PEARSON
Addison Wesley

C++ Common Knowledge

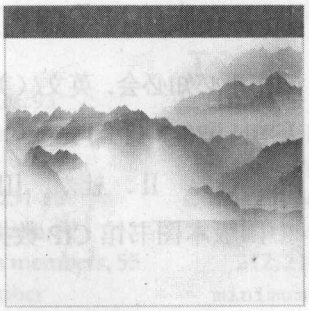
Essential Intermediate programming

[美] Stephen C. Dewhurst 著

C++必知必会

(英文版)

人民邮电出版社
POSTS & TELECOM PRESS



C++必知必会

（英文版）

C++ common knowledge
Essential Intermediate Programing

[美] Stephen C. Dewhurst 著

人民邮电出版社
北 京

图书在版编目（CIP）数据

C++必知必会. 英文/（美）杜赫斯特（Dewhurst, S. C.）著. —北京：人民邮电出版社，2007.7

ISBN 978-7-115-15568-9

Ⅰ. C… Ⅱ. 杜… Ⅲ. C语言—程序设计—英文 Ⅳ. TP312

中国版本图书馆CIP数据核字（2006）第147804号

C++必知必会（英文版）

◆ 著 ［美］Stephen C. Dewhurst

责任编辑 付 飞

◆ 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 电子函件 315@ptpress.com.cn

网址 http://www.ptpress.com.cn

北京顺义振华印刷厂印刷

新华书店总店北京发行所经销

◆ 开本：800×1000 1/16

印张：16.5

字数：329千字 2007年7月第1版

印数：1－3 000册 2007年7月北京第1次印刷

著作权合同登记号 图字：01-2006-6575号

ISBN 978-7-115-15568-9/TP

定价：35.00元

读者服务热线：(010)67132705 印装质量热线：(010)67129223

内容提要

本书描述了C++编程和设计中必须掌握但却不易掌握的主题，这些主题涉及的范围较广，包括指针操作模板、泛型编程、异常处理、内存分配、设计模式等。作者根据本人以及其他有经验的管理人员和培训老师的经验总结，对与这些主题相关的知识进行了精心挑选，最终浓缩成63条。每一条款所包含的内容均为进行产品级C++编程所需的关键知识。作者称这些知识为C++程序员必备的“常识”，其实非意味着简单或平庸，而是“必不可少”。

本书适合于中、高级C++程序员，也适合C或Java程序员转向C++程序设计时参考。

Acknowledgments

Peter Gordon, editor *on ne peut plus extraordinaire*, withstood my kvetching about the state of education in the C++ community for an admirably long time before suggesting that I do something about it. This book is the result. Kim Boedigheimer somehow managed to keep the entire project on track without even once making violent threats to the author.

The expert technical reviewers—Matthew Johnson, Moataz Kamel, Dan Saks, Clovis Tondo, and Matthew Wilson—pointed out several errors and many infelicities of language in the manuscript, helping to make this a better book. A stubborn individual, I haven't followed *all* their recommendations, so any errors or infelicities that remain are entirely my fault.

Some of the material in this book appeared, in slightly different form, in my "Common Knowledge" column for *C/C++ Users Journal*, and much of the material appeared in the "Once, Weakly" Web column on semantics.org. I received many insightful comments on both print and Web articles from Chuck Allison, Attila Fehér, Kevlin Henney, Thorsten Ottosen, Dan Saks, Terje Slettebø, Herb Sutter, and Leor Zolman. Several in-depth discussions with Dan Saks improved my understanding of the difference between template specialization and instantiation and helped me clarify the distinction between overloading and the appearance of overloading under ADL and infix operator lookup.

This book relies on less direct contributions as well. I'm indebted to Brandon Goldfedder for the algorithm analogy to patterns that appears in the item on design patterns and to Clovis Tondo both for motivation and for his assistance in finding qualified reviewers. I've had the good fortune over the years to teach courses based on Scott Meyers's *Effective C++*, *More Effective C++*, and *Effective STL* books. This has allowed me to observe firsthand what background information was commonly missing from students who wanted to profit from these industry-standard, intermediate-level C++ books, and those observations have helped to

shape the set of topics treated in this book. Andrei Alexandrescu's work inspired me to experiment with template metaprogramming rather than do what I was supposed to be doing, and both Herb Sutter's and Jack Reeves's work with exceptions has helped me to understand better how exceptions should be employed.

I'd also like to thank my neighbors and good friends Dick and Judy Ward, who periodically ordered me away from my computer to work the local cranberry harvest. For one whose professional work deals primarily in simplified abstractions of reality, it's intellectually healthful to be shown that the complexity involved in convincing a cranberry vine to bear fruit is a match for anything a C++ programmer may attempt with template partial specialization.

Sarah G. Hewins and David R. Dewhurst provided, as always, both valuable assistance and important impediments to this project.

I like to think of myself as a quiet person of steady habits, given more to calm reflection than strident demand. However, like those who undergo a personality transformation once they're behind the wheel of an automobile, when I get behind a manuscript I become a different person altogether. Addison-Wesley's terrific team of behavior modification professionals saw me through these personality issues. Chanda Leary-Coutu worked with Peter Gordon and Kim Boedigheimer to translate my rantings into rational business proposals and shepherd them through the powers-that-be. Molly Sharp and Julie Nahil not only turned an awkward word document into the graceful pages you see before you, they managed to correct many flaws in the manuscript while allowing me to retain my archaic sentence structure, unusual diction, and idiosyncratic hyphenation. In spite of my constantly changing requests, Richard Evans managed to stick to the schedule and produce not one, but two separate indexes. Chuti Prasertsith designed a gorgeous, cranberry-themed cover. Many thanks to all.

A Note on Typographical Conventions

As mentioned in the preface, these items frequently reference one another. Rather than simply mention the item number, which would force an examination of the table of contents to determine just what was being referenced, the title of the item is italicized and rendered in full. To permit easy reference to the item, the item number and page on which it appears are appended as subscripts. For example, the item referenced *Eat Your Vegetables* [64, 256] tells us that the item entitled "Eat Your Vegetables" is item 64, which can be found on page 256.

Code examples appear in fixed-width font to better distinguish them from the running text. Incorrect or inadvisable code examples appear with a gray background, and correct and proper code appears with no background.

前　言

“一本成功的书不是由书的内容构成的，而是由该书省略的内容所构成的。”

——马克·吐温

……尽可能简单，但不过分简单。

——阿尔伯特·爱因斯坦

……一个怀疑读者能力的作家根本不能称其为作家，只不过是个阴谋家而已。

——E. B. 怀特

Herb Sutter 接手 C++ Report 的编辑工作后很快就邀我为之写一个专栏，主题由我来定。我将该专栏命名为“Common Knowledge（常识）”。用 Herb 的话来说，该专栏预期为“对每一位职业 C++程序员应该知道但未必总是知道的基础知识之定期概述”。然而，在以那样的风格写了一些专栏文章后，我对模板元编程（template metapro gramming）技术的兴趣日渐浓厚，故而此后“Common Knowledge”中讨论的一些主题距离“Common”越来越远。

然而，在 C++程序设计界，当初促使我选定这个专栏的问题仍然存在。在我的培训和咨询工作中，常常会遇到下面几类人员：

- 领域专家，他们是专家级的 C 程序员，但对 C++只有一些基本的认知（并可能对 C++怀有敌意）；
- 直接从大学雇来的新手，他们有才干，但对 C++语言只有理论上的认识，缺乏 C++产品开发经验；
- 专家级的 Java 程序员，他们仅有少量的 C++经验，并有以 Java 编程的方式来从事 C++编程的倾向；
- C++程序员，他们具有若干年维护现有 C++应用程序的经验，但从未经受过学习超出维护所需基础范围的任何知识的挑战。

我希望能立刻进行建设性的工作，但许多我共事过或培训过的人在有能力承担业务之前，都需要接受各种各样的关于C++语言特性、模式以及编程技术的预备教育。更糟的是，我怀疑大多数C++代码都至少忽略了其中一些基本要素，因而不具备大多数C++专家所认可的产品级质量。

这本书致力于解决这个具有普遍性的问题，它提供了每一位职业C++程序员需要知道的常识，并将这些常识精简至本质以便读者高效而精确地吸收。其中有不少信息可以从其他途径获得，而有些知识则是所有专家级C++程序员都知道的未成文信息的完整摘要。本书的优势在于将这些材料集中于一处，并依据我多年的培训和咨询经验进行了遴选。经验表明，这些语言特性、概念和技术都是最常被误解同时也是最有用的。

也许构成本书的63个简短条款最重要的地方在于它们省略掉的东西，而不是它们所包含的东西。许多主题都可以进行更复杂的讨论。作者如果忽略这些复杂性而只作一般性的描述可能会误导读者，但对一个主题的全部复杂性进行专家级的讨论又可能会使读者无所适从。本书采取的方式是在讨论每一个主题时过滤掉那些“不必要”的复杂性。我希望留下的是产品级C++编程所必需的知识的纯粹、完整的精华。较真儿的C++语言专家会意识到我省略了对一些有趣甚至重要的问题（从理论的角度来说）的讨论，但我省略的那些东西通常并不会影响阅读和编写产品级C++代码的能力。

写作这本书的另一个动机来自于我在一次会议上同一群知名C++专家的谈话。这些专家对于一件事情颇感沮丧，那就是他们认为现代C++是如此复杂，以至于“普通”程序员已经不能理解它了（比如，在模板和名字空间上下文中的名字绑定问题。是的，解决这样的问题确实需要普通C++程序员下更多的功夫）。在我看来，应该说其实我们的态度有些过于自负了，我们的沮丧也是不合情理的。我们这些“专家”们自己就不存在这样的问题，实际上，使用C++编程就像说一门（远比C++复杂的）自然语言那样容易，尽管我们不能完全分析我们所说的每一句话的语法结构。本书不断出现的一个主题是，虽然对特定语言特性细节的完整描述可能让人望而生畏，但日常使用的语言特性都是直观而自然的。

不妨考虑一下函数重载。有关它的完整描述占据了很大一块标准文档，并占据了许多C++教程的一整章（甚至多章）。然而，当我们面对如下代码时：

```
void f( int );
void f( const char * );
//...
f( "Hello" );
```

一个职业C++程序员是不可能不知道哪一个f被调用的。有关重载函数调用解析规则的完整知识当然是有意义的，但很少用到。这个道理也适用于其他许多看上去很复杂的C++语言特性和惯用法。

这并不是说本书中出现的所有材料都很简单，它们“尽可能简单，但不过分简单”。在C++编程以及任何其他值得从事的智力活动中，许多重要的细节都无法写在“索引卡片”

上。此外，这并不是一本“傻瓜”书。我觉得自己对那些挤出宝贵时间来阅读我的书的读者负有极大的责任。我尊重这些读者，并且努力与他们交流，就像我亲自与同事交流一样。我认为给职业人员看八级难度的东西算不上写作，不过是想迎合少数人的胃口而已。

本书中的许多条款描述的都是一些简单的误解，这些误解都是我曾反复见到的，只要指出即可（例如成员函数查找的作用域顺序、重写（override）和重载（overload）之间的区别等）。另外一些条款则论述那些正在成为职业 C++程序员所必需的但常常又被错误地认为过于困难而避免使用的知识（例如类模板局部特化（template partial specialization）和模板的模板参数（template template parameter）等）。为此我受到了一些专家级审稿人的批评，说我在模板问题上花费的篇幅过多（约占全书的 1/3），而这些知识并非真的是“常识”。然而，其中每一位专家又都指出有一两个甚至好几个模板主题应该包含于本书之中。有趣的是，这些建议中几乎没有重叠，每一个和模板有关的条款都至少有一个支持者。

这就是构成本书所含条款的问题的症结。我并不认为有哪一位读者对本书每一个条款所谈的主题都一无所知，很可能有的读者会熟悉本书中的所有条款。显然，如果某一位读者对某个特定的主题不熟悉，我认为阅读本书应该会从中受益。即使某一位读者已经熟悉某个主题，我还是希望他能从一个全新的角度来阅读它，这样也许能够澄清一些轻微的误解或进一步加深对该主题的理解。这本书可能还有一个作用，那就是可以节省更有经验的 C++程序员的宝贵时间。那些能干的 C++程序员常常发现他们一再被问及同样的问题，从而影响了他们自身的工作。希望“先读一读《C++必知必会》，再来和我讨论这个问题”的回答方式能够为这些 C++专家节省大量的时间，使他们得以将自己的专家经验用在更复杂的问题上，而这些问题才是需要专家来解决的。

开始我试图将这 63 个条款分组放到各章中，那样显得更加整洁，但这些条款自己却并不这么认为，它们倾向于彼此簇拥在一起。就这一点而言，有些很明显，有些则有点出乎意料。举个例子，与异常和资源管理有关的条款形成了相当自然的一个组；而不那么明显的是，“能力查询”、“指针比较的含义”、“虚构造函数与 Prototype 模式”、“Factory Method 模式”以及“协变返回类型”这几个条款之间的关系紧密得有点出乎意料，因此最好安排在一起。“指针算术”和“智能指针”放在一块儿，而不是和出现于本书较早部分的指针和数组方面的资料放在一起。与其将章式结构强加于这些自然的分组上，不如让各个条款自由结合。当然，这些条款涉及的主题之间存在许多其他相互关系，这是简单的线形顺序难以表现出来的，因此条款中还频繁出现内部交叉引用。所以说，它们是一个聚集但紧密连接的共同体。

尽管本书写作的主要指导思想是短小精悍，但对一个主题的讨论有时会包括一些辅助性的细节，尽管它们和眼前讨论的主题并不直接相关。对于该主题的讨论来说，这些细节并不是必需的，但读者由此可注意到特定程序或技术的存在。例如，在好几个条款中都出现的 Heap 模板例子可以让读者顺便了解到有用但很少被讨论到的 STL 堆算法，而对 placement new 的讨论则勾画出许多标准库组件所用到的复杂缓冲区管理技术基础。只要这

么做看起来很自然，我就会利用机会将辅助性话题的讨论混入某个特定的具名条款中。因此，条款“RAII”包含了对构造函数和析构函数激活顺序的简短讨论，条款“模板实参推导”讨论了用于特化类模板的辅助函数的使用，条款“赋值和初始化并不相同”则混入了对计算性的构造函数的讨论。这本书的条款数目很容易翻倍，但是，就像这些聚集的条款自身一样，辅助性话题和具体条款的相关性使该主题被放置于合适的上下文之中，并且有助于读者高效、精确地吸收这些知识。

我很不情愿地加入了几个很难用本书这样条款简洁的风格写作的主题。特别是有关设计模式和标准模板库设计的主题，看上去短得可笑，而且并不完整。它们的出现只是为了消除一些常见的误解，并强调这些主题的重要性，从而鼓励读者去学习有关该主题更多的东西。

就像团聚在一起度假的家庭成员交换各自的趣事一样，提供例子早已成为我们编程文化的一部分，因此 Shape、String、Stack 以及任何其他常见的“嫌犯”都一一露面。对这些基准例子达成共识，可以使我们的交流像使用设计模式那样高效。例如“设想我希望 rotate（旋转）一个 Shape，除了……之外”，或者“当连接两个 String 时……”这些常见的例子更适合于交流，可以避免费时的背景介绍，比如“你知道当你的兄弟被逮捕时的表现吗？呃，前些天……”

和我以前写的书不同，本书试图避免传递对一些糟糕的编程实践以及对 C++语言特性误用的评判——那是其他一些书的目标，其中最好的一些已被我列于“参考书目”之中（不过，我并没有完全成功地避免说教的倾向，本书中还是提到了一些糟糕的编程实践，虽然只是顺带一提）。一句话，本书的目的在于以尽可能高效的方式告诉读者产品级 C++编程技术的本质。

Stephen C. Dewhurst
马萨诸塞州　卡沃尔

Contents

Item 1 Data Abstraction

A "type" is a set of operations, and an "abstract data type" is a set of operations with an implementation. When we identify objects in a problem domain, the first question we should ask about them is, "What can I do with this object?" not "How is this object implemented?" Therefore, if a natural description of a problem involves employees, contracts, and payroll records, then the programming language used to solve the problem should contain `Employee`, `Contract`, and `PayrollRecord` types. This allows an efficient, two-way translation between the problem domain and the solution domain, and software written this way has less "translation noise" and is simpler and more correct.

In a general-purpose programming language like C++, we don't have application-specific types like `Employee`. Instead, we have something better: the language facilities to create sophisticated abstract data types. The purpose of an abstract data type is, essentially, to extend the programming language into a particular problem domain.

No universally accepted procedure exists for designing abstract data types in C++. This aspect of programming still has its share of inspiration and artistry, but most successful approaches follow a set of similar steps.

1. Choose a descriptive name for the type. If you have trouble choosing a name for the type, you don't know enough about what you want to implement. Go think some more. An abstract data type should represent a single, well-defined concept, and the name for that concept should be obvious.
2. List the operations that the type can perform. An abstract data type is defined by what you can do with it. Remember initialization (constructors), cleanup (destructor), copying (copy operations), and conversions (nonexplicit single-argument constructors and conversion operators). Never, ever, simply provide a bunch of get/set operations on the data members of the implementation. That's not data abstraction; that's laziness and lack of imagination.
3. Design an interface for the type. The type should be, as Scott Meyers tells us, "easy to use correctly and hard to use incorrectly." An

abstract data type extends the language; do proper language design. Put yourself in the place of the user of your type, and write some code with your interface. Proper interface design is as much a question of psychology and empathy as technical prowess.

4. Implement the type. Don't let the implementation affect the interface of the type. Implement the contract promised by the type's interface. Remember that the implementations of most abstract data types will change much more frequently than their interfaces.

Item 2 Polymorphism

The topic of polymorphism is given mystical status in some programming texts and is ignored in others, but it's a simple, useful concept that the C++ language supports. According to the standard, a "polymorphic type" is a class type that has a virtual function. From the design perspective, a "polymorphic object" is an object with more than one type, and a "polymorphic base class" is a base class that is designed for use by polymorphic objects.

Consider a type of financial option, `AmOption`, as shown in Figure 1.

An `AmOption` object has four types: It is simultaneously an `AmOption`, an `Option`, a `Deal`, and a `Priceable`. Because a type is a set of operations (see *Data Abstraction* [1, 1] and *Capability Queries* [27, 93]), an `AmOption` object can be manipulated through any one of its four interfaces. This means that an `AmOption` object can be manipulated by code that is written to the `Deal`, `Priceable`, and `Option` interfaces, thereby allowing the implementation of `AmOption` to leverage and reuse all that code. For a polymorphic type such as `AmOption`, the most important things inherited from its base classes are their interfaces, not their implementations. In

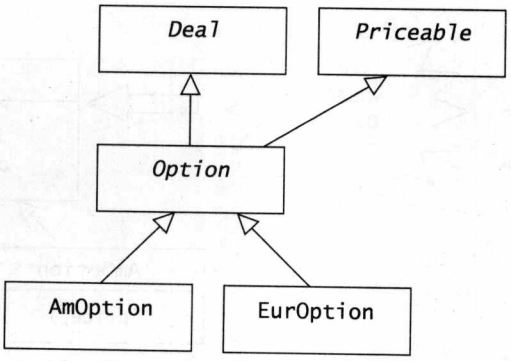


Figure 1 Polymorphic leveraging in a financial option hierarchy. An American option has four types.