivation）来限制结构性复杂度

序列派生（参见第5章的描述）在对抗编译期实参复杂性的影响方面是一个威力极大的武器，对于类似于vector的序列更是如此。很大程度上这要归结于它们的迭代器的结构。回忆一下

tiny_iterator的声明：

```
template <class Tiny, class Pos>
struct tiny_iterator;
```

一个tiny_iterator特化标记（mentions）它以模板实参所遍历的序列的整个名字。如果我们推断序列具有较大的容量（capacity），那就很容易看到

```
mpl::vector_iterator<my_vector, mpl::int_<3> >
```

可能比操纵下面的东西更快：

```
mpl::vector_iterator<mpl::vector30<int,..., Foo>, mpl::int_<3> >
```

附录D　MPL可移植性摘要

许多相当符合标准的编译器都可以处理MPL。我们没有测试现有的每一款编译器，但表D.1列出了一些正文中描述的已知可以处理MPL的编译器，无需任何特别的用户干预。谨记这并非一个完整列表，你可以参考配书光碟，以便了解更详细的可移植性报告。

表D.1　一些无需用户干预（采用迂回解决方式）的编译器

编译器	版　本
Comeau	4.3.3
GCC	3.2.2
GCC	3.3.1
GCC	3.4
Intel C++	7.1
Intel C++	8.0
Metrowerks CodeWarrior	8.3
Metrowerks CodeWarrior	9.2
Microsoft Visual C++	7.1

表D.2中列出的编译器对模板的支持不够完善，需要来自用户的一些帮助，如表中最后一列所示。参见配书光碟，以便了解有关这些迂回方案的细节。

表D.2　需要用户干预的编译器

编译器	版　本	有问题的领域
Borland C++	5.5.1	Lambda表达式、整型常量表达式
Borland C++	5.6.4	Lambda表达式、整型常量表达式
GCC	2.95.3	Lambda表达式
Microsoft Visual C++	6.0 sp5	Lambda表达式、ETI①
Microsoft Visual C++	7.0	Lambda表达式、ETI

① ETI即“Early Template Instantiation（早期模板实例化）”，有关此问题的解释参见配书光碟或http://www.boost.org/libs/mpl/doc/tutorial/eti.html。

最后，一些编译器的模板系统问题太多了，以至于即使有用户的帮助，我们还是无法使MPL工作（见表D.3）。请注意一个事实，出现在这个列表中编译器版本并不意味着将来的版本也无法处理MPL，一些厂商正在努力工作来修正这个问题，但是，到本书写作时，还是有一些厂商无动于衷。

表D.3　已知的不能处理MPL的编译器

编译器	版本
HP aCC	03.55
Sun CC	5.6

参考文献

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. ISBN: 0201704315, Boston, MA: Addison-Wesley, 2001.
《C++设计新思维》侯捷 /于春景 译 台湾碁峰信息股份有限公司/华中科技大学出版社，2003年。

[AS01a] David Abrahams and Jeremy Siek . "Policy Adaptors and the Boost Iterator Adaptor Library," Second Workshop on C++ Template Programming. October 2001. http://www.boost-consulting.com/writing/iterator_adaptors0.pdf.

[AS01b] David Abrahams and Jeremy Siek. "Boost Iterator Adaptor Library." November 2001. http://www.boost-consulting.com/writing/iterator_adaptors0.html.

[Bent86] J. L. Bentley. "Programming pearls: Little languages." *Communications of the ACM*, 29(8):711-721, August 1986. http://doi.acm.org/10.1145/6424.315691.

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. ISBN: 0471197130, New York: Wiley, 1998.

[BN94] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. ISBN: 0201533936, Reading, MA: Addison-Wesley, 1994.

[CDHM01] Steve Cleary, Beman Dawes, Howard Hinnant, and John Maddock. "Compressed Pair." http://www.boost.org/libs/utility/compressed_pair.htm.

[Cop96] James O. Coplien. "A Curiously Recurring Template Pattern." In Stanley B. Lippman, editor, *C++ Gems*, pp. 135144, New York: Cambridge University Press, 1996.

[Dew02] Steve Dewhurst, "A Bit-Wise Typeof Operator, Part 1," *Computer User's Journal* 20(8), August 2002.
"A Bit-Wise Typeof Operator, Part 2," *Computer User's Journal* 20(10), October 2002.
"A Bit-Wise Typeof Operator, Part 3," *Computer User's Journal* 20(10), December 2002.

[Dimov02] Peter Dimov. "Boost: bind.hpp documentation." http://www.boost.org/libs/bind.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. ISBN: 0201633612, Reading, MA: Addison-Wesley, 1999.

[GotW50] Herb Sutter. "Guru of the Week #50: Using Standard Containers." http://www. gotw.ca/gotw/050.htm.

[Guz03] Joel de Guzman. "MAJOR BREAK-THROUGH !!! Yabadabadoo... Must Read :-)."

http://sf.net/mailarchive/forum.php?thread_id=529112&forum_id=1595.

[Guz04] Joel de Guzman, Hartmut Kaizer, et. al. "The Spirit Parser Framework." http://spirit. sf.net.

[Heer02] Jan Heering and Marjan Mernik. "Domain-Specific Languages for Software Engineering." Proceedings of the 35th Hawaii International Conference on System Sciences2002. http://www.computer.org/proceedings/hicss/1435/volume9/14350279.pdf.

[Hudak89] Paul Hudak. "Conception, evolution, and application of functional programming languages." ISSN:0360-0300, pp. 359-411, ACM Computing Surveys 21, no. 3, New York: ACM Press, 1989. http://doi.acm.org/10.1145/72551.72554.

[IBM54] "Preliminary Report: Specifications for the IBM Mathematical FORmula TRANslation System FORTRAN." Programming Research Group, Applied Science Division, IBM Corporation, November 10, 1954.

[ISO98] ANSI/ISO C++ Committee. *Programming Languages*—C++. ISO/IEC 14882: 1998(E). American National Standards Institute, New York, 1998. http://webstore. ansi.org/ansidocstore/find.asp?find_spec=14882

[Joh79] Stephen C. Johnson. *Yacc: Yet Another Compiler Compiler*. UNIX Programmer's Manual, Vol. 2b, pp. 353-387, Holt, Reinhart, and Winston, 1979.

[KM00] Andrew Koenig and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. ISBN: 020170353X, Boston, MA: Addison-Wesley, 2000.

[KV89] Thomas Keffer and Allan Vermeulen. *Math.h++ Introduction and Reference Manual*. Corvallis, Oregon: Rogue Wave Software, 1989.

[LaFre00] David Lafreniere. "State Machine Design in C++." *C++ Report*, May 2000. http://www.cuj.com/documents/s=8039/cuj0005lafrenie/.

[Land01] Walter Landry. "Implementing a High Performance Tensor Library." Second Workshop on C++ Template Programming, 2001. http://citeseer.nj.nec.com/landry01implementing.html.

[Mad00] John Maddock. "Static Assertions." http://www.boost.org/libs/static_assert.

[Mart98] Robert Martin. "UML Tutorial: Finite State Machines." *C++ Report*, June 1998. http://www.objectmentor.com/resources/articles/umlfsm.pdf.

[MK04] Paul Mensonides and Vesa Karvonen. "The Boost Preprocessor Library." http://www.boost.org/libs/preprocessor.

[MS00a] Brian McNamara and Yannis Smaragdakis. "Static Intefaces in C++." First Workshop on C++ Template Programming, Erfurt, Germany, October 10, 2000. http:// oonumerics.org/tmpw00/.

[MS00b] Brian McNamara and Yannis Smaragdakis. "FC++." 2000-2003. http://www.cc. gatech.edu/~yannis/fc++.

[n1185] Herb Sutter. "vector<bool> Is Nonconforming, and Forces Optimization Choice."

http://www.gotw.ca/publications/N1185.pdf.

[n1211] Herb Sutter. "vector<bool>: More Problems, Better Solutions." http://www.gotw.ca/ publications/N1211.pdf.

[n1424] John Maddock. "A Proposal to Add Type Traits to the Standard Library." http:// anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1424.htm.

[n1519] John Maddock. "Type Traits Issue List." http://anubis.dkuug.dk/jtc1/sc22/wg21/ docs/papers/2003/n1519.htm.

[n1521] Gabriel dos Reis. "Generalized Constant Expressions." document number N1521-03-0104. http://www.openstd.org/jctl/wg21/docs/papers/2003/nl1521.pdf.

[n1550] David Abrahams, Jeremy Siek, and Thomas Witt. "New Iterator Concepts." http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1550.htm.

[Nas03] Alexander Nasonov. "Re: boost::tuple to MPL sequence." http://lists.boost.org/MailArchives/boost/msg46771.php.

[Sey96] John Seymour, Views— A C++ Standard Template Library Extension. http://www.zeta. org/au/~jon/STLviews/doc/views.html.

[SL00] Jeremy Siek and Andrew Lumsdaine. "Concept Checking: Binding Parametric Polymorphism in C++." First Workshop on C++ Template Programming, Erfurt, Germany, October 10, 2000. http://oonumerics.org/tmpw00/.

[SS75] Gerald Sussman and Guy L. Steele Jr. "Scheme: An interpreter for extended lambda calculus." MIT AI Memo 349, Massachusetts Institute of Technology, May 1975.

[Strou03] Bjarne Stroustrup. *The C++ Standard: Incorporating Technical Corrigendum No. 1*. BS ISO/IEC 14882:2003. ISBN: 0470846747, New York: Wiley, 2003.

[Unruh94] Erwin Unruh. "Prime number computation." ANSI X3J16-94-0075/ISO WG21-462. 1994.

[Veld95a] Todd Veldhuizen. "Blitz++." http://www.oonumerics.org/blitz/.

[Veld95b] Todd Veldhuizen. "Using C++ Template Metaprograms." C++ *Report*, SIGS Publications Inc., ISSN 1040-6042, Vol. 7, No. 4, pp. 36-43, May 1995.

[Veld04] Todd Veldhuizen. *Active Libraries and Universal Languages*. Doctoral Dissertation, Indiana University, Computer Science, 17 May 2004. http://www.cs.chalmers.se/~ tveldhui/papers/2004/dissertation.pdf.

[VJ02] David Vandervoode and Nicolai M. Josuttis. *C++ 模板: The Complete Guide*. ISBN: 0201734842, Boston, MA: Addison-Wesley, 2002.
《C++ Templates 全览》侯捷/荣耀/姜宏 译 台湾 碁峰信息股份有限公司，2004年。

[WP99] Martin Weiser and Gary Powell, The View Template Library. http://www.zib.de/weiser/vtl.

[WP00] Martin Weiser and Gary Powell, The View Template Library. First Workshop on C++ Template Programming, Erfurt, Germany, October 10, 2000.