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

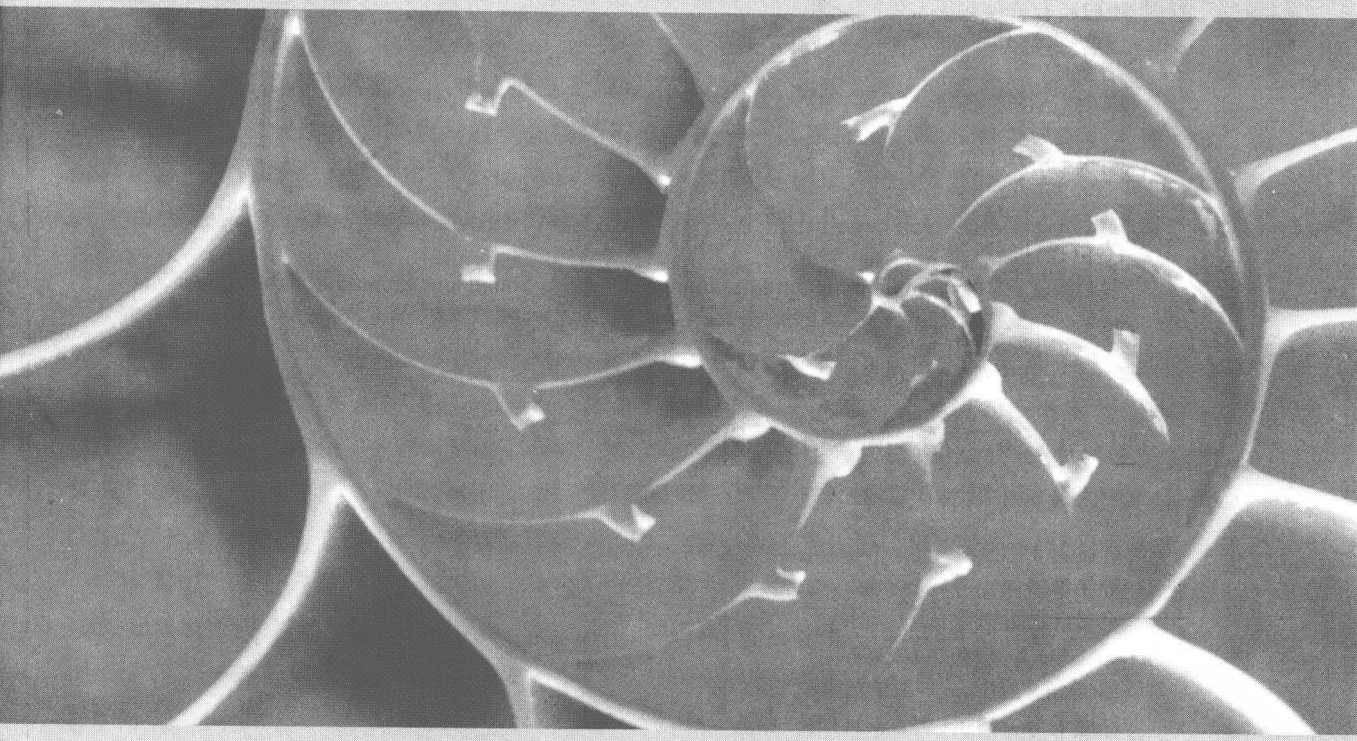
光盘

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。