HZ BOOKS 华章科技

C++标准委员会成员和IBM XL编译器中国开发团队共同撰写，权威性毋庸置疑

系统、深入、详尽地讲解了C++11新标准中的新语言特性、新标准库特性、对原有特性的改进，以及所有这些新特性的应用

华章原创精品

深入理解C++11

C++11新特性解析与应用

Understanding C++11

Analysis and Application of New Features

（加）Michael Wong

IBM XL编译器中国开发团队 著

机械工业出版社
China Machine Press

深入理解 C++11:

C++ 11 新特性解析与应用

Michael Wong　IBM XL 编译器中国开发团队　著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深入理解 C++11 ：C++11 新特性解析与应用 / Michael Wong, IBM XL 编译器中国开发团队著 . —北京 ：机械工业出版社，2013.6
（原创精品系列）

ISBN 978-7-111-42660-8

Ⅰ. 深… Ⅱ. ① M… ② I… Ⅲ. C 语言 – 程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2013）第 110482 号

本书法律顾问 北京市展达律师事务所

国内首本全面深入解读 C++11 新标准的专著，由 C++ 标准委员会代表和 IBM XL 编译器中国开发团队共同撰写。不仅详细阐述了 C++11 标准的设计原则，而且系统地讲解了 C++11 新标准中的所有新语言特性、新标准库特性、对原有特性的改进，以及如何应用所有这些新特性。

全书一共 8 章 ：第 1 章从设计思维和应用范畴两个维度对 C++11 新标准中的所有特性进行了分类，呈现了 C++11 新特性的原貌 ；第 2 章讲解了在保证与 C 语言和旧版 C++ 标准充分兼容的原则下增加的一些新特性 ；第 3 章讲解了具有广泛可用性、能与其他已有的或者新增的特性结合起来使用的、具有普适性的一些新特性；第 4 章讲解了 C++11 新标准对原有一些语言特性的改进，这些特性不仅能让 C++ 变得更强大，还能提升程序员编写代码的效率 ；第 5 章讲解了 C++11 在安全方面所做的改进，主要涵盖枚举类型安全和指针安全两个方面的内容；第 6 章讲解了为了进一步提升和挖掘 C++ 程序性能和让 C++ 能更好地适应各种新硬件的发展而设计的新特性，如多核、多线程、并行编程方面的新特性 ；第 7 章讲解了一些颠覆 C++ 一贯设计思想的新特性，如 lambda 表达式等 ；第 8 章讲解了 C++11 为了解决 C++ 编程中各种典型实际问题而做出的有效改进，如对 Unicode 的深入支持等。附录中则介绍了 C++11 标准与其他相关标准的兼容性和区别、C++11 中弃用的特性、编译器对 C++11 的支持情况，以及学习 C++11 的相关资源。

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：孙海亮
印刷
2013 年 6 月第 1 版第 1 次印刷
186mm × 240 mm • 20.5 印张
标准书号：ISBN 978-7-111-42660-8
定 价：69.00 元

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 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

免责声明

本书言论仅为作者根据自身实践经验所得，仅代表个人观点，不代表 IBM 公司的官方立场和看法，特此声明。

HZ BOOKS
华章图书

IBM XL 编译器中国开发团队简介

IBM 拥有悠久的编译器开发历史（始于 20 世纪 80 年代），在全球拥有将近 400 名高素质工程师组成的研发团队，其中包括许多世界知名的研究学者和技术专家。IBM 一直以来都是编程语言的制定者和倡导者之一，并将长期在编译领域进行研发和投资。IBM XL 编译器中国开发团队于 2010 年在上海成立，现拥有编译器前端开发人员（C/C++）、后端开发人员，测试人员，以及性能分析人员等。团队与 IBM 北美编译器团队紧密合作，共同开发、测试和发布基于 POWER 系统的 AIX 及 Linux 平台下的 XL C/C++ 和 XL Fortran 系列产品，并对其提供技术支持。虽然团队成立时间不长，但已于 2012 年成功发布最新版本的 XL C/C++ for Linux V12.1 & XL Fortran for Linux V14.1，并获得 7 项发明专利。团队成员拥有丰富的编译器开发经验，对编译技术、编程语言、性能优化和并行计算等都有一定的研究，也对 C++11 标准的各种新特性有较早的研究和理解，并正在实际参与 C++11 新特性的开发工作。

欢迎广大读者关注 IBM XL 编译器中国开发团队博客：http://ibm.co/HK0GCx，及新浪微博：www.weibo.com/ibmcompiler，并与我们一起学习、探讨 C++11 和编译技术。

作者个人简介

官孝峰　毕业于上海交通大学软件学院。多年致力于底层软件开发，先后供职于 AMD 上海研发中心，IBM 上海研发中心。在嵌入式系统及应用、二进制翻译、图形驱动等领域有丰富的实践经验。2010 年加入 IBM XL 中国编译器中国开发团队，负责 XL C/C++/Fortran 编译器后端开发工作，专注于编译器后端优化、代码生成，以及语言标准等领域。在 C++11 标准制定后，率先对标准进行了全面深入的研究，并组织团队成员对新语言标准进行学习探讨。是数项国内外专利的发明人，并曾于 DeveloperWorks 社区发表英文论文一篇。

陈晶　毕业于西安交通大学通信与信息工程专业，有多年的编译器文档开发写作经验。2008 年起供职于 IBM 上海研发中心，一直致力于研究 C++11 标准的各项新特性，并负责该部分的技术文档撰写。精通 C/C++ 语言，对编译器领域有浓厚的兴趣。负责工作部门内部的编译器产品技术培训。在 DeveloperWorks 社区发表过多篇论文。拥有一项国内专利。

任剑钢　毕业于复旦大学计算机专业。2010 年加入 IBM XL 中国编译器中国开发团队，先后从事 XL Fortran 编译器前端、XL C/C++/Fortran 编译器后端等各种开发工作。对于技术敏感而热衷，擅长 C/C++/Java 等多种编程语言，现专注于编译器代码优化技术。拥有一项专利，并带领团队在 IBM Connections 平台的技术拓展大赛中赢得大奖。

朱鸿伟　毕业于浙江大学计算机科学与技术专业。资深软件开发工程师，有多年系统底层软件开发和 Linux 环境开发经验。一直致力于 C/C++ 语言、编译器、Linux 内核等相关技术的研究与实践，关注新技术和开源社区，对于 Linux 内核以及 Linux 平台软件的开发有着浓厚的兴趣，曾参与 Linux 开源软件的开发与设计。2010 年加入 IBM XL 中国编译器中国开发团队，现专注于编译器前端技术研究与开发工作。

张青山　毕业于福州大学计算机系。从事嵌入式开发多年，曾致力于 Linux 内核和芯片驱动程序的开发、及上层应用程序的编写。2010 年加入 IBM XL 编译器中国开发团队，负责 XL C++ 编译器前端的研发工作。对 C99、C++98、C++11 等语言标准及编译理论有深入理解，

并实际参与 C++11 前端各种特性的实现。此外还致力于编译器兼容性的研究和开发。

纪金松　中国科学技术大学计算机体系结构专业博士，有多年编译器开发经验。2008 年起先后就职于 Marvell（上海）、腾讯上海研究院、IBM 上海研发中心。一直致力于系统底层软件的研究与实践，在编译器后端优化、二进制工具链、指令集优化、可重构计算等相关领域有丰富的实战经验。在国内外杂志和会议中发表过 10 多篇论文，并拥有多项国内外专利。

郭久福　毕业于华东理工大学控制理论与控制工程专业。拥有近 10 年的系统软件开发经验，曾就职于柯达开发中心、HP 中国以及 IBM 中国软件实验室。对 C 语言标准以及 C++ 语言标准有深入研究，近年来致力于编译器前端的开发和研究。

林科文　毕业于复旦大学计算机软件与理论专业，有多年底层系统开发经验。2010 年起供职于 IBM 上海研发中心，现从事编译器后端开发工作。一直致力于系统软件的研究和实践，以及编译器代码优化等领域。活跃于 DeveloperWorks 社区，是三项国内专利的发明人。

班怀芸　毕业于南京理工大学计算机应用技术专业。资深测试工程师，在测试领域耕耘多年，曾任职于 Alcatel-Lucent 公司，有丰富的项目经验。2011 年加入 IBM XL 编译器中国开发团队，现从事和 C/C++ 语言相关的测试领域研究，负责用编译器实现相关需求分析、自动化测试方案制定及实现等工作。持续关注语言和测试技术的发展，对 C/C++ 新特性和语言标准有较深的理解。

蒋健　毕业于复旦大学计算机科学系，资深编译器技术专家。先后供职于 Intel、Marvell、Microsoft 及 IBM 等各公司编译器开发部门，参与并领导业界知名的编译器后端多种相关研发工作，并且拥有近十项编译器方面的专利。现负责 XL C/C++/Fortran 编译器后端开发，并领导 IBM XL 中国编译器中国开发团队各种技术工作。

宋瑞　毕业于北京大学微电子学院，在软件测试领域工作多年，对于测试架构以及软件的发布流程具有丰富的经验，曾就职于 Synopsys 和 Apache design solution。2011 年加入 IBM XL 编译器中国开发团队，从事和编译器、C/C++ 语言相关的研究与测试，对 C/C++ 新语言特性和标准有较深的理解和研究。

刘志鹏　毕业于南京理工大学计算机应用专业，2010 年加入 IBM XL 中国编译器中国开发团队。先后从事 Fortran、C++ 前端的开发工作。现致力于 C++ 语言新标准、语言兼容性等研究与开发。对编译器优化技术、Linux 内核开发有浓厚兴趣，擅长 C/C++/Java 等多种语言。

毛一赠　毕业于上海交通大学，现任职于 IBM 编译器中国开发团队，从事 XL C/C++ 编译器前端的开发工作，在此领域有多年研究经验。具有丰富的实际项目经验并持续关注语言发展。擅长 C++、C、Java 等语言，对 C++ 模板有深入研究。此外对 C#/VB/perl/Jif/Fabric 等语言也均有所涉猎。热爱编程与新技术，活跃于 DeveloperWorks 等社区，发表过技术博客数篇。

张寅林　毕业于上海交通大学信息安全工程专业。2010 年起加入 IBM XL 编译器中国开发团队，专注于编译器后端性能优化开发工作。对于编译器优化算法，Linux 操作系统体系结构与实现有浓厚的兴趣，擅长 C/C++/Python 编程语言。目前从事 IBM 企业级存储服务器的开发工作。

刘林　东南大学计算机科学与工程学院硕士，有多年底层系统开发经验。2010—2012 年间就职于 IBM XL 编译器中国开发团队，先后从事编译器测试及后端开发工作。对嵌入式、Linux 操作系统体系结构与实现有浓厚的兴趣。

Preface

If you are holding this book in the store, you might be wondering why you should read this book among many C++ books. First, you should know this book is about the latest new C++11 (codenamed C++0x) Standard ratified at the end of 2011. This new Standard is almost like a new language, with many new language and library features but there is a strong emphasis for compatibility with the last C++98/03 Standard during design. At the time of printing of this book in 2013, this is one of the first few C++11 books published. All books that do not mention C++11 will invariably be talking about C++98/03.

What makes this book different is that it is written by Chinese writers, in its original Chinese language. In fact, all of us are on the C++ compiler team for the IBM xlC++ compiler, which has been adding C++11 features since 2008.

For my part, I am the C++ Standard representative for IBM and Canada and have been working in compilers for 20 years, and is the author of several C++11 features while leading the IBM C++ Compiler team.

For C++ users who read Chinese, many prefer to read an original Chinese language book, rather than a translated book, even if they can read other languages. While very well written also by experts from the C++ Standard Committee, these non-Chinese books' translation can take time, or result in words or meanings that are loss in translation. The translator has a tough job as technical books contain many jargon and new words that may not have an exact meaning in Chinese. Different translators may not use the same word, even within the same book. These are reasons that lead to a slow dissemination of C++ knowledge and slows the adoption of C++11 in Chinese.

These are all reasons that lead to weak competitiveness. We aim to improve that competiveness with a book written by Chinese-speaking writers, with a uniform language for jargons, who understand the technology gap as many of the writers work in the IBM Lab in Shanghai. We know there are many Chinese-speaking C++ enthusiasts who are eager to learn and use the updates to their favorite language. The newness of C++11 also demands a strong candidate in the beginner to intermediate level of C++11, which is the level of this book.

You should do well reading this book, if you are：

- ❑an experienced C programmer who wants to see what C++11 can do for you.
- ❑already a C++98/03 programmer who wants to learn the new C++11 language.
- ❑anyone interested in learning the new C++11 language.

We structure this book using the design principles that Professor Bjarne Stroustrup, the father of C++ followed in designing C++11 through the Standard Committee. In fact, we separated this book into chapters based on those design principles. These design principles are outlined in the first chapter. The remaining chapters separate every C++11 language features under those design classifications. For each feature, it will explain the motivation for the feature, the rules, and how it is used, taking from the approved C++11 papers that proposed these features. A further set of appendices will outline the current state of the art of compiler support for C++11, incompatibilities, deprecated features, and links to the approved papers.

After reading this book, you should be able to answer questions such as:

- ❑What is a lambda and the best way to use it?
- ❑How is decltype and auto type inference related?
- ❑What is move semantics and how does it solve the forwarding problem?
- ❑I want to understand default and deleted as well as explicit overrides.
- ❑What did they replace exception specifications with and how does noexcept work?
- ❑What are atomics and the new memory model?
- ❑How do you do parallel programming in C++11?

What we do not cover are the C++11 changes to the Standard library. This part could be a book itself and we may continue with this as Book II. This means we will not talk about the new algorithms, or new class libraries, but we will talk about atomics since most compilers implement atomics as a language feature rather than a library feature for efficiency reason. However, in the C++11 Standard, atomics is listed as a library feature simply because it could be implemented at worst as a library, but very few compilers would do that. This book is also not trying to teach you C++. For that, we particularly recommend Professor Stroustrup's book Programming principles & Practice Using C++ which is based on an excellent course he taught at Texas A&M University on programming.

This book could be read chapter by chapter if you are interested in every feature of C++11. More likely, you would want to learn about certain C++11 feature and want to target that feature. But while reading about that feature, you might explore other features that fall under the same design guideline.

We hope you find this book useful in your professional or personal learning. We learnt too during our journey of collaborating in writing this book, as we wrote while building the IBM C++ compiler and making it C++11 compliant.

The work of writing a book is long but it is well worth it. While I have been thinking about writing this book while working on the C++ Standard, it was really Xiao Feng Guan who motivated me to start really stop thinking and start doing it for real. He continued to motivate and lead others through their writing assignment process and completed the majority of the work of organizing

this book. I also wish to thank many who have been my mentors officially and unofficially. There are too many to mention but people such as Bjarne Stroustrup, Herb Sutter, Hans Boehm, Anthony Williams, Scott Meyers and many others have been my teachers and great examples of leaders since I started reading their books and watching how they work within large groups. IBM has generously provided the platform, the time, and the facility to allow all of us to exceed ourselves, if only just a little to help the next generation of programmers. Thank you above all to my family Sophie, Cameron, Spot the Cat, and Susan for lending my off-time to work on this book.

Michael

序

当你在书店里拿起这本书的时候，可能最想问的就是：这么多 C++ 的书籍，为什么需要选择这一本？回答这个问题首先需要知道的是，这是一本关于在 2011 年年底才制定通过的 C++11（代码 C++0x）的新标准的书籍。这个新标准看起来就像是一门新的语言，不仅有很多的新语言特性、标准库特性，而且在设计时就考虑了高度兼容于旧有的 C++98/03 标准。在 2013 年出版的 C++ 的书籍中，本书是少数几部关于 C++11 的书籍之一，而其他的，则会是仅讲解 C++98/03 而未提及 C++11 的书籍。

相比于其他书籍，本书还有个显著特点——绝大多数章节都是由中国作者编写。事实上，本书所有作者均来自 IBM XL C++ 编译器开发团队。而团队对于 C++11 新特性的开发，早在 2008 年就开始了。

而我则是一位 IBM 和加拿大的 C++ 标准委员会的代表。我在编译器领域已工作了 20 多年。除了是 IBM C++ 编译器开发团队的领导者之外，还是一些 C++11 特性的作者。

对于使用中文的 C++ 用户而言，很多人还是喜欢阅读原生的中文图书，而非翻译版本，即使是在他们具备阅读其他语言能力的时候。虽然 C++ 标准委员会的专家也在编写一些高质量的书籍，但是书籍从翻译到出版通常需要较长时间，而且一些词语或者意义都可能在翻译中丢失。而翻译者通常也会觉得技术书籍的翻译是门苦差，很多行话、术语通常难以找到准确的中文表达方式。这么一来不同的翻译者会使用不同的术语，即使是在同一本书中，有时同一术语也会翻译成不同的中文。这些状况都是 C++ 知识传播的阻碍，会拖慢 C++11 语言被中国程序员接受的进程。

基于以上种种原因，我们决定本书让母语是中文，并且了解国内外技术差距的 IBM 上海实验室的同事编写。我们知道，在中国有非常多的 C++ 狂热爱好者正等着学习关于自己最爱的编程语言的新知识。而新的 C++11 也会招来大量的初级、中级用户，而本书也正好能满足这些人的需求。

所以，如果你属于以下几种状况之一，将会非常适合阅读本书：

- C 语言经验非常丰富且正在期待着看看 C++11 新功能的读者。
- 使用 C++98/03 并期待使用新的 C++11 的程序员。
- 任何对新的 C++11 语言感兴趣的人。

在本书中，我们引述了 C++ 之父 Bjarne Stroustrup 教授关于 C++11 的设计原则。而事实上，本书的章节划分也是基于这些设计原则的，读者在第 1 章可以找到相关信息，而剩余章节则是基于该原则对每个 C++11 语言进行的划分。对于每个特性，本书将根据其相关的论文展开描述，讲解如设计的缘由、语法规则、如何使用等内容。而书后的附录，则包括当前的

C++11 编译器支持状况、不兼容性、废弃的特性，以及论文的链接等内容。

在读完本书后，读者应该能够回答以下问题：

- ❑什么是 lambda，及怎么样使用它是最好的？
- ❑decltype 和 auto 类型推导有什么关系？
- ❑什么是移动语义，以及（右值引用）是如何解决转发问题的？
- ❑default/deleted 函数以及 override 是怎么回事？
- ❑异常描述符被什么替代了？ noexcept 是如何工作的？
- ❑什么是原子类型以及新的内存模型？
- ❑如何在 C++11 中做并行编程？

对于标准程序库，我们在本书中并没有介绍。这部分内容可能会成为我们下一本书的内容。这意味着我们将在下一本书中不仅会描述新的算法、新的类库，还会更多地描述原子类型。虽然出于性能考虑，大多数的编译器都是通过语言特性的方式来实现原子类型的，但在 C++11 标准中，原子类型却被视为一种库特性，因其可以通过库的方式来实现。同样的，这样一本书也不会教读者基础的 C++ 知识，如果读者想了解这方面的内容，我们推荐 Stroustrup 教授的《Programming principles & Practice Using C++》（中文译为：《C++ 程序设计原理与实践》，华章公司已出版）。该书是 Stroustrup 教授以其在德克萨斯 A&M 大学教授的课程为基础编写的。

对 C++11 特性感兴趣的读者可以顺序阅读本书。当然，读者也可以直接阅读自己感兴趣的章节，但是读者阅读时肯定会发现，这些特性基本和其他的特性一样，遵从了相同的设计准则。

我们也希望本书对你的职业或者个人学习起到积极的作用。当然，我们在合作写作本书，以及在为 IBM C++ 编译器开发 C++11 特性时，也颇有收获。

本书的编写经历了较长的时间，但这是值得的。我在 C++ 标准委员会工作的时候，只是在考虑写这样一本书，而官孝峰则让我从这样的考虑转到了动手行动。继而他还激励并领导其他成员共同参与，最终完成了本书。

此外，我要感谢我的一些正式的以及非正式的导师，比如 Bjarne Stroustrup、Herb Sutter、Hans Boehm、Anthony Williams、Scott Meyers，以及许多其他人，通过阅读他们的著作，或观察他们在委员会中的工作，我学会了很多。当然，更要感谢 IBM 为我们提供的平台、时间，以及各种便利，因为有了这些最终我们才能够超越自我，为新一代的程序员做一些事情，即使这样的事情可能微不足道。还要感谢的是我的家人，Sophie、Cameron、Spot（猫）和 Susan，让我能够在空闲时间完成书籍编写。

Michael

前　言

为什么要写这本书

相比其他语言的频繁更新，C++ 语言标准已经有十多年没有真正更新过了。而上一次标准制定，正是面向对象概念开始盛行的时候。较之基于过程的编程语言，基于面向对象、泛型编程等概念的 C++ 无疑是非常先进的，而 C++98 标准的制定以及各种符合标准的编译器的出现，又在客观上推动了编程方法的革命。因此在接下来的很多年中，似乎人人都在学习并使用 C++。商业公司在邀请 C++ 专家为程序员讲课，学校里老师在为学生绘声绘色地讲解面向对象编程，C++ 的书籍市场也是百花齐放，论坛、BBS 的 C++ 板块则充斥了大量各种关于 C++ 的讨论。随之而来的，招聘启事写着“要求熟悉 C++ 编程”，派生与继承成为了面试官审视毕业生基础知识的重点。凡此种种，不一而足。于是 C++ 语言“病毒性”地蔓延到各种编程环境，成为了使用最为广泛的编程语言之一。

十来年的时光转瞬飞逝，各种编程语言也在快马加鞭地向前发展。如今流行的编程语言几乎无一不支持面向对象的概念。即使是古老的语言，也通过了制定新标准，开始支持面向对象编程。随着 Web 开发、移动开发逐渐盛行，一些新流行起来的编程语言，由于在应用的快速开发、调试、部署上有着独特的优势，逐渐成为了这些新领域中的主流。不过这并不意味着 C++ 正在失去其阵地。身为 C 的“后裔”，C++ 继承了 C 能够进行底层操作的特性，因此，使用 C/C++ 编写的程序往往具有更佳的运行时性能。在构建包括操作系统的各种软件层，以及构建一些对性能要求较高的应用程序时，C/C++ 往往是最佳选择。更一般地讲，即使是由其他语言编写的程序，往往也离不开由 C/C++ 编写的编译器、运行库、操作系统，或者虚拟机等提供支持。因此，C++ 已然成为了编程技术中的中流砥柱。如果用个比喻来形容 C++，那么可以说这十来年 C++ 正是由“锋芒毕露”的青年时期走向“成熟稳重”的中年时期。

不过十来年对于编程语言来说也是个很长的时间，长时间的沉寂甚至会让有的人认为，C++ 就是这样一种语言：特性稳定，性能出色，易于学习而难于精通。长时间使用 C++ 的程序员也都熟悉了 C++ 毛孔里每一个特性，甚至是现实上的一些细微的区别，比如各种编译器对 C++ 扩展的区别，也都熟稔于心。于是这个时候，C++11 标准的横空出世，以及 C++ 之父 Bjarne Stroustrup 的一句“看起来像一门新语言”的说法，无疑让很多 C++ 程序员有些诚惶诚恐：C++11 是否又带来了编程思维的革命？ C++11 是否保持了对 C++98 及 C 的兼容？旧有的 C++ 程序到了 C++11 是否需要被推倒重来？

事实上这些担心都是多余的。相比于 C++98 带来的面向对象的革命性，C++11 带来的

却并非“翻天覆地”式的改变。很多时候，程序员保持着“C++98 式”的观点来看待 C++11 代码也同样是合理的。因为在编程思想上，C++11 依然遵从了一贯的面向对象的思想，并深入加强了泛型编程的支持。从我们的观察来看，C++11 更多的是对步入“成熟稳重”的中年时期的 C++ 的一种改造。比如，像 auto 类型推导这样的新特性，展现出的是语言的亲和力；而右值引用、移动语义的特性，则着重于改变一些使用 C++ 程序库时容易发生的性能不佳的状况。当然，C++11 中也有局部的创新，比如 lambda 函数的引入，以及原子类型的设计等，都体现了语言与时俱进的活力。语言的诸多方面都在 C++11 中再次被锤炼，从而变得更加合理、更加条理清晰、更加易用。C++11 对 C++ 语言改进的每一点，都呈现出了经过长时间技术沉淀的编程语言的特色与风采。所以从这个角度上看，学习 C++11 与 C++98 在思想上是一脉相承的，程序员可以用较小的代价对 C++ 的知识进行更新换代。而在现实中，只要修改少量已有代码（甚至不修改），就可以使用 C++11 编译器对旧有代码进行升级编译而获得新标准带来的好处，这也非常具有实用性。因此，从很多方面来看，C++ 程序员都应该乐于升级换代已有的知识，而学习及使用 C++11 也正是大势所趋。

在本书开始编写的时候，C++11 标准刚刚发布一年，而本书出版的时候，C++11 也只不过才诞生了两年。这一两年，各个编译器厂商或者组织都将支持 C++11 新特性作为了一项重要工作。不过由于 C++11 的语言特性非常的多，因此本书在接近完成时，依然没有一款编译器支持 C++11 所有的新特性。但从从业者的角度看，C++11 迟早会普及，也迟早会成为 C++ 程序员的首选，因此即使现阶段编译器对 C++ 新特性的支持还不充分，但还是有必要在这个时机推出一本全面介绍 C++11 新特性的中文图书。希望通过这样的图书，使得更多的中国程序员能够最快地了解 C++11 新语言标准的方方面面，并且使用最新的 C++11 编译器来从各方面提升自己编写的 C++ 程序。

读者对象

本书针对的对象是已经学习过 C++，并想进一步学习、了解 C++11 的程序员。这里我们假定读者已经具备了基本的 C++ 编程知识，并掌握了一定的 C++ 编程技巧（对于 C++ 的初学者来说，本书阅读起来会有一定的难度）。通过本书，读者可以全面而详细地了解 C++11 对 C++ 进行的改造。无论是试图进行更加精细的面向对象程序编写，或是更加容易地进行泛型编程，或是更加轻松地改造使用程序库等，读者都会发现 C++11 提供了更好的支持。

本书作者和书籍支持

本书的作者都是编译器行业的从业者，主要来自于 IBM XL 编译器中国开发团队。IBM XL 编译器中国开发团队创立于 2010 年，拥有编译器前端、后端、性能分析、测试等各方面的人员，工作职责涵盖了 IBM XL C/C++ 及 IBM XL Fortran 编译器的开发、测试、发布等与编译器产品相关的方方面面。虽然团队成立时间不长，成员却都拥有比较丰富的编译器开发经验，对 C++11 的新特性也有较好的理解。此外，IBM 北美编译器团队成员 Michael（他是 C++ 标准委员会的成员）也参加了本书的编写工作。在书籍的编写上，Michael 为本书

拟定了提纲、确定了章节主题，并直接编写了本书的首章。其余作者则分别对 C++11 各种新特性进行了详细研究讨论，并完成了书稿其余各章的撰写工作。在书稿完成后，除了请 Michael 为本书的部分章节进行了审阅并提出修改意见外，我们又邀请了 IBM 中国信息开发部及 IBM 北京编译器团队的一些成员对本书进行了详细的审阅。虽然在书籍的策划、编写、审阅上我们群策群力，尽了最大的努力，以保证书稿质量，不过由于 C++11 标准发布时间不长，理解上的偏差在所难免，因此本书也可能在特性描述中存在一些不尽如人意或者错误的地方，希望读者、同行等一一为我们指出纠正。我们也会通过博客（http://ibm.co/HK0GCx）、微博（www.weibo.com/ibmcompiler）发布与本书相关的所有信息，并与本书读者共同讨论、进步。

如何阅读本书

读者在书籍阅读中可能会发现，本书的一些章节对 C++ 基础知识要求较高，而某些特性很可能很难应用于自己的编程实践。这样的情况应该并不少见，但这并不是这门语言缺乏亲和力，或是读者缺失了背景知识，这诚然是由于 C++ 的高成熟度导致的。在 C++11 中，不少新特性都会局限于一些应用场景，比如说库的编写，而编写库却通常不是每个程序员必须的任务。为了避免这样的状况，本书第 1 章对 C++11 的语言新特性进行了分类，因此读者可以选择按需阅读，对不想了解的部分予以略过。一些本书的使用约定，读者也可以在第 1 章中找到。

致谢

在这里我们要对 IBM 中国信息开发部的陈晶（作者之一）、卢昉、付琳，以及 IBM 北京编译器团队的冯威、许小羽、王颖对本书书稿详尽细致的审阅表示感谢，同时也对他们专业的工作素养表示由衷的钦佩。此外，我们也要感谢 IBM XL 编译器中国开发团队的舒蓓、张嗣元两位经理在本书编写过程中给予的大力支持。而 IBM 图书社区的刘慎峰及华章图书的杨福川编辑的辛勤工作则保证了本书的顺利出版，在这里我们也要对他们以及负责初审工作的孙海亮编辑说声谢谢。此外，我们还要感谢各位作者的家人在书籍编写过程中给予作者的体谅与支持。最后要感谢的是本书的读者，感谢你们对本书的支持，希望通过这本书，我们能够一起进入 C++ 编程的新时代。

IBM XL 编译器中国开发团队

目　录

免责声明
序
前言

第 1 章　新标准的诞生……1
1.1　曙光：C++11 标准的诞生……1
1.1.1　C++11/C++0x（以及 C11/C1x）——新标准诞生……1
1.1.2　什么是 C++11/C++0x……2
1.1.3　新 C++ 语言的设计目标……3
1.2　今时今日的 C++……5
1.2.1　C++ 的江湖地位……5
1.2.2　C++11 语言变化的领域……5
1.3　C++11 特性的分类……7
1.4　C++ 特性一览……11
1.4.1　稳定性与兼容性之间的抉择……11
1.4.2　更倾向于使用库而不是扩展语言来实现特性……12
1.4.3　更倾向于通用的而不是特殊的手段来实现特性……13
1.4.4　专家新手一概支持……13
1.4.5　增强类型的安全性……14
1.4.6　与硬件紧密合作……14
1.4.7　开发能够改变人们思维方式的特性……15
1.4.8　融入编程现实……16
1.5　本书的约定……17
1.5.1　关于一些术语的翻译……17
1.5.2　关于代码中的注释……17
1.5.3　关于本书中的代码示例与实验平台……18

第 2 章　保证稳定性和兼容性……19
2.1　保持与 C99 兼容……19
2.1.1　预定义宏……19
2.1.2　__func__ 预定义标识符……20
2.1.3　_Pragma 操作符……22
2.1.4　变长参数的宏定义以及 __VA_ARGS__……22
2.1.5　宽窄字符串的连接……23
2.2　long long 整型……23
2.3　扩展的整型……25
2.4　宏 __cplusplus……26
2.5　静态断言……27
2.5.1　断言：运行时与预处理时……27
2.5.2　静态断言与 static_assert……28
2.6　noexcept 修饰符与 noexcept 操作符……32
2.7　快速初始化成员变量……36
2.8　非静态成员的 sizeof……39
2.9　扩展的 friend 语法……40
2.10　final/override 控制……44
2.11　模板函数的默认模板参数……48
2.12　外部模板……50
2.12.1　为什么需要外部模板……50
2.12.2　显式的实例化与外部模板的声明……52
2.13　局部和匿名类型作模板实参……54
2.14　本章小结……55

第 3 章　通用为本，专用为末……57
3.1　继承构造函数……57

3.2 委派构造函数 ····· 62
3.3 右值引用：移动语义和完美转发 ····· 68
3.3.1 指针成员与拷贝构造 ····· 68
3.3.2 移动语义 ····· 69
3.3.3 左值、右值与右值引用 ····· 75
3.3.4 std::move：强制转化为右值 ····· 80
3.3.5 移动语义的一些其他问题 ····· 82
3.3.6 完美转发 ····· 85
3.4 显式转换操作符 ····· 89
3.5 列表初始化 ····· 92
3.5.1 初始化列表 ····· 92
3.5.2 防止类型收窄 ····· 96
3.6 POD 类型 ····· 98
3.7 非受限联合体 ····· 106
3.8 用户自定义字面量 ····· 110
3.9 内联名字空间 ····· 113
3.10 模板的别名 ····· 118
3.11 一般化的 SFINEA 规则 ····· 119
3.12 本章小结 ····· 121
第 4 章 新手易学，老兵易用 ····· 123
4.1 右尖括号 > 的改进 ····· 123
4.2 auto 类型推导 ····· 124
4.2.1 静态类型、动态类型与类型推导 ····· 124
4.2.2 auto 的优势 ····· 126
4.2.3 auto 的使用细则 ····· 130
4.3 decltype ····· 134
4.3.1 typeid 与 decltype ····· 134
4.3.2 decltype 的应用 ····· 136
4.3.3 decltype 推导四规则 ····· 140
4.3.4 cv 限制符的继承与冗余的符号 ··· 143
4.4 追踪返回类型 ····· 145
4.4.1 追踪返回类型的引入 ····· 145
4.4.2 使用追踪返回类型的函数 ····· 146
4.5 基于范围的 for 循环 ····· 150
4.6 本章小结 ····· 153
第 5 章 提高类型安全 ····· 155
5.1 强类型枚举 ····· 155
5.1.1 枚举：分门别类与数值的名字 ··· 155
5.1.2 有缺陷的枚举类型 ····· 156
5.1.3 强类型枚举以及 C++11 对原有枚举类型的扩展 ····· 160
5.2 堆内存管理：智能指针与垃圾回收 ····· 163
5.2.1 显式内存管理 ····· 163
5.2.2 C++11 的智能指针 ····· 164
5.2.3 垃圾回收的分类 ····· 167
5.2.4 C++ 与垃圾回收 ····· 169
5.2.5 C++11 与最小垃圾回收支持 ····· 170
5.2.6 垃圾回收的兼容性 ····· 172
5.3 本章小结 ····· 173
第 6 章 提高性能及操作硬件的能力 ····· 174
6.1 常量表达式 ····· 174
6.1.1 运行时常量性与编译时常量性 ··· 174
6.1.2 常量表达式函数 ····· 176
6.1.3 常量表达式值 ····· 178
6.1.4 常量表达式的其他应用 ····· 180
6.2 变长模板 ····· 183
6.2.1 变长函数和变长的模板参数 ····· 183
6.2.2 变长模板：模板参数包和函数参数包 ····· 185
6.2.3 变长模板：进阶 ····· 189
6.3 原子类型与原子操作 ····· 196
6.3.1 并行编程、多线程与 C++11 ····· 196
6.3.2 原子操作与 C++11 原子类型 ····· 197
6.3.3 内存模型，顺序一致性与 memory_order ····· 203
6.4 线程局部存储 ····· 214
6.5 快速退出：quick_exit 与 at_quick_exit ····· 216
6.6 本章小结 ····· 219

第 7 章　为改变思考方式而改变……220
7.1　指针空值——nullptr……220
7.1.1　指针空值：从 0 到 NULL，再到 nullptr……220
7.1.2　nullptr 和 nullptr_t……223
7.1.3　一些关于 nullptr 规则的讨论……225
7.2　默认函数的控制……227
7.2.1　类与默认函数……227
7.2.2　“= default”与“= deleted”……230
7.3　lambda 函数……234
7.3.1　lambda 的一些历史……234
7.3.2　C++11 中的 lambda 函数……235
7.3.3　lambda 与仿函数……238
7.3.4　lambda 的基础使用……240
7.3.5　关于 lambda 的一些问题及有趣的实验……243
7.3.6　lambda 与 STL……247
7.3.7　更多的一些关于 lambda 的讨论…254
7.4　本章小结……256
第 8 章　融入实际应用……258
8.1　对齐支持……258
8.1.1　数据对齐……258
8.1.2　C++11 的 alignof 和 alignas……261
8.2　通用属性……267
8.2.1　语言扩展到通用属性……267
8.2.2　C++11 的通用属性……268
8.2.3　预定义的通用属性……270
8.3　Unicode 支持……274
8.3.1　字符集、编码和 Unicode……274
8.3.2　C++11 中的 Unicode 支持……276
8.3.3　关于 Unicode 的库支持……280
8.4　原生字符串字面量……284
8.5　本章小结……286
附录 A　C++11 对其他标准的不兼容项目……287
附录 B　弃用的特性……294
附录 C　编译器支持……301
附录 D　相关资源……304

第 1 章

新标准的诞生

从最初的代号 C++0x 到最终的名称 C++11，C++ 的第二个真正意义上的标准姗姗来迟。可以想象，这个迟来的标准必定遭遇了许多的困难，而 C++ 标准委员会应对这些困难的种种策略，则构成新的 C++ 语言基因，我们可以从新的 C++11 标准中逐一体会。而客观上，这些基因也决定了 C++11 新特性的应用范畴。在本章中，我们会从设计思维和应用范畴两个维度对所有的 C++11 新特性进行分类，并依据这种分类对一些特性进行简单的介绍，从而一览 C++11 的全景。

1.1 曙光：C++11 标准的诞生

1.1.1 C++11/C++0x（以及 C11/C1x）——新标准诞生

2011 年 11 月，在印第安纳州布卢明顿市，“八月印第安纳大学会议”（August Indiana University Meeting）缓缓落下帷幕。这次会议的结束，意味着长久以来以 C++0x 为代号的 C++11 标准终于被 C++ 标准委员会批准通过。至此，C++ 新标准尘埃落定。从 C++98 标准通过的时间开始计算，C++ 标准委员会，即 WG21，已经为新标准工作了 11 年多的时间。对于一个编程语言标准而言，11 年显然是个非常长的时间。其间我们目睹了面向对象编程的盛极，也见证了泛型编程的风起云涌，还见证了 C++ 后各种新的流行编程语言的诞生。不过在新世纪第二个 10 年的伊始，C++ 的标准终于二次来袭。

事实上，在 2003 年 WG21 曾经提交了一份技术勘误表（Technical Corrigendum，简称 TC1）。这次修订使得 C++03 这个名字已经取代了 C++98 成为 C++11 之前的最新 C++ 标准名称。不过由于 TC1 主要是对 C++98 标准中的漏洞进行修复，核心语言规则部分则没有改动，因此，人们还是习惯地把两个标准合称为 C++98/03 标准。

注意 在本书中，但凡是 C++98 和 C++03 标准没有差异时，我们都会沿用 C++98/03 这样的俗称，或者直接简写为 C++98。如果涉及 TC1 中所提出的微小区别，我们会使用 C++98 和 C++03 来分别指代两种 C++ 标准。

C++11 是一种新语言的开端。虽然设计 C++11 的目的是为了要取代 C++98/03，不过相比于 C++03 标准，C++11 则带来了数量可观的变化，这包括了约 140 个新特性，以及对

C++03 标准中约 600 个缺陷的修正。因此，从这个角度看来 C++11 更像是从 C++98/03 中孕育出的一种新语言。正如当年 C++98/03 为 C++ 引入了如异常处理、模板等许多让人耳目一新的新特性一样，C++11 也通过大量新特性的引入，让 C++ 的面貌焕然一新。这些全新的特性以及相应的全新的概念，都是我们要在本书中详细描述的。

1.1.2 什么是 C++11/C++0x

C++0x 是 WG21 计划取代 C++98/03 的新标准代号。这个代号还是在 2003 年的时候取的。当时委员会乐观地估计，新标准会在 21 世纪的第一个 10 年内完成。从当时看毕竟还有 6 年的时间，确实无论如何也该好了。不过 2010 新年钟声敲响的时候，WG21 内部却还在为一些诸如哪些特性该放弃，哪些特性该被削减的议题而争论。于是所有人只好接受这个令人沮丧的事实：新标准没能准时发布。好在委员会成员保持着乐观的情绪，还常常相互开玩笑说，x 不是一个 0 到 9 的十进制数，而应该是一个十六进制数，我们还可以有 A、B、C、D、E、F。虽然这是个玩笑，但也有点认真的意思，如果需要，WG21 会再使用“额外”的 6 年，在 2015 年之前完成标准。不过众所周知的，WG21“只”再花了两年时间就完成了 C++11 标准。

注意 C 语言标准委员会（C committee）WG14 也几乎在同时开始致力于取代 C99 标准。不过相比于 WG21，WG14 对标准完成的预期更加现实。因为他们使用的代号是 C1x，这样新的 C 标准完成的最后期限将是 2019 年。事实上 WG14 并没用那么长时间，他们最终在 2011 年通过了提案，也就是 C11 标准。

从表 1-1 中可以看到 C++ 从诞生到最新通过的 C++11 标准的编年史。

表 1-1 C++ 发展编年史

日 期	事 件
1990 年	*The Annotated C++ Reference Manual*，M.A.Ellis 和 B.Stroustrup 著。主要描述了 C++ 核心语言，没有涉及库
1998 年	第一个国际化的 C++ 语言标准：IOS/IEC 15882:1998。包括了对核心语言及 STL、locale、iostream、numeric、string 等诸多特性的描述
2003 年	第二个国际化的 C++ 语言标准：IOS/IEC 15882:2003。核心语言及库与 C++98 保持了一致，但包含了 TC1（Technical Corrigendum 1, 技术勘误表 1）。自此，C++03 取代了 C++98
2005 年	TR1（Technical Report 1，技术报告 1）：IOS/IEC TR 19768:2005。核心语言不变。TR1 作为标准的非规范出版物，其包含了 14 个可能进入新标准的新程序库
2007 年 9 月	SC22 注册（特性）表决。通过了 C++0x 中核心特性
2008 年 9 月	SC22 委员会草案（Committee Draft, CD）表决。基本上所有 C++0x 的核心特性都完成了，新的 C++0x 标准草稿包括了 13 个源自 TR1 的库及 70 个库特性，修正了约 300 个库缺陷。此外，新标准草案还包括了 70 多个语言特性及约 300 个语言缺陷的修正
2010 年 3 月	SC22 最终委员会草案（Final Committee Draft, FCD）表决。所有核心特性都已经完成，处理了各国代表的评议

（续）

日　期	事　件
2011 年 11 月	JTC1 C++11 最终国际化标准草案（Final Draft International Standard, FDIS）发布，即 IOS/IEC 15882:2011。新标准在核心语言部分和标准库部分都进行了很大的改进，这包括 TR1 的大部分内容。但整体的改进还是与先前的 C++ 标准兼容的
2012 年 2 月	在 ANSI 和 ISO 商店可以以低于原定价的价格买到 C++11 标准

注意 语言标准的发布通常有两种——规范的（Normative）及不规范的（Non-normative）。前者表示内容通过了批准（ratified），因此是正式的标准，而后者则不是。不过不规范的发布通常是有积极意义的，比方说 TR1，它就是不规范的标准，但是后来很多 TR1 的内容都成为了 C++11 标准的一部分。

图 1-1 比较了两个语言标准委员会（WG21，WG14）制定新标准的工作进程，其中一些重要时间点都标注了出来。

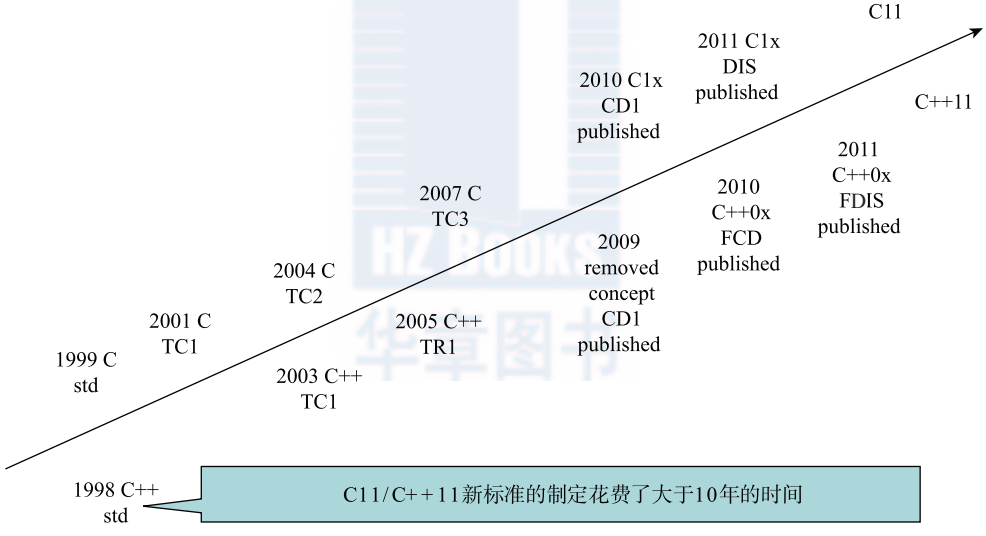

图 1-1　WG21 和 WG14 制定新语言标准的工作进程

1.1.3　新 C++ 语言的设计目标

如果读者已经学习过 C++98/03，就可以发现 C++98/03 的设计目标如下：

- ❑比 C 语言更适合系统编程（且与 C 语言兼容）。
- ❑支持数据抽象。
- ❑支持面向对象编程。
- ❑支持泛型编程。

这些特点使得面向对象编程和泛型编程在过去的 10 ~ 20 年内成为编程界的明星。不过从那时开始，C++ 的发展就不仅仅是靠学者的远见前瞻去推动的，有时也会借由一些“奇缘”而演进。比方说，C++ 模板就是这样一个“奇缘”。它使得 C++ 近乎成为了一种函数式编程语言，而且还使得 C++ 程序员拥有了模板元编程的能力。但是凡事有两面，C++98/03 中的一些较为激进的特性，比如说动态异常处理、输出模板，现在回顾起来则是不太需要的。当然，这是由于我们有了“后见之明”，或者由于这些特性在新情况下不再适用，又或者它们影响了 C++11 的新特性的设计。因此一部分这样的特性已经被 C++11 弃用了。在附录 B 中我们会一一列出这些弃用的特性，并分析其被弃用的原因。

而 C++11 的整体设计目标如下：

- 使得 C++ 成为更好的适用于系统开发及库开发的语言。
- 使得 C++ 成为更易于教学的语言（语法更加一致化和简单化）。
- 保证语言的稳定性，以及和 C++03 及 C 语言的兼容性。

我们可以分别解释一下。

首先，使 C++ 成为更好的适用于系统开发及库开发的语言，意味着标准并不只是注重为某些特定的领域提供专业化功能，比如专门为 Windows 开发提供设计，或者专门为数值计算提供设计。标准希望的是使 C++ 能够对各种系统的编程都作出贡献。

其次，使得 C++ 更易于教学，则意味着 C++11 修复了许多令程序员不安的语言“毒瘤”。这样一来，C++ 语法显得更加一致化，新手使用起来也更容易上手，而且有了更好的语法保障。其实语言复杂也有复杂的好处，比如 ROOTS、DEALII 等一些复杂科学运算的算法，它们的作者非常喜爱泛型编程带来的灵活性，于是 C++ 语言最复杂的部分正好满足了他们的需求。但是在这个世界上，新手总是远多于专家。而即使是专家，也常常只是精通自己的领域。因此语言不应该复杂到影响人们的学习。本书作者之一也是 WG21 中的一员，从结果上看，无论读者怎么看待 C++11，委员会大多数人都认同 C++11 达成了易于教学这个目标（即使其中还存在着些看似严重的小缺陷）。

最后，则是语言的稳定性。经验告诉我们，伟大的编程语言能够长期存活下来的原因还是因为语言的设计突出了实用性。事实上，在标准制定过程中，委员会承担了很多压力，这些压力源自于大家对加入更多语言特性的期盼——每一个人都希望将其他编程语言中自己喜欢的特性加入到新的 C++ 中。对于这些热烈而有些许盲目的期盼，委员会成员在 Bjarne Stroustrup 教授的引导下，选择了不断将许多无关的特性排除在外。其目的是防止 C++ 成为一个千头万绪的但功能互不关联的语言。而如同现在看到的那样，C++11 并非大而无序，相反地，许多特性可以良好地协作，进而达到“1 + 1 > 2”的效果。可以说，有了这些努力，今天的读者才能够使用稳定而强大的 C++11，而不用担心语言本身存在着混乱状况甚至是冲突。

值得一提的是，虽然在取舍新语言特性方面标准委员会曾面临过巨大压力，但与此同

时，标准委员会却没有收集到足够丰富的库的新特性。作为一种通用型语言，C++ 是否是成功，通常会依赖于不同领域中 C++ 的使用情况，比如科学计算、游戏、制造业、网络编程等。在 C++11 通过的标准库中，服务于各个领域的新特性确实还是太少了。因此很有可能在下一个版本的 C++ 标准制定中，如何标准化地使用库将成为热门话题，标准委员会也准备好了接受来自这方面的压力。

1.2 今时今日的 C++

1.2.1 C++ 的江湖地位

如今 C++ 依旧位列通用编程语言三甲，不过似乎没有以前那么流行了。事实上，编程语言排名通常非常难以衡量。比如，某位教授或学生用了 C++ 来教授课程应该被计算在内吗？在新的联合攻击战斗机（Joint Strike Fighter，JSF-35）的航空电子设备中使用了 C++ 编程应该计算在内吗？又或者 C++ 被用于一款流行的智能手机操作系统的编程中算不算呢？再或者是 C++ 被用于编写最流行的在线付费搜索引擎，或用于构建一款热门的第一人称射击游戏的引擎，或用于构建最热门的社交网络的代码库，这些都该计算在内吗？事实上，据我们所知，以上种种都使用了 C++ 编程。而且在构建致力于沟通软硬件的系统编程中，C++ 也常常是必不可少的。甚至，C++ 还常用于设计和编写编程语言。因此我们可以认为，编程语言价值的衡量标准应该包括数量、新颖性、质量，以及以上种种，都应该纳入“考核”。这样一来，结论就很明显了：C++ 无处不在。

1.2.2 C++11 语言变化的领域

如果谁说 C++11 只是对 C++ 语言做了大幅度的改进，那么他很可能就错过了 C++11 精彩的地方。事实上，读罢本书后，读者只需要看一眼代码，就可以说出代码究竟是 C++98/03 的，还是 C++11 的。C++11 为程序员创造了很多更有效、更便捷的代码编写方式，程序员可以用简短的代码来完成 C++98/03 中同样的功能，简单到你惊呼“天哪，怎么能这么简单”。从一些简单的数据统计上看，比起 C++98/03，C++11 大大缩短了代码编写量，依情况最多可以将代码缩短 30% ~ 80%。

那么 C++11 相对于 C++98/03 有哪些显著的增强呢？事实上，这包括以下几点：

- ❑通过内存模型、线程、原子操作等来支持本地并行编程（Native Concurrency）。
- ❑通过统一初始化表达式、auto、declytype、移动语义等来统一对泛型编程的支持。
- ❑通过 constexpr、POD（概念）等更好地支持系统编程。
- ❑通过内联命名空间、继承构造函数和右值引用等，以更好地支持库的构建。

表 1-2 列出了 C++11 批准通过的，且本书将要涉及的语言特性。这是一张相当长的表，而且一个个陌生的词汇足以让新手不知所措。不过现在还不是了解它们的时候。但看过这张

表，读者至少会有这样一种感觉：C++11 的确像是一门新的语言。如果我们将 C++98/03 标准中的特性和 C++11 放到一起，C++11 则像是个恐怖的“编程语言范型联盟”。利用它不仅仅可以写出面向对象语言的代码，也可以写出过程式编程语言代码、泛型编程语言代码、函数式编程语言代码、元编程编程语言代码，或者其他。多范型的支持使得 C++11 语言的“硬能力”几乎在编程语言中“无出其右”。

表 1-2 C++11 主要的新语言特性（中英文对照）

中文翻译	英文名称	备注
__cplusplus 宏	__cplusplus macro	
对齐支持	alignment support	
通用属性	general attribute	
原子操作	atomic operation	
auto 类型推导（初始化类型推导）	auto (type deduction from 18nitialize)	
C99 特性	C99	
强类型枚举	enum class (scoped and strongly typed enums)	
复制及再抛出异常	copy and rethrow exception	本书未讲解
常量表达式	constexpr	
decltype	decltype	
函数的默认模板参数	default template parameters for function	
显式默认和删除的函数（默认的控制）	defaulted and deleted functions (control of defaults)	
委托构造函数	delegating constructors	
并行动态初始化和析构	Dynamic Initialization and Destruction with Concurrency	本书未讲解
显式转换操作符	explicit conversion operators	
扩展的 friend 语法	extended friend syntax	
扩展的整型	extended integer types	
外部模板	extern templates	
一般化的 SFINAE 规则	generalized SFINAE rules	
统一的初始化语法和语义	Uniform initialization syntax and semantics	
非受限联合体	unrestricted union	
用户定义的字面量	user-defined literals	
变长模板	variadic templates	
类成员初始化	in-class member initializers	
继承构造函数	inherited constructors	
初始化列表	initializer lists	
lambda 函数	lambda	
局部类型用作模板参数	local classes as template arguments	
long long 整型	long long integers	
内存模型	memory model	
移动语义（参见右值引用）	move semantics（see rvalue references）	
内联名字空间	Inline namespace	

（续）

中文翻译	英文名称	备　注
防止类型收窄	Preventing narrowing	
指针空值	nullptr	
POD	POD（plain old data）	
基于范围的 for 语句	range-based for statement	
原生字符串字面量	raw string literals	
右值引用	rvalue reference	
静态断言	static assertions	
追踪返回类型语法	trailing return type syntax	
模板别名	template alias	
线程本地的存储	thread-local storage	
Unicode	Unicode	

而从另一个角度看，编程中程序员往往需要将实物、流程、概念等进行抽象描述。但通常情况下，程序员需要抽象出的不仅仅是对象，还有一些其他的概念，比如类型、类型的类型、算法，甚至是资源的生命周期，这些实际上都是 C++ 语言可以描述的。在 C++11 中，这些抽象概念常常被实现在库中，其使用将比在 C++98/03 中更加方便，更加好用。从这个角度上讲，C++11 则是一种所谓的“轻量级抽象编程语言”（Lightweight Abstraction Programming Language）。其好处就是程序员可以将程序设计的重点更多地放在设计、实现，以及各种抽象概念的运用上。

总的来说，灵活的静态类型、小的抽象概念、绝佳的时间与空间运行性能，以及与硬件紧密结合工作的能力都是 C++11 突出的亮点。而反观 C++98/03，其最强大的能力则可能是体现在能够构建软件基础架构，或构建资源受限及资源不受限的项目上。因此，C++11 也是 C++ 在编程语言领域上一次“泛化”与进步。

要实现表 1-2 中的各种特性，需要编译器完成大量的工作。对于大多数编译器供应商来说，只能分阶段地发布若干个编译版本，逐步支持所有特性（罗马从来就不是一天建成的，对吧）。大多数编译器已经开始了对 C++11 特性的支持。有 3 款编译器甚至从 2008 年前就开始支持 C++11 了：IBM 的 XL C/C++ 编译器从版本 10.1 开始。GNU 的 GCC 编译器从版本 4.3 开始，英特尔编译器从版本 10.1 开始。而微软则从 Visual Studio 2010 开始。最近，苹果的 clang/llvm 编译器也从 2010 年的版本 2.8 开始支持 C++11 新特性，并且急速追赶其他编译器供应商。在本书附录 C 中，读者可以找到现在情况下各种编译器对 C++11 的支持情况。

1.3　C++11 特性的分类

从设计目标上说，能够让各个特性协同工作是设计 C++11/0x 中最为关键的部分。委员会总希望通过特性协作取得整体大于个体的效果，但这也是语言设计过程中最困难的一点。

因此相比于其他的各种考虑，WG21 更专注于以下理念：

- ❑保持语言的稳定性和兼容性（Maintain stability and compatibility）。
- ❑更倾向于使用库而不是扩展语言来实现特性（Prefer libraries to language extensions）。
- ❑更倾向于通用的而不是特殊的手段来实现特性（Prefer generality to specialization）。
- ❑专家新手一概支持（Support both experts and novices）。
- ❑增强类型的安全性（Increase type safety）。
- ❑增强代码执行性能和操作硬件的能力（Improve performance and ability to work directly with hardware）。
- ❑开发能够改变人们思维方式的特性（Make only changes that change the way people think）。
- ❑融入编程现实（Fit into the real world）。

根据这些设计理念可以对新特性进行分类。在本书中，我们的核心章节（第 2 ~ 8 章）也会按照这样的方式进行划分。在可能的时候，我们也会为每个理念取个有趣一点儿的中文名字。

而从使用上，Scott Mayers 则为 C++11 创建了另外一种有效的分类方式，Mayers 根据 C++11 的使用者是类的使用者，还是库的使用者，或者特性是广泛使用的，还是库的增强的来区分各个特性。具体地，可以把特性分为以下几种：

- ❑类作者需要的（class writer，简称为“类作者”）
- ❑库作者需要的（library writer，简称为“库作者”）
- ❑所有人需要的（everyone，简称为“所有人”）
- ❑部分人需要的（everyone else，简称为“部分人”）

那么我们可以结合这种分类再来看一下可以怎样来学习所有的特性。下面我们通过设计理念和用户群对 C++11 特性进行分类，如表 1-3 所示。

由于 C++11 的新特性非常多，因此本书不准备涵盖所有内容。我们粗略地将特性划分为核心语言特性和库特性。而从 C++11 标准的章节划分来看（读者可以从网站上搜到接近于最终版本的草稿，正式的标准需要通过购买获得），本书将涉及 C++11 标准中第 1 ~ 16 章的语言特性部分（在 C++11 语言标准中，第 1 ~ 16 章涵盖了核心语言特性，第 17 ~ 30 章涉及库特性），而标准库将不在本书中描述。当然，这会导致许多灰色地带，因为如同我们提到的，我们总是倾向于使用库而不是语言扩展来实现一些特性，那么实际上，讲解语言核心特性也必然涉及库的内容。典型的，原子操作（atomics）就是这样一个例子。因此，在本书的编写中，我们只是不对标准库进行专门的讲解，而与核心内容相关的库内容，我们还是会有所描述的。

表 1-3　根据设计理念和用户群对 C++11 新特性进行划分

理　念	特性名称（中英文）	用户群
保持语言的稳定性和兼容性（Maintain stability and compatibility）	C99	部分人
	函数的默认模板参数 default template parameters for function	所有人
	扩展的 friend 语法 extended friend syntax	部分人
	扩展的整型 extended integer types	部分人
	外部模板 extern templates	部分人
	类成员的初始化 in-class member initializers	部分人
	局部类用作模板参数 local classes as template arguments	部分人
	long long 整型 long long integers	部分人
	__cplusplus	部分人
	noexcept	库作者
	override/final 控制 Override/final controls	部分人
	静态断言 static assertions	库作者
	类成员的 sizeof sizeof class data members	部分人
更倾向于通用的而不是特殊化的手段来实现特性（Prefer generality to specialization）	继承构造函数 Inherited constructor	类作者
	移动语义，完美转发，引用折叠 Move semantics, perfect forwarding, reference collapse	类作者
	委托构造函数 delegating constructors	类作者
	显式转换操作符 explicit conversion operators	库作者
	统一的初始化语法和语义，初始化列表，防止收窄 Uniform initialization syntax and semantics, initializer lists, Preventing narrowing	所有人
	非受限联合体 unrestricted unions (generalized)	部分人
	用户自定义字面量 UDL	部分人
	一般化 SFINAE 规则 generalized SFINAE rules	库作者
	内联名字空间 Inline Namespace	部分人
	PODs	部分人
	模板别名 template alias	所有人
专家新手一概支持（Support both experts and novices）	右尖括号 Right angle bracket	所有人
	auto	所有人
	基于范围的 for 语句 Ranged For	所有人
	decltype	库作者
	追踪返回类型语法（扩展的函数声明语法）Trailing return type syntax (extended function declaration syntax)	所有人
增强类型的安全性（Increase type safety）	强类型枚举 Strong enum	部分人
	unique_ptr, shared_ptr	类作者
	垃圾回收 ABI Garbage collection ABI	库作者

（续）

理　念	特性名称（中英文）	用户群
增强性能和操作硬件的能力（Improve performance and ability to work directly with hardware）	常量表达式 constexpr	类作者
	原子操作 / 内存模型 atomics/mm	所有人
	复制和再抛出异常 copying and rethrowing exceptions	所有人
	并行动态初始化和析构 Dynamic Initialization and Destruction with Concurrency	所有人
	变长模板 Variadic template	库作者
	线程本地的存储 thread-local storage	所有人
	快速退出进程 quick_exit Abandoning a process	所有人
开发能够改变人们思维方式的特性（Make only changes that change the way people think）	指针空值 nullptr	所有人
	显示默认和删除的函数（默认的控制）defaulted and deleted functions (control of defauls)	类作者
	lambdas	所有人
融入编程现实（Fit into the real world）	对齐支持 Alignments	部分人
	通用属性 Attributes [[carries dependency]] [[noreturn]]	部分人
	原生字符串字面量 raw string literals	所有人
	Unicode unicode characters	所有人

而之前我们提到过的“更倾向于使用库而不是扩展语言来实现特性”理念的部分，如果有可能，我们会在另一本书或者本书的下一个版本中来进行讲解。下面列出了属于该设计理念下的库特性：

- ❑算法增强 Algorithm improvements
- ❑容器增强 Container improvements
- ❑分配算符 Scoped allocators
- ❑std::array
- ❑std::forward_list
- ❑无序容器 Unordered containers
- ❑sts::tuple
- ❑类型特性 Type traits
- ❑std::function, std::bind
- ❑unique_ptr
- ❑shared_ptr
- ❑weak_ptr
- ❑线程 Threads
- ❑互斥 Mutex
- ❑锁 Locks

❑条件变量 Condition variables
❑时间工具 Time utilities
❑std::future, std::promises
❑std::async
❑随机数 Random numbers
❑正则表达式 regex

1.4 C++ 特性一览

接下来，我们会一窥 C++11 中的各种特性，了解它们的来历、用途、特色等。可能这部分对于还没有开始阅读正文的读者来说有些困难。如果有机会，我们建议读者在读完全书后再回到这里，这也是全书最好的总结。

1.4.1 稳定性与兼容性之间的抉择

通常在语言设计中，不破坏现有的用户代码和增加新的能力，这二者是需要同时兼顾的。就像之前的 C 一样，如今 C++ 在各种代码中、开源库中，或用户的硬盘中都拥有上亿行代码，那么当 C++ 标准委员会要改变一个关键字的意义，或者发明一个新的关键字时，原有代码就很可能发生问题。因为有些代码可能已经把要加入的这个准关键字用作了变量或函数的名字。

语言的设计者或许能够完全不考虑兼容性，但说实话这是个丑陋的做法，因为来自习惯的力量总是超乎人的想象。因此 C++11 只是在非常必要的情况下才引入新的关键字。WG21 在加入这些关键字的时候非常谨慎，至少从谷歌代码搜索（Google Code Search）的结果看来，这些关键字没有被现有的开源代码频繁地使用。不过谷歌代码搜索只会搜索开源代码，私人的或者企业的代码库（codebase）是不包含在内的。因此这些数据可能还有一定的局限性，不过至少这种方法可以避免一些问题。而 WG21 中也有很多企业代表，他们也会帮助 WG21 确定这些关键字是否会导致自己企业代码库中代码不兼容的问题。

C++11 的新关键字如下：

❑alignas
❑alignof decltype
❑auto（重新定义）
❑static_assert
❑using（重新定义）
❑noexcept
❑export（弃用，不过未来可能留作他用）
❑nullptr

❑constexpr

❑thread_local

这些新关键字都是相对于 C++98/03 来说的。当然，引入它们可能会破坏一些 C++98/03 代码，甚至更为糟糕的是，可能会悄悄地改变了原有 C++98/03 程序的目的。static_assert 就是这样一个例子。为了降低它与已有程序变量冲突的可能性，WG21 将这个关键字的名字设计得很长，甚至还包含了下划线，可以说命名丑得不能再丑了，不过在一些情况下，它还是会发生冲突，比如：

```
static_assert(4<=sizeof(int), "error:small ints");
```

这行代码的意图是确定编译时（不是运行时）系统的 int 整型的长度不小于 4 字节，如果小于，编译器则会报错说系统的整型太小了。在 C++11 中这是一段有效的代码，在 C++98/03 中也可能是有效的，因为程序员可能已经定义了一个名为 static_assert 的函数，以用于判断运行时的 int 整型大小是否不小于 4。显然这与 C++11 中的 static_assert 完全不同。

实际上，在 C++11 中还有两个具有特殊含义的新标识符：override、final。这两个标识符如何被编译器解释与它们所在的位置有关。如果这两个标识符出现在成员函数之后，它们的作用是标注一个成员函数是否可以被重载。不过读者实际上也可以在 C++11 代码中定义出 override、final 这样名称的变量。而在这样的情况下，它们只是标识了普通的变量名称而已。

我们主要会在第 2 章中看到相关的特性的描述。

1.4.2 更倾向于使用库而不是扩展语言来实现特性

相比于语言核心功能的稳定，库则总是能随时为程序员提供快速上手的、方便易用的新功能。库的能量是巨大的，Boost[⊖]和一些公司私有的库（如 Qt、POOMA）的快速成长就说明了这一点。而且库有一个很大的优势，就是其改动不需要编译器实现新特性（只要接口保持一致即可），当然，更重要的是库可以用于支持不同领域的编程。这样一来，通常读者不需要非常精通 C++ 就能使用它们。

不过这些优点并不是被广泛认可的。狂热的语言爱好者总是觉得功能加入语言特性，由编译器实现了才是王道，而库只是第二选择。不过 WG21 的看法跟他们相反。事实上，如果可能，WG21 会尽量将一个语言特性转为库特性来实现。比较典型的如 C++11 中的线程，它被实现为库特性的一部分：std::thread，而不是一个内置的“线程类型”。同样的，C++11 中没有内置的关联数组（associative array）类型，而是将它们实现为如 std::unorder_map 这样的库。再者，C++11 也没有像其他语言一样在语言核心部分加入正则表达式功能，而是实现为

⊖ 在 C++ 的众多开源库，最为出名的应该是 Boost。Boost 是一个无限制的开源库，在设计和审阅的时候，都常常有 C++ 标准委员会的人参与。

std::regex 库。这样一来，C++ 语言可以尽量在保持较少的核心语言特性的同时，通过标准库扩大其功能。

从传统意义上讲，库可能是通过提供头文件来实现的。当然，有些时候库的提供者也会将一些实现隐藏在二进制代码库存档（archive）文件中。不过并非所有的库都是通过这样的方式提供的。事实上，库也有可能实现于编译器内部。比如 C++11 中的原子操作等许多内容，就通常不是在头文件或库存档中实现的。编译器会在内部就将原子操作实现为具体的机器指令，而无需在稍后去链接实实在在的库进行存档。而之所以将原子操作的内容放在库部分，也是为了满足将原子操作作为库实现的自由。从这个意义上讲，原子操作并非纯粹的“库”，因此也被我们选择性地纳入了本书的讲解中。

1.4.3 更倾向于通用的而不是特殊的手段来实现特性

如我们说到的，如果将无数互不相关的小特性加入 C++ 中，而且不加选择地批准通过，C++ 将成为一个令人眼花缭乱的“五金店”，不幸的是，这个五金店的产品虽然各有所长，凑在一起却是一盘散沙，缺乏战斗力。所以 WG21 更希望从中抽象出更为通用的手段而不是加入单独的特性来“练成”C++11 的“十八般武艺”。

显式类型转换操作符是一个很好的例子。在 C++98/03 中，可以用在构造函数前加上 explicit 关键字来声明构造函数为显式构造，从而防止程序员在代码中“不小心”将一些特定类型隐式地转换为用户自定义类型。不过构造函数并不是唯一会导致产生隐式类型转换的方法，在 C++98/03 中类型转换操作符也可以参与隐式转换，而程序员的意图则可能只是希望类型转换操作符在显式转换时发生。这是 C++98/03 的疏忽，不过在 C++11 中，我们已经可以做到这点了。

其他的一些新特性，比如继承构造函数、移动语义等，在本书的第 3 章中我们均会涉及。

1.4.4 专家新手一概支持

如果 C++ 只是适合专家的语言，那它就不可能是一门成功的语言。C++ 中虽然有许多专家级的特性，但这并不是必须学习的。通常程序员只需要学习一定的知识就可以使用 C++。而在 C++11 中，从易用的角度出发，修缮了很多特性，也铲除了许多带来坏声誉的“毒瘤”，比如一度被群起而攻之的“毒瘤”——双右尖括号。在 C++98/03 中，由于采用了最长匹配的解析规则（maximal munch parsing rule），编译器会在解析符号时尽可能多地“吸收”符号。这样一来，在模板解析的时候，编译器就会将原本是“模板的模板”识别为右移，并“理直气壮”地抛出一条令人绝望的错误信息：模板参数中不应该存在的右移。如今这个问题已经在 C++11 中被修正。模板参数内的两个右尖括号会终结模板参数，而不会导致编译器错误。当然从实现上讲，编译器只需要在原来报错的地方加入一些上下文的判断就可以避免

这样的错误了。比如：

```
vector<list<int>> veclist: //C++11 中有效，C++98/03 中无效
```

另一个 C++11 易于上手的例子则是统一初始化语法的引入。C++ 继承了 C 语言中所谓的“集合初始化语法”（aggregate initialization syntax，比如 a[] = {0, 1,}; ），而在设计类的时候，却只定义了形式单一的构造函数的初始化语法，比如 A a(0, 1)。所以在使用 C++98/03 的时候，编写模板会遇到障碍，因为模板作者无法知道模板用户会使用哪种类型来初始化模板。对于泛型编程来说，这种不一致则会导致不能总是进行泛型编程。而在 C++11 中，标准统一了变量初始化方法，所以模板作者可以总是在模板编写中采用集合初始化（初始化列表）。进一步地，集合初始化对于类型收窄还有一定的限制。而类型收窄也是许多让人深夜工作的奇特错误的源头。因此在 C++11 中使用了初始化列表，就等同于拥有了防止收窄和泛型编程的双重好处。

读者可以在第 4 章看到 C++11 是如何增进语言对新手的支持的。

1.4.5 增强类型的安全性

绝对的类型安全对编程语言来说几乎是不可能达到的，不过在编译时期捕捉更多的错误则是非常有益的。在 C++98/03 中，枚举类会退化为整型，因此常会与其他的枚举类型混淆。这个类型的不安全根源还是在于兼容 C 语言。在 C 中枚举用起来非常便利，在 C++ 中却是类型系统的一个大“漏勺”。因此在 C++11 中，标准引入了新的“强类型枚举”来解决这个问题。

```
enum class Color { red, blue, green };
int x = Color::red;     //C++98/03 中允许，C++11 中错误：不存在 Color->int 的转换
Color y = 7;     //C++98/03 中，C++11 中错误：不存在 int->Color conversion 的转换
Color z = red;     //C++98/03 中允许，C++11 中错误：red 不在作用域内
Color c = Color::red;     //C++98/03 中错误，C++11 中允许
```

在第 5 章中，我们会详细讲解诸如此类能够增强类型安全的 C++11 特性。

1.4.6 与硬件紧密合作

在 C++ 编程中，嵌入式编程是一个非常重要的领域。虽然一些方方圆圆的智能设备外表光鲜亮丽，但是植根于其中的技术基础也常常会是 C++。在 C++11 中，常量表达式以及原子操作都是可以用于支持嵌入式编程的重要特性。这些特性对于提高性能、降低存储空间都大有好处，比如 ROM。

C++98/03 中也具备 const 类型，不过它对只读内存（ROM）支持得不够好。这是因为在 C++ 中 const 类型只在初始化后才意味着它的值应该是常量表达式，从而在运行时不能被改变。不过由于初始化依旧是动态的，这对 ROM 设备来说并不适用。这就要求在动态初始化前就将常量计算出来。为此标准增加了 constexpr，它让函数和变量可以被编译时的常量取

代，而从效果上说，函数和变量在固定内存设备中要求的空间变得更少，因而对于手持、桌面等用于各种移动控制的小型嵌入式设备（甚至心率调整器）的 ROM 而言，C++11 也支持得更好。

在 C++11，我们甚至拥有了直接操作硬件的方法。这里指的是 C++11 中引入的原子类型。C++11 通过引入内存模型，为开发者和系统建立了一个高效的同步机制。作为开发者，通常需要保证线程程序能够正确同步，在程序中不会产生竞争。而相对地，系统（可能是编译器、内存系统，或是缓存一致性机制）则会保证程序员编写的程序（使用原子类型）不会引入数据竞争。而且为了同步，系统会自行禁止某些优化，又保证其他的一些优化有效。除非编写非常底层的并行程序，否则系统的优化对程序员来讲，基本上是透明的。这可能是 C++11 中最大、最华丽的进步。而就算程序员不乐意使用原子类型，而要使用线程，那么使用标准的互斥变量 mutex 来进行临界区的加锁和开锁也就够了。而如果读者还想要疯狂地挖掘并行的速度，或试图完全操控底层，或想找点麻烦，那么无锁（lock-free）的原子类型也可以满足你的各种“野心”。内存模型的机制会保证你不会犯错。只有在使用与系统内存单位不同的位域的时候，内存模型才无法成功地保证同步。比如说下面这个位域的例子，这样的位域常常会引发竞争（跨了一个内存单元），因为这破坏了内存模型的假定，编译器不能保证这是没有竞争的。

```
struct {int a:9; int b:7;}
```

不过如果使用下面的字符位域则不会引发竞争，因为字符位域可以被视为是独立内存位置。而在 C++98/03 中，多线程程序中该写法却通常会引发竞争。这是因为编译器可能将 a 和 b 连续存放，那么对 b 进行赋值（互斥地）的时候就有可能在 a 没有被上锁的情况下一起写掉了。原因是在单线程情况下常被视为普通的安全的优化，却没有考虑到多线程情况下的复杂性。C++11 则在这方面做出了较好的修正。

```
struct {char a; char b;}
```

与硬件紧密合作的能力使得 C++ 可以在任何系统编程中继续保持领先的位置，比如说构建设备驱动或操作系统内核，同时在一些像金融、游戏这样需要高性能后台守护进程的应用中，C++ 的参与也会大大提升其性能。

我们会在第 6 章看到相关特性的描述。

1.4.7 开发能够改变人们思维方式的特性

C++11 中一个小小的 lambda 特性是如何撬动编程世界的呢？从一方面讲，lambda 只是对 C++98/03 中带有 operator() 的局部仿函数（函数对象）包装后的“语法甜点”。事实上，在 C++11 中 lambda 也被处理为匿名的仿函数。当创建 lambda 函数的时候，编译器内部会生成这样一个仿函数，并从其父作用域中取得参数传递给 lambda 函数。不过，真正会改变人们思维方式的是，lambda 是一个局部函数，这在 C++98/03 中我们只能模仿实现该特性。

此外，当程序员开始越来越多地使用 C++11 中先进的并行编程特性时，lambda 会成为一个非常重要的语法。程序员将会发现到处都是奇怪的“lambda 笑脸”，即 ;} [⊖]，而且程序员也必须习惯在各种上下文中阅读翻译 lambda 函数。顺带一提，lambda 笑脸常会出现在每一个 lambda 表达式的终结部分。

另一个人们会改变思维方式的地方则是如何让一个成员函数变得无效。在 C++98/03 中，我们惯用的方法是将成员函数声明为私有的。如果读者不知道这种方法的用意，很可能在阅读代码的时候产生困惑。不过今天的读者非常幸运，因为在 C++11 中不再需要这样的手段。在 C++11 中我们可以通过显式默认和删除的特性，清楚明白地将成员函数设为删除的。这无疑改变了程序员编写和阅读代码的方式，当然，思考问题的方式也就更加直截了当了。

我们会在第 7 章中看到相关特性的描述。

1.4.8 融入编程现实

现实世界中的编程往往都有特殊的需求。比如在访问因特网的时候我们常常需要输入 URL，而 URL 通常都包含了斜线“/”。要在 C++ 中输入斜线却不是件容易的事，通常我们需要转义字符“\/”的配合，否则斜线则可能被误认为是除法符号。所以如果读者在写网络地址或目录路径的时候，代码最终看起来就是一堆倒胃口的反斜线的组合，而且会让内容变得晦涩。而 C++11 中的原生字符串常量则可免除“转义”的需要，也可以帮助程序员清晰地呈现网络地址或文件系统目录的真实内容。

另一方面，如今 GNU 的属性（attribute）几乎无所不在，所有的编译器都在尝试支持它，以用于修饰类型、变量和函数等。不过 __attribute__((attribute-name)) 这样的写法，除了不怎么好看外，每一个编译器可能还都有它自己的变体，比如微软的属性就是以 __declspec 打头的。因此在 C++11 中，我们看到了通用属性的出现。

不过 C++11 引入通用属性更大的原因在于，属性可以在不引入额外的关键字的情况下，为编译提供额外的信息。因此，一些可以实现为关键字的特性，也可以用属性来实现（在某些情况下，属性甚至还可以在代码中放入程序供应商的名字，不过这样做存在一些争议）。这在使用关键字还是将特性实现为一个通用属性间就会存在权衡。不过最后标准委员会认为，在现在的情况下，在 C++11 中的通用属性不能破坏已有的类型系统，也不应该在代码中引起语义的歧义。也就是说，有属性的和没有属性的代码在编译时行为是一致的。所以 C++11 标准最终选择创建很少的几个通用属性——noreturn 和 carrier_dependency（其实 final、override 也一度是热门“人选”）。

属性的真正强大之处在于它们能够让编译器供应商创建他们自己的语言扩展，同时不

⊖ lambda 笑脸是一种编写 lambda 函数的编程风格，即在 lambda 函数结束时将分号与括号连写，看起来就是一个 ;} 形式的笑脸。而实际在本书第 7 章中没有采用 lambda 笑脸的编程风格。

会干扰语言或等待特性的标准化。它们可以被用于在一些域内加入特定的“方言”，甚至是在不用 pragma 语法的情况下扩展专有的并行机制（如果读者了解 OpenMP，对此会有一些体会）。

我们将在第 8 章中看到相关的描述。

1.5 本书的约定

1.5.1 关于一些术语的翻译

在 C++11 标准中，我们会涉及很多已有的或新建的术语。在本书中，这些术语我们会尽量翻译，但不求过度翻译。

在已有翻译且翻译意义已经被广为接受的情况下，我们会使用已有的翻译词汇。比如说将 class 翻译为“类”，或者将 template 翻译为“模板”。这样翻译已经为中文读者广为接受，本书则会沿用这样的译法。

而已有翻译但是意义并没有被广为接受的情况下，本书中则会考虑保留英文原文。比如说将“URL”翻译为“统一资源定址器”在我们看来就是一种典型的不良情况。通常将这样的术语翻译为中文会阻碍读者的理解。而大多数能够阅读本书的读者也会具有基本的英文阅读能力和一些常识性的计算机知识，因此本书将保留原文，以期望能够帮助读者更好地理解涉及术语的部分。

对于还没有广泛被认同的中文翻译的术语，我们会采用审慎的态度。一些时候，如果英文确实有利于理解，我们会尝试以注释的方式提供一个中文的解释，而在文中保持英文。如果翻译成中文非常利于理解，则会提供一个中文的翻译，在注释中留下英文。

1.5.2 关于代码中的注释

在本书中，如果可能我们会将一些形如 cout、printf 打印至标准输出 / 错误的内容放在代码的注释中，从读书的经验来看，我们认为这样是最方便阅读的。比如：

```
int a = 2012;
cout << "hello, world" << endl;    // hello, world
cout << a << " is doomed" << endl; // 2012 is doomed
```

同时，一些关键的、有助于读者理解代码的解释也会放在注释中。在通常情况下，注释中有了打印结果的语句不会再有其他的代码解释。如果有，我们将会以逗号将其分开。比如：

```
cout << "hello world" << endl;    // hello world, 打印 "hello world"
```

1.5.3 关于本书中的代码示例与实验平台

在本书的编写中，我们一共使用了 3 种编译器对代码进行编译，即 IBM 的 xlC++、GNU 的 g++，以及 llvm 的 clang++。我们使用的这 3 种编译器都是开发中的版本，其中 xlC++ 使用的是开发中的版本 13，g++ 使用的是开发中的版本 4.8，而 clang++ 则使用的是开发中的版本 3.2。

本书的代码大多数由作者原创，少量使用了 C++11 标准提案中的案例，以及一些网上资源。由于本书编写时，还没有编译器提供对 C++11 所有特性的完整支持，所以通常我们都会将使用的编译器、编译时采用的编译选项罗列在代码处。在本书的代码中，我们会以 g++ 编译为主，但这并不意味着其他编译器无法编译通过这些代码示例。从我们现在看到的结果而言，使用相同特性的代码，编译器的支持往往不存在很大的个体差别（这也是设立标准的意义所在）。而具体的编译器支持，读者则可以通过附录 C 获得相关的信息。

我们的代码运行平台之一是一台运行在 IBM Power 服务器上的 SUSE Linux Enterprise Server 11 (x86_64) 的虚拟机（从我们的实验看来，在该虚拟机上并没有出现与实体机器不一致之处，而不同的 Linux 也不会对我们的实验产生影响）。运行平台之二则是一台运行于 SUSE Linux Enterprise Server 10 SP2 (ppc) 的 IBM Power5+ 服务器。

第②章

保证稳定性和兼容性

作为 C 语言的嫡亲，C++ 有一个众所周知的特性——对 C 语言的高度兼容。这样的兼容性不仅体现在程序员可以较为容易地将 C 代码“升级”为 C++ 代码上，也体现在 C 代码可以被 C++ 的编译器所编译上。新的 C++11 标准也并不例外。在 C++11 中，设计者总是保证在不破坏原有设计的情况下，增加新的特性，以充分保证语言的稳定性与兼容性。本章中的新特性基本上都遵循了该设计思想。

2.1 保持与 C99 兼容

☞类别：部分人

在 C11 之前最新的 C 标准是 1999 年制定的 C99 标准。而第一个 C++ 语言标准却出现在 1998 年（通常被称为 C++98），随后的 C++03 标准也只对 C++98 进行了小的修正。这样一来，虽然 C 语言发展中的大多数改进都被引入了 C++ 语言标准中，但还是存在着一些属于 C99 标准的“漏网之鱼”。所以 C++11 将对以下 C99 特性的支持也都纳入了新标准中：

- ❑C99 中的预定义宏
- ❑__func__ 预定义标识符
- ❑_Pragma 操作符
- ❑不定参数宏定义以及 __VA_ARGS__
- ❑宽窄字符串连接

这些特性并不像语法规则一样常用，并且有的 C++ 编译器实现也都先于标准地将这些特性实现，因此可能大多数程序员没有发现这些不兼容。但将这些 C99 的特性在 C++11 中标准化无疑可以更广泛地保证两者的兼容性。我们来分别看一下。

2.1.1 预定义宏

除去语法规范等，包括标准库的接口函数定义、相关的类型、宏、常量等也都会被发布在语言标准中。相较于 C89 标准，C99 语言标准增加一些预定义宏。C++11 同样增加了对这些宏的支持。我们可以看一下表 2-1。

表 2-1 C++11 中与 C99 兼容的宏

宏 名 称	功 能 描 述
__STDC_HOSTED__	如果编译器的目标系统环境中包含完整的标准 C 库，那么这个宏就定义为 1，否则宏的值为 0
__STDC__	C 编译器通常用这个宏的值来表示编译器的实现是否和 C 标准一致。C++11 标准中这个宏是否定义以及定成什么值由编译器来决定
__STDC_VERSION__	C 编译器通常用这个宏来表示所支持的 C 标准的版本，比如 1999mmL。C++11 标准中这个宏是否定义以及定成什么值将由编译器来决定
__STDC_ISO_10646__	这个宏通常定义为一个 yyyymmL 格式的整数常量，例如 199712L，用来表示 C++ 编译环境符合某个版本的 ISO/IEC 10646 标准

使用这些宏，我们可以查验机器环境对 C 标准和 C 库的支持状况，如代码清单 2-1 所示。

代码清单 2-1

```
#include <iostream>
using namespace std;

int main() {
    cout << "Standard Clib: " << __STDC_HOSTED__ << endl;   // Standard Clib: 1
    cout << "Standard C: " << __STDC__ << endl;             // Standard C: 1
    // cout << "C Stardard version: " << __STDC_VERSION__ << endl;
    cout << "ISO/IEC " << __STDC_ISO_10646__ << endl;       // ISO/IEC 200009
}

// 编译选项 :g++ -std=c++11 2-1-1.cpp
```

在我们的实验机上，__STDC_VERSION__ 这个宏没有定义（也是符合标准规定的，如表 2-1 所示），其余的宏都可以打印出一些常量值。

预定义宏对于多目标平台代码的编写通常具有重大意义。通过以上的宏，程序员通过使用 #ifdef/#endif 等预处理指令，就可使得平台相关代码只在适合于当前平台的代码上编译，从而在同一套代码中完成对多平台的支持。从这个意义上讲，平台信息相关的宏越丰富，代码的多平台支持越准确。

不过值得注意的是，与所有预定义宏相同的，如果用户重定义（#define）或 #undef 了预定义的宏，那么后果是“未定义”的。因此在代码编写中，程序员应该注意避免自定义宏与预定义宏同名的情况。

2.1.2 __func__ 预定义标识符

很多现实的编译器都支持 C99 标准中的 __func__ 预定义标识符功能，其基本功能就是

返回所在函数的名字。我们可以看看下面这个例子，如代码清单 2-2 所示。

代码清单 2-2

```
#include <string>
#include <iostream>
using namespace std;
const char* hello() { return __func__; }
const char* world() { return __func__; }

int main(){
    cout << hello() << ", " << world() << endl; // hello, world
}

// 编译选项 :g++ -std=c++11 2-1-2.cpp
```

在代码清单 2-2 中，我们定义了两个函数 hello 和 world。利用 __func__ 预定义标识符，我们返回了函数的名字，并将其打印出来。事实上，按照标准定义，编译器会隐式地在函数的定义之后定义 __func__ 标识符。比如上述例子中的 hello 函数，其实际的定义等同于如下代码：

```
const char* hello() {
    static const char* __func__ = "hello";
    return __func__;
}
```

__func__ 预定义标识符对于轻量级的调试代码具有十分重要的作用。而在 C++11 中，标准甚至允许其使用在类或者结构体中。我们可以看看下面这个例子，如代码清单 2-3 所示。

代码清单 2-3

```
#include <iostream>
using namespace std;

struct TestStruct {
    TestStruct () : name(__func__) {}
    const char *name;
};

int main() {
    TestStruct ts;
    cout << ts.name << endl;    // TestStruct
}
// 编译选项 :g++ -std=c++11 2-1-3.cpp
```

从代码清单 2-3 可以看到，在结构体的构造函数中，初始化成员列表使用 __func__ 预定义标识符是可行的，其效果跟在函数中使用一样。不过将 __fun__ 标识符作为函数参数的默

认值是不允许的，如下例所示：

```
void FuncFail( string func_name = __func__) {};// 无法通过编译
```

这是由于在参数声明时，__func__ 还未被定义。

2.1.3 _Pragma 操作符

在 C/C++ 标准中，#pragma 是一条预处理的指令（preprocessor directive）。简单地说，#pragma 是用来向编译器传达语言标准以外的一些信息。举个简单的例子，如果我们在代码的头文件中定义了以下语句：

```
#pragma once
```

那么该指令会指示编译器（如果编译器支持），该头文件应该只被编译一次。这与使用如下代码来定义头文件所达到的效果是一样的。

```
#ifndef THIS_HEADER
#define THIS_HEADER
// 一些头文件的定义
#endif
```

在 C++11 中，标准定义了与预处理指令 #pragma 功能相同的操作符 _Pragma。_Pragma 操作符的格式如下所示：

```
_Pragma （字符串字面量）
```

其使用方法跟 sizeof 等操作符一样，将字符串字面量作为参数写在括号内即可。那么要达到与上例 #pragma 类似的效果，则只需要如下代码即可。

```
_Pragma("once");
```

而相比预处理指令 #pragma，由于 _Pragma 是一个操作符，因此可以用在一些宏中。我们可以看看下面这个例子：

```
#define CONCAT(x) PRAGMA(concat on #x)
#define PRAGMA(x) _Pragma(#x)
CONCAT( ..\concat.dir )
```

这里，CONCAT(..\concat.dir) 最终会产生 _Pragma(concat on "..\concat.dir") 这样的效果（这里只是显示语法效果，应该没有编译器支持这样的 _Pragma 语法）。而 #pragma 则不能在宏中展开，因此从灵活性上来讲，C++11 的 _Pragma 具有更大的灵活性。

2.1.4 变长参数的宏定义以及 __VA_ARGS__

在 C99 标准中，程序员可以使用变长参数的宏定义。变长参数的宏定义是指在宏定义中参数列表的最后一个参数为省略号，而预定义宏 __VA_ARGS__ 则可以在宏定义的实现部分

替换省略号所代表的字符串。比如：

```
#define PR(...) printf(__VA_ARGS__)
```

就可以定义一个 printf 的别名 PR。事实上，变长参数宏与 printf 是一对好搭档。我们可以看如代码清单 2-4 所示的一个简单的变长参数宏的应用。

代码清单 2-4

```
#include <stdio.h>

#define LOG(...)  {\
    fprintf(stderr,"%s: Line %d:\t", __FILE__, __LINE__);\
    fprintf(stderr, __VA_ARGS__);\
    fprintf(stderr,"\n");\
}

int main() {
    int x = 3;
    // 一些代码 ...
    LOG("x = %d", x); // 2-1-5.cpp: Line 12:     x = 3
}
// 编译选项 :g++ -std=c++11 2-1-5.cpp
```

在代码清单 2-4 中，定义 LOG 宏用于记录代码位置中一些信息。程序员可以根据 stderr 产生的日志追溯到代码中产生这些记录的位置。引入这样的特性，对于轻量级调试，简单的错误输出都是具有积极意义的。

2.1.5 宽窄字符串的连接

在之前的 C++ 标准中，将窄字符串（char）转换成宽字符串（wchar_t）是未定义的行为。而在 C++11 标准中，在将窄字符串和宽字符串进行连接时，支持 C++11 标准的编译器会将窄字符串转换成宽字符串，然后再与宽字符串进行连接。

事实上，在 C++11 中，我们还定义了更多种类的字符串类型（主要是为了更好地支持 Unicode），更多详细的内容，读者可以参见 8.3 与 8.4 节。

2.2 long long 整型

☞类别：部分人

相比于 C++98 标准，C++11 整型的最大改变就是多了 long long。但事实上，long long 整型本来就离 C++ 标准很近，早在 1995 年，long long 就被提议写入 C++98 标准，却被 C++ 标准委员会拒绝了。而后来，long long 类型却进入了 C99 标准，而且也事实上也被很多编译器支持。于是辗转地，C++ 标准委员会又掉头决定将 long long 纳入 C++11 标准。

long long 整型有两种：long long 和 unsigned long long。在 C++11 中，标准要求 long long 整型可以在不同平台上有不同的长度，但至少有 64 位。我们在写常数字面量时，可以使用 LL 后缀（或是 ll）标识一个 long long 类型的字面量，而 ULL（或 ull、Ull、uLL）表示一个 unsigned long long 类型的字面量。比如：

```
long long int lli = -9000000000000000000LL;
unsigned long long int ulli = -9000000000000000000ULL;
```

就定义了一个有符号的 long long 变量 lli 和无符号的 unsigned long long 变量 ulli。事实上，在 C++11 中，还有很多与 long long 等价的类型。比如对于有符号的，下面的类型是等价的：long long、signed long long、long long int、signed long long int；而 unsigned long long 和 unsigned long long int 也是等价的。

同其他的整型一样，要了解平台上 long long 大小的方法就是查看 <climits>（或 <limits.h> 中的宏）。与 long long 整型相关的一共有 3 个：LLONG_MIN、LLONG_MAX 和 ULLONG_MIN，它们分别代表了平台上最小的 long long 值、最大的 long long 值，以及最大的 unsigned long long 值。可以看看下面这个例子，如代码清单 2-5 所示。

代码清单 2-5

```
#include <climits>
#include <cstdio>
using namespace std;

int main() {
    long long ll_min = LLONG_MIN;
    long long ll_max = LLONG_MAX;
    unsigned long long ull_max = ULLONG_MAX;

    printf("min of long long: %lld\n", ll_min); // min of long long:
    -9223372036854775808
    printf("max of long long: %lld\n", ll_max); // max of long long:
    9223372036854775807
    printf("max of unsigned long long: %llu\n", ull_max);   // max of unsigned
    long long: 18446744073709551615
}
// 编译选项 :g++ -std=c++11 2-2-1.cpp
```

在代码清单 2-5 中，将以上 3 个宏打印了出来，对于 printf 函数来说，输出有符号的 long long 类型变量可以用符号 %lld，而无符号的 unsigned long long 则可以采用 %llu。18446744073709551615 用 16 进制表示是 0xFFFFFFFFFFFFFFFF（16 个 F），可知在我们的实验机上，long long 是一个 64 位的类型。

2.3 扩展的整型

☞ 类别：部分人

程序员常会在代码中发现一些整型的名字，比如 UINT、__int16、u64、int64_t，等等。这些类型有的源自编译器的自行扩展，有的则是来自某些编程环境（比如工作在 Linux 内核代码中），不一而足。而事实上，在 C++11 中一共只定义了以下 5 种标准的有符号整型：

- ❑signed char
- ❑short int
- ❑int
- ❑long int
- ❑long long int

标准同时规定，每一种有符号整型都有一种对应的无符号整数版本，且有符号整型与其对应的无符号整型具有相同的存储空间大小。比如与 signed int 对应的无符号版本的整型是 unsigned int。

在实际的编程中，由于这 5 种基本的整型适用性有限，所以有时编译器出于需要，也会自行扩展一些整型。在 C++11 中，标准对这样的扩展做出了一些规定。具体地讲，除了标准整型（standard integer type）之外，C++11 标准允许编译器扩展自有的所谓扩展整型（extended integer type）。这些扩展整型的长度（占用内存的位数）可以比最长的标准整型（long long int，通常是一个 64 位长度的数据）还长，也可以介于两个标准整数的位数之间。比如在 128 位的架构上，编译器可以定义一个扩展整型来对应 128 位的的整数；而在一些嵌入式平台上，也可能需要扩展出 48 位的整型；不过 C++11 标准并没有对扩展出的类型的名称有任何的规定或建议，只是对扩展整型的使用规则做出了一定的限制。

简单地说，C++11 规定，扩展的整型必须和标准类型一样，有符号类型和无符号类型占用同样大小的内存空间。而由于 C/C++ 是一种弱类型语言[⊖]，当运算、传参等类型不匹配的时候，整型间会发生隐式的转换，这种过程通常被称为整型的提升（Integral promotion）。比如如下表达式：

```
(int) a + (long long)b
```

通常就会导致变量 (int)a 被提升为 long long 类型后才与 (long long)b 进行运算。而无论是扩展的整型还是标准的整型，其转化的规则会由它们的“等级”（rank）决定。而通常情况，我们认为有如下原则：

- ❑长度越大的整型等级越高，比如 long long int 的等级会高于 int。

⊖ 关于 C/C++ 是强类型语言还是弱类型语言存在一些争议，请参见 http://stackoverflow.com/questions/430182/is-c-strongly-typed。

- ❑长度相同的情况下，标准整型的等级高于扩展类型，比如 long long int 和 _int64 如果都是 64 位长度，则 long long int 类型的等级更高。
- ❑相同大小的有符号类型和无符号类型的等级相同，long long int 和 unsigned long long int 的等级就相同。

而在进行隐式的整型转换的时候，一般是按照低等级整型转换为高等级整型，有符号的转换为无符号。这种规则其实跟 C++98 的整型转换规则是一致的。

在这样的规则支持下，如果编译器定义一些自有的整型，即使这样自定义的整型由于名称并没有被标准收入，因而可移植性并不能得到保证，但至少编译器开发者和程序员不用担心自定义的扩展整型与标准整型间在使用规则上（尤其是整型提升）存在着不同的认识了。

比如在一个 128 位的构架上，编译器可以定义 _int128_t 为 128 位的有符号整型（对应的无符号类型为 _uint128_t）。于是程序员可以使用 _int128_t 类型保存形如 +92233720368547758070 的超长整数（长于 64 位的自然数）。而不用查看编译器文档我们也会知道，一旦遇到整型提升，按照上面的规则，比如 _int128_t a，与任何短于它的类型的数据 b 进行运算（比如加法）时，都会导致 b 被隐式地转换为 _int128_t 的整型，因为扩展的整型必须遵守 C++11 的规范。

2.4 宏 __cplusplus

☞ **类别：部分人**

在 C 与 C++ 混合编写的代码中，我们常常会在头文件里看到如下的声明：

```
#ifdef __cplusplus
extern "C" {
#endif
// 一些代码
#ifdef __cplusplus
}
#endif
```

这种类型的头文件可以被 #include 到 C 文件中进行编译，也可以被 #include 到 C++ 文件中进行编译。由于 extern "C" 可以抑制 C++ 对函数名、变量名等符号（symbol）进行名称重整（name mangling），因此编译出的 C 目标文件和 C++ 目标文件中的变量、函数名称等符号都是相同的（否则不相同），链接器可以可靠地对两种类型的目标文件进行链接。这样该做法成为了 C 与 C++ 混用头文件的典型做法。

鉴于以上的做法，程序员可能认为 __cplusplus 这个宏只有“被定义了”和“未定义”两种状态。事实上却并非如此，__cplusplus 这个宏通常被定义为一个整型值。而且随着标准变化，__cplusplus 宏一般会是一个比以往标准中更大的值。比如在 C++03 标准中，__cplusplus

的值被预定为 199711L，而在 C++11 标准中，宏 __cplusplus 被预定义为 201103L。这点变化可以为代码所用。比如程序员在想确定代码是使用支持 C++11 编译器进行编译时，那么可以按下面的方法进行检测：

```
#if __cplusplus < 201103L
    #error "should use C++11 implementation"
#endif
```

这里，使用了预处理指令 #error，这使得不支持 C++11 的代码编译立即报错并终止编译。读者可以使用 C++98 编译器和 C++11 的编译器分别实验一下其效果。

2.5 静态断言

☞ 类别：库作者

2.5.1 断言：运行时与预处理时

断言（assertion）是一种编程中常用的手段。在通常情况下，断言就是将一个返回值总是需要为真的判别式放在语句中，用于排除在设计的逻辑上不应该产生的情况。比如一个函数总需要输入在一定的范围内的参数，那么程序员就可以对该参数使用断言，以迫使在该参数发生异常的时候程序退出，从而避免程序陷入逻辑的混乱。

从一些意义上讲，断言并不是正常程序所必需的，不过对于程序调试来说，通常断言能够帮助程序开发者快速定位那些违反了某些前提条件的程序错误。在 C++ 中，标准在 <cassert> 或 <assert.h> 头文件中为程序员提供了 assert 宏，用于在运行时进行断言。我们可以看看下面这个例子，如代码清单 2-6 所示。

代码清单 2-6

```
#include <cassert>
using namespace std;
// 一个简单的堆内存数组分配函数
char * ArrayAlloc(int n) {
    assert(n > 0);  // 断言，n 必须大于 0
    return new char [n];
}

int main (){
    char* a = ArrayAlloc(0);
}
// 编译选项 :g++ 2-5-1.cpp
```

在代码清单 2-6 中，我们定义了一个 ArrayAlloc 函数，该函数的唯一功能就是在堆上分配字节长度为 n 的数组并返回。为了避免意外发生，函数 ArrayAlloc 对参数 n 进行了断言，

要求其大于 0。而 main 函数中对 ArrayAlloc 的使用却没有满足这个条件，那么在运行时，我们可以看到如下结果：

```
a.out: 2-5-1.cpp:6: char* ArrayAlloc(int): Assertion `n > 0' failed.
Aborted
```

在 C++ 中，程序员也可以定义宏 NDEBUG 来禁用 assert 宏。这对发布程序来说还是必要的。因为程序用户对程序退出总是敏感的，而且部分的程序错误也未必会导致程序全部功能失效。那么通过定义 NDEBUG 宏发布程序就可以尽量避免程序退出的状况。而当程序有问题时，通过没有定义宏 NDEBUG 的版本，程序员则可以比较容易地找到出问题的位置。事实上，assert 宏在 <cassert> 中的实现方式类似于下列形式：

```
#ifdef  NDEBUG
# define assert(expr)           (static_cast<void> (0))
#else
...
#endif
```

可以看到，一旦定义了 NDBUG 宏，assert 宏将被展开为一条无意义的 C 语句（通常会被编译器优化掉）。

在 2.4 节中，我们还看到了 #error 这样的预处理指令，而事实上，通过预处理指令 #if 和 #error 的配合，也可以让程序员在预处理阶段进行断言。这样的用法也是极为常见的，比如 GNU 的 cmathcalls.h 头文件中（在我们实验机上，该文件位于 /usr/include/bits/cmathcalls.h），我们会看到如下代码：

```
#ifndef _COMPLEX_H
#error "Never use <bits/cmathcalls.h> directly; include <complex.h> instead."
#endif
```

如果程序员直接包含头文件 <bits/cmathcalls.h> 并进行编译，就会引发错误。#error 指令会将后面的语句输出，从而提醒用户不要直接使用这个头文件，而应该包含头文件 <complex.h>。这样一来，通过预处理时的断言，库发布者就可以避免一些头文件的引用问题。

2.5.2 静态断言与 static_assert

通过 2.5.1 节的例子可以看到，断言 assert 宏只有在程序运行时才能起作用。而 #error 只在编译器预处理时才能起作用。有的时候，我们希望在编译时能做一些断言。比如下面这个例子，如代码清单 2-7 所示。

代码清单 2-7

```
#include <cassert>
using namespace std;
```

```
// 枚举编译器对各种特性的支持，每个枚举值占一位
enum FeatureSupports {
    C99         =   0x0001,
    ExtInt      =   0x0002,
    SAssert     =   0x0004,
    NoExcept    =   0x0008,
    SMAX        =   0x0010,
};

// 一个编译器类型，包括名称、特性支持等
struct Compiler{
    const char * name;
    int spp;    // 使用 FeatureSupports 枚举
};

int main() {
    // 检查枚举值是否完备
    assert((SMAX - 1) == (C99 | ExtInt | SAssert | NoExcept));

    Compiler a = {"abc", (C99 | SAssert)};
    // ...
    if (a.spp & C99) {
        // 一些代码 ...
    }
}
// 编译选项 :g++ 2-5-2.cpp
```

代码清单 2-7 所示的是 C 代码中常见的“按位存储属性”的例子。在该例中，我们编写了一个枚举类型 FeatureSupports，用于列举编译器对各种特性的支持。而结构体 Compiler 则包含了一个 int 类型成员 spp。由于各种特性都具有“支持”和“不支持”两种状态，所以为了节省存储空间，我们让每个 FeatureSupports 的枚举值占据一个特定的比特位置，并在使用时通过“或”运算压缩地存储在 Compiler 的 spp 成员中（即 bitset 的概念）。在使用时，则可以通过检查 spp 的某位来判断编译器对特性是否支持。

有的时候这样的枚举值会非常多，而且还会在代码维护中不断增加。那么代码编写者必须想出办法来对这些枚举进行校验，比如查验一下是否有重位等。在本例中程序员的做法是使用一个“最大枚举”SMAX，并通过比较 SMAX - 1 与所有其他枚举的或运算值来验证是否有枚举值重位。可以想象，如果 SAssert 被误定义为 0x0001，表达式 (SMAX - 1) == (C99 | ExtInt | SAssert | NoExcept) 将不再成立。

在本例中我们使用了断言 assert。但 assert 是一个运行时的断言，这意味着不运行程序我们将无法得知是否有枚举重位。在一些情况下，这是不可接受的，因为可能单次运行代码并不会调用到 assert 相关的代码路径。因此这样的校验最好是在编译时期就能完成。

在一些 C++ 的模板的编写中，我们可能也会遇到相同的情况，比如下面这个例子，如代

码清单 2-8 所示。

代码清单 2-8

```
#include <cassert>
#include <cstring>
using namespace std;

template <typename T, typename U> int bit_copy(T& a, U& b){
    assert(sizeof(b) == sizeof(a));
    memcpy(&a,&b,sizeof(b));
};

int main() {
    int a = 0x2468;
    double b;
    bit_copy(a, b);
}
// 编译选项 :g++ 2-5-3.cpp
```

代码清单 2-8 中的 assert 是要保证 a 和 b 两种类型的长度一致，这样 bit_copy 才能够保证复制操作不会遇到越界等问题。这里我们还是使用 assert 的这样的运行时断言，但如果 bit_copy 不被调用，我们将无法触发该断言。实际上，正确产生断言的时机应该在模板实例化时，即编译时期。

代码清单 2-7 和代码清单 2-8 这类问题的解决方案是进行编译时期的断言，即所谓的“静态断言”。事实上，利用语言规则实现静态断言的讨论非常多，比较典型的实现是开源库 Boost 内置的 BOOST_STATIC_ASSERT 断言机制（利用 sizeof 操作符）。我们可以利用“除 0”会导致编译器报错这个特性来实现静态断言。

```
#define assert_static(e) \
    do { \
        enum { assert_static__ = 1/(e) }; \
    } while (0)
```

在理解这段代码时，读者可以忽略 do while 循环以及 enum 这些语法上的技巧。真正起作用的只是 1/(e) 这个表达式。把它应用到代码清单 2-8 中，就会得到代码清单 2-9。

代码清单 2-9

```
#include <cstring>
using namespace std;

#define assert_static(e) \
    do { \
        enum { assert_static__ = 1/(e) }; \
    } while (0)
```

```
template <typename T, typename U> int bit_copy(T& a, U& b){
    assert_static(sizeof(b) == sizeof(a));
    memcpy(&a,&b,sizeof(b));
};

int main() {
    int a = 0x2468;
    double b;
    bit_copy(a, b);
}
// 编译选项 :g++ -std=c++11 2-5-4.cpp
```

结果如我们预期的，在模板实例化时我们会得到编译器的错误报告，读者可以实验一下在自己本机运行的结果。在我们的实验机上会输出比较长的错误信息，主要信息是除零错误。当然，读者也可以尝试一下 Boost 库内置的 BOOST_STATIC_ASSERT，输出的主要信息是 sizeof 错误。但无论是哪种方式的静态断言，其缺陷都是很明显的：诊断信息不够充分，不熟悉该静态断言实现的程序员可能一时无法将错误对应到断言错误上，从而难以准确定位错误的根源。

在 C++11 标准中，引入了 static_assert 断言来解决这个问题。static_assert 使用起来非常简单，它接收两个参数，一个是断言表达式，这个表达式通常需要返回一个 bool 值；一个则是警告信息，它通常也就是一段字符串。我们可以用 static_assert 替换一下代码清单 2-9 中 bit_copy 的声明。

```
template <typename t, typename u> int bit_copy(t& a, u& b){
     static_assert(sizeof(b) == sizeof(a),"the parameters of bit_copy must have
     same width.");
};
```

那么再次编译代码清单 2-9 的时候，我们就会得到如下信息：

```
error: static assertion failed: "the parameters of bit_copy should have same width."
```

这样的错误信息就非常清楚，也非常有利于程序员排错。而由于 static_assert 是编译时期的断言，其使用范围不像 assert 一样受到限制。在通常情况下，static_assert 可以用于任何名字空间，如代码清单 2-10 所示。

代码清单 2-10

```
static_assert(sizeof(int) == 8, "This 64-bit machine should follow this!");
int main() { return 0; }
// 编译选项 :g++ -std=c++11 2-5-5.cpp
```

而在 C++ 中，函数则不可能像代码清单 2-10 中的 static_assert 这样独立于任何调用之外运行。因此将 static_assert 写在函数体外通常是较好的选择，这让代码阅读者可以较容易发

现 static_assert 为断言而非用户定义的函数。而反过来讲，必须注意的是，static_assert 的断言表达式的结果必须是在编译时期可以计算的表达式，即必须是常量表达式。如果读者使用了变量，则会导致错误，如代码清单 2-11 所示。

代码清单 2-11

```
int positive(const int n) {
    static_assert(n > 0, "value must >0");
}
// 编译选项 :g++ -std=c++11 -c 2-5-6.cpp
```

代码清单 2-11 使用了参数变量 n（虽然是个 const 参数），因而 static_assert 无法通过编译。对于此例，如果程序员需要的只是运行时的检查，那么还是应该使用 assert 宏。

2.6 noexcept 修饰符与 noexcept 操作符

☞ **类别：库作者**

相比于断言适用于排除逻辑上不可能存在的状态，异常通常是用于逻辑上可能发生的错误。在 C++98 中，我们看到了一套完整的不同于 C 的异常处理系统。通过这套异常处理系统，C++ 拥有了远比 C 强大的异常处理功能。

在异常处理的代码中，程序员有可能看到过如下的异常声明表达形式：

```
void excpt_func() throw(int, double) { ... }
```

在 excpt_func 函数声明之后，我们定义了一个动态异常声明 throw(int, double)，该声明指出了 excpt_func 可能抛出的异常的类型。事实上，该特性很少被使用，因此在 C++11 中被弃用了（参见附录 B），而表示函数不会抛出异常的动态异常声明 throw() 也被新的 noexcept 异常声明所取代。

noexcept 形如其名地，表示其修饰的函数不会抛出异常。不过与 throw() 动态异常声明不同的是，在 C++11 中如果 noexcept 修饰的函数抛出了异常，编译器可以选择直接调用 std::terminate() 函数来终止程序的运行，这比基于异常机制的 throw() 在效率上会高一些。这是因为异常机制会带来一些额外开销，比如函数抛出异常，会导致函数栈被依次地展开（unwind），并依帧调用在本帧中已构造的自动变量的析构函数等。

从语法上讲，noexcept 修饰符有两种形式，一种就是简单地在函数声明后加上 noexcept 关键字。比如：

```
void excpt_func() noexcept;
```

另外一种则可以接受一个常量表达式作为参数，如下所示：

```
void excpt_func() noexcept （常量表达式）;
```

常量表达式的结果会被转换成一个 bool 类型的值。该值为 true，表示函数不会抛出异常，反之，则有可能抛出异常。这里，不带常量表达式的 noexcept 相当于声明了 noexcept(true)，即不会抛出异常。

在通常情况下，在 C++11 中使用 noexcept 可以有效地阻止异常的传播与扩散。我们可以看看下面这个例子，如代码清单 2-12 所示。

代码清单 2-12

```
#include <iostream>
using namespace std;
void Throw() { throw 1; }
void NoBlockThrow() { Throw(); }
void BlockThrow() noexcept { Throw(); }

int main() {
    try {
        Throw();
    }
    catch(...) {
        cout << "Found throw." << endl;      // Found throw.
    }

    try {
        NoBlockThrow();
    }
    catch(...) {
        cout << "Throw is not blocked." << endl;     // Throw is not blocked.
    }

    try {
        BlockThrow();   // terminate called after throwing an instance of 'int'
    }
    catch(...) {
        cout << "Found throw 1." << endl;
    }
}
// 编译选项 :g++ -std=c++11 2-6-1.cpp
```

在代码清单 2-12 中，我们定义了 Throw 函数，该函数的唯一作用是抛出一个异常。而 NoBlockThrow 是一个调用 Throw 的普通函数，BlockThrow 则是一个 noexcept 修饰的函数。从 main 的运行中我们可以看到，NoBlockThrow 会让 Throw 函数抛出的异常继续抛出，直到 main 中的 catch 语句将其捕捉。而 BlockThrow 则会直接调用 std::terminate 中断程序的执行，从而阻止了异常的继续传播。从使用效果上看，这与 C++98 中的 throw() 是一样的。

而 noexcept 作为一个操作符时，通常可以用于模板。比如：

```
template <class T>
  void fun() noexcept(noexcept(T())) {}
```

这里，fun 函数是否是一个 noexcept 的函数，将由 T() 表达式是否会抛出异常所决定。这里的第二个 noexcept 就是一个 noexcept 操作符。当其参数是一个有可能抛出异常的表达式的时候，其返回值为 false，反之为 true（实际 noexcept 参数返回 false 还包括一些情况，这里就不展开讲了）。这样一来，我们就可以使模板函数根据条件实现 noexcept 修饰的版本或无 noexcept 修饰的版本。从泛型编程的角度看来，这样的设计保证了关于“函数是否抛出异常”这样的问题可以通过表达式进行推导。因此这也可以视作 C++11 为了更好地支持泛型编程而引入的特性。

虽然 noexcept 修饰的函数通过 std::terminate 的调用来结束程序的执行的方式可能会带来很多问题，比如无法保证对象的析构函数的正常调用，无法保证栈的自动释放等，但很多时候，“暴力”地终止整个程序确实是很简单有效的做法。事实上，noexcept 被广泛地、系统地应用在 C++11 的标准库中，用于提高标准库的性能，以及满足一些阻止异常扩散的需求。

比如在 C++98 中，存在着使用 throw() 来声明不抛出异常的函数。

```
template<class T> class A {
  public:
    static constexpr T min() throw() { return T(); }
    static constexpr T max() throw() { return T(); }
    static constexpr T lowest() throw() { return T(); }
...
```

而在 C++11 中，则使用 noexcept 来替换 throw()。

```
template<class T> class A {
  public:
    static constexpr T min() noexcept { return T(); }
    static constexpr T max() noexcept  { return T(); }
    static constexpr T lowest() noexcept { return T(); }
...
```

又比如，在 C++98 中，new 可能会包含一些抛出的 std::bad_alloc 异常。

```
void* operator new(std::size_t) throw(std::bad_alloc);
void* operator new[](std::size_t) throw(std::bad_alloc);
```

而在 C++11 中，则使用 noexcept(false) 来进行替代。

```
void* operator new(std::size_t) noexcept(false);
void* operator new[](std::size_t) noexcept(false);
```

当然，noexcept 更大的作用是保证应用程序的安全。比如一个类析构函数不应该抛出异常，那么对于常被析构函数调用的 delete 函数来说，C++11 默认将 delete 函数设置成 noexcept，就可以提高应用程序的安全性。

```
void operator delete(void*) noexcept;
void operator delete[](void*) noexcept;
```

而同样出于安全考虑，C++11 标准中让类的析构函数默认也是 noexcept(true) 的。当然，如果程序员显式地为析构函数指定了 noexcept，或者类的基类或成员有 noexcept(false) 的析构函数，析构函数就不会再保持默认值。我们可以看看下面的例子，如代码清单 2-13 所示。

代码清单 2-13

```
#include <iostream>
using namespace std;

struct A {
    ~A() { throw 1; }
};

struct B {
    ~B() noexcept(false) { throw 2; }
};

struct C {
    B b;
};

int funA() { A a; }
int funB() { B b; }
int funC() { C c; }

int main() {
    try {
        funB();
    }
    catch(...){
        cout << "caught funB." << endl; // caught funB.
    }

    try {
        funC();
    }
    catch(...){
        cout << "caught funC." << endl; // caught funC.
    }

    try {
        funA(); // terminate called after throwing an instance of 'int'
    }
    catch(...){
        cout << "caught funA." << endl;
    }
}
// 编译选项 :g++ -std=c++11 2-6-2.cpp
```

在代码清单 2-13 中，无论是析构函数声明为 noexcept(false) 的类 B，还是包含了 B 类型成员的类 C，其析构函数都是可以抛出异常的。只有什么都没有声明的类 A，其析构函数被默认为 noexcept(true)，从而阻止了异常的扩散。这在实际的使用中，应该引起程序员的注意。

2.7 快速初始化成员变量

☞ **类别：部分人**

在 C++98 中，支持了在类声明中使用等号“=”加初始值的方式，来初始化类中静态成员常量。这种声明方式我们也称之为“就地”声明。就地声明在代码编写时非常便利，不过 C++98 对类中就地声明的要求却非常高。如果静态成员不满足常量性，则不可以就地声明，而且即使常量的静态成员也只能是整型或者枚举型才能就地初始化。而非静态成员变量的初始化则必须在构造函数中进行。我们来看看下面的例子，如代码清单 2-14 所示。

代码清单 2-14

```
class Init{
public:
    Init(): a(0){}
    Init(int d): a(d){}
private:
    int a;
    const static int b = 0;
    int c = 1;          // 成员，无法通过编译
    static int d = 0;   // 成员，无法通过编译
    static const double e = 1.3;          // 非整型或者枚举，无法通过编译
    static const char * const f = "e";  // 非整型或者枚举，无法通过编译
};
// 编译选项 :g++ -c 2-7-1.cpp
```

在代码清单 2-14 中，成员 c、静态成员 d、静态常量成员 e 以及静态常量指针 f 的就地初始化都无法通过编译（这里，使用 g++ 的读者可能发现就地初始化 double 类型静态常量 e 是可以通过编译的，不过这实际是 GNU 对 C++ 的一个扩展，并不遵从 C++ 标准）。在 C++11 中，标准允许非静态成员变量的初始化有多种形式。具体而言，除了初始化列表外，在 C++11 中，标准还允许使用等号 = 或者花括号 {} 进行就地的非静态成员变量初始化。比如：

```
struct init{ int a = 1; double b {1.2}; };
```

在这个名叫 init 的结构体中，我们给了非静态成员 a 和 b 分别赋予初值 1 和 1.2。这在 C++11 中是一个合法的结构体声明。虽然这里采用的一对花括号 {} 的初始化方法读者第一次见到，不过在第 3 章中，读者会在 C++ 对于初始化表达式的改动发现，花括号式的集合

（列表）初始化已经成为 C++11 中初始化声明的一种通用形式，而其效果类似于 C++98 中使用圆括号 () 对自定义变量的表达式列表初始化。不过在 C++11 中，对于非静态成员进行就地初始化，两者却并非等价的，如代码清单 2-15 所示。

代码清单 2-15

```
#include <string>
using namespace std;

struct C {
    C(int i):c(i){};
    int c;
};

struct init {
    int a = 1;
    string b("hello");  // 无法通过编译
    C c(1);             // 无法通过编译
};
// 编译选项 :g++ -std=c++11 -c 2-7-2.cpp
```

从代码清单 2-15 中可以看到，就地圆括号式的表达式列表初始化非静态成员 b 和 c 都会导致编译出错。

在 C++11 标准支持了就地初始化非静态成员的同时，初始化列表这个手段也被保留下来了。如果两者都使用，是否会发生冲突呢？我们来看下面这个例子，如代码清单 2-16 所示。

代码清单 2-16

```
#include <iostream>
using namespace std;

struct Mem {
    Mem() { cout << "Mem default, num: " << num << endl; }
    Mem(int i): num(i) { cout << "Mem, num: " << num << endl; }

    int num = 2;  // 使用 = 初始化非静态成员
};

class Group {
public:
    Group() { cout << "Group default. val: " << val << endl; }
    Group(int i): val('G'), a(i) { cout << "Group. val: " << val << endl; }
    void NumOfA() { cout << "number of A: " << a.num << endl; }
    void NumOfB() { cout << "number of B: " << b.num << endl; }

private:
    char val{'g'};  // 使用 {} 初始化非静态成员
```

```
    Mem a;
    Mem b{19};       // 使用 {} 初始化非静态成员
};

int main() {
    Mem member;          // Mem default, num: 2

    Group group;         // Mem default, num: 2
                         // Mem, num: 19
                         // Group default. val: g

    group.NumOfA();      // number of A: 2
    group.NumOfB();      // number of B: 19

    Group group2(7);     // Mem, num: 7
                         // Mem, num: 19
                         // Group. val: G

    group2.NumOfA();     // number of A: 7
    group2.NumOfB();     // number of B: 19
}
// 编译选项 :g++ 2-7-3.cpp -std=c++11
```

在代码清单 2-16 中，我们定义了有两个初始化函数的类 Mem，此外还定义了包含两个 Mem 对象的 Group 类。类 Mem 中的成员变量 num，以及 class Group 中的成员变量 a、b、val，采用了与 C++98 完全不同的初始化方式。读者可以从 main 函数的打印输出中看到，就地初始化和初始化列表并不冲突。程序员可以为同一成员变量既声明就地的列表初始化，又在初始化列表中进行初始化，只不过初始化列表总是看起来“后作用于”非静态成员。也就是说，初始化列表的效果总是优先于就地初始化的。

相对于传统的初始化列表，在类声明中对非静态成员变量进行就地列表初始化可以降低程序员的工作量。当然，我们只在有多个构造函数，且有多个成员变量的时候可以看到新方式带来的便利。我们来看看下面的例子，如代码清单 2-17 所示。

代码清单 2-17

```
#include <string>
using namespace std;

class Mem {
public:
    Mem(int i): m(i){}

private:
    int m;
};
```

```
class Group {
public:
    Group(){}                       // 这里就不需要初始化 data、mem、name 成员了
    Group(int a): data(a) {}        // 这里就不需要初始化 mem、name 成员了
    Group(Mem m) : mem(m) {}        // 这里就不需要初始化 data、name 成员了
    Group(int a, Mem m, string n): data(a), mem(m), name(n){}

private:
    int data = 1;
    Mem mem{0};
    string name{"Group"};
};
// 编译选项 :g++ 2-7-4.cpp -std=c++11 -c
```

在代码清单 2-17 中，Group 有 4 个构造函数。可以想象，如果我们使用的是 C++98 的编译器，我们不得不在 Group()、Group(int a)，以及 Group(Mem m) 这 3 个构造函数中将 data、mem、name 这 3 个成员都写进初始化列表。但如果使用的是 C++11 的编译器，那么通过对非静态成员变量的就地初始化，我们就可以避免重复地在初始化列表中写上每个非静态成员了（在 C++98 中，我们还可以通过调用公共的初始化函数来达到类似的目的，不过目前在书写的复杂性及效率性上远低于 C++11 改进后的做法）。

此外，值得注意的是，对于非常量的静态成员变量，C++11 则与 C++98 保持了一致。程序员还是需要到头文件以外去定义它，这会保证编译时，类静态成员的定义最后只存在于一个目标文件中。不过对于静态常量成员，除了 const 关键字外，在本书第 6 章中我们会看到还可以使用 constexpr 来对静态常量成员进行声明。

2.8 非静态成员的 sizeof

☞ **类别：部分人**

从 C 语言被发明开始，sizeof 就是一个运算符，也是 C 语言中除了加减乘除以外为数不多的特殊运算符之一。而在 C++ 引入类（class）类型之后，sizeof 的定义也随之进行了拓展。不过在 C++98 标准中，对非静态成员变量使用 sizeof 是不能够通过编译的。我们可以看看下面的例子，如代码清单 2-18 所示。

代码清单 2-18

```
#include <iostream>
using namespace std;

struct People {
public:
    int hand;
    static People * all;
```

```
};

int main() {
    People p;
    cout << sizeof(p.hand) << endl;          // C++98 中通过，C++11 中通过
    cout << sizeof(People::all) << endl;     // C++98 中通过，C++11 中通过
    cout << sizeof(People::hand) << endl;    // C++98 中错误，C++11 中通过
}
// 编译选项 :g++ 2-8-1.cpp
```

注意最后一个 sizeof 操作。在 C++11 中，对非静态成员变量使用 sizeof 操作是合法的。而在 C++98 中，只有静态成员，或者对象的实例才能对其成员进行 sizeof 操作。因此如果读者只有一个支持 C++98 标准的编译器，在没有定义类实例的时候，要获得类成员的大小，我们通常会采用以下的代码：

```
sizeof(((People*)0)->hand);
```

这里我们强制转换 0 为一个 People 类的指针，继而通过指针的解引用获得其成员变量，并用 sizeof 求得该成员变量的大小。而在 C++11 中，我们无需这样的技巧，因为 sizeof 可以作用的表达式包括了类成员表达式。

```
sizeof(People::hand);
```

可以看到，无论从代码的可读性还是编写的便利性，C++11 的规则都比强制指针转换的方案更胜一筹。

2.9 扩展的 friend 语法

☞ **类别：部分人**

friend 关键字在 C++ 中是一个比较特别的存在。因为我们常常会发现，一些面向对象程序语言，比如 Java，就没有定义 friend 关键字。friend 关键字用于声明类的友元，友元可以无视类中成员的属性。无论成员是 public、protected 或是 private 的，友元类或友元函数都可以访问，这就完全破坏了面向对象编程中封装性的概念。因此，使用 friend 关键字充满了争议性。在通常情况下，面向对象程序开发的专家会建议程序员使用 Get/Set 接口来访问类的成员，但有的时候，friend 关键字确实会让程序员少写很多代码。因此即使存在争论，friend 还是在很多程序中被使用到。而 C++11 对 friend 关键字进行了一些改进，以保证其更加好用。我们可以看看下面的例子，如代码清单 2-19 所示。

代码清单 2-19

```
class Poly;
typedef Poly P;

class LiLei {
```

```
    friend class Poly;  // C++98 通过，C++11 通过
};

class Jim {
    friend Poly;        // C++98 失败，C++11 通过
};

class HanMeiMei {
    friend P;           // C++98 失败，C++11 通过
};
// 编译选项 :g++ -std=c++11 2-9-1.cpp
```

在代码清单 2-19 中，我们声明了 3 个类型：LiLei、Jim 和 HanMeiMei，它们都有一个友元类型 Poly。从编译通过与否的状况中我们可以看出，在 C++11 中，声明一个类为另外一个类的友元时，不再需要使用 class 关键字。本例中的 Jim 和 HanMeiMei 就是这样一种情况，在 HanMeiMei 的声明中，我们甚至还使用了 Poly 的别名 P，这同样是可行的。

虽然在 C++11 中这是一个小的改进，却会带来一点应用的变化——程序员可以为类模板声明友元了。这在 C++98 中是无法做到的。比如下面这个例子，如代码清单 2-20 所示。

代码清单 2-20

```
class P;

template <typename T> class People {
    friend T;
};

People<P> PP;   // 类型 P 在这里是 People 类型的友元
People<int> Pi; // 对于 int 类型模板参数，友元声明被忽略
// 编译选项 :g++ -std=c++11 2-9-2.cpp
```

从代码清单 2-20 中我们看到，对于 People 这个模板类，在使用类 P 为模板参数时，P 是 People<P> 的一个 friend 类。而在使用内置类型 int 作为模板参数的时候，People<int> 会被实例化为一个普通的没有友元定义的类型。这样一来，我们就可以在模板实例化时才确定一个模板类是否有友元，以及谁是这个模板类的友元。这是一个非常有趣的小特性，在编写一些测试用例的时候，使用该特性是很有好处的。我们看看下面的例子，该例子源自一个实际的测试用例，如代码清单 2-21 所示。

代码清单 2-21

```
// 为了方便测试，进行了危险的定义
#ifdef UNIT_TEST
#define private public
#endif
class Defender {
```

```
public:
    void Defence(int x, int y){}
    void Tackle(int x, int y){}

private:
    int pos_x = 15;
    int pos_y = 0;
    int speed = 2;
    int stamina = 120;
};

class Attacker {
public:
    void Move(int x, int y){}
    void SpeedUp(float ratio){}

private:
    int pos_x = 0;
    int pos_y = -30;
    int speed = 3;
    int stamina = 100;
};

#ifdef UNIT_TEST
class Validator {
public:
    void Validate(int x, int y, Defender & d){}
    void Validate(int x, int y, Attacker & a){}
};

int main() {
    Defender d;
    Attacker a;
    a.Move(15, 30);
    d.Defence(15, 30);
    a.SpeedUp(1.5f);
    d.Defence(15, 30);
    Validator v;
    v.Validate(7, 0, d);
    v.Validate(1, -10, a);
    return 0;
}
#endif
// 编译选项 :g++ 2-9-3.cpp -std=c++11 -DUNIT_TEST
```

在代码清单 2-21 所示的这个例子中，测试人员的目的是在一系列函数调用后，检查 Attacker 类变量 a 和 Defender 类变量 d 中成员变量的值是否符合预期。这里，按照封装的思想，所有成员变量被声明为 private 的。但 Attacker 和 Defender 的开发者为了方便，并没有为每个成员写 Get 函数，也没有为 Attacker 和 Defender 增加友元定义。而测试人员为了能够

快速写出测试程序，采用了比较危险的做法，即使用宏将 private 关键字统一替换为 public 关键字。这样一来，类中的 private 成员就都成了 public 的。这样的做法存在 4 个缺点：一是如果侥幸程序中没有变量包含 private 字符串，该方法可以正常工作，但反之，则有可能导致严重的编译时错误；二是这种做法会降低代码可读性，因为改变了一个常见关键字的意义，没有注意到这个宏的程序员可能会非常迷惑程序的行为；三是如果在类的成员定义时不指定关键字（如 public、private、protect 等），而使用默认的 private 成员访问限制，那么该方法也不能达到目的；四则很简单，这样的做法看起来也并不漂亮。

不过由于有了扩展的 friend 声明，在 C++11 中，我们可以将 Defender 和 Attacker 类改良一下。我们看看下面的例子，如代码清单 2-22 所示。

代码清单 2-22

```
template <typename T> class DefenderT {
public:
    friend T;
    void Defence(int x, int y){}
    void Tackle(int x, int y){}

private:
    int pos_x = 15;
    int pos_y = 0;
    int speed = 2;
    int stamina = 120;
};

template <typename T> class AttackerT {
public:
    friend T;
    void Move(int x, int y){}
    void SpeedUp(float ratio){}

private:
    int pos_x = 0;
    int pos_y = -30;
    int speed = 3;
    int stamina = 100;
};

using Defender = DefenderT<int>;     // 普通的类定义，使用 int 做参数
using Attacker = AttackerT<int>;

#ifdef UNIT_TEST
class Validator {
public:
    void Validate(int x, int y, DefenderTest & d){}
    void Validate(int x, int y, AttackerTest & a){}
```

```
};

using DefenderTest = DefenderT<Validator>;  // 测试专用的定义，Validator 类成为友元
using AttackerTest = AttackerT<Validator>;

int main() {
    DefenderTest d;
    AttackerTest a;
    a.Move(15, 30);
    d.Defence(15, 30);
    a.SpeedUp(1.5f);
    d.Defence(15, 30);
    Validator v;
    v.Validate(7, 0, d);
    v.Validate(1, -10, a);
    return 0;
}
#endif
// 编译选项 :g++ 2-9-4.cpp -std=c++11 -DUNIT_TEST
```

在代码清单 2-22 中，我们把原有的 Defender 和 Attacker 类定义为模板类 DefenderT 和 AttackerT。而在需要进行测试的时候，我们使用 Validator 为模板参数，实例化出 DefenderTest 及 AttackerTest 版本的类，这个版本的特点是，Validator 是它们的友元，可以任意访问任何成员函数。而另外一个版本则是使用 int 类型进行实例化的 Defender 和 Attacker，按照 C++11 的定义，它们不会有友元。因此这个版本保持了良好的封装性，可以用于提供接口用于常规使用。

值得注意的是，在代码清单 2-22 中，我们使用了 using 来定义类型的别名，这跟使用 typedef 的定义类型的别名是完全一样的。使用 using 定义类型别名是 C++11 中的一个新特性，我们可以在 3.10 节中看到相关的描述。

2.10 final/override 控制

☞ **类别：部分人**

在了解 C++11 中的 final/override 关键字之前，我们先回顾一下 C++ 关于重载的概念。简单地说，一个类 A 中声明的虚函数 fun 在其派生类 B 中再次被定义，且 B 中的函数 fun 跟 A 中 fun 的原型一样（函数名、参数列表等一样），那么我们就称 B 重载（overload）了 A 的 fun 函数。对于任何 B 类型的变量，调用成员函数 fun 都是调用了 B 重载的版本。而如果同时有 A 的派生类 C，却并没有重载 A 的 fun 函数，那么调用成员函数 fun 则会调用 A 中的版本。这在 C++ 中就实现多态。

在通常情况下，一旦在基类 A 中的成员函数 fun 被声明为 virtual 的，那么对于其派生类

B 而言，fun 总是能够被重载的（除非被重写了）。有的时候我们并不想 fun 在 B 类型派生类中被重载，那么，C++98 没有方法对此进行限制。我们看看下面这个具体的例子，如代码清单 2-23 所示。

代码清单 2-23

```
#include <iostream>
using namespace std;

class MathObject{
public:
    virtual double Arith() = 0;
    virtual void Print() = 0;
};

class Printable : public MathObject{
public:
   double Arith() = 0;
   void Print() // 在 C++98 中我们无法阻止该接口被重写
   {
       cout << "Output is: " << Arith() << endl;
   }
};

class Add2 : public Printable {
public:
    Add2(double a, double b): x(a), y(b) {}
    double Arith() { return x + y; }
private:
    double x, y;
};

class Mul3 : public Printable {
public:
    Mul3(double a, double b, double c): x(a), y(b), z(c) {}
    double Arith() { return x * y * z; }
private:
    double x, y, z;
};
// 编译选项 :g++ 2-10-1.cpp
```

在代码清单 2-23 中，我们的基础类 MathObject 定义了两个接口：Arith 和 Print。类 Printable 则继承于 MathObject 并实现了 Print 接口。接下来，Add2 和 Mul3 为了使用 MathObject 的接口和 Printable 的 Print 的实现，于是都继承了 Printable。这样的类派生结构，在面向对象的编程中非常典型。不过倘若这里的 Printable 和 Add2 是由两个程序员完成的，Printable 的编写者不禁会有一些忧虑，如果 Add2 的编写者重载了 Print 函数，那么他所期望的统一风格的打印方式将不复存在。

对于 Java 这种所有类型派生于单一元类型（Object）的语言来说，这种问题早就出现了。因此 Java 语言使用了 final 关键字来阻止函数继续重写。final 关键字的作用是使派生类不可覆盖它所修饰的虚函数。C++11 也采用了类似的做法，如代码清单 2-24 所示的例子。

代码清单 2-24

```
struct Object{
    virtual void fun() = 0;
};

struct Base : public Object {
    void fun() final;    // 声明为 final
};

struct Derived : public Base {
    void fun();      // 无法通过编译
};
// 编译选项 :g++ -c -std=c++11 2-10-2.cpp
```

在代码清单 2-24 中，派生于 Object 的 Base 类重载了 Object 的 fun 接口，并将本类中的 fun 函数声明为 final 的。那么派生于 Base 的 Derived 类对接口 fun 的重载则会导致编译时的错误。同理，在代码清单 2-23 中，Printable 的编写者如果要阻止派生类重载 Print 函数，只需要在定义时使用 final 进行修饰就可以了。

读者可能注意到了，在代码清单 2-23 及代码清单 2-24 两个例子当中，final 关键字都是用于描述一个派生类的。那么基类中的虚函数是否可以使用 final 关键字呢？答案是肯定的，不过这样将使用该虚函数无法被重载，也就失去了虚函数的意义。如果不想成员函数被重载，程序员可以直接将该成员函数定义为非虚的。而 final 通常只在继承关系的“中途”终止派生类的重载中有意义。从接口派生的角度而言，final 可以在派生过程中任意地阻止一个接口的可重载性，这就给面向对象的程序员带来了更大的控制力。

在 C++ 中重载还有一个特点，就是对于基类声明为 virtual 的函数，之后的重载版本都不需要再声明该重载函数为 virtual。即使在派生类中声明了 virtual，该关键字也是编译器可以忽略的。这带来了一些书写上的便利，却带来了一些阅读上的困难。比如代码清单 2-23 中的 Printable 的 Print 函数，程序员无法从 Printable 的定义中看出 Print 是一个虚函数还是非虚函数。另外一点就是，在 C++ 中有的虚函数会“跨层”，没有在父类中声明的接口有可能是祖先的虚函数接口。比如在代码清单 2-23 中，如果 Printable 不声明 Arith 函数，其接口在 Add2 和 Mul3 中依然是可重载的，这同样是在父类中无法读到的信息。这样一来，如果类的继承结构比较长（不断地派生）或者比较复杂（比如偶尔多重继承），派生类的编写者会遇到信息分散、难以阅读的问题（虽然有时候编辑器会进行提示，不过编辑器不是总是那么有效）。而自己是否在重载一个接口，以及自己重载的接口的名字是否有拼写错误等，都非常

不容易检查。

在 C++11 中为了帮助程序员写继承结构复杂的类型，引入了虚函数描述符 override，如果派生类在虚函数声明时使用了 override 描述符，那么该函数必须重载其基类中的同名函数，否则代码将无法通过编译。我们来看一下如代码清单 2-25 所示的这个简单的例子。

代码清单 2-25

```
struct Base {
    virtual void Turing() = 0;
    virtual void Dijkstra() = 0;
    virtual void VNeumann(int g) = 0;
    virtual void DKnuth() const;
    void Print();
};

struct DerivedMid: public Base {
    // void VNeumann(double g);
    // 接口被隔离了，曾想多一个版本的 VNeumann 函数
};

struct DerivedTop : public DerivedMid {
    void Turing() override;
    void Dikjstra() override;           // 无法通过编译，拼写错误，并非重载
    void VNeumann(double g) override;   // 无法通过编译，参数不一致，并非重载
    void DKnuth() override;             // 无法通过编译，常量性不一致，并非重载
    void Print() override;              // 无法通过编译，非虚函数重载
};
// 编译选项 :g++ -c -std=c++11 2-10-3.cpp
```

在代码清单 2-25 中，我们在基类 Base 中定义了一些 virtual 的函数（接口）以及一个非 virtual 的函数 Print。其派生类 DerivedMid 中，基类的 Base 的接口都没有重载，不过通过注释可以发现，DerivedMid 的作者曾经想要重载出一个“void VNeumann(double g)”的版本。这行注释显然迷惑了编写 DerivedTop 的程序员，所以 DerivedTop 的作者在重载所有 Base 类的接口的时候，犯下了 3 种不同的错误：

- ❑函数名拼写错，Dijkstra 误写作了 Dikjstra。
- ❑函数原型不匹配，VNeumann 函数的参数类型误做了 double 类型，而 DKnuth 的常量性在派生类中被取消了。
- ❑重写了非虚函数 Print。

如果没有 override 修饰符，DerivedTop 的作者可能在编译后都没有意识到自己犯了这么多错误。因为编译器对以上 3 种错误不会有任何的警示。这里 override 修饰符则可以保证编译器辅助地做一些检查。我们可以看到，在代码清单 2-25 中，DerivedTop 作者的 4 处错误都无法通过编译。

此外，值得指出的是，在 C++ 中，如果一个派生类的编写者自认为新写了一个接口，而实际上却重载了一个底层的接口（一些简单的名字如 get、set、print 就容易出现这样的状况），出现这种情况编译器还是爱莫能助的。不过这样无意中的重载一般不会带来太大的问题，因为派生类的变量如果调用了该接口，除了可能存在的一些虚函数开销外，仍然会执行派生类的版本。因此编译器也就没有必要提供检查“非重载”的状况。而检查“一定重载”的 override 关键字，对程序员的实际应用则会更有意义。

还有值得注意的是，如我们在第 1 章中提到的，final/override 也可以定义为正常变量名，只有在其出现在函数后时才是能够控制继承 / 派生的关键字。通过这样的设计，很多含有 final/override 变量或者函数名的 C++98 代码就能够被 C++ 编译器编译通过了。但出于安全考虑，建议读者在 C++11 代码中应该尽可能地避免这样的变量名称或将其定义在宏中，以防发生不必要的错误。

2.11 模板函数的默认模板参数

☞ 类别：所有人

在 C++11 中模板和函数一样，可以有默认的参数。这就带来了一定的复杂性。我可以通过代码清单 2-26 所示的这个简单的模板函数的例子来回顾一下函数模板的定义。

代码清单 2-26

```
#include <iostream>
using namespace std;

// 定义一个函数模板
template <typename T> void TempFun(T a) {
    cout << a << endl;
}

int main() {
    TempFun(1);      // 1, ( 实例化为 TempFun<const int>(1))
    TempFun("1");    // 1, ( 实例化为 TempFun<const char *>("1"))
}
// 编译选项 :g++ 2-11-1.cpp
```

在代码清单 2-26 中，当编译器解析到函数调用 fun(1) 的时候，发现 fun 是一个函数模板。这时候编译器就会根据实参 1 的类型 const int 推导实例化出模板函数 void TempFun<const int>(int)，再进行调用。相应的，对于 fun("1") 来说也是类似的，不过编译器实例化出的模板函数的参数的类型将是 const char *。

函数模板在 C++98 中与类模板一起被引入，不过在模板类声明的时候，标准允许其有默认模板参数。默认的模板参数的作用好比函数的默认形参。然而由于种种原因，C++98 标准

却不支持函数模板的默认模板参数。不过在 C++11 中，这一限制已经被解除了，我们可以看看下面这个例子，如代码清单 2-27 所示。

代码清单 2-27

```
void DefParm(int m = 3) {}  // c++98 编译通过，c++11 编译通过
template <typename T = int>
    class DefClass {};      // c++98 编译通过，c++11 编译通过
template <typename T = int>
    void DefTempParm() {};  // c++98 编译失败，c++11 编译通过
// 编译选项 :g++ -c -std=c++11 2-11-1.cpp
```

可以看到，DefTempParm 函数模板拥有一个默认参数。使用仅支持 C++98 的编译器编译，DefTempParm 的编译会失败，而支持 C++11 的编译器则毫无问题。不过在语法上，与类模板有些不同的是，在为多个默认模板参数声明指定默认值的时候，程序员必须遵照“从右往左”的规则进行指定。而这个条件对函数模板来说并不是必须的，如代码清单 2-28 所示。

代码清单 2-28

```
template<typename T1, typename T2 = int> class DefClass1;
template<typename T1 = int, typename T2> class DefClass2;   // 无法通过编译

template<typename T, int i = 0> class DefClass3;
template<int i = 0, typename T> class DefClass4;            // 无法通过编译

template<typename T1 = int, typename T2> void DefFunc1(T1 a, T2 b);
template<int i = 0, typename T> void DefFunc2(T a);
// 编译选项 :g++ -c -std=c++11 2-11-2.cpp
```

从代码清单 2-28 中可以看到，不按照从右往左定义默认类模板参数的模板类 DefClass2 和 DefClass4 都无法通过编译。而对于函数模板来说，默认模板参数的位置则比较随意。可以看到 DefFunc1 和 DefFunc2 都为第一个模板参数定义了默认参数，而第二个模板参数的默认值并没有定义，C++11 编译器却认为没有问题。

函数模板的参数推导规则也并不复杂。简单地讲，如果能够从函数实参中推导出类型的话，那么默认模板参数就不会被使用，反之，默认模板参数则可能会被使用。我们可以看看下面这个来自于 C++11 标准草案的例子，如代码清单 2-29 所示。

代码清单 2-29

```
template <class T, class U = double>
void f(T t = 0, U u = 0);

void g() {
```

```
    f(1, 'c');       // f<int,char>(1,'c')
    f(1);            // f<int,double>(1,0), 使用了默认模板参数 double
    f();             // 错误 : T 无法被推导出来
    f<int>();        // f<int,double>(0,0), 使用了默认模板参数 double
    f<int,char>();   // f<int,char>(0,0)
}
// 编译选项 :g++ -std=c++11 2-11-3.cpp
```

在代码清单 2-29 中，我们定义了一个函数模板 f，f 同时使用了默认模板参数和默认函数参数。可以看到，由于函数的模板参数可以由函数的实参推导而出，所以在 f(1) 这个函数调用中，我们实例化出了模板函数的调用应该为 f<int,double>(1,0)，其中，第二个类型参数 U 使用了默认的模板类型参数 double，而函数实参则为默认值 0。类似地，f<int>() 实例化出的模板函数第二参数类型为 double，值为 0。而表达式 f() 由于第一类型参数 T 的无法推导，从而导致了编译的失败。而通过这个例子我们也可以看到，默认模板参数通常是需要跟默认函数参数一起使用的。

还有一点应该强调一下，模板函数的默认形参不是模板参数推导的依据。函数模板参数的选择，总是由函数的实参推导而来的，这点读者在使用中应当注意。

2.12 外部模板

☞ **类别：部分人**

2.12.1 为什么需要外部模板

“外部模板”是 C++11 中一个关于模板性能上的改进。实际上，“外部”（extern）这个概念早在 C 的时候已经就有了。通常情况下，我们在一个文件中 a.c 中定义了一个变量 int i，而在另外一个文件 b.c 中想使用它，这个时候我们就会在没有定义变量 i 的 b.c 文件中做一个外部变量的声明。比如：

```
extern int i;
```

这样做的好处是，在分别编译了 a.c 和 b.c 之后，其生成的目标文件 a.o 和 b.o 中只有 i 这个符号[⊖]的一份定义。具体地，a.o 中的 i 是实在存在于 a.o 目标文件的数据区中的数据，而在 b.o 中，只是记录了 i 符号会引用其他目标文件中数据区中的名为 i 的数据。这样一来，在链接器（通常由编译器代为调用）将 a.o 和 b.o 链接成单个可执行文件（或者库文件）c 的时候，c 文件的数据区也只会有一个 i 的数据（供 a.c 和 b.c 的代码共享）。

而如果 b.c 中我们声明 int i 的时候不加上 extern 的话，那么 i 就会实实在在地既存在于 a.o 的数据区中，也存在于 b.o 的数据区中。那么链接器在链接 a.o 和 b.o 的时候，就会报告

⊖ 符号（symbol）是编译器 / 链接器的术语，读者可以简单地将它想象为一个变量名字。

错误，因为无法决定相同的符号是否需要合并。

而对于函数模板来说，现在我们遇到的几乎是一模一样的问题。不同的是，发生问题的不是变量（数据），而是函数（代码）。这样的困境是由于模板的实例化带来的。

注意 这里我们以函数模板为例，因为其只涉及代码，讲解起来比较直观。如果是类模板，则有可能涉及数据，不过其原理都是类似的。

比如，我们在一个 test.h 的文件中声明了如下一个模板函数：

```
template <typename T> void fun(T) {}
```

在第一个 test1.cpp 文件中，我们定义了以下代码：

```
#include "test.h"
void test1() { fun(3); }
```

而在另一个 test2.cpp 文件中，我们定义了以下代码：

```
#include "test.h"
void test2() { fun(4); }
```

由于两个源代码使用的模板函数的参数类型一致，所以在编译 test1.cpp 的时候，编译器实例化出了函数 fun<int>(int)，而当编译 test2.cpp 的时候，编译器又再一次实例化出了函数 fun<int>(int)。那么可以想象，在 test1.o 目标文件和 test2.o 目标文件中，会有两份一模一样的函数 fun<int>(int) 代码。

代码重复和数据重复不同。数据重复，编译器往往无法分辨是否是要共享的数据；而代码重复，为了节省空间，保留其中之一就可以了（只要代码完全相同）。事实上，大部分链接器也是这样做的。在链接的时候，链接器通过一些编译器辅助的手段将重复的模板函数代码 fun<int>(int) 删除掉，只保留了单个副本。这样一来，就解决了模板实例化时产生的代码冗余问题。我们可以看看图 2-1 中的模板函数的编译与链接的过程示意。

不过读者也注意到了，对于源代码中出现的每一处模板实例化，编译器都需要去做实例化的工作；而在链接时，链接器还需要移除重复的实例化代码。很明显，这样的工作太过冗余，而在广泛使用模板的项目中，由于编译器会产生大量冗余代码，会极大地增加编译器的编译时间和链接时间。解决这个问题的方法基本跟变量共享的思路是一样的，就是使用“外部的”模板。

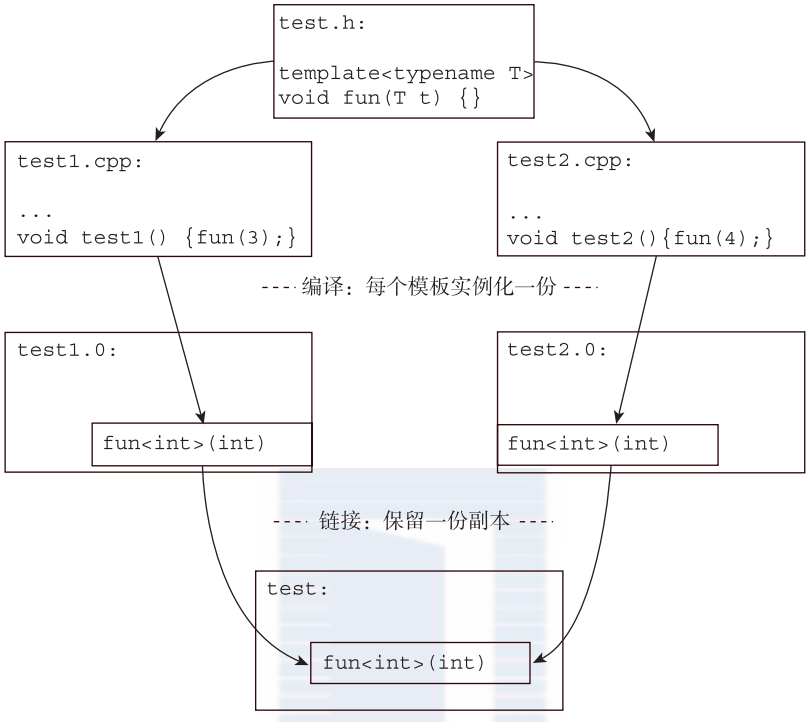

图 2-1 模板函数的编译与链接

2.12.2 显式的实例化与外部模板的声明

外部模板的使用实际依赖于C++98中一个已有的特性，即显式实例化（Explicit Instantiation）。显式实例化的语法很简单，比如对于以下模板：

```
template <typename T> void fun(T) {}
```

我们只需要声明：

```
template void fun<int>(int);
```

这就可以使编译器在本编译单元中实例化出一个 fun<int>(int) 版本的函数（这种做法也被称为强制实例化）。而在 C++11 标准中，又加入了外部模板（Extern Template）的声明。语法上，外部模板的声明跟显式的实例化差不多，只是多了一个关键字 extern。对于上面的例子，我们可以通过：

```
extern template void fun<int>(int);
```

这样的语法完成一个外部模板的声明。

那么回到一开始我们的例子，来修改一下我们的代码。首先，在 test1.cpp 做显式地实例化：

```
#include "test.h"
template void fun<int>(int); // 显示地实例化
void test1() { fun(3); }
```

接下来，在 test2.cpp 中做外部模板的声明：

```
#include "test.h"
extern template void fun<int>(int); // 外部模板的声明
void test1() { fun(3); }
```

这样一来，在 test2.o 中不会再生成 fun<int>(int) 的实例代码。整个模板的实例化流程如图 2-2 所示。

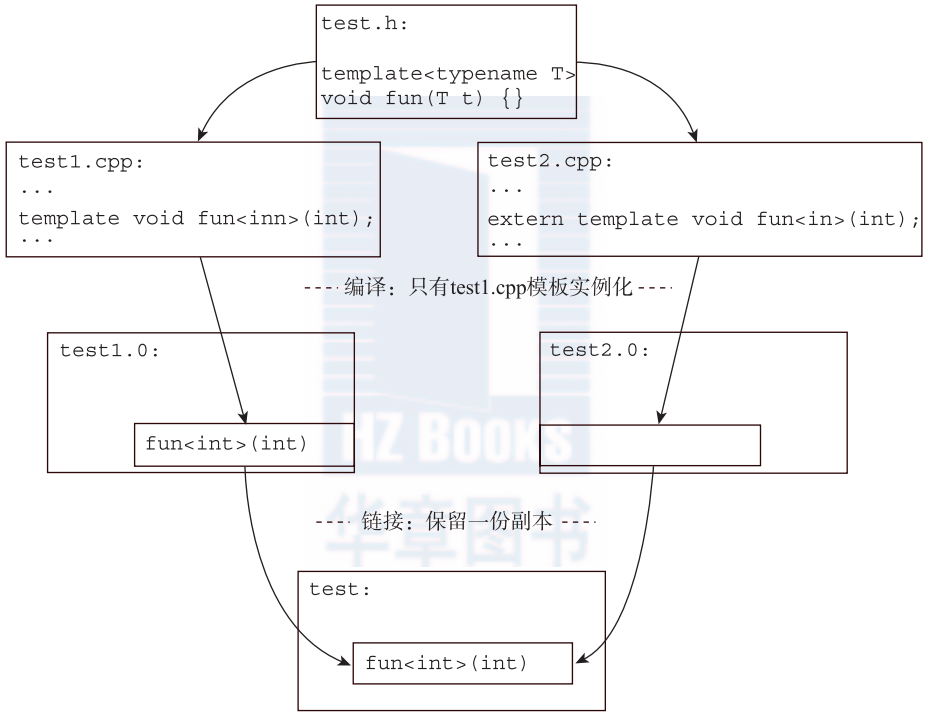

图 2-2　模板函数的编译与链接（使用外部模板声明）

可以看到，由于 test2.o 不再包含 fun<int>(int) 的实例，因此链接器的工作很轻松，基本跟外部变量的做法是一样的，即只需要保证让 test1.cpp 和 test2.cpp 共享一份代码位置即可。而同时，编译器也不用每次都产生一份 fun<int>(int) 的代码，所以可以减少编译时间。这里也可以把外部模板声明放在头文件中，这样所有包含 test.h 的头文件就可以共享这个外部模板声明了。这一点跟使用外部变量声明是完全一致的。

在使用外部模板的时候，我们还需要注意以下问题：如果外部模板声明出现于某个编译单元中，那么与之对应的显示实例化必须出现于另一个编译单元中或者同一个编译单元的后

续代码中；外部模板声明不能用于一个静态函数（即文件域函数），但可以用于类静态成员函数（这一点是显而易见的，因为静态函数没有外部链接属性，不可能在本编译单元之外出现）。

在实际上，C++11 中“模板的显式实例化定义、外部模板声明和使用”好比“全局变量的定义、外部声明和使用”方式的再次应用。不过相比于外部变量声明，不使用外部模板声明并不会导致任何问题。如我们在本节开始讲到的，外部模板定义更应该算作一种针对编译器的编译时间及空间的优化手段。很多时候，由于程序员低估了模板实例化展开的开销，因此大量的模板使用会在代码中产生大量的冗余。这种冗余，有的时候已经使得编译器和链接器力不从心。但这并不意味着程序员需要为四五十行的代码写很多显式模板声明及外部模板声明。只有在项目比较大的情况下。我们才建议用户进行这样的优化。总的来说，就是在既不忽视模板实例化产生的编译及链接开销的同时，也不要过分担心模板展开的开销。

2.13 局部和匿名类型作模板实参

☞ **类别：部分人**

在 C++98 中，标准对模板实参的类型还有一些限制。具体地讲，局部的类型和匿名的类型在 C++98 中都不能做模板类的实参。比如，如代码清单 2-30 所示的代码在 C++98 中很多都无法编译通过。

代码清单 2-30

```
template <typename T> class X {};
template <typename T> void TempFun(T t){};
struct A{} a;
struct {int i;}b;           // b 是匿名类型变量
typedef struct {int i;}B;   // B 是匿名类型

void Fun()
{
    struct C {} c;          // C 是局部类型

    X<A> x1;    // C++98 通过，C++11 通过
    X<B> x2;    // C++98 错误，C++11 通过
    X<C> x3;    // C++98 错误，C++11 通过
    TempFun(a); // C++98 通过，C++11 通过
    TempFun(b); // C++98 错误，C++11 通过
    TempFun(c); // C++98 错误，C++11 通过
}
// 编译选项 :g++ -std=c++11 2-13-1.cpp
```

在代码清单 2-30 中，我们定义了一个模板类 X 和一个模板函数 TempFun，然后分别用普通的全局结构体、匿名的全局结构体，以及局部的结构体作为参数传给模板。可以看到，

使用了局部的结构体 C 及变量 c，以及匿名的结构体 B 及变量 b 的模板类和模板函数，在 C++98 标准下都无法通过编译。而除了匿名的结构体之外，匿名的联合体以及枚举类型，在 C++98 标准下也都是无法做模板的实参的。如今看来这都是不必要的限制。所以在 C++11 中标准允许了以上类型做模板参数的做法 , 故而用支持 C++11 标准的编译器编译以上代码，代码清单 2-30 所示代码可以通过编译。

不过值得指出的是，虽然匿名类型可以被模板参数所接受了，但并不意味着以下写法可以被接受，如代码清单 2-31 所示。

代码清单 2-31

```
template <typename T> struct MyTemplate { };

int main() {
    MyTemplate<struct { int a; }> t; // 无法编译通过 , 匿名类型的声明不能在模板实参位置
    return 0;
}
// 编译选项 :g++ -std=c++11 2-13-2.cpp
```

在代码清单 2-31 中，我们把匿名的结构体直接声明在了模板实参的位置。这种做法非常直观，但却不符合 C/C++ 的语法。在 C/C++ 中，即使是匿名类型的声明，也需要独立的表达式语句。要使用匿名结构体作为模板参数，则可如同代码清单 2-30 一样对匿名结构体作别名。此外在第 4 章我们还会看到使用 C++11 独有的类型推导 decltype，也可以完成相同的功能。

2.14 本章小结

在本章中，我们可以看到 C++11 大大小小共 17 处改动。这 17 处改动，主要都是为保持 C++ 的稳定性以及兼容性而增加的。

比如为了兼容 C99，C++11 引入了 4 个 C99 的预定的宏、__func__ 预定义标识符、_Pragma 操作符、变长参数定义，以及宽窄字符连接等概念。这些都是错过了 C++98 标准，却进入了 C99 的一些标准，为了最大程度地兼容 C，C++ 将这些特性全都纳入 C++11。而由于标准的更新，C++11 也更新了 __cplusplus 宏的值，以表示新的标准的到来。而为了稳定性，C++11 不仅纳入了 C99 中的 long long 类型，还将扩展整型的规则预先定义好。这样一来，就保证了各个编译器扩展内置类型遵守统一的规则。此外，C++11 还将做法不一的静态断言做成了编译器级别的支持，以方便程序员使用。而通过抛弃 throw() 异常描述符和新增可以推导是否抛出异常的 noexcept 异常描述符，C++11 又对标准库大量代码做了改进。

在类方面，C++11 先是对非静态成员的初始化做了改进，同时允许 sizeof 直接作用于类的成员，再者 C++11 对 friend 的声明予以了一定扩展，以方便通过模板的方式指定某个类是

否是其他类或者函数的友元。而 final 和 override 两个关键字的引入，则又为对象编程增加了实用的功能。而在模板方面，C++11 则把默认模板参数的概念延伸到了模板函数上。而且局部类型和匿名类型也可以用做模板的实参。这两个约束的解除，使得模板的使用中需要记忆的规则又少了一些。而外部模板声明的引入，C++11 又为很看重编译性能的用户提供了一些优化编译时间和内存消耗的方法。

在读者读完并理解了这些特性之后，会发现它们几乎像是一台轰鸣作响的机器上的螺丝钉、润滑油、电线丝。C++ 标准委员会则通过这些小修小补，让 C++11 已有的特性看起来更加成熟，更加完美。在这一章里，虽然有的特性会带来一些“小欣喜”，但我们还看不到脱胎换骨、让人眼前一亮的新特性。不过这些零散的特性又确实非常重要，是 C++ 发展中必要的“维护”过程的必然结果。

不过如同我们讲到的，C++11 其实已经看起来像一门新的语言了。在接下来的几章中，我们会看到更多更“闪亮”的新特性。如果读者已经等不及了，那么请现在就翻开下一页。

第③章

通用为本，专用为末

C++11 的设计者总是希望从各种方案中抽象出更为通用的方法来构建新的特性。这意味着 C++11 中的新特性往往具有广泛的可用性，可以与其他已有的，或者新增的语言特性结合起来进行自由的组合，或者提升已有特性的通用性。这与在语言缺陷上“打补丁”的做法有着本质的不同，但也在一定程度上拖慢了 C++11 标准的制定。不过现在一切都已经尘埃落定了。在本章里读者可以看到这些经过反复斟酌制定的新特性，并体会其“普适”的特性。当然，要对一些形如右值引用、移动语义的复杂新特性做到融会贯通，则需要读者反复揣摩。

3.1 继承构造函数

☞ **类别：类作者**

C++ 中的自定义类型——类，是 C++ 面向对象的基石。类具有可派生性，派生类可以自动获得基类的成员变量和接口（虚函数和纯虚函数，这里我们指的都是 public 派生）。不过基类的非虚函数则无法再被派生类使用了。这条规则对于类中最为特别的构造函数也不例外，如果派生类要使用基类的构造函数，通常需要在构造函数中显式声明。比如下面的例子：

```
struct A { A(int i) {} };
struct B : A { B(int i): A(i) {} };
```

B 派生于 A，B 又在构造函数中调用 A 的构造函数，从而完成构造函数的“传递”。这在 C++ 代码中非常常见。当然，这样的设计有一定的好处，尤其是 B 中有成员的时候。如代码清单 3-1 所示的例子。

代码清单 3-1

```
struct A { A(int i) {} };
struct B : A {
    B(int i): A(i), d(i) {}
    int d;
};
// 编译选项 :g++ -c 3-1-1.cpp
```

在代码清单 3-1 中我们看到，派生于结构体 A 的结构体 B 拥有一个成员变量 d，那么在 B 的构造函数 B(int i) 中，我们可以在初始化其基类 A 的同时初始化成员 d。从这个意义上

讲，这样的构造函数设计也算是非常合理的。

不过合情合理并不等于合用，有的时候，我们的基类可能拥有数量众多的不同版本的构造函数——这样的情况并不少见，我们在 2.7 节中就曾经看到过这样的例子。那么倘若基类中有大量的构造函数，而派生类却只有一些成员函数时，那么对于派生类而言，其构造就等同于构造基类。这时候问题就来了，在派生类中我们写的构造函数完完全全就是为了构造基类。那么为了遵从于语法规则，我们还需要写很多的“透传”的构造函数。我们可以看看下面这个例子，如代码清单 3-2 所示。

代码清单 3-2

```
struct A {
    A(int i) {}
    A(double d, int i) {}
    A(float f, int i, const char* c) {}
    // ...
};

struct B : A {
    B(int i): A(i) {}
    B(double d, int i) : A(d, i) {}
    B(float f, int i, const char* c) : A(f, i, c){}
    // ...
    virtual void ExtraInterface(){}
};
// 编译选项 :g++ -c 3-1-2.cpp
```

在代码清单 3-2 中，我们的基类 A 有很多的构造函数的版本，而继承于 A 的派生类 B 实际上只是添加了一个接口 ExtraInterface。那么如果我们在构造 B 的时候想要拥有 A 这样多的构造方法的话，就必须一一“透传”各个接口。这无疑是相当不方便的。

事实上，在 C++ 中已经有了一个好用的规则，就是如果派生类要使用基类的成员函数的话，可以通过 using 声明（using-declaration）来完成。我们可以看看下面这个例子，如代码清单 3-3 所示。

代码清单 3-3

```
#include <iostream>
using namespace std;

struct Base {
    void f(double i){ cout << "Base:" << i << endl; }
};

struct Derived : Base {
    using Base::f;
```

```
    void f(int i) { cout << "Derived:" << i << endl; }
};

int main() {
    Base b;
    b.f(4.5);  // Base:4.5

    Derived d;
    d.f(4.5);  // Base:4.5
}
// 编译选项 :g++ 3-1-3.cpp
```

在代码清单 3-3 中，我们的基类 Base 和派生类 Derived 声明了同名的函数 f，不过在派生类中的版本跟基类有所不同。派生类中的 f 函数接受 int 类型为参数，而基类中接受 double 类型的参数。这里我们使用了 using 声明，声明派生类 Derived 也使用基类版本的函数 f。这样一来，派生类中实际就拥有了两个 f 函数的版本。可以看到，我们在 main 函数中分别定义了 Base 变量 b 和 Derived 变量 d，并传入浮点字面常量 4.5，结果都会调用到基类的接受 double 为参数的版本。

在 C++11 中，这个想法被扩展到了构造函数上。子类可以通过使用 using 声明来声明继承基类的构造函数。那我们要改造代码清单 3-2 所示的例子就非常容易了，如代码清单 3-4 所示。

代码清单 3-4

```
struct A {
    A(int i) {}
    A(double d, int i) {}
    A(float f, int i, const char* c) {}
    // ...
};

struct B : A {
    using A::A;      // 继承构造函数
    // ...
    virtual void ExtraInterface(){}
};
```

这里我们通过 using A::A 的声明，把基类中的构造函数悉数继承到派生类 B 中。这样我们在代码清单 3-2 中的“透传”构造函数就不再需要了。而且更为精巧的是，C++11 标准继承构造函数被设计为跟派生类中的各种类默认函数（默认构造、析构、拷贝构造等）一样，是隐式声明的。这意味着如果一个继承构造函数不被相关代码使用，编译器不会为其产生真正的函数代码。这无疑比“透传”方案总是生成派生类的各种构造函数更加节省目标代码空间。

不过继承构造函数只会初始化基类中成员变量，对于派生类中的成员变量，则无能为力。不过配合我们 2.7 节中的类成员的初始化表达式，为派生类成员变量设定一个默认值还是没有问题的。

在代码清单 3-5 中我们就同时使用了继承构造函数和成员变量初始化两个 C++11 的特性。这样就可以解决一些继承构造函数无法初始化的派生类成员问题。如果这样仍然无法满足需求的话，程序员只能自己来实现一个构造函数，以达到基类和成员变量都能够初始化的目的。

代码清单 3-5

```
struct A {
    A(int i) {}
    A(double d, int i) {}
    A(float f, int i, const char* c) {}
    // ...
};

struct B : A {
    using A::A;
    int d {0};
};

int main() {
    B b(356);    // b.d 被初始化为 0
}
```

有的时候，基类构造函数的参数会有默认值。对于继承构造函数来讲，参数的默认值是不会被继承的。事实上，默认值会导致基类产生多个构造函数的版本，这些函数版本都会被派生类继承。比如代码清单 3-6 所示的这个例子。

代码清单 3-6

```
struct A {
    A (int a = 3, double = 2.4){}
}

struct B : A{
    using A::A;
};
```

可以看到，在代码清单 3-6 中，我们的基类的构造函数 A (int a = 3, double = 2.4) 有一个接受两个参数的构造函数，且两个参数均有默认值。那么 A 到底有多少个可能的构造函数的版本呢？

事实上，B 可能从 A 中继承来的候选继承构造函数有如下一些：

❑A(int = 3, double = 2.4); 这是使用两个参数的情况。
❑A(int = 3); 这是减掉一个参数的情况。
❑A(const A &); 这是默认的复制构造函数。
❑A(); 这是不使用参数的情况。

相应地，B 中的构造函数将会包括以下一些：

❑B(int, double); 这是一个继承构造函数。
❑B(int); 这是减少掉一个参数的继承构造函数。
❑B(const B &); 这是复制构造函数，这不是继承来的。
❑B(); 这是不包含参数的默认构造函数。

可以看见，参数默认值会导致多个构造函数版本的产生，因此程序员在使用有参数默认值的构造函数的基类的时候，必须小心。

而有的时候，我们还会遇到继承构造函数“冲突”的情况。这通常发生在派生类拥有多个基类的时候。多个基类中的部分构造函数可能导致派生类中的继承构造函数的函数名、参数（有的时候，我们也称其为函数签名）都相同，那么继承类中的冲突的继承构造函数将导致不合法的派生类代码，如代码清单 3-7 所示。

代码清单 3-7

```
struct A { A(int) {} };
struct B { B(int) {} };

struct C: A, B {
    using A::A;
    using B::B;
};
```

在代码清单 3-7 中，A 和 B 的构造函数会导致 C 中重复定义相同类型的继承构造函数。这种情况下，可以通过显式定义继承类的冲突的构造函数，阻止隐式生成相应的继承构造函数来解决冲突。比如：

```
struct C: A, B {
    using A::A;
    using B::B;
    C(int){}
};
```

其中的构造函数 C(int) 就很好地解决了代码清单 3-7 中继承构造函数的冲突问题。

另外我们还需要了解的一些规则是，如果基类的构造函数被声明为私有成员函数，或者派生类是从基类中虚继承的，那么就不能够在派生类中声明继承构造函数。此外，如果一旦使用了继承构造函数，编译器就不会再为派生类生成默认构造函数了，那么形如代码清单

3-8 中这样的情况，程序员就必须注意继承构造函数没有包含一个无参数的版本。在本例中，变量 b 的定义应该是不能够通过编译的。

代码清单 3-8

```
struct A { A (int){} };
struct B : A{ using A::A; };

B b;     // B 没有默认构造函数
```

在我们编写本书的时候，还没有编译器实现了继承构造函数这个特性，所以本节中代码清单 3-4 至代码清单 3-8 的例子都仅供读者参考，因为我们并没有实际编译过。但是编译器对继承构造函数的支持应该很快就要完成了，比如 g++ 就计划在 4.8 版本中提供支持。可能本书出版的时候，读者就已经可以进行实验了。

3.2 委派构造函数

☞ **类别：类作者**

与继承构造函数类似的，委派构造函数也是 C++11 中对 C++ 的构造函数的一项改进，其目的也是为了减少程序员书写构造函数的时间。通过委派其他构造函数，多构造函数的类编写将更加容易。

首先我们可以看看代码清单 3-9 中构造函数代码冗余的例子。

代码清单 3-9

```
class Info {
public:
    Info() : type(1), name('a') { InitRest(); }
    Info(int i) : type(i), name('a') { InitRest(); }
    Info(char e): type(1), name(e) { InitRest(); }

private:
    void InitRest() { /* 其他初始化 */ }
    int  type;
    char name;
    // ...
};
// 编译选项 :g++ -c 3-2-1.cpp
```

在代码清单 3-9 中，我们声明了一个 Info 的自定义类型。该类型拥有 2 个成员变量以及 3 个构造函数。这里的 3 个构造函数都声明了初始化列表来初始化成员 type 和 name，并且都调用了相同的函数 InitRest。可以看到，除了初始化列表有的不同，而其他的部分，3 个构造函数基本上是相似的，因此其代码存在着很多重复。

读者可能会想到 2.7 节中我们对成员初始化的方法，那么我们用该方法来改写一下这个例子，如代码清单 3-10 所示。

代码清单 3-10

```
class Info {
public:
    Info() { InitRest(); }
    Info(int i) : type(i) { InitRest(); }
    Info(char e): name(e) { InitRest(); }

private:
    void InitRest() { /* 其他初始化 */ }
    int  type {1};
    char name {'a'};
    // ...
};
// 编译选项 :g++ -c -std=c++11 3-2-2.cpp
```

在代码清单 3-10 中，我们在 Info 成员变量 type 和 name 声明的时候就地进行了初始化。可以看到，构造函数确实简单了不少，不过每个构造函数还是需要调用 InitRest 函数进行初始化。而现实编程中，构造函数中的代码还会更长，比如可能还需要调用一些基类的构造函数等。那能不能在一些构造函数中连 InitRest 都不用调用呢？

答案是肯定的，但前提是我们能够将一个构造函数设定为“基准版本”，比如本例中 Info() 版本的构造函数，而其他构造函数可以通过委派“基准版本”来进行初始化。按照这个想法，我们可能会如下编写构造函数：

```
Info() { InitRest(); }
Info(int i) { this->Info(); type = i; }
Info(char e) { this->Info(); name = e; }
```

这里我们通过 this 指针调用我们的“基准版本”的构造函数。不过可惜的是，一般的编译器都会阻止 this->Info() 的编译。原则上，编译器不允许在构造函数中调用构造函数，即使参数看起来并不相同。

当然，我们还可以开发出一个更具有“黑客精神”的版本：

```
Info() { InitRest(); }
Info(int i) { new (this) Info(); type = i; }
Info(char e) { new (this) Info(); name = e; }
```

这里我们使用了 placement new 来强制在本对象地址（this 指针所指地址）上调用类的构造函数。这样一来，我们可以绕过编译器的检查，从而在 2 个构造函数中调用我们的“基准版本”。这种方法看起来不错，却是在已经初始化一部分的对象上再次调用构造函数，因此虽然针对这个简单的例子在我们的实验机上该做法是有效的，却是种危险的做法。

在 C++11 中，我们可以使用委派构造函数来达到期望的效果。更具体的，C++11 中的委派构造函数是在构造函数的初始化列表位置进行构造的、委派的。我们可以看看代码清单 3-11 所示的这个例子。

代码清单 3-11

```
class Info {
public:
    Info() { InitRest(); }
    Info(int i) : Info() { type = i; }
    Info(char e): Info() { name = e; }

private:
    void InitRest() { /* 其他初始化 */ }
    int  type {1};
    char name {'a'};
    // ...
};
// 编译选项 :g++ -c -std=c++11 3-2-3.cpp
```

可以看到，在代码清单 3-11 中，我们在 Info(int) 和 Info(char) 的初始化列表的位置，调用了“基准版本”的构造函数 Info()。这里我们为了区分被调用者和调用者，称在初始化列表中调用“基准版本”的构造函数为委派构造函数（delegating constructor），而被调用的“基准版本”则为目标构造函数（target constructor）。在 C++11 中，所谓委派构造，就是指委派函数将构造的任务委派给了目标构造函数来完成这样一种类构造的方式。

当然，在代码清单 3-11 中，委派构造函数只能在函数体中为 type、name 等成员赋初值。这是由于委派构造函数不能有初始化列表造成的。在 C++ 中，构造函数不能同时“委派”和使用初始化列表，所以如果委派构造函数要给变量赋初值，初始化代码必须放在函数体中。比如：

```
struct Rule1 {
    int i;
    Rule1(int a): i(a) {}
    Rule1(): Rule1(40), i(1) {} // 无法通过编译
};
```

Rule1 的委派构造函数 Rule1() 的写法就是非法的。我们不能在初始化列表中既初始化成员，又委托其他构造函数完成构造。

这样一来，代码清单 3-11 中的代码的初始化就不那么令人满意了，因为初始化列表的初始化方式总是先于构造函数完成的（实际在编译完成时就已经决定了）。这会可能致使程序员犯错（稍后解释）。不过我们可以稍微改造一下目标构造函数，使得委派构造函数依然可以在初始化列表中初始化所有成员，如代码清单 3-12 所示。

代码清单 3-12

```
class Info {
public:
    Info() : Info(1, 'a') { }
    Info(int i) : Info(i, 'a') { }
    Info(char e): Info(1, e) { }

private:
    Info(int i, char e): type(i), name(e) { /* 其他初始化 */ }
    int  type;
    char name;
    // ...
};
// 编译选项 :g++ -c -std=c++11 3-2-4.cpp
```

在代码清单 3-12 中，我们定义了一个私有的目标构造函数 Info(int, char)，这个构造函数接受两个参数，并将参数在初始化列表中初始化。而且由于这个目标构造函数的存在，我们可以不再需要 InitRest 函数了，而是将其代码都放入 Info(int, char) 中。这样一来，其他委派构造函数就可以委托该目标构造函数来完成构造。

事实上，在使用委派构造函数的时候，我们也建议程序员抽象出最为“通用”的行为做目标构造函数。这样做一来代码清晰，二来行为也更加正确。读者可以比较一下代码清单 3-11 和代码清单 3-12 中 Info 的定义，这里我们假设代码清单 3-11、代码清单 3-12 中注释行的“其他初始化”位置的代码如下：

```
type += 1;
```

那么调用 Info(int) 版本的构造函数会得到不同的结果。比如如果做如下一个类型的声明：

```
Info f(3);
```

这个声明对代码清单 3-11 中的 Info 定义而言，会导致成员 f.type 的值为 3，（因为 Info(int) 委托 Info() 初始化，后者调用 InitRest 将使得 type 的值为 4。不过 Info(int) 函数体内又将 type 重写为 3 ）。而依照代码清单 3-12 中的 Info 定义，f.type 的值将最终为 4。从代码编写者角度看，代码清单 3-12 中 Info 的行为会更加正确。这是由于在 C++11 中，目标构造函数的执行总是先于委派构造函数而造成的。因此避免目标构造函数和委托构造函数体中初始化同样的成员通常是必要的，否则则可能发生代码清单 3-11 错误。

而在构造函数比较多的时候，我们可能会拥有不止一个委派构造函数，而一些目标构造函数很可能也是委派构造函数，这样一来，我们就可能在委派构造函数中形成链状的委派构造关系，如代码清单 3-13 所示。

代码清单 3-13

```
class Info {
public:
    Info() : Info(1) { }    // 委派构造函数
    Info(int i) : Info(i, 'a') { } // 既是目标构造函数，也是委派构造函数
    Info(char e): Info(1, e) { }

private:
    Info(int i, char e): type(i), name(e) { /* 其他初始化 */ } // 目标构造函数
    int  type;
    char name;
    // ...
};
// 编译选项 :g++ -c -std=c++11 3-2-5.cpp
```

代码清单 3-13 所示就是这样一种链状委托构造，这里我们使 Info() 委托 Info(int) 进行构造，而 Info(int) 又委托 Info(int, char) 进行构造。在委托构造的链状关系中，有一点程序员必须注意，就是不能形成委托环（delegation cycle）。比如：

```
struct Rule2 {
    int i, c;
    Rule2(): Rule2(2) {}
    Rule2(int i): Rule2('c') {}
    Rule2(char c): Rule2(2) {}
};
```

Rule2 定义中，Rule2()、Rule2(int) 和 Rule2(char) 都依赖于别的构造函数，形成环委托构造关系。这样的代码通常会导致编译错误。

委派构造的一个很实际的应用就是使用构造模板函数产生目标构造函数，如代码清单 3-14 所示。

代码清单 3-14

```
#include <list>
#include <vector>
#include <deque>
using namespace std;

class TDConstructed {
    template<class T> TDConstructed(T first, T last) :
        l(first, last) {}
    list<int> l;

public:
    TDConstructed(vector<short> & v):
        TDConstructed(v.begin(), v.end()) {}
    TDConstructed(deque<int> & d):
```

```
        TDConstructed(d.begin(), d.end()) {}
};
// 编译选项 :g++ -c -std=c++11 3-2-6.cpp
```

在代码清单 3-14 中，我们定义了一个构造函数模板。而通过两个委派构造函数的委托，构造函数模板会被实例化。T 会分别被推导为 vector<short>::iterator 和 deque<int>::iterator 两种类型。这样一来，我们的 TDConstructed 类就可以很容易地接受多种容器对其进行初始化。这无疑比罗列不同类型的构造函数方便了很多。可以说，委托构造使得构造函数的泛型编程也成为了一种可能。

此外，在异常处理方面，如果在委派构造函数中使用 try 的话，那么从目标构造函数中产生的异常，都可以在委派构造函数中被捕捉到。我们可以看看代码清单 3-15 所示的例子。

代码清单 3-15

```
#include <iostream>
using namespace std;

class DCExcept {
public:
    DCExcept(double d)
        try : DCExcept(1, d) {
            cout << "Run the body." << endl;
            // 其他初始化
        }
        catch(...) {
            cout << "caught exception." << endl;
        }
private:
    DCExcept(int i, double d){
        cout << "going to throw!" << endl;
        throw 0;
    }
    int type;
    double data;
};

int main() {
    DCExcept a(1.2);
}
// 编译选项 :g++ -std=c++11 3-2-7.cpp
```

在代码清单 3-15 中，我们在目标构造函数 DCExcept(int, double) 抛出了一个异常，并在委派构造函数 DCExcept(int) 中进行捕捉。编译运行该程序，我们在实验机上获得以下输出：

```
going to throw!
caught exception.
```

```
terminate called after throwing an instance of 'int'
Aborted
```

可以看到，由于在目标构造函数中抛出了异常，委派构造函数的函数体部分的代码并没有被执行。这样的设计是合理的，因为如果函数体依赖于目标构造函数构造的结果，那么当目标构造函数构造发生异常的情况下，还是不要执行委派构造函数函数体中的代码为好。

其实，在 Java 等一些面向对象的编程语言中，早已经支持了委派构造函数这样的功能。因此，相比于继承构造函数，委派构造函数的设计和实现都比较早。而通过成员的初始化、委派构造函数，以及继承构造函数，C++ 中的构造函数的书写将进一步简化，这对程序员尤其是库的编写者来说，无疑是有积极意义的。

3.3 右值引用：移动语义和完美转发

☞ 类别：类作者

3.3.1 指针成员与拷贝构造

对 C++ 程序员来说，编写 C++ 程序有一条必须注意的规则，就是在类中包含了一个指针成员的话，那么就要特别小心拷贝构造函数的编写，因为一不小心，就会出现内存泄露。我们来看看代码清单 3-16 中的例子。

代码清单 3-16

```
#include <iostream>
using namespace std;

class HasPtrMem {
public:
    HasPtrMem(): d(new int(0)) {}
    HasPtrMem(const HasPtrMem & h):
        d(new int(*h.d)) {} // 拷贝构造函数，从堆中分配内存，并用 *h.d 初始化
    ~HasPtrMem() { delete d; }
    int * d;
};

int main() {
    HasPtrMem a;
    HasPtrMem b(a);
    cout << *a.d << endl;   // 0
    cout << *b.d << endl;   // 0
}   // 正常析构
// 编译选项 :g++ 3-3-1.cpp
```

在代码清单 3-16 中，我们定义了一个 HasPtrMem 的类。这个类包含一个指针成员，该

成员在构造时接受一个 new 操作分配堆内存返回的指针，而在析构的时候则会被 delete 操作用于释放之前分配的堆内存。在 main 函数中，我们声明了 HasPtrMem 类型的变量 a，又使用 a 初始化了变量 b。按照 C++ 的语法，这会调用 HasPtrMem 的拷贝构造函数。这里的拷贝构造函数由编译器隐式生成，其作用是执行类似于 memcpy 的按位拷贝。这样的构造方式有一个问题，就是 a.d 和 b.d 都指向了同一块堆内存。因此在 main 作用域结束的时候，a 和 b 的析构函数纷纷被调用，当其中之一完成析构之后（比如 b），那么 a.d 就成了一个“悬挂指针”（dangling pointer），因为其不再指向有效的内存了。那么在该悬挂指针上释放内存就会造成严重的错误。

这个问题在 C++ 编程中非常经典。这样的拷贝构造方式，在 C++ 中也常被称为“浅拷贝”（shollow copy）。而在未声明构造函数的情况下，C++ 也会为类生成一个浅拷贝的构造函数。通常最佳的解决方案是用户自定义拷贝构造函数来实现“深拷贝”（deep copy），我们来看看代码清单 3-17 中的修正方法。

代码清单 3-17

```
#include <iostream>
using namespace std;

class HasPtrMem {
public:
    HasPtrMem(): d(new int(0)) {}
    HasPtrMem(HasPtrMem & h):
        d(new int(*h.d)) {} // 拷贝构造函数，从堆中分配内存，并用 *h.d 初始化
    ~HasPtrMem() { delete d; }
    int * d;
};

int main() {
    HasPtrMem a;
    HasPtrMem b(a);
    cout << *a.d << endl;   // 0
    cout << *b.d << endl;   // 0
}   // 正常析构
// 编译选项 :g++ 3-3-2.cpp
```

在代码清单 3-17 中，我们为 HasPtrMem 添加了一个拷贝构造函数。拷贝构造函数从堆中分配新内存，将该分配来的内存的指针交还给 d，又使用 *(h.d) 对 *d 进行了初始化。通过这样的方法，就避免了悬挂指针的困扰。

3.3.2 移动语义

拷贝构造函数中为指针成员分配新的内存再进行内容拷贝的做法在 C++ 编程中几乎被视为是不可违背的。不过在一些时候，我们确实不需要这样的拷贝构造语义。我们可以看看代

码清单 3-18 所示的例子。

代码清单 3-18

```
#include <iostream>
using namespace std;

class HasPtrMem {
public:
    HasPtrMem(): d(new int(0)) {
        cout << "Construct: " << ++n_cstr << endl;
    }
    HasPtrMem(const HasPtrMem & h): d(new int(*h.d)) {
        cout << "Copy construct: " << ++n_cptr << endl;
    }
    ~HasPtrMem() {
        cout << "Destruct: " << ++n_dstr << endl;
    }
    int * d;
    static int n_cstr;
    static int n_dstr;
    static int n_cptr;
};

int HasPtrMem::n_cstr = 0;
int HasPtrMem::n_dstr = 0;
int HasPtrMem::n_cptr = 0;

HasPtrMem GetTemp() { return HasPtrMem(); }

int main() {
    HasPtrMem a = GetTemp();
}
// 编译选项 :g++ 3-3-3.cpp -fno-elide-constructors
```

在代码清单 3-18 中，我们声明了一个返回一个 HasPtrMem 变量的函数。为了记录构造函数、拷贝构造函数，以及析构函数调用的次数，我们使用了一些静态变量。在 main 函数中，我们简单地声明了一个 HasPtrMem 的变量 a，要求它使用 GetTemp 的返回值进行初始化。编译运行该程序，我们可以看到下面的输出：

```
Construct: 1
Copy construct: 1
Destruct: 1
Copy construct: 2
Destruct: 2
Destruct: 3
```

这里构造函数被调用了一次，这是在 GetTemp 函数中 HasPtrMem() 表达式显式地调用了

构造函数而打印出来的。而拷贝构造函数则被调用了两次。这两次一次是从 GetTemp 函数中 HasPtrMem() 生成的变量上拷贝构造出一个临时值，以用作 GetTemp 的返回值，而另外一次则是由临时值构造出 main 中变量 a 调用的。对应地，析构函数也就调用了 3 次。这个过程如图 3-1 所示。

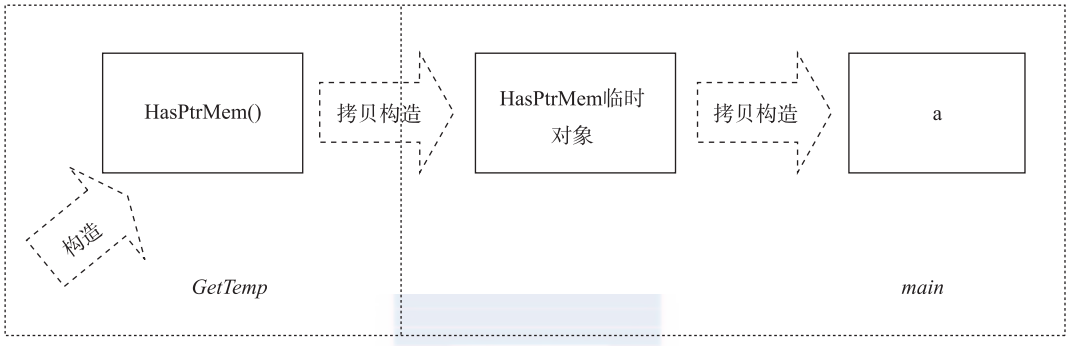

图 3-1　函数返回时的临时变量与拷贝

最让人感到不安就是拷贝构造函数的调用。在我们的例子里，类 HasPtrMem 只有一个 int 类型的指针。而如果 HasPtrMem 的指针指向非常大的堆内存数据的话，那么拷贝构造的过程就会非常昂贵。可以想象，这种情况一旦发生，a 的初始化表达式的执行速度将相当堪忧。而更为令人堪忧的是，临时变量的产生和销毁以及拷贝的发生对于程序员来说基本上是透明的，不会影响程序的正确性，因而即使该问题导致程序的性能不如预期，也不易被程序员察觉（事实上，编译器常常对函数返回值有专门的优化，我们在本节结束时会提到）。

让我们把目光再次聚集在临时对象上，即图 3-1 中的 main 函数的部分。按照 C++ 的语义，临时对象将在语句结束后被析构，会释放它所包含的堆内存资源。而 a 在拷贝构造的时候，又会被分配堆内存。这样的一去一来似乎并没有太大的意义，那么我们是否可以在临时对象构造 a 的时候不分配内存，即不使用所谓的拷贝构造语义呢？

在 C++11 中，答案是肯定的。我们可以看看如图 3-2 所示的示意图。

图 3-2 中的上半部分可以看到从临时变量中拷贝构造变量 a 的做法，即在拷贝时分配新的堆内存，并从临时对象的堆内存中拷贝内容至 a.d。而构造完成后，临时对象将析构，因此其拥有的堆内存资源会被析构函数释放。而图 3-2 的下半部分则是一种“新”方法（实际跟我们在代码清单 3-1 中做得差不多），该方法在构造时使得 a.d 指向临时对象的堆内存资源。同时我们保证临时对象不释放所指向的堆内存（下面解释怎么做），那么在构造完成后，临时对象被析构，a 就从中“偷”到了临时对象所拥有的堆内存资源。

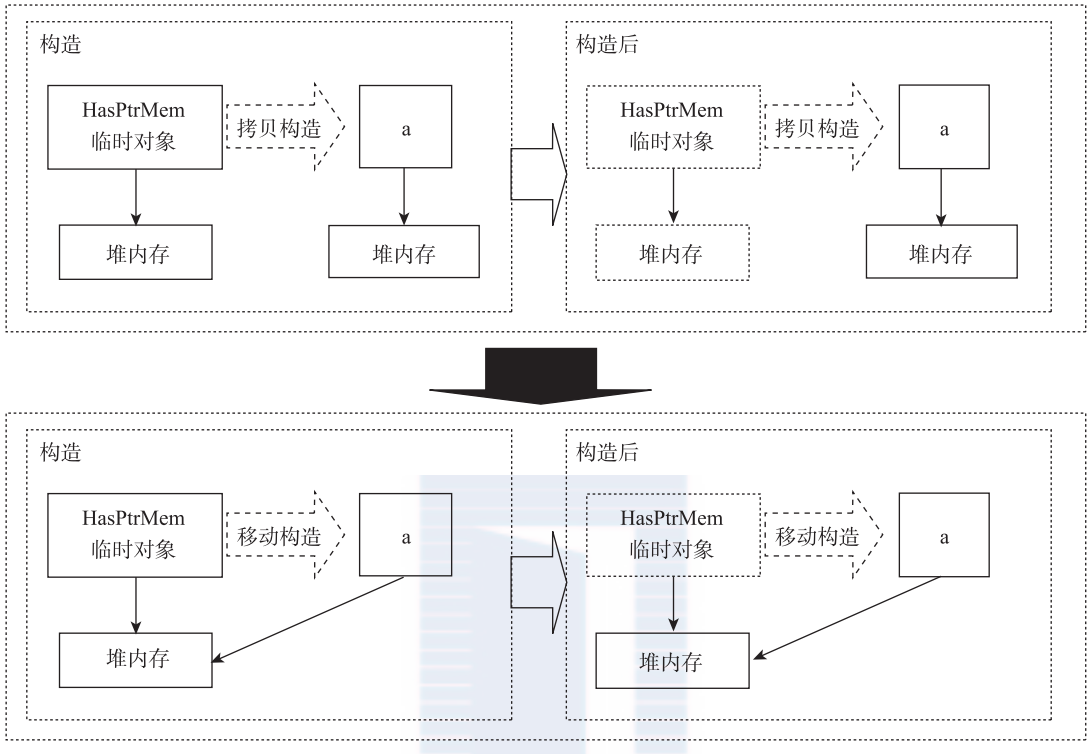

图 3-2 拷贝构造与移动构造

在 C++11 中，这样的“偷走”临时变量中资源的构造函数，就被称为“移动构造函数”。而这样的“偷”的行为，则称之为“移动语义”（move semantics）。当然，换成白话的中文，可以理解为“移为己用”。我们可以看看代码清单 3-19 中是如何来实现这种移动语义的。

代码清单 3-19

```
#include <iostream>
using namespace std;

class HasPtrMem {
public:
    HasPtrMem(): d(new int(3)) {
        cout << "Construct: " << ++n_cstr << endl;
    }
    HasPtrMem(const HasPtrMem & h): d(new int(*h.d)) {
        cout << "Copy construct: " << ++n_cptr << endl;
    }
    HasPtrMem(HasPtrMem && h): d(h.d) { // 移动构造函数
        h.d = nullptr;                  // 将临时值的指针成员置空
        cout << "Move construct: " << ++n_mvtr << endl;
    }
```

```
    ~HasPtrMem() {
        delete d;
        cout << "Destruct: " << ++n_dstr << endl;
    }
    int * d;
    static int n_cstr;
    static int n_dstr;
    static int n_cptr;
    static int n_mvtr;
};

int HasPtrMem::n_cstr = 0;
int HasPtrMem::n_dstr = 0;
int HasPtrMem::n_cptr = 0;
int HasPtrMem::n_mvtr = 0;

HasPtrMem GetTemp() {
    HasPtrMem h;
    cout << "Resource from " <<  __func__ << ": " << hex << h.d << endl;
    return h;
}

int main() {
    HasPtrMem a = GetTemp();
    cout << "Resource from " <<  __func__ << ": " << hex << a.d << endl;
}
// 编译选项 :g++ -std=c++11 3-3-4.cpp -fno-elide-constructors
```

相比于代码清单 3-18，代码清单 3-19 中的 HasPtrMem 类多了一个构造函数 HasPtrMem (HasPtrMem &&)，这个就是我们所谓的移动构造函数。与拷贝构造函数不同的是，移动构造函数接受一个所谓的“右值引用”的参数，关于右值我们接下来会解释，读者可以暂时理解为临时变量的引用。可以看到，移动构造函数使用了参数 h 的成员 d 初始化了本对象的成员 d（而不是像拷贝构造函数一样需要分配内存，然后将内容依次拷贝到新分配的内存中），而 h 的成员 d 随后被置为指针空值 nullptr（请参见 7.1 节，这里等同于 NULL）。这就完成了移动构造的全过程。

这里所谓的“偷”堆内存，就是指将本对象 d 指向 h.d 所指的内存这一条语句，相应地，我们还将 h 的成员 d 置为指针空值。这其实也是我们“偷”内存时必须做的。这是因为在移动构造完成之后，临时对象会立即被析构。如果不改变 h.d（临时对象的指针成员）的话，则临时对象会析构掉本是我们“偷”来的堆内存。这样一来，本对象中的 d 指针也就成了一个悬挂指针，如果我们对指针进行解引用，就会发生严重的运行时错误。

为了看看移动构造的效果，我们让 GetTemp 和 main 函数分别打印变量 h 和变量 a 中的指针 h.d 和 a.d，在我们的实验机上运行的结果如下：

```
Construct: 1
```

```
Resource from GetTemp: 0x603010
Move construct: 1
Destruct: 1
Move construct: 2
Destruct: 2
Resource from main: 0x603010
Destruct: 3
```

可以看到，这里没有调用拷贝构造函数，而是调用了两次移动构造函数，移动构造的结果是，GetTemp 中的 h 的指针成员 h.d 和 main 函数中的 a 的指针成员 a.d 的值是相同的，即 h.d 和 a.d 都指向了相同的堆地址内存。该堆内存在函数返回的过程中，成功地逃避了被析构的“厄运”，取而代之地，成为了赋值表达式中的变量 a 的资源。如果堆内存不是一个 int 长度的数据，而是以 MByte 为单位的堆空间，那么这样的移动带来的性能提升将非常惊人。

或许读者会质疑说：为什么要这么费力地添加移动构造函数呢？完全可以选择改变 GetTemp 的接口，比如直接传一个引用或者指针到 GetTemp 的参数中去，效果应该也不差。其实从性能上来讲，这样的做法确实毫无问题，甚至只好不差。不过从使用的方便性上来讲效果不好。如果函数返回临时值的话，可以在单条语句里完成很多计算，比如我们可以很自然地写出如下语句：

```
Caculate(GetTemp(), SomeOther(Maybe(), Useful(Values, 2)));
```

但如果通过传引用或者指针的方法而不返回值的话，通常就需要很多语句来完成上面的工作。可能是像下面这样的代码：

```
string *a; vector b;     // 事先声明一些变量用于传递返回值
...
Useful(Values, 2, a);     // 最后一个参数是指针，用于返回结果
SomeOther(Maybe(), a, b);     // 最后一个参数是引用，用于返回结果
Caculate(GetTemp(), b);
```

两者在代码编写效率和可读性上都存在着明显的差别。而即使声明这些传递返回值的变量为全局的，函数再将这些引用和指针都作为返回值返回给调用者，我们也需要在 Caculate 调用之前声明好所有的引用和指针。这无疑是繁琐的工作。函数返回临时变量的好处就是不需要声明变量，也不需要知道生命期。程序员只需要按照最自然的方式，使用最简单的语句就可以完成大量的工作。

那么再回到移动语义上来，还有一个最为关键的问题没有解决，那就是移动构造函数何时会被触发。之前我们只是提到了临时对象，一旦我们用到的是个临时变量，那么移动构造语义就可以得到执行。那么，在 C++ 中如何判断产生了临时对象？如何将其用于移动构造函数？是否只有临时变量可以用于移动构造？……读者可能还有很多问题。要回答这些问题，需要先了解一下 C++ 中的“值”是如何分类的。

注意 事实上，移动语义并不是什么新的概念，在 C++98/03 的语言和库中，它已经存

在了，比如：

- ❑在某些情况下拷贝构造函数的省略（copy constructor elision in some contexts）
- ❑智能指针的拷贝（auto_ptr “copy”）
- ❑链表拼接（list::splice）
- ❑容器内的置换（swap on containers）

以上这些操作都包含了从一个对象向另外一个对象的资源转移（至少概念上）的过程，唯一欠缺的是统一的语法和语义的支持，来使我们可以使用通用的代码移动任意的对象（就像我们今天可以使用通用的代码来拷贝任意对象一样）。如果能够任意地使用对象的移动而不是拷贝，那么标准库中的很多地方的性能都会大大提高。

3.3.3 左值、右值与右值引用

在 C 语言中，我们常常会提起左值（lvalue）、右值（rvalue）这样的称呼。而在编译程序时，编译器有时也会在报出的错误信息中会包含左值、右值的说法。不过左值、右值通常不是通过一个严谨的定义而为人所知的，大多数时候左右值的定义与其判别方法是一体的。一个最为典型的判别方法就是，在赋值表达式中，出现在等号左边的就是“左值”，而在等号右边的，则称为“右值”。比如：

```
a = b + c;
```

在这个赋值表达式中，a 就是一个左值，而 b + c 则是一个右值。这种识别左值、右值的方法在 C++ 中依然有效。不过 C++ 中还有一个被广泛认同的说法，那就是可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值。那么这个加法赋值表达式中，&a 是允许的操作，但 &(b + c) 这样的操作则不会通过编译。因此 a 是一个左值，(b + c) 是一个右值。

这些判别方法通常都非常有效。更为细致地，在 C++11 中，右值是由两个概念构成的，一个是将亡值（xvalue，eXpiring Value），另一个则是纯右值（prvalue，Pure Rvalue）。

其中纯右值就是 C++98 标准中右值的概念，讲的是用于辨识临时变量和一些不跟对象关联的值。比如非引用返回的函数返回的临时变量值（我们在前面多次提到了）就是一个纯右值。一些运算表达式，比如 1 + 3 产生的临时变量值，也是纯右值。而不跟对象关联的字面量值，比如：2、‘c’、true，也是纯右值。此外，类型转换函数的返回值、lambda 表达式（见 7.3 节）等，也都是右值。

而将亡值则是 C++11 新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用 T&& 的函数返回值、std::move 的返回值（稍后解释），或者转换为 T&& 的类型转换函数的返回值（稍后解释）。而剩余的，可以标识函数、对象的值都属于左值。在 C++11 的程序中，所有的值必属于左值、将亡值、纯右值三者之一。

注意 事实上，之所以我们只知道一些关于左值、右值的判断而很少听到其真正的定义的一个原因就是——很难归纳。而且即使归纳了，也需要大量的解释。

在 C++11 中，右值引用就是对一个右值进行引用的类型。事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。通常情况下，我们只能是从右值表达式获得其引用。比如：

```
T && a = ReturnRvalue();
```

这个表达式中，假设 ReturnRvalue 返回一个右值，我们就声明了一个名为 a 的右值引用，其值等于 ReturnRvalue 函数返回的临时变量的值。

为了区别于 C++98 中的引用类型，我们称 C++98 中的引用为“左值引用”（lvalue reference）。右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名。

在上面的例子中，ReturnRvalue 函数返回的右值在表达式语句结束后，其生命也就终结了（通常我们也称其具有表达式生命期），而通过右值引用的声明，该右值又“重获新生”，其生命期将与右值引用类型变量 a 的生命期一样。只要 a 还“活着”，该右值临时量将会一直“存活”下去。

所以相比于以下语句的声明方式：

```
T b = ReturnRvalue();
```

我们刚才的右值引用变量声明，就会少一次对象的析构及一次对象的构造。因为 a 是右值引用，直接绑定了 ReturnRvalue() 返回的临时量，而 b 只是由临时值构造而成的，而临时量在表达式结束后会析构因应就会多一次析构和构造的开销。

不过值得指出的是，能够声明右值引用 a 的前提是 ReturnRvalue 返回的是一个右值。通常情况下，右值引用是不能够绑定到任何的左值的。比如下面的表达式就是无法通过编译的。

```
int c;
int && d = c;
```

相对地，在 C++98 标准中就已经出现的左值引用是否可以绑定到右值（由右值进行初始化）呢？比如：

```
T & e = ReturnRvalue();
const T & f = ReturnRvalue();
```

这样的语句是否能够通过编译呢？这里的答案是：e 的初始化会导致编译时错误，而 f 则不会。

出现这样的状况的原因是，在常量左值引用在 C++98 标准中开始就是个“万能”的引用类型。它可以接受非常量左值、常量左值、右值对其进行初始化。而且在使用右值对其初始化的时候，常量左值引用还可以像右值引用一样将右值的生命期延长。不过相比于右值引用所引用的右值，常量左值所引用的右值在它的“余生”中只能是只读的。相对地，非常量左值只能接受非常量左值对其进行初始化。

既然常量左值引用在 C++98 中就已经出现，读者可能会努力地搜索记忆，想找出在 C++ 中使用常量左值绑定右值的情况。不过可能一切并不如愿。这是因为，在 C++11 之前，左值、右值对于程序员来说，一直呈透明状态。不知道什么是左值、右值，并不影响写出正确的 C++ 代码。引用的是左值和右值通常也并不重要。不过事实上，在 C++98 通过左值引用来绑定一个右值的情况并不少见，比如：

```
const bool & judgement = true;
```

就是一个使用常量左值引用来绑定右值的例子。不过与如下声明相比较看起来似乎差别不大。

```
const bool judgement = true;
```

可能很多程序员都没有注意到其中的差别（从语法上讲，前者直接使用了右值并为其“续命”，而后者的右值在表达式结束后就销毁了）。

事实上，即使在 C++98 中，我们也常可以使用常量左值引用来减少临时对象的开销，如代码清单 3-20 所示。

代码清单 3-20

```
#include <iostream>
using namespace std;

struct Copyable {
    Copyable() {}
    Copyable(const Copyable &o) {
        cout << "Copied" << endl;
    }
};

Copyable ReturnRvalue() { return Copyable(); }
void AcceptVal(Copyable) {}
void AcceptRef(const Copyable & ) {}

int main() {
    cout << "Pass by value: " << endl;
    AcceptVal(ReturnRvalue()); // 临时值被拷贝传入
    cout << "Pass by reference: " << endl;
    AcceptRef(ReturnRvalue()); // 临时值被作为引用传递
}
```

```
// 编译选项 :g++ 3-3-5.cpp -fno-elide-constructors
```

在代码清单 3-20 中，我们声明了结构体 Copyable，该结构体的唯一的作用就是在被拷贝构造的时候打印一句话：Copied。而两个函数，AcceptVal 使用了值传递参数，而 AcceptRef 使用了引用传递。在以 ReturnRvalue 返回的右值为参数的时候，AcceptRef 就可以直接使用产生的临时值（并延长其生命期），而 AcceptVal 则不能直接使用临时对象。

编译运行代码清单 3-20，可以得到以下结果：

```
Pass by value:
Copied
Copied
Pass by reference:
Copied
```

可以看到，由于使用了左值引用，临时对象被直接作为函数的参数，而不需要从中拷贝一次。读者可以自行分析一下输出结果，这里就不赘述了。而在 C++11 中，同样地，如果在代码清单 3-20 中以右值引用为参数声明如下函数：

```
void AcceptRvalueRef(Copyable && ) {}
```

也同样可以减少临时变量拷贝的开销。进一步地，还可以在 AcceptRvalueRef 中修改该临时值（这个时候临时值由于被右值引用参数所引用，已经获得了函数时间的生命期）。不过修改一个临时值的意义通常不大，除非像 3.3.2 节一样使用移动语义。

就本例而言，如果我们这样实现函数：

```
void AcceptRvalueRef(Copyable && s) {
   Copyable news = std::move(s);
}
```

这里 std::move 的作用是强制一个左值成为右值（看起来很奇怪？这个我们会在下面一节中解释）。该函数就是使用右值来初始化 Copyable 变量 news。当然，如同我们在上小节提到的，使用移动语义的前提是 Copyable 还需要添加一个以右值引用为参数的移动构造函数，比如：

```
Copyable(Copyable &&o) { /* 实现移动语义 */ }
```

这样一来，如果 Copyable 类的临时对象（即 ReturnRvalue 返回的临时值）中包含一些大块内存的指针，news 就可以如同代码清单 3-19 一样将临时值中的内存“窃”为己用，从而从这个以右值引用参数的 AcceptRvalueRef 函数中获得最大的收益。事实上，右值引用的来由从来就跟移动语义紧紧相关。这是右值存在的一个最大的价值（另外一个价值是用于转发，我们会在后面的小节中看到）。

对于本例而言，很有趣的是，读者也可以思考一下：如果我们不声明移动构造函数，而只声明一个常量左值的构造函数会发生什么？如同我们刚才提到的，常量左值引用是个“万

能”的引用类型，无论左值还是右值，常量还是非常量，一概能够绑定。那么如果 Copyable 没有移动构造函数，下列语句：

```
Copyable news = std::move(s);
```

将调用以常量左值引用为参数的拷贝构造函数。这是一种非常安全的设计——移动不成，至少还可以执行拷贝。因此，通常情况下，程序员会为声明了移动构造函数的类声明一个常量左值为参数的拷贝构造函数，以保证在移动构造不成时，可以使用拷贝构造（不过，我们也会在之后看到一些特殊用途的反例）。

为了语义的完整，C++11 中还存在着常量右值引用，比如我们通过以下代码声明一个常量右值引用。

```
const T && crvalueref = ReturnRvalue();
```

但是，一来右值引用主要就是为了移动语义，而移动语义需要右值是可以被修改的，那么常量右值引用在移动语义中就没有用武之处；二来如果要引用右值且让右值不可以更改，常量左值引用往往就足够了。因此在现在的情况下，我们还没有看到常量右值引用有何用处。

表 3-1 中，我们列出了在 C++11 中各种引用类型可以引用的值的类型。值得注意的是，只要能够绑定右值的引用类型，都能够延长右值的生命期。

表 3-1　C++11 中引用类型及其可以引用的值类型

引用类型	可以引用的值类型				注　　记
	非常量左值	常量左值	非常量右值	常量右值	
非常量左值引用	Y	N	N	N	无
常量左值引用	Y	Y	Y	Y	全能类型，可用于拷贝语义
非常量右值引用	N	N	Y	N	用于移动语义、完美转发
常量右值引用	N	N	Y	Y	暂无用途

有的时候，我们可能不知道一个类型是否是引用类型，以及是左值引用还是右值引用（这在模板中比较常见）。标准库在 <type_traits> 头文件中提供了 3 个模板类：is_rvalue_reference、is_lvalue_reference、is_reference，可供我们进行判断。比如：

```
cout << is_rvalue_reference<string &&>::value;
```

我们通过模板类的成员 value 就可以打印出 stirng && 是否是一个右值引用了。配合第 4 章中的类型推导操作符 decltype，我们甚至还可以对变量的类型进行判断。当读者搞不清楚引用类型的时候，不妨使用这样的小工具实验一下。

3.3.4 std::move：强制转化为右值

在 C++11 中，标准库在 <utility> 中提供了一个有用的函数 std::move，这个函数的名字具有迷惑性，因为实际上 std::move 并不能移动任何东西，它唯一的功能是将一个左值强制转化为右值引用，继而我们可以通过右值引用使用该值，以用于移动语义。从实现上讲，std::move 基本等同于一个类型转换：

```
static_cast<T&&>(lvalue);
```

值得一提的是，被转化的左值，其生命期并没有随着左右值的转化而改变。如果读者期望 std::move 转化的左值变量 lvalue 能立即被析构，那么肯定会失望了。我们来看代码清单 3-21 所示的例子。

代码清单 3-21

```
#include <iostream>
using namespace std;

class Moveable{
public:
    Moveable():i(new int(3)) {}
    ~Moveable() { delete i; }
    Moveable(const Moveable & m): i(new int(*m.i)) { }
    Moveable(Moveable && m):i(m.i) {
        m.i = nullptr;
    }
    int* i;
};

int main() {
    Moveable a;

    Moveable c(move(a));     // 会调用移动构造函数
    cout << *a.i << endl;    // 运行时错误
}
// 编译选项 :g++ -std=c++11 3-3-6.cpp -fno-elide-constructors
```

在代码清单 3-21 中，我们为类型 Moveable 定义了移动构造函数。这个函数定义本身没有什么问题，但调用的时候，使用了 Moveable c(move(a)); 这样的语句。这里的 a 本来是一个左值变量，通过 std::move 将其转换为右值。这样一来，a.i 就被 c 的移动构造函数设置为指针空值。由于 a 的生命期实际要到 main 函数结束才结束，那么随后对表达式 *a.i 进行计算的时候，就会发生严重的运行时错误。

这是个典型误用 std::move 的例子。当然，标准库提供该函数的目的不是为了让程序员搬起石头砸自己的脚。事实上，要使用该函数，必须是程序员清楚需要转换的时候。比如上例中，程序员应该知道被转化为右值的 a 不可以再使用。不过更多地，我们需要转换成为右

值引用的还是一个确实生命期即将结束的对象。我们来看看代码清单 3-22 所示的正确例子。

代码清单 3-22

```
#include <iostream>
using namespace std;

class HugeMem{
public:
    HugeMem(int size): sz(size > 0 ? size : 1) {
        c = new int[sz];
    }
    ~HugeMem() { delete [] c; }
    HugeMem(HugeMem && hm): sz(hm.sz), c(hm.c) {
        hm.c = nullptr;
    }
    int * c;
    int sz;
};
class Moveable{
public:
    Moveable():i(new int(3)), h(1024) {}
    ~Moveable() { delete i; }
    Moveable(Moveable && m):
        i(m.i), h(move(m.h)) {      // 强制转为右值，以调用移动构造函数
        m.i = nullptr;
    }
    int* i;
    HugeMem h;
};

Moveable GetTemp() {
    Moveable tmp = Moveable();
    cout << hex << "Huge Mem from " << __func__
        << " @" << tmp.h.c << endl; // Huge Mem from GetTemp @0x603030
    return tmp;
}

int main() {
    Moveable a(GetTemp());
    cout << hex << "Huge Mem from " << __func__
        << " @" << a.h.c << endl;   // Huge Mem from main @0x603030
}
// 编译选项 :g++ -std=c++11 3-3-7.cpp -fno-elide-constructors
```

在代码清单 3-22 中，我们定义了两个类型：HugeMem 和 Moveable，其中 Moveable 包含了一个 HugeMem 的对象。在 Moveable 的移动构造函数中，我们就看到了 std::move 函数的使用。该函数将 m.h 强制转化为右值，以迫使 Moveable 中的 h 能够实现移动构造。这里

可以使用 std::move，是因为 m.h 是 m 的成员，既然 m 将在表达式结束后被析构，其成员也自然会被析构，因此不存在代码清单 3-21 中的生存期不对的问题。另外一个问题可能是 std::move 使用的必要性。这里如果不使用 std::move(m.h) 这样的表达式，而是直接使用 m.h 这个表达式将会怎样？

其实这是 C++11 中有趣的地方：可以接受右值的右值引用本身却是个左值。这里的 m.h 引用了一个确定的对象，而且 m.h 也有名字，可以使用 &m.h 取到地址，因此是个不折不扣的左值。不过这个左值确确实实会很快“灰飞烟灭”，因为拷贝构造函数在 Moveable 对象 a 的构造完成后也就结束了。那么这里使用 std::move 强制其为右值就不会有问题了。而且，如果我们不这么做，由于 m.h 是个左值，就会导致调用 HugeMem 的拷贝构造函数来构造 Moveable 的成员 h（虽然这里没有声明，读者可以自行添加实验一下）。如果是这样，移动语义就没有能够成功地向类的成员传递。换言之，还是会由于拷贝而导致一定的性能上的损失。

事实上，为了保证移动语义的传递，程序员在编写移动构造函数的时候，应该总是记得使用 std::move 转换拥有形如堆内存、文件句柄等资源的成员为右值，这样一来，如果成员支持移动构造的话，就可以实现其移动语义。而即使成员没有移动构造函数，那么接受常量左值的构造函数版本也会轻松地实现拷贝构造，因此也不会引起大的问题。

3.3.5 移动语义的一些其他问题

我们在前面多次提到，移动语义一定是要修改临时变量的值。那么，如果这样声明移动构造函数：

```
Moveable(const Moveable &&)
```

或者这样声明函数：

```
const Moveable ReturnVal();
```

都会使得的临时变量常量化，成为一个常量右值，那么临时变量的引用也就无法修改，从而导致无法实现移动语义。因此程序员在实现移动语义一定要注意排除不必要的 const 关键字。

在 C++11 中，拷贝 / 移动构造函数实际上有以下 3 个版本：

```
T Object(T &)
T Object(const T &)
T Object(T &&)
```

其中常量左值引用的版本是一个拷贝构造版本，而右值引用版本是一个移动构造版本。默认情况下，编译器会为程序员隐式地生成一个（隐式表示如果不被使用则不生成）移动构造函数。不过如果程序员声明了自定义的拷贝构造函数、拷贝赋值函数、移动赋值函数、析

构函数中的一个或者多个，编译器都不会再为程序员生成默认版本。默认的移动构造函数实际上跟默认的拷贝构造函数一样，只能做一些按位拷贝的工作。这对实现移动语义来说是不够的。通常情况下，如果需要移动语义，程序员必须自定义移动构造函数。当然，对一些简单的、不包含任何资源的类型来说，实现移动语义与否都无关紧要，因为对这样的类型而言，移动就是拷贝，拷贝就是移动。

同样地，声明了移动构造函数、移动赋值函数、拷贝赋值函数和析构函数中的一个或者多个，编译器也不会再为程序员生成默认的拷贝构造函数。所以在 C++11 中，拷贝构造 / 赋值和移动构造 / 赋值函数必须同时提供，或者同时不提供，程序员才能保证类同时具有拷贝和移动语义。只声明其中一种的话，类都仅能实现一种语义。

其实，只实现一种语义在类的编写中也是非常常见的。比如说只有拷贝语义的类型——事实上在 C++11 之前我们见过大多数的类型的构造都是只使用拷贝语义的。而只有移动语义的类型则非常有趣，因为只有移动语义表明该类型的变量所拥有的资源只能被移动，而不能被拷贝。那么这样的资源必须是唯一的。因此，只有移动语义构造的类型往往都是“资源型”的类型，比如说智能指针，文件流等，都可以视为“资源型”的类型。在本书的第 5 章中，就可以看到标准库中的仅可移动的模板类：unique_ptr。一些编译器，如 vs2011，现在也把 ifstream 这样的类型实现为仅可移动的。

在标准库的头文件 <type_traits> 里，我们还可以通过一些辅助的模板类来判断一个类型是否是可以移动的。比如 is_move_constructible、is_trivially_move_constructible、is_nothrow_move_constructible，使用方法仍然是使用其成员 value。比如：

```
cout << is_move_constructible<UnknownType>::value;
```

就可以打印出 UnknowType 是否可以移动，这在一些情况下还是非常有用的。

而有了移动语义，还有一个比较典型的应用是可以实现高性能的置换（swap）函数。看看下面这段 swap 模板函数代码：

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

如果 T 是可以移动的，那么移动构造和移动赋值将会被用于这个置换。代码中，a 先将自己的资源交给 tmp，随后 b 再将资源交给 a，tmp 随后又将从 a 中得到的资源交给 b，从而完成了一个置换动作。整个过程，代码都只会按照移动语义进行指针交换，不会有资源的释放与申请。而如果 T 不可移动却是可拷贝的，那么拷贝语义会被用来进行置换。这就跟普通的置换语句是相同的了。因此在移动语义的支持下，我们仅仅通过一个通用的模板，就可能

更高效地完成置换，这对于泛型编程来说，无疑是具有积极意义的。

另外一个关于移动构造的话题是异常。对于移动构造函数来说，抛出异常有时是件危险的事情。因为可能移动语义还没完成，一个异常却抛出来了，这就会导致一些指针就成为悬挂指针。因此程序员应该尽量编写不抛出异常的移动构造函数，通过为其添加一个 noexcept 关键字，可以保证移动构造函数中抛出来的异常会直接调用 terminate 程序终止运行，而不是造成指针悬挂的状态。而标准库中，我们还可以用一个 std::move_if_noexcept 的模板函数替代 move 函数。该函数在类的移动构造函数没有 noexcept 关键字修饰时返回一个左值引用从而使变量可以使用拷贝语义，而在类的移动构造函数有 noexcept 关键字时，返回一个右值引用，从而使变量可以使用移动语义。我们来看一下代码清单 3-23 所示的例子。

代码清单 3-23

```
#include <iostream>
#include <utility>
using namespace std;

struct Maythrow {
    Maythrow() {}
    Maythrow(const Maythrow&) {
        std::cout << "Maythorow copy constructor." << endl;
    }
    Maythrow(Maythrow&&) {
        std::cout << "Maythorow move constructor." << endl;
    }
};

struct Nothrow {
    Nothrow() {}
    Nothrow(Nothrow&&) noexcept {
        std::cout << "Nothorow move constructor." << endl;
    }
    Nothrow(const Nothrow&) {
        std::cout << "Nothorow move constructor." << endl;
    }
};

int main() {
    Maythrow m;
    Nothrow n;

    Maythrow mt = move_if_noexcept(m);  // Maythorow copy constructor.
    Nothrow nt = move_if_noexcept(n);   // Nothorow move constructor.
    return 0;
}
// 编译选项 :g++ -std=c++11 3-3-8.cpp
```

在代码清单 3-23 中，可以清楚地看到 move_if_noexcept 的效果。事实上，move_if_noexcept 是以牺牲性能保证安全的一种做法，而且要求类的开发者对移动构造函数使用 noexcept 进行描述，否则就会损失更多的性能。这是库的开发者和使用者必须协同平衡考虑的。

还有一个与移动语义看似无关，但偏偏有些关联的话题是，编译器中被称为 RVO/NRVO 的优化（RVO, Return Value Optimization，返回值优化，或者 NRVO，Named Return Value optimization)。事实上，在本节中大量的代码都使用了 -fno-elide-constructors 选项在 g++/clang++ 中关闭这个优化，这样可以使读者在代码中较为容易地利用函数返回的临时量右值。

但若在编译的时候不使用该选项的话，读者会发现很多构造和移动都被省略了。对于下面这样的代码，一旦打开 g++/clang++ 的 RVO/NRVO，从 ReturnValue 函数中 a 变量拷贝 / 移动构造临时变量，以及从临时变量拷贝 / 移动构造 b 的二重奏就通通没有了。

```
A ReturnRvalue() { A a(); return a; }
A b = ReturnRvalue();
```

b 变量实际就使用了 ReturnRvalue 函数中 a 的地址，任何的拷贝和移动都没有了。通俗地说，就是 b 变量直接“霸占”了 a 变量。这是编译器中一个效果非常好的一个优化。不过 RVO/NRVO 并不是对任何情况都有效。比如有些情况下，一些构造是无法省略的。还有一些情况，即使 RVO/NRVO 完成了，也不能达到最好的效果。但结论是明显的，移动语义可以解决编译器无法解决的优化问题，因而总是有用的。

3.3.6 完美转发

所谓完美转发（perfect forwarding），是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另外一个函数。比如：

```
template <typename T>
void IamForwording(T t) {    IrunCodeActually(t);    }
```

这个简单的例子中，IamForwording 是一个转发函数模板。而函数 IrunCodeActually 则是真正执行代码的目标函数。对于目标函数 IrunCodeActually 而言，它总是希望转发函数将参数按照传入 Iamforwarding 时的类型传递（即传入 IamForwording 的是左值对象，IrunCodeActually 就能获得左值对象，传入 IamForwording 的是右值对象，IrunCodeActually 就能获得右值对象），而不产生额外的开销，就好像转发者不存在一样。

这似乎是一件非常容易的事情，但实际却并不简单。在上面例子中，我在 IamForwording 的参数中使用了最基本类型进行转发，该方法会导致参数在传给 IrunCodeActually 之前就产生了一次额外的临时对象拷贝。因此这样的转发只能说是正确的转发，但谈不上完美。

所以通常程序员需要的是一个引用类型，引用类型不会有拷贝的开销。其次，则需要考虑转发函数对类型的接受能力。因为目标函数可能需要能够既接受左值引用，又接受右值引

用。那么如果转发函数只能接受其中的一部分，我们也无法做到完美转发。结合表 3-1，我们会想到“万能”的常量左值类型。不过以常量左值为参数的转发函数却会遇到一些尴尬，比如：

```
void IrunCodeActually(int t){}
template <typename T>
void IamForwording(const T & t) {    IrunCodeActually(t);     }
```

这里，由于目标函数的参数类型是非常量左值引用类型，因此无法接受常量左值引用作为参数，这样一来，虽然转发函数的接受能力很高，但在目标函数的接受上却出了问题。那么我们可能就需要通过一些常量和非常量的重载来解决目标函数的接受问题。这在函数参数比较多的情况下，就会造成代码的冗余。而且依据表 3-1，如果我们的目标函数的参数是个右值引用的话，同样无法接受任何左值类型作为参数，间接地，也就导致无法使用移动语义。

那 C++11 是如何解决完美转发的问题的呢？实际上，C++11 是通过引入一条所谓“引用折叠”（reference collapsing）的新语言规则，并结合新的模板推导规则来完成完美转发。

在 C++11 以前，形如下列语句：

```
typedef const int T;
typedef T& TR;
TR& v = 1;     // 该声明在 C++98 中会导致编译错误
```

其中 TR& v = 1 这样的表达式会被编译器认为是不合法的表达式，而在 C++11 中，一旦出现了这样的表达式，就会发生引用折叠，即将复杂的未知表达式折叠为已知的简单表达式，具体如表 3-2 所示。

表 3-2 C++11 中的引用折叠规则

TR 的类型定义	声明 v 的类型	v 的实际类型
T&	TR	A&
T&	TR&	A&
T&	TR&&	A&
T&&	TR	A&&
T&&	TR&	A&
T&&	TR&&	A&&

这个规则并不难记忆，因为一旦定义中出现了左值引用，引用折叠总是优先将其折叠为左值引用。而模板对类型的推导规则就比较简单，当转发函数的实参是类型 X 的一个左值引用，那么模板参数被推导为 X& 类型，而转发函数的实参是类型 X 的一个右值引用的话，那么模板的参数被推导为 X&& 类型。结合以上的引用折叠规则，就能确定出参数的实际类型。进一步，我们可以把转发函数写成如下形式：

```
template <typename T>
```

```
void IamForwording(T && t) {
   IrunCodeActually(static_cast<T &&>(t));
}
```

注意 对于完美转发而言，右值引用并非“天生神力”，只是C++11新引入了右值，因此为其新定下了引用折叠的规则，以满足完美转发的需求。

注意一下，我们不仅在参数部分使用了T &&这样的标识，在目标函数传参的强制类型转换中也使用了这样的形式。比如我们调用转发函数时传入了一个X类型的左值引用，可以想象，转发函数将被实例化为如下形式：

```
void IamForwording(X& && t) {
   IrunCodeActually(static_cast<X& &&>(t));
}
```

应用上引用折叠规则，就是：

```
void IamForwording(X& t) {
   IrunCodeActually(static_cast<X&>(t));
}
```

这样一来，我们的左值传递就毫无问题了。实际使用的时候，IrunCodeActually如果接受左值引用的话，就可以直接调用转发函数。不过读者可能发现，这里调用前的static_cast没有什么作用。事实上，这里的static_cast是留给传递右值用的。

而如果我们调用转发函数时传入了一个X类型的右值引用的话，我们的转发函数将被实例化为：

```
void IamForwording(X&& && t) {
   IrunCodeActually(static_cast<X&& &&>(t));
}
```

应用上引用折叠规则，就是：

```
void IamForwording(X&& t) {
   IrunCodeActually(static_cast<X&&>(t));
}
```

这里我们就看到了static_cast的重要性。如我们在上面几个小节中讲到的，对于一个右值而言，当它使用右值引用表达式引用的时候，该右值引用却是个不折不扣的左值，那么我们想在函数调用中继续传递右值，就需要使用std::move来进行左右值的转换。而std::move通常就是一个static_cast。不过在C++11中，用于完美转发的函数却不再叫作move，而是另外一个名字：forward。所以我们可以把转发函数写成这样：

```
template <typename T>
void IamForwording(T && t) {
   IrunCodeActually(forward(t));
```

```
}
```

move 和 forward 在实际实现上差别并不大。不过标准库这么设计，也许是为了让每个名字对应于不同的用途，以应对未来可能的扩展（虽然现在我们使用 move 可能也能通过完美转发函数的编译，但这并不是推荐的做法）。

我们来看一个完美转发的例子，如代码清单 3-24 所示。

代码清单 3-24

```
#include <iostream>
using namespace std;

void RunCode(int && m) { cout << "rvalue ref" << endl; }
void RunCode(int & m) { cout << "lvalue ref" << endl; }
void RunCode(const int && m) { cout << "const rvalue ref" << endl; }
void RunCode(const int & m) { cout << "const lvalue ref" << endl; }

template <typename T>
void PerfectForward(T &&t) {   RunCode(forward<T>(t));    }

int main() {
    int a;
    int b;
    const int c = 1;
    const int d = 0;

    PerfectForward(a);          // lvalue ref
    PerfectForward(move(b));    // rvalue ref
    PerfectForward(c);          // const lvalue ref
    PerfectForward(move(d));    // const rvalue ref
}
// 编译选项 :g++ -std=c++11 3-3-9.cpp
```

在代码清单 3-24 中，我们使用了表 3-1 中的所有 4 种类型的值对完美转发进行测试，可以看到，所有的转发都被正确地送到了目的地。

完美转发的一个作用就是做包装函数，这是一个很方便的功能。我们对代码清单 3-24 中的转发函数稍作修改，就可以用很少的代码记录单参数函数的参数传递状况，如代码清单 3-25 所示。

代码清单 3-25

```
#include <iostream>
using namespace std;

template <typename T, typename U>
void PerfectForward(T &&t, U& Func) {
```

```
    cout << t << "\tforwarded..." << endl;
    Func(forward<T>(t));
}

void RunCode(double && m) {}
void RunHome(double && h) {}
void RunComp(double && c) {}

int main() {
    PerfectForward(1.5, RunComp);   // 1.5     forwarded...
    PerfectForward(8, RunCode);     // 8       forwarded...
    PerfectForward(1.5, RunHome);   // 1.5     forwarded...
}
// 编译选项：g++ -std=c++11 3-3-10.cpp
```

当然，读者可以尝试将该例子变得更复杂一点，以更加符合实际的需求。事实上，在C++11 标准库中我们可以看到大量完美转发的实际应用，一些很小巧好用的函数，比如make_pair、make_unique 等在 C++11 都通过完美转发实现了。这样一来，就减少了一些函数版本的重复（const 和非 const 版本的重复），并能够充分利用移动语义。无论从运行性能的提高还是从代码编写的简化上，完美转发都堪称完美。

3.4 显式转换操作符

☞ **类别：库作者**

在 C++ 中，有个非常好也非常坏的特性，就是隐式类型转换。隐式类型转换的“自动性”可以让程序员免于层层构造类型。但也是由于它的自动性，会在一些程序员意想不到的地方出现严重的但不易被发现的错误。我们可以先看看代码清单 3-26 所示的这个例子。

代码清单 3-26

```
#include <iostream>
using namespace std;

struct Rational1 {
    Rational1(int n = 0, int d = 1): num(n), den(d) {
        cout << __func__ << "(" << num << "/" << den << ")" << endl;
    }
    int num;    // Numerator（被除数）
    int den;    // Denominator（除数）
};

struct Rational2 {
    explicit Rational2(int n = 0, int d = 1): num(n), den(d) {
        cout << __func__ << "(" << num << "/" << den << ")" << endl;
    }
```

```
    int num;
    int den;
};

void Display1(Rational1 ra){
    cout << "Numerator: " << ra.num <<" Denominator: "<< ra.den <<endl;
}
void Display2(Rational2 ra){
    cout << "Numerator: "<< ra.num <<" Denominator: "<< ra.den<<endl;
}

int main(){
    Rational1 r1_1 = 11;                // Rational1(11/1)
    Rational1 r1_2(12);                 // Rational1(12/1)

    Rational2 r2_1 = 21;                // 无法通过编译
    Rational2 r2_2(22);                 // Rational2(22/1)

    Display1(1);                        // Rational1(1/1)
                                        // Numerator: 1 Denominator: 1
    Display2(2);                        // 无法通过编译
    Display2(Rational2(2));             // Rational2(2/1)
                                        // Numerator: 2 Denominator: 1
    return 0;
}
// 编译选项 :g++ -std=c++11 3-4-1.cpp
```

在代码清单 3-26 中，声明了两个类型 Rational1 和 Rational2。两者在代码上的区别不大，只不过 Rational1 的构造函数 Rational1(int,int) 没有 explicit 关键字修饰，这意味着该构造函数可以被隐式调用。因此，在定义变量 r1_1 的时候，字面量 11 就会成功地构造出 Rational1(11, 1) 这样的变量，Rational2 却不能从字面量 21 中构造，这是因为其构造函数由于使用了关键字 explicit 修饰，禁止被隐式构造，因此会导致编译失败。相同的情况也出现在函数 Display2 上，由于字面量 2 不能隐式地构造出 Rational2 对象，因此表达式 Display2(2) 的编译同样无法通过。

这里虽然 Display1(1) 编译成功，不过如果不是结合了上面 Rational1 的定义，我们很容易在阅读代码的时候产生误解。按照习惯，程序员会误认为 Display1 是个打印整型数的函数。因此，使用了 explicit 这个关键字保证对象的显式构造在一些情况下都是必须的。

不过同样的机制并没有出现在自定义的类型转换符上。这就允许了一个逆向的过程，从自定义类型转向一个已知类型。这样虽然出现问题的几率远小于从已知类型构造自定义类型，不过有的时候，我们确实应该阻止会产生歧义的隐式转换。让我们来看看代码清单 3-27 所示的例子，该例子来源于 C++11 提案。

代码清单 3-27

```
#include <iostream>
using namespace std;

template <typename T>
class Ptr {
public:
    Ptr(T* p): _p(p) {}
    operator bool() const {
        if (_p != 0)
            return true;
        else
            return false;
    }
private:
    T* _p;
};

int main() {
    int a;
    Ptr<int> p(&a);

    if (p)       // 自动转换为bool型，没有问题
        cout << "valid pointer." << endl;    // valid pointer.
    else
        cout << "invalid pointer." << endl;

    Ptr<double> pd(0);
    cout << p + pd << endl; // 1，相加，语义上没有意义
}
// 编译选项 :g++ 3-4-2.cpp
```

在代码清单 3-27 中，我们定义了一个指针模板类型 Ptr。为了方便判断指针是否有效，我们为指针编写了自定义类型转换到 bool 类型的函数，这样一来，我们就可以通过 if(p) 这样的表达式来轻松地判断指针是否有效。不过这样的转换使得 Ptr<int> 和 Ptr<double> 两个指针的加法运算获得了语法上的允许。不过明显地，我们无法看出其语义上的意义。

在 C++11 中，标准将 explicit 的使用范围扩展到了自定义的类型转换操作符上，以支持所谓的“显式类型转换”。explicit 关键字作用于类型转换操作符上，意味着只有在直接构造目标类型或显式类型转换的时候可以使用该类型。我们可以看看代码清单 3-28 所示的例子。

代码清单 3-28

```
class ConvertTo {};
class Convertable {
public:
    explicit operator ConvertTo () const { return ConvertTo(); }
```

```
};
void Func(ConvertTo ct) {}
void test() {
    Convertable c;
    ConvertTo ct(c);          // 直接初始化，通过
    ConvertTo ct2 = c;        // 拷贝构造初始化，编译失败
    ConvertTo ct3 = static_cast<ConvertTo>(c);  // 强制转化，通过
    Func(c);                  // 拷贝构造初始化，编译失败
}
// 编译选项：g++ -std=c++11 3-4-3.cpp
```

在代码清单 3-28 中，我们定义了两个类型 ConvertTo 和 Convertable，Convertable 定义了一个显式转换到 ConvertTo 类型的类型转换符。那么对于 main 中 ConvertTo 类型的 ct 变量而言，由于其直接初始化构造于 Convertable 变量 c，所以可以编译通过。而做强制类型转换的 ct3 同样通过了编译。而 ct2 由于需要从 c 中拷贝构造，因而不能通过编译。此外，我们使用函数 Func 的时候，传入 Convertable 的变量 c 的也会导致参数的拷贝构造，因此也不能通过编译。

如果我们把该方法用于代码清单 3-27 中，可以发现我们预期的事情就发生了，if(p) 可以通过编译，因为可以通过 p 直接构造出 bool 类型的变量。而 p + pd 这样的语句就无法通过编译了，这是由于全局的 operator + 并不接受 bool 类型变量为参数，而 Convertable 也不能直接构造出适用于 operator + 的 int 类型的变量造成的（不过读者可以尝试一下使用 p && pd 这样的表达式，是能够通过编译的）。这样一来，程序的行为将更加良好。

可以看到，所谓显式类型转换并没完全禁止从源类型到目标类型的转换，不过由于此时拷贝构造和非显式类型转换不被允许，那么我们通常就不能通过赋值表达式或者函数参数的方式来产生这样一个目标类型。通常通过赋值表达式和函数参数进行的转换有可能是程序员的一时疏忽，而并非本意。那么使用了显式类型转换，这样的问题就会暴露出来，这也是我们需要显式转换符的一个重要原因。

3.5 列表初始化

☞ **类别：所有人**

3.5.1 初始化列表

在 C++98 中，标准允许使用花括号 "{}" 对数组元素进行统一的集合（列表）初始值设定，比如：

```
int arr[5] = {0};
int arr[] = {1, 2, 3, 4} ;
```

这些都是合法的表达式。不过一些自定义类型，却无法享受这样便利的初始化。通常，