原版风暴 · 深入 C++ 系列

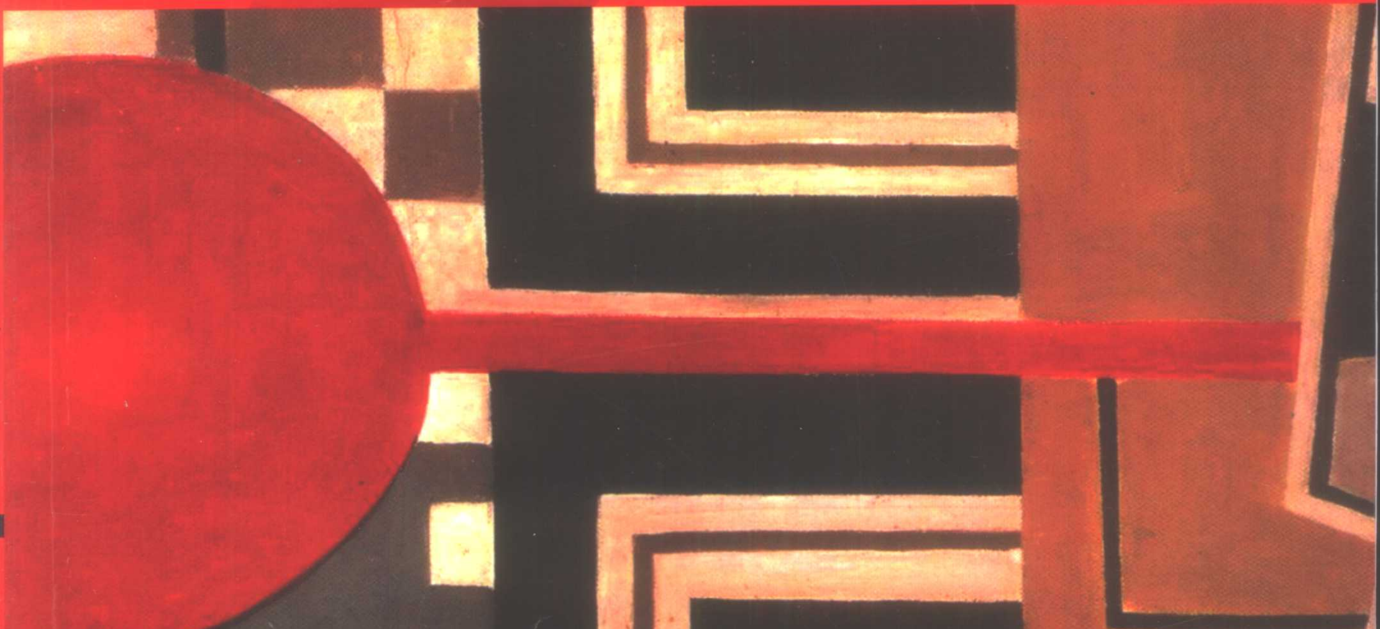
PEARSON Addison Wesley

Modern C++ Design Generic Programming and Design Patterns Applied

C++ 设计新思维

（影印版）

[美] Andrei Alexandrescu 著



Pearson Education

中国电力出版社
www.infopower.com.cn

原版风暴 • 深入 C++ 系列

Modern C++ Design Generic Programming and Design Patterns Applied

C++ 设计新思维

（影印版）

[美] Andrei Alexandrescu　著

中国电力出版社

Modern C++ design : generic programming and design patterns applied (ISBN 0-201-70431-5)
Andrei Alexandrescu
Copyright © 2001 Addison Wesley .
Original English Language Edition Published by Addison Wesley .
All rights reserved.
Reprinting edition published by PEARSON EDUCATION NORTH ASIA LTD and CHINA ELECTRIC POWER PRESS, Copyright © 2003.

本书影印版由 Pearson Education 授权中国电力出版社在中国境内（香港、澳门特别行政区和台湾地区除外）独家出版、发行。
未经出版者书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有 Pearson Education 防伪标签，无标签者不得销售。
北京市版权局著作合同登记号：图字：01-2003-2437
For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macao SAR).
仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区和中国台湾地区）销售发行。

图书在版编目（CIP）数据
C++设计新思维 / （美）亚力山德雷斯库著. —影印本. —北京：中国电力出版社，2003
（原版风暴 · 深入 C++系列）
ISBN 7-5083-1496-4
Ⅰ.C... Ⅱ.亚... Ⅲ.C 语言—程序设计—英文 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 023079 号

责任编辑：闫宏
丛 书 名：原版风暴 · 深入C++系列
书　　名：C++设计新思维（影印版）
编　　著：（美）Andrei Alexandrescu
出 版 者：中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）88515918　传真：（010）88423191
印　　刷：北京地矿印刷厂
发 行 者：新华书店总店北京发行所
开　　本：787×1092 1/16　　**印　　张**：21.75
书　　号：ISBN 7-5083-1496-4
版　　次：2003年6月北京第一版
印　　次：2003年6月第一次印刷
定　　价：39.00 元

Foreword

by Scott Meyers

In 1991, I wrote the first edition of *Effective* C++. The book contained almost no discussions of templates, because templates were such a recent addition to the language, I knew almost nothing about them. What little template code I included, I had verified by e-mailing it to other people, because none of the compilers to which I had access offered support for templates.

In 1995, I wrote *More Effective* C++. Again I wrote almost nothing about templates. What stopped me this time was neither a lack of knowledge of templates (my initial outline for the book included an entire chapter on the topic) nor shortcomings on the part of my compilers. Instead, it was a suspicion that the C++ community's understanding of templates was about to undergo such dramatic change, anything I had to say about them would soon be considered trite, superficial, or just plain wrong.

There were two reasons for that suspicion. The first was a column by John Barton and Lee Nackman in the January 1995 C++ *Report* that described how templates could be used to perform typesafe dimensional analysis with zero runtime cost. This was a problem I'd spent some time on myself, and I knew that many had searched for a solution, but none had succeeded. Barton and Nackman's revolutionary approach made me realize that templates were good for a lot more than just creating containers of `T`.

As an example of their design, consider this code for multiplying two physical quantities of arbitrary dimensional type:

```
template<int m1, int l1, int t1, int m2, int l2, int t2>
Physical<m1+m2, l1+l2, t1+t2> operator*(Physical<m1, l1, t1> lhs,
                                         Physical<m2, l2, t2> rhs)
{
    return Physical<m1+m2, l1+l2, t1+t2>::unit*lhs.value()*rhs.value();
}
```

Even without the context of the column to clarify this code, it's clear that this function template takes six parameters, none of which represents a type! This use of templates was such a revelation to me, I was positively giddy.

Shortly thereafter, I started reading about the STL. Alexander Stepanov's elegant library design, where containers know nothing about algorithms; algorithms know nothing about containers; iterators act like pointers (but may be objects instead); containers and algorithms accept function pointers and function objects with equal aplomb; and library clients may extend the library without having to inherit from any base classes or redefine any virtual functions, made me feel—as I had when I read Barton and Nackman's work—like I knew almost *nothing* about templates.

So I wrote almost nothing about them in *More Effective* C++. How could I? My understanding of templates was still at the containers-of-T stage, while Barton, Nackman, Stepanov, and others were demonstrating that such uses barely scratched the surface of what templates could do.

In 1998, Andrei Alexandrescu and I began an e-mail correspondence, and it was not long before I recognized that I was again about to modify my thinking about templates. Where Barton, Nackman, and Stepanov had stunned me with what templates could *do*, however, Andrei's work initially made more of an impression on me for *how* it did what it did.

One of the simplest things he helped popularize continues to be the example I use when introducing people to his work. It's the `CTAssert` template, analogous in use to the `assert` macro, but applied to conditions that can be evaluated during compilation. Here it is:

```
template<bool> struct CTAssert;
template<> struct CTAssert<true> {};
```

That's it. Notice how the general template, `CTAssert`, is never defined. Notice how there is a specialization for `true`, but not for `false`. In this design, what's *missing* is at least as important as what's present. It makes you look at template code in a new way, because large portions of the "source code" are deliberately omitted. That's a very different way of thinking from the one most of us are used to. (In this book, Andrei discusses the more sophisticated `CompileTimeChecker` template instead of `CTAssert`.)

Eventually, Andrei turned his attention to the development of template-based implementations of popular language idioms and design patterns, especially the GoF* patterns. This led to a brief skirmish with the Patterns community, because one of their fundamental tenets is that patterns cannot be represented in code. Once it became clear that Andrei was automating the generation of pattern *implementations* rather than trying to encode patterns themselves, that objection was removed, and I was pleased to see Andrei and one of the GoF (John Vlissides) collaborate on two columns in the C++ *Report* focusing on Andrei's work.

In the course of developing the templates to generate idiom and pattern implementations, Andrei was forced to confront the variety of design decisions that all implementers face. Should the code be thread safe? Should auxiliary memory come from the heap, from the stack, or from a static pool? Should smart pointers be checked for nullness prior to dereferencing? What should happen during program shutdown if one Singleton's destructor tries to use another Singleton that's already been destroyed? Andrei's goal was to offer his clients all possible design choices while mandating none.

*"GoF" stands for "Gang of Four" and refers to Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, authors of the definitive book on patterns, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

此为试读,需要完整PDF请访问: www.ertongbook.com

His solution was to encapsulate such decisions in the form of *policy classes*, to allow clients to pass policy classes as template parameters, and to provide reasonable default values for such classes so that most clients could ignore them. The results can be astonishing. For example, the Smart Pointer template in this book takes only 4 policy parameters, but it can generate over 300 different smart pointer types, each with unique behavioral characteristics! Programmers who are content with the default smart pointer behavior, however, can ignore the policy parameters, specify only the type of object pointed to by the smart pointer, and reap the benefits of a finely crafted smart pointer class with virtually no effort.

In the end, this book tells three different technical stories, each compelling in its own way. First, it offers new insights into the power and flexibility of C++ templates. (If the material on typelists doesn't knock your socks off, it's got to be because you're already barefoot.) Second, it identifies orthogonal dimensions along which idiom and pattern implementations may differ. This is critical information for template designers and pattern implementers, but you're unlikely to find this kind of analysis in most idiom or pattern descriptions. Finally, the source code to Loki (the template library described in this book) is available for free download, so you can study Andrei's implementation of the templates corresponding to the idioms and patterns he discusses. Aside from providing a nice stress test for your compilers' support for templates, this source code serves as an invaluable starting point for templates of your own design. Of course, it's also perfectly respectable (and completely legal) to use Andrei's code right out of the box. I know he'd want you to take advantage of his efforts.

From what I can tell, the template landscape is changing almost as quickly now as it was in 1995 when I decided to avoid writing about it. At the rate things continue to develop, I may *never* write about templates. Fortunately for all of us, some people are braver than I am. Andrei is one such pioneer. I think you'll get a lot out of his book. I did.

Scott Meyers
September 2000

Foreword

by John Vlissides

What's left to say about C++ that hasn't already been said? Plenty, it turns out. This book documents a convergence of programming techniques—generic programming, template metaprogramming, object-oriented programming, and design patterns—that are well understood in isolation but whose synergies are only beginning to be appreciated. These synergies have opened up whole new vistas for C++, not just for programming but for software design itself, with profound implications for software analysis and architecture as well.

Andrei's generic components raise the level of abstraction high enough to make C++ begin to look and feel like a design specification language. Unlike dedicated design languages, however, you retain the full expressiveness and familiarity of C++. Andrei shows you how to program in terms of design concepts: singletons, visitors, proxies, abstract factories, and more. You can even vary implementation trade-offs through template parameters, with positively no runtime overhead. And you don't have to blow big bucks on new development tools or learn reams of methodological mumbo jumbo. All you need is a trusty, late-model C++ compiler—and this book.

Code generators have held comparable promise for years, but my own research and practical experience have convinced me that, in the end, code generation doesn't compare. You have the round-trip problem, the not-enough-code-worth-generating problem, the inflexible-generator problem, the inscrutable-generated-code problem, and of course the I-can't-integrate-the-bloody-generated-code-with-my-own-code problem. Any one of these problems may be a showstopper; together, they make code generation an unlikely solution for most programming challenges.

Wouldn't it be great if we could realize the theoretical benefits of code generation—quicker, easier development, reduced redundancy, fewer bugs—without the drawbacks? That's what Andrei's approach promises. Generic components implement good designs in easy-to-use, mixable-and-matchable templates. They do pretty much what code generators do: produce boilerplate code for compiler consumption. The difference is that they do it within C++, not apart from it. The result is seamless integration with application code.

You can also use the full power of the language to extend, override, and otherwise tweak the designs to suit your needs.

Some of the techniques herein are admittedly tricky to grasp, especially the template metaprogramming in Chapter 3. Once you've mastered that, however, you'll have a solid foundation for the edifice of generic componentry, which almost builds itself in the ensuing chapters. In fact, I would argue that the metaprogramming material of Chapter 3 alone is worth the book's price—and there are ten other chapters full of insights to profit from. "Ten" represents an order of magnitude. Even so, the return on your investment will be far greater.

John Vlissides
IBM T.J. Watson Research
September 2000

Preface

You might be holding this book in a bookstore, asking yourself whether you should buy it. Or maybe you are in your employer's library, wondering whether you should invest time in reading it. I know you don't have time, so I'll cut to the chase. If you have ever asked yourself how to write higher-level programs in C++, how to cope with the avalanche of irrelevant details that plague even the cleanest design, or how to build reusable components that you don't have to hack into each time you take them to your next application, then this book is for you.

Imagine the following scenario. You come from a design meeting with a couple of printed diagrams, scribbled with your annotations. Okay, the event type passed between these objects is not `char` anymore; it's `int`. You change one line of code. The smart pointers to `Widget` are too slow; they should go unchecked. You change one line of code. The object factory needs to support the new `Gadget` class just added by another department. You change one line of code.

You have changed the design. Compile. Link. Done.

Well, there is something wrong with this scenario, isn't there? A much more likely scenario is this: You come from the meeting in a hurry because you have a pile of work to do. You fire a global search. You perform surgery on code. You add code. You introduce bugs. You remove the bugs . . . that's the way a programmer's job is, right? Although this book cannot possibly promise you the first scenario, it is nonetheless a resolute step in that direction. It tries to present C++ as a newly discovered language for software architects.

Traditionally, code is the most detailed and intricate aspect of a software system. Historically, in spite of various levels of language support for design methodologies (such as object orientation), a significant gap has persisted between the blueprints of a program and its code because the code must take care of the ultimate details of the implementation and of many ancillary tasks. The intent of the design is, more often than not, dissolved in a sea of quirks.

This book presents a collection of reusable design artifacts, called *generic components,* together with the techniques that make them possible. These generic components bring their users the well-known benefits of libraries, but in the broader space of system architecture. The coding techniques and the implementations provided focus on tasks and issues that traditionally fall in the area of design, activities usually done *before* coding. Because of their high level, generic components make it possible to map intricate architectures to code in unusually expressive, terse, and easy-to-maintain ways.

Three elements are reunited here: design patterns, generic programming, and C++. These elements are combined to achieve a very high rate of reuse, both horizontally and vertically. On the horizontal dimension, a small amount of library code implements a combinatorial—and essentially open-ended—number of structures and behaviors. On the vertical dimension, the generality of these components makes them applicable to a vast range of programs.

This book owes much to design patterns, powerful solutions to ever-recurring problems in object-oriented development. Design patterns are distilled pieces of good design—recipes for sound, reusable solutions to problems that can be encountered in many contexts. Design patterns concentrate on providing a suggestive lexicon for designs to be conveyed. They describe the problem, a time-proven solution with its variants, and the consequences of choosing each variant of that solution. Design patterns go above and beyond anything a programming language, no matter how advanced, could possibly express. By following and combining certain design patterns, the components presented in this book tend to address a large category of concrete problems.

Generic programming is a paradigm that focuses on abstracting types to a narrow collection of functional requirements and on implementing algorithms in terms of these requirements. Because algorithms define a strict and narrow interface to the types they operate on, the same algorithm can be used against a wide collection of types. The implementations in this book use generic programming techniques to achieve a minimal commitment to specificity, extraordinary terseness, and efficiency that rivals carefully handcrafted code.

C++ is the only implementation tool used in this book. You will not find in this book code that implements nifty windowing systems, complex networking libraries, or clever logging mechanisms. Instead, you will find the fundamental components that make it easy to implement all of the above, and much more. C++ has the breadth necessary to make this possible. Its underlying C memory model ensures raw performance, its support for polymorphism enables object-oriented techniques, and its templates unleash an incredible code generation machine. Templates pervade all the code in the book because they allow close cooperation between the user and the library. The user of the library literally controls the way code is generated, in ways constrained by the library. The role of a generic component library is to allow user-specified types and behaviors to be combined with generic components in a sound design. Because of the static nature of the techniques used, errors in mixing and matching the appropriate pieces are usually caught during compile time.

This book's manifest intent is to create generic components—preimplemented pieces of design whose main characteristics are flexibility, versatility, and ease of use. Generic components do not form a framework. In fact, their approach is complementary—whereas a framework defines interdependent classes to foster a specific object model, generic components are lightweight design artifacts that are independent of each other, yet can be mixed and matched freely. They can be of great help in *implementing* frameworks.

Audience

The intended audience of this book falls into two main categories. The first category is that of experienced C++ programmers who want to master the most modern library writing techniques. The book presents new, powerful C++ idioms that have surprising capabilities, some of which weren't even thought possible. These idioms are of great help in writing high-level libraries. Intermediate C++ programmers who want to go a step further will certainly find the book useful, too, especially if they invest a bit of perseverance. Although pretty hard-core C++ code is sometimes presented, it is thoroughly explained.

The second category consists of busy programmers who need to get the job done without undergoing a steep learning investment. They can skim the most intricate details of implementation and concentrate on *using* the provided library. Each chapter has an introductory explanation and ends with a Quick Facts section. Programmers will find these features a useful reference in understanding and using the components. The components can be understood in isolation, are very powerful yet safe, and are a joy to use.

You need to have a solid working experience with C++ and, above all, the desire to learn more. A degree of familiarity with templates and the Standard Template Library (STL) is desirable.

Having an acquaintance with design patterns (Gamma et al. 1995) is recommended but not mandatory. The patterns and idioms applied in the book are described in detail. However, this book is not a pattern book—it does not attempt to treat patterns in full generality. Because patterns are presented from the pragmatic standpoint of a library writer, even readers interested mostly in patterns may find the perspective refreshing, if constrained.

Loki

The book describes an actual C++ library called Loki. Loki is the god of wit and mischief in Norse mythology, and the author's hope is that the library's originality and flexibility will remind readers of the playful Norse god. All the elements of the library live in the namespace `Loki`. The namespace is not mentioned in the coding examples because it would have unnecessarily increased indentation and the size of the examples. Loki is freely available; you can download it from http://www.awl.com/cseng/titles/0-201-70431-5.

Except for its threading part, Loki is written exclusively in standard C++. This, alas, means that many current compilers cannot cope with parts of it. I implemented and tested Loki using Metrowerks' CodeWarrior Pro 6.0 and Comeau C++ 4.2.38, both on Windows. It is likely that KAI C++ wouldn't have any problem with the code, either. As vendors release new, better compiler versions, you will be able to exploit everything Loki has to offer.

Loki's code and the code samples presented throughout the book use a popular coding standard originated by Herb Sutter. I'm sure you will pick it up easily. In a nutshell,

- Classes, functions, and enumerated types look `LikeThis`.
- Variables and enumerated values look `likeThis`.
- Member variables look `likeThis_`.
- Template parameters are declared with `class` if they can be only a user-defined type, and with `typename` if they can also be a primitive type.

此为试读，需要完整PDF请访问：www.ertongbook.com

Organization

The book consists of two major parts: techniques and components. Part I (Chapters 1 to 4) describes the C++ techniques that are used in generic programming and in particular in building generic components. A host of C++-specific features and techniques are presented: policy-based design, partial template specialization, typelists, local classes, and more. You may want to read this part sequentially and return to specific sections for reference.

Part II builds on the foundation established in Part I by implementing a number of generic components. These are not toy examples; they are industrial-strength components used in real-world applications. Recurring issues that C++ developers face in their day-to-day activity, such as smart pointers, object factories, and functor objects, are discussed in depth and implemented in a generic way. The text presents implementations that address basic needs and solve fundamental problems. Instead of explaining what a body of code does, the approach of the book is to discuss problems, take design decisions, and implement those decisions gradually.

Chapter 1 presents policies—a C++ idiom that helps in creating flexible designs.
Chapter 2 discusses general C++ techniques related to generic programming.
Chapter 3 implements typelists, which are powerful type manipulation structures.
Chapter 4 introduces an important ancillary tool: a small-object allocator.
Chapter 5 introduces the concept of generalized functors, useful in designs that use the Command design pattern.
Chapter 6 describes Singleton objects.
Chapter 7 discusses and implements smart pointers.
Chapter 8 describes generic object factories.
Chapter 9 treats the Abstract Factory design pattern and provides implementations of it.
Chapter 10 implements several variations of the Visitor design pattern in a generic manner.
Chapter 11 implements several multimethod engines, solutions that foster various trade-offs.

The design themes cover many important situations that C++ programmers have to cope with on a regular basis. I personally consider object factories (Chapter 8) a cornerstone of virtually any quality polymorphic design. Also, smart pointers (Chapter 7) are an important component of many C++ applications, small and large. Generalized functors (Chapter 5) have an incredibly broad range of applications. Once you have generalized functors, many complicated design problems become very simple. The other, more specialized, generic components, such as Visitor (Chapter 10) or multimethods (Chapter 11), have important niche applications and stretch the boundaries of language support.

Acknowledgments

I would like to thank my parents for diligently taking care of the longest, toughest part of them all.

It should be stressed that this book, and much of my professional development, wouldn't have existed without Scott Meyers. Since we met at the C++ World Conference in 1998, Scott has constantly helped me do more and do better. Scott was the first person who enthusiastically encouraged me to develop my early ideas. He introduced me to John Vlissides, catalyzing another fruitful cooperation; lobbied Herb Sutter to accept me as a columnist for C++ *Report;* and introduced me to Addison-Wesley, practically forcing me into starting this book, at a time when I still had trouble understanding New York sales clerks. Ultimately, Scott helped me all the way through the book with reviews and advice, sharing with me all the pains of writing, and none of the benefits.

Many thanks to John Vlissides, who, with his sharp insights, convinced me of the problems with my solutions and suggested much better ones. Chapter 9 exists because John insisted that "things could be done better."

Thanks to P. J. Plauger and Marc Briand for encouraging me to write for the C/C++ *Users Journal*, at a time when I still believed that magazine columnists were inhabitants of another planet.

I am indebted to my editor, Debbie Lafferty, for her continuous support and sagacious advice.

My colleagues at RealNetworks, especially Boris Jerkunica and Jim Knaack, helped me very much by fostering an atmosphere of freedom, emulation, and excellence. I am grateful to them all for that.

I also owe much to all participants in the comp.lang.c++.moderated and comp.std.c++ Usenet newsgroups. These people greatly and generously contributed to my understanding of C++.

I would like to address thanks to the reviewers of early drafts of the manuscript: Mihail Antonescu, Bob Archer (my most thorough reviewer of all), Allen Broadman, Ionut Burete, Mirel Chirita, Steve Clamage, James O. Coplien, Doug Hazen, Kevlin Henney, John Hickin, Howard Hinnant, Sorin Jianu, Zoltan Kormos, James Kuyper, Lisa Lippincott, Jonathan H. Lundquist, Petru Marginean, Patrick McKillen, Florin Mihaila, Sorin Oprea, John Potter, Adrian Rapiteanu, Monica Rapiteanu, Brian Stanton, Adrian Steflea, Herb Sutter, John Torjo, Florin Trofin, and Cristi Vlasceanu. They all have invested significant efforts in reading and providing input, without which this book wouldn't have been half of what it is.

Thanks to Greg Comeau for providing me with his top-notch C++ compiler for free.

Last but not least, I would like to thank all my family and friends for their continuous encouragement and support.

Modern C++ Design

The C++ In-Depth Series

Bjarne Stroustrup, Editor

"I have made this letter longer than usual, because I lack the time to make it short."
—Blaise Pascal

The advent of the ISO/ANSI C++ standard marked the beginning of a new era for C++ programmers. The standard offers many new facilities and opportunities, but how can a real-world programmer find the time to discover the key nuggets of wisdom within this mass of information? **The C++ In-Depth Series** minimizes learning time and confusion by giving programmers concise, focused guides to specific topics.

Each book in this series presents a single topic, at a technical level appropriate to that topic. The Series' practical approach is designed to lift professionals to their next level of programming skills. Written by experts in the field, these short, in-depth monographs can be read and referenced without the distraction of unrelated material. The books are cross-referenced within the Series, and also reference *The C++ Programming Language* by Bjarne Stroustrup.

As you develop your skills in C++, it becomes increasingly important to separate essential information from hype and glitz, and to find the in-depth content you need in order to grow. The C++ In-Depth Series provides the tools, concepts, techniques, and new approaches to C++ that will give you a critical edge.

Titles in the Series

Accelerated C++: Practical Programming by Example, Andrew Koenig and Barbara E. Moo

The Boost Graph Library: User Guide and Reference Manual, Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine

C++ In-Depth Box Set, Bjarne Stroustrup, Andrei Alexandrescu, Andrew Koenig, Barbara E. Moo, Stanley B. Lippman, and Herb Sutter

C++ Network Programming, Volume 1: Mastering Complexity Using ACE and Patterns, Douglas C. Schmidt and Stephen D. Huston

Essential C++, Stanley B. Lippman

Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

Modern C++ Design: Generic Programming and Design Patterns Applied, Andrei Alexandrescu

More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions, Herb Sutter

C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks, Douglas C. Schmidt and Stephen D. Huston

For more information, check out the series Web site at http://www.awprofessional.com/series/indepth/

Contents

此为试读，需要完整PDF请访问：www.ertongbook.com