PEARSON

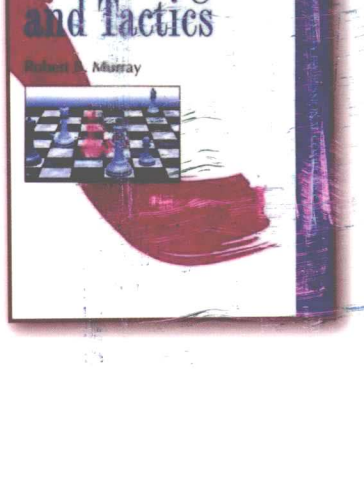
C和C++实务精选
品味岁月积淀，读享技术菁华

C++编程惯用法
——高级程序员常用方法和技巧

[美] Robert B. Murray 著　　王昕 译

C++ Strategies and Tactics

- ACCU主席Francis Glassborow倾力推荐
- 传授如何在C++中作出选择的专家级读本
- 阐释如何使用C++进行更好编程的真知灼见



人民邮电出版社
POSTS & TELECOM PRESS

PEARSON

C++编程惯用法

——高级程序员常用方法和技巧

[美] Robert B. Murray 著　　王昕 译

人民邮电出版社
北京

图书在版编目（CIP）数据

C++编程惯用法 : 高级程序员常用方法和技巧 /
(美) 莫瑞 (Murray, R. B.) 著 ; 王昕译. -- 北京 : 人
民邮电出版社, 2012.10
ISBN 978-7-115-29084-7

Ⅰ. ①C… Ⅱ. ①莫… ②王… Ⅲ. ①C语言－程序设
计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2012)第172707号

版 权 声 明

C++编程惯用法——高级程序员常用方法和技巧

◆ 著　　[美] Robert B. Murray
译　　王　昕
责任编辑　傅道坤

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800 × 1000　1/16
印张：15
字数：303 千字　　2012 年 10 月第 1 版
印数：1-3 000 册　　2012 年 10 月北京第 1 次印刷

著作权合同登记号　图字：01-2011-5635 号

ISBN 978-7-115-29084-7

定价：39.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

内容提要

在本书中，C++专家 Robert B . Murray 与我们分享了他宝贵的经验和建议，以帮助初中级 C++程序员得到进一步的提高。

本书总共分为 11 章，先后介绍了抽象、类、句柄、继承、多重继承、考虑继承的设计、模板的高级用法、重用、异常以及向 C++的移植等相关的诸多话题。在书中，作者大量采用了实际开发中的代码作为示例，向读者展示了那些有用的编程策略，并对那些有害的做法进行了警示。为了帮助读者更好地理解，在每一章结束前，在该章中介绍过的主要内容都被放到了一个列表中，此外，书中还给出了一些问题来激励读者们进行更多的思考和讨论。

本书适合具有一定 C++编程经验的程序员和项目经理阅读，也适合对 C++编程的高级主题感兴趣的读者参考。

序

在专家看来，C++可以帮助软件设计者和程序员构造出模块化、易维护以及高性能的系统。然而，对新手来说，这门语言的复杂程度是一个不小的威胁。C++中有着许多不同的语言特性，只有具有一定经验之后，我们才会知道各种特性的适用情况。

本书的目的就是加速读者的这种学习进程。大部分成功的C++程序员并不会简单地从语言规则中复述出某些章节或是条款，相反，他们总是习惯于自己以往工作中所获得的行之有效的那些惯用法和技巧。我们的目的就是帮助那些C++的新手学习到那些在实践中最有效的惯用法。同时，我们也会在书中指出那些我们经常接触到的C++缺陷。

在本书中，我们的讨论并不试图覆盖整个语言。那些有关语言语义的精确定义可以查看参考手册。我们主要帮助读者构造出可以被那些不是C++语言专家的人所理解的程序。我们不但讨论了那些使程序变得优雅及高效的技巧，也展示了使它们更容易被理解和维护的方法。

致谢

本书中的想法和编程惯用法几乎都不是我发明的。我的目的就是把它们（在我8年的C++使用过程中从其他人那学到的并被我认为是最重要的策略和战术）用一种C++新手也能学会的方式展现给读者。这些收获中的部分来自于实际的开发项目中（这些项目被我们从 C 移植到C++上）的经验，另外一些则来自于和其他高手的讨论。

关于模板以及函数库设计的许多好的想法（包括那些隐藏在本书所给出的容器类后的想法）都来自于最初由Martin Carroll、Andrew Koenig以及Jonathan Shopiro所设计的USL标准组件库[1]中的类。对于在本书中出现的任何错误，则都由我负责。Andrew Koenig是一个C++语言专家，他对我的帮助非常大。另外，那些参与过我所举行过的几期“C++策略与战术”研讨会的人也帮助启发了我书写本书，并对书中的想法进行提炼。其他的重要的想法来自于Tom Cargill、John Carolan、Jim Coplien、Mark Linton、Gerald Schwarz，当然，还有Bjarne Stroustrup，要不是他发明了C++语言，本书也就不可能出现了。

Brian Kernighan阅读过本书的多次草稿，他极好的反馈意见起了很大的帮助作用。同时我还要感谢David Annatone、Steve Buroff、Tom Cargill、Bill Hopkins、Cay Horstman、Lorraine Juhl、

[1] 译注：关于USL标准组件的文章参见于1993年6月的《C++ Report》中Martin Carroll所写的《Design of the USL Standard Components》。

Peter Juhl、Stan Lippman、Dennis Mancl、Scott Meyers、Barbara Moo、Lorraine Weisbrot Murray、Bjarne Stroustrup、Clovis Tondo、Steve Vinoski 以及 Christopher Van Wyk，他们对本书早期的草稿提出了意见。此外，Lorraine Weisbrot Murray[1] 对我的鼓励、理解、支持以及爱使得这一切努力都成为了可行的。

[1] 译注：Lorraine Weisbrot Murray 是 Robert B. Murray 的妻子（经 Robert B. Murray 证实）。

前 言

C++是一门规模庞大的编程语言。只了解 C++规则的程序员就和只知道棋子如何移动的棋手一样（失败）。为了获得成功，还要学习一些相关的法则和策略。

本书所面对的读者是使用 C++语言的初级及中级程序员们，他们一方面期待学到更多有关使用这门语言的知识，一方面又没有时间去参加一个大学学期那么长的一门课程。我们假设读者已经了解了一些 C++的基础知识——如：什么是成员函数？如何使用 public、private 以及 protected 等。本书的读者没有必要是一个 C++专家，在每次提及一些该语言中比较高级的话题（如多继承）时，我们都会给出一个“回顾”栏目对此进行简短的总结。我们关注的是那些被证明在实际中十分有效的策略。在本书中，我们将避免多谈理论，而尽量为读者给出例子及实用的建议。

我们尽量保证不偏离惯例。在本书中，没有一个想法或技术会超出语言的基础，它们大多来自于有着多年 C++经验的程序员日常的实践中。很显然，最新的想法总是具有最大的风险。我们希望能够避免将 C++的初级用户引导到那些想法上去。除了第 10 章之外[1]，本书中所有的技术都经过了实践的证明。

我们也无意将读者变成 C++专家。C++中也包含了一些阴暗的角落，我们会在书中将它们一一指出并教导读者如何避免它们。写出依赖于语言中定义含糊、意义微妙的规则的程序是不好的，即便作者本人清楚它的意义并保证它可以正确运行，下一个来维护这段代码的人也可能未必能够做到这点。比较好的做法是：坚持使用语言中那些被广泛使用及理解的部分来书写程序。

关于示例代码

本书中的示例代码都通过了编译(或是取自一个通过编译的文件)。为了使代码尽可能短小，我们在此略掉了那些和主题无关的周边代码（如#include 指令）。

本书中的示例大多只处理接口，而不是实现。因此，为了清晰起见，大部分的示例代码都略去了类的私有部分：

---

[1] 译注：在本书写作过程中。异常还只是 C++中的一个新增特性，但到了现在，在 C++中异常已经发展得比较成熟了。

```
class  Something {
    //私有部分被省略……
public:
    Something();
);
```

本书的组织

在第 1 章中，我们讨论了如何选择一种合适的高阶抽象方式来实现设计，并着重阐述了抽象和其实现之间的区别。

后续的两章主要关注如何在 C++的类中实现较高层次的抽象。第 2 章的讨论涵盖了自高阶的设计策略到底层的接口和实现细节。在第 3 章中，我们阐述了几种句柄在类实现中的用法。

接下来的 3 章介绍了继承。第 4 章中讨论了公用、保护、私用继承的区别，并给出了如何选择使用各种继承方式的参考意见。第 5 章中的讨论主要针对超过基类不止一个的继承，我们讨论了应该如何使用它以及有关多继承的一些难以掌握的细节之处。在第 6 章中，我们讲述了一些有关如何构建类的方法，以使得其他人可以把这些类作为他们的基类来发展他们自己的继承体系。

模板，作为 C++语言的一个新特性，其功能非常强大，它必将对人们以后编写 C++程序造成极大的影响。我们在第 7 章和第 8 章中详细地讲解了模板。由于大部分 C++用户对于模板还不甚了解，我们开始只是以基础为始，先描述一些用来构建有效模板的技巧，接着才构建各种各样的模板。包括：使用模板来实现智能指针，一些简单的容器，最后是一个接近实际应用的容器（一个 List 模板）。然后我们才讨论一些关于模板的高级技巧，如：如何对那个 List 模板进行重新组织，使其更快、更小，并且具有更多功能。

第 9 章中讨论了如何把一段可工作代码加入到重用库中：重用的困难程度要远远超过其字面意思。我们在本章中也展示了如何使用前几章中介绍的技巧来提高代码的健壮性、使其更容易使用以及运行得更快。

C++中新的语言特性——异常——可以使我们在编写程序时不再需要经常去进行出错情况的检测，但它也很容易被人们误用。在第 10 章中，我们讲述了如何使用异常以及不适合使用它的情况。工业界目前也正在学习如何使用异常，这也使这一章显得稍微有点投机。

将一个项目从 C 平台移植到 C++平台所牵涉到的不止是更改一个编译器那么简单：我们必须在项目开发中习惯一种新的开发软件的方式。我们将会在第 11 章中简单讲述一些有关采用 C++开发的项目中经常碰到的技术及人们自身上的问题。

问题

除本章外，本书中的其他章节都以一个“小结”加一个“问题”作为结束。在“小结”中，我们将会以列表的方式给出每一章中的主要概念。那些问题是用来促使读者去思考以及讨论有关技术和人们自身方面的问题，以使得他们可以更容易地理解和记忆每一章中给出的内容。问题中的一部分具有“标准”答案，但更多的则没有，我们衷心地希望读者就此进行的讨论会对他们产生帮助。

语言规则

在本书编写过程中，C++语言的当前“官方”标准文档是 Ellis 和 Stroustrup 所编写的《Annotated C++ Reference Manual》（也经常简称为 ARM）[1]。ARM 是 ISO/ANSI C++标准委员会（一个由来自工业界以及学术界的自愿者所组成的团体，他们正在制订一个会被 ANSI 和 ISO 同时采纳的 C++语言标准）用以制订标准文档的基础。除非 ISO/ANSI 委员会明确提出了某些不同之处，否则在本书中所提到的“语言规则”都是基于 ARM 之上的。实际上，（ISO/ANSI 委员会与 ARM 之间的）大部分的区别都微不足道。我们也鼓励所有的 C++用户都能够遵循标准化结果。

[1] 译注：目前 C++的标准文档是 1998 年 7 月所发布的 ISO 14882 文档（也称 C++98）。

目 录

第1章
抽象

数据抽象（data abstraction）是面向对象设计的一个重要概念。数据抽象要优先于面向对象的设计；然而，随着C++这样直接支持数据抽象的语言变得流行起来，它的应用范围也变得越来越广泛。

抽象数据类型（abstract data type，也称为ADT）是一种由用户定义、拥有明显不同的两部分的类型：

一个公用的接口（public interface），用于指定用户使用该类型的方式，而一个私用的实现（private implementation），在类型内部使用，以提供公用接口所给定的功能。

在C++中，private和public关键字用于指定类声明中哪部分是实现，哪部分又是接口。通过这种方法，编译器就可以确保类的使用者不会绕过类的接口而直接访问其私用成员。然而，另一个事实就是，类有私用成员也不表示它的设计就很好。

对于private和public来说，它们本身还存在着一个远比低层次语言规则更重要的思想。这个思想还从未被C++编译器显式地检测过，但它对于设计模块化、易维护的程序来说却是至关重要的。它表述的是一种概念，那就是——每个编程问题的解决方案都可以被划分为两部分：一个抽象模型（abstract model），它是在程序员和程序用户之间都能取得一致意见的、用于描述该问题以及其解决方案的智力模型；对于该模型的实现（implementation），就是程序员使用特定的方法使计算机可以表述出这种抽象。在本章中，我们将会通过对一个类的设计、细化以及记录过程，来逐步体现出抽象的过程。如我们将看到的那样，“简单地书写一页手册，然后让代码来适合它”这种方式在程序设计中是行不通的。

下面是一些理由，用于说明为什么一个经过详细考虑并记录下来的抽象模型对于程序设计来说很重要：

它有助于帮助其他人来理解如何使用你所设计的类。如果你正试图使用一个链表类，你的主要的（也是最初的）考虑应该不是包含这个类的头文件的名字，也不是该类中成员函数的名字以及其参数类型等。相对来说，你会更注重那些有关该类抽象模型的基础问题，如：它提供了什么操作？能不能在链表上进行回退？能不能在常数时间内访问到链表的头和尾？同一对象

能不能在一个链表中出现多次？链表中包含的是对象本身呢，还是对对象的引用？如果链表包含的是对象的引用，那么由谁来负责创建及销毁这些对象？当链表中的某个对象被销毁时，链表应该有什么样的调整？等等。

对这些问题的回答将会直接影响到那些使用该链表类的应用程序的设计。（例如，如果你的应用程序需要能够在链表上双向移动，那么一个单向链表类在此处就不再适合了。）

如果你不能理解并记录下你的类所支持的抽象模型，那么用户就可能会选择不使用你所提供的类（更糟的是，用户决定使用你的类，然后却发现他们做了一个错误的选择）。

抽象模型是你与你的用户之间的一个协议。由于抽象模型对用户的程序设计有着很重要的影响，你将发现对它进行不向上兼容的改动会有多困难，有时那样的改动甚至就是不可能做到的。例如，一个去除掉链表中后退功能的决议，对公开使用的 List 类来说将会是一场灾难。为什么这样说呢？因为某些 List 的用户可能会将其应用程序的设计完全围绕着它能够支持后退这种抽象来进行。这种在抽象模型中的不兼容改动肯定会使某些已有的用户陷入困境。如果一个类已经被广泛使用，那么它的每一个特性都将至少被一个用户所使用，如果你轻率地决定去除掉某个特性，必将在特定的用户群中引起骚乱。

这就意味着，在你决定将你的库发布给客户时，对库的抽象模型进行细化和完善尤为重要。在实现中产生的错误通常都可以很容易地在下一个版本中得到更正；但在抽象模型中产生的问题（除去冗余错误之外）则将持续存在于类的整个生命周期中。

在记录抽象模型的过程中，我们常常可以发现设计中的重要缺陷。在软件项目设计的早期，我们通常会过高地估计我们对于问题以及我们所提出的解决方案的理解程度。在将早期的模糊想法用精确的言语表示清楚的过程（即抽象模型的记录过程）中，我们就能更关注于那些我们以前从未考虑到的方面。毕竟，在认识不清晰的时候，我们是无法提出合理的解决方案的。

清晰的抽象模型文档有助于其他人重新构造出你的类的新版本（这包括继承或是重新实现两种方式）。要想实现出一个能够与现有代码共同工作的新版本的类，你所需做的不止是在成员函数的名字以及类型签名上做到和最初的代码完全匹配；更重要的是，新版本的类必须还能够符合旧版本的抽象模型的要求。

一旦理解了抽象模型，我们就可以避免“以实现来驱动设计”的情况出现。不管其承认与否，许多的软件设计者在设计一个新类的接口时，他们的脑海中都已经有了一个“明显”的实现方案。这就会使得他们在设计中不自觉地将抽象模型向这种实现方式靠拢。这种做法不但不能提供一个用户易于理解的、并且认为就应该如此的接口，它还会使实现的细节遍布于接口之中，导致以后对实现进行改动变得异常困难。

当然，其他极端的做法也同样会导致麻烦的出现（如：设计出一个完全不顾及实现方案的可能性的接口）。一个接口，不管它有多么优雅，如果它不可能被实现出来，或是实现它需要一

些让人无法接受的性能上的损失，那么对用户来说它就起不到任何帮助作用。在这些考量之间取得一个合理的平衡是类设计中最具有挑战性的部分。最后说明一点，设计抽象模型和设计实现方案应该是两个独立的行为。但尽管如此，这并不意味着我们需要用不同的人手来分开处理这两部分，重要的是，开发人员必须知道自己在某个特定的时间时，自己到底是在负责抽象呢，还是负责实现？

仅仅考虑抽象（而不是实现），我们有几种常用的方法。在抽象的过程中决定“什么应该有，什么不应该有”是面向对象设计者的一个关键技巧。在下一小节中，我们将开始构造出一个简单的抽象模型。

1.1　有关电话号码的抽象模型

在本节中，我们将对用于表示电话号码的类的抽象模型进行最初的探讨。我们期望用这个类来代指各种电话应用程序（如交换机系统、账单系统等）中的电话号码。

我们应该怎么开始我们的第一步呢？一个不错的方法就是：用一句话来描述该对象是用来干什么的（注意，不是“是什么”）。这种描述方式应该尽可能的抽象，尽量不要涉及对象的内部结构。这种“官面总结（executive summary）”应该尽可能的简单，简单到即便是行政人员也可以轻而易举地理解它。

例如，下面就是对于电话号码的一个糟糕的官面总结：

“电话号码包括 3 位地区号码，后面紧接着 3 位交换机号码，最后是 4 位数字。”

这样的描述并没有提及电话号码是用来做什么，或者它和谁交互的问题。相反的是，它简单地把电话号码的结构描述了一遍。这不但没有解释清楚电话号码是做什么的，它还限制了电话号码的范围。例如，通过 modem 所拨打的电话号码可以含有非数字的字符（如&就用于告诉 modem 来等待第二次的拨号音，另外还有#和*这两个特殊的按键）。更严重的就是，它还限制了在美国和加拿大之间能够合法通话的电话号码集。

下面的修改就要稍微好一些：

“一个电话号码指定了世界上的某一部特定电话。”

此处我们并没有丝毫提及电话号码的结构，它只是阐述了另一个用于电话号码的抽象。

当你写下这样的句子后，请花一些时间来仔细琢磨它。请像律师对待一份合法的合同那样仔细检查其中的每个单词。这句话中有什么明显的漏洞吗？它所隐含表达的意思又是什么？

让我们先来注意“指定”这个词。它意味着什么呢？一个电话号码是不是唯一指定一部电话呢？换句话说，能不能有两部以上的电话共用同一个号码呢？事实上，在许多的商业机构中，

大部分电话都是共用一个电话号码的：在有电话进来时，会有一个本地的交换机将它转到一部空闲的电话上面去。同样，对于一个特定的电话来说，电话号码也不是唯一的。当你拨打 555-1234 时。对方的电话就取决于你拨打时所处的区域。

这样，我们就得知了：一个电话号码并不代表唯一一部电话，它的含义取决于使用它的区域。为了反映出这种更新，我们进行了如下的官面总结：

“当与呼叫电话关连起来时，电话号码就是决定被呼电话的关键。”

这样，我们就把电话号码与呼出电话的关系给涵盖进去了，并且也隐式地表达了被呼电话的唯一性。但让我们仔细看看这句话的后半部分。是不是说你拨打电话后就可以得到被呼电话呢？也许如此吧，但更精确的说法是：拨打电话会导致一个到被呼电话的连接的建立。这两种说法之间有什么不同呢？它们之间的不同完全取决于电话用户是否会对连接本身有兴趣。那么，在两台电话间的连接有什么属性会让人们感兴趣呢？当然有了：一个连接有一个开始时间（start time），一个结束时间（end time），以及一定的开销。它们不属于这两台电话，而仅仅是电话连接本身的属性。

拨号必定产生一个被呼电话的结论并不总是正确的：有时线路可能比较繁忙；有时也可能因为某些其他原因会导致呼叫失败……所有这些都不会产生我们所期望的被呼电话。有鉴于此，我们将上面的抽象结论继续细化为：

“当从呼叫电话拨号后，电话号码就是决定可能连接到的被呼电话的关键。”

上面的句子与我们开始时所给出的抽象表述相比，它更贴近实际应用中的抽象模型。然而，要想真正地了解它的含义，我们还必须定义一些它所依赖的其他抽象模型。例如：

“（电话间的）连接表示的是在两部（或多部）电话间的逻辑连接。”

1.1.1 什么不应该出现在抽象模型中？

没有出现在抽象模型中的东西和存在于模型中的一样重要。在我们上面给出的那个电话号码的抽象模型中，我们并没有假设出：一个电话号码有多少位；它是否包含非数字字符；电话与电话号码之间是否是一一对应的以及它们间的连接是如何建立的。

在保持实用性的前提下，我们通过尽可能地将该模型最小化以使其尽可能通用化。我们提高了它应付今后未知变化的能力，以避免频繁地修改我们的抽象模型（但实现细节可能有所改变）。例如：蜂窝电话的出现或者是标准电话系统中新增的按键都不致迫使我们修改我们的抽象模型。

1.1.2 如果存在疑问，暂不考虑

在吃不准某个特殊的概念是否应该包含于抽象模型中时，通常的安全做法是不去考虑它，