C++23 高级编程

(第 6 版)

(上册)

[比] 马克 • 格雷瓜尔(Marc Gregoire) 著

王志强 张兴 李畅 何荣华 译

清華大學出版社

北 京

北京市版权局著作权合同登记号　图字：01-2024-1476

图书在版编目(CIP)数据

C++23高级编程 : 第6版 / (比) 马克·格雷瓜尔著 ; 王志强等译. -- 北京 : 清华大学出版社, 2025. 6.
ISBN 978-7-302-69398-7

I. TP312.8

中国国家版本馆CIP数据核字第20255MC030号

责任编辑：王　军　韩宏志
封面设计：高娟妮
版式设计：恒复文化
责任校对：马遥遥
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-83470000　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市春园印刷有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张**：65.5　　**字　　数**：1925 千字
版　　次：2025 年 7 月第 1 版　　**印　　次**：2025 年 7 月第 1 次印刷
定　　价：239.00 元(全二册)

产品编号：102405-01

推 荐 序 一

给《C++20 高级编程(第 5 版)》写了推荐序后，时隔 3 年，我很高兴看到更新版的《C++23 高级编程》问世了。

本书延续之前版本备受赞誉的优点，是一本真正从程序员视角出发、专为程序员量身打造的实用教材，能帮助初、中级 C++程序员全面提升其 C++专业技能。作者以深厚的专业功底和丰富的一线编程经验，精心打造了这本佳作，提供了完整的到 C++23 为止的 C++语言介绍，从入门语法、高级技巧到软件工程，做了全方位、无死角的介绍，每个知识点都讲解得细致入微。无论你想系统学习 C++，还是要找其中某些语法点的参考资料，都将收获实实在在的帮助。

与某些 C++教材不同，本书不是简单地在以前的版本上打个小补丁出来卖钱，而是根据当前 C++标准(C++23)进行了全面更新。从最基本的 Hello World 程序，读者就能看到与传统的 C++程序的不同：代码使用了 import std 和 std::println 这两个新特性，而非使用#include 和 std::cout。而后面讨论较新特性的时候，也莫不如此，比如，几乎处处可以看到模块和 println。类似地，另外一些“较新”的 C++特性(如 C++17 的 string_view 和结构化绑定，C++20 的三向比较运算符，等等)，也较早得到介绍，并在书中多次出现。如果你新学 C++的话，可不带历史包袱地看到一门现代的高性能编程语言。如果你之前学过一点 C++，也可细细品味现代 C++带来的不同，特别是模块引入后对代码组织产生的巨大影响。

既然是个新版本，中文书名里又有“C++23”，本书当然要重点描述 C++23 的特性。除了标准库模块和 print/println，C++23 的主要新特性——如显式对象参数、mdspan 和 Unicode 表示改进——书里也都进行了介绍。在描述 C++23 特性时，书的侧边通常会出现特殊的带圈“C++23”标记，非常清晰。

本书的英文主书名与上一版相同，都是 *Professional C++*，自然，它希望你能专业地对待 C++编程。本书的内容编排也十分合理。

第 I 部分包括三章，以较短篇幅介绍了 C++的主要语法和功能，让你快速上手 C++。

第 II 部分的三章不讲语言了，转而讲软件设计。毕竟，我们使用 C++的目的是设计出好的软件。写程序不是工作目的。

第 III 部分是本书的重点，介绍 C++编码方法。该部分占了全书的大部分章节和篇幅，具体讲解 C++中的各个重要特性和库。日常用到的绝大部分功能，都会在这一部分讲到，如内存管理、基本模板、泛型、错误处理、容器、时间工具等。

第 IV 部分较为简短，有三章，描述了一些高级技巧，仍然是 C++的技术内容。

第 V 部分又超越了 C++语言本身，用了整整 7 章的篇幅讨论 C++项目的软件工程问题，从软件生命周期、测试、调试、设计模式等各个角度进行探讨。这一部分的广度大于深度，对于项目经验不多的程序员来说，尤其有用。

要说我对这本书有什么意见的话，最主要就是作者对广度的追求了。对于某些不推荐(如 bind)或不常用的特性，作者仍有所着墨；而对另一些内容(如协程)，作者只是一带而过。因为某些特性不常用，描述中也容易发生问题，如 14.8.3 节的 function-try-blocks 例子中包含错误行为，29.3.2 节的 launder 例子也有更简单的写法。不过，这也是作者自己的定位选择吧。此外，作者给出的建议里，至少有一

项我持保留意见：作者建议把非 final 类的构造函数以外的方法全部设为 virtual——这个建议，从我对 C++的理解看，也许能适用于某些领域，但绝非放之四海皆准。事实上，标准库的类没有一个标成 final，只有少数类中用到了 virtual。

无论如何，作为一本中级教材，本书的表现堪称出色。作者的绝大部分建议都极具实用性和可靠性。相信随着学习程度的加深，读者自会有能力去辨别和领悟其中的精妙之处。

好书要有好的翻译。本书的译者有两位我打过交道，都是热心于 C++知识传播的年轻才俊。初览译稿，便感受到内容相当清晰晓畅，对于这么厚重的一本大块头教材，也是殊为不易了。在此衷心盼望广大 C++学习者都能从此书中受益，成长为更优秀的 C++程序员。

吴咏炜
Boolan 首席技术咨询师
《C++实战：核心技术与最佳实践》作者

推 荐 序 二

《C++高级编程》第 1 版是 Nicholas A. Solter 和 Scott J. Kleper 这两位颇具编程实战开发经验，曾在斯坦福大学任教编程课程，并具有多年 C++应用开发经验的专家撰写的。该书的特点是总结了面向工程开发，特别是 OOP 技术的全套经验，用足了 C++ STL 标准库、设计模式，展示了优秀的代码效率、策略和风格。

《C++高级编程》第 2 版改版正值推出 C++11 标准的大变革前夜，由 Marc Gregoire 接手续写。Marc Gregoire 更是一位 C++大咖，长期跟踪 C++发展前沿，从事 C++应用研究与开发多年，与前两位作者的开发经历十分相似，并因其 C++开发技能而荣获 2007 年度微软 MVP 称号。

《C++高级编程》第 3 版和第 4 版，是 Marc Gregoire 在综合应用开发技能的基础上，分别总结了 C++14 标准和 C++17 标准的特征而写就的。

到了《C++高级编程》第 5 版，中文书名直接改为《C++20 高级编程(第 5 版)》，如此更为贴切，总结了 C++20 标准的特征，又视为 C++应用开发方法上的新里程碑。最新的第 6 版《C++高级编程》，中文书名为《C++23 高级编程(第 6 版)》，总结了很多 C++23 标准的特性，紧跟 C++发展潮流。

王志强(“程序喵大人”主理人)本就是 C++大咖，他辟有专门的编程公众号，常年发表 C++博文，产出文章颇丰，他与我在 C++开发者俱乐部@IncrediClub 群相识，志趣相投，互相仰慕。

《C++23 高级编程(第 6 版)》和《C++20 高级编程(第 5 版)》都由王志强领衔翻译。我激情盈胸，这般心境，无以言表。看了几个样章，就觉得翻译语词确切而通畅，对于 C++编程爱好者极易产生喜感。

学 C++难，难在学了一些 C++的点点滴滴，却不知如何上手编程应用，不知融会贯通，此书便可起到点拨的作用。看王志强之译作，入其所在 C++开发者群，得其群助而诚可谓众人拾柴火焰高，必将事半而功倍。

钱能
浙江工业大学教授

推 荐 序 三

对于 C++社区而言，2011 年是个分水岭。那一年发布的 C++11 标准打破了此前长达十余年的版本沉寂。那一年 Bjarne Stroustrup 访华时，我在北京采访了他，并从他那里确认 C++的版本发布将进入以三年为周期的“火车发车模式”；即从 2011 年起，每三年必发布新版本，而所有赶不上某一版本的语言特性将延迟至下一版本或更后面的版本发布。换言之，以 Bjarne 为首的 C++标准委员会对于长期以来“版本等特性”造成的版本发布一再延误及由此带来的语言市场份额丢失痛定思痛，宁愿以“特性等版本”的做法矫枉过正。也是从那个时候起，C++的发展驶入快车道，也确实每三年发布一个新版本，目前最新的版本是 C++23，而 C++26 也早已开始了关于哪些新特性会进入的热烈讨论，并对这些新特性展开了范围广泛的测试和反馈。而从 C++11 开始的版本，被统称为“现代 C++”，以彰显与旧时代的分道扬镳。

的确，现代 C++带来了太多强大的新特性，这些新特性从核心语法层次深深地改变了这个有着近 40 年历史的语言。通过 auto 和 decltype，C++甚至在撰写第一行代码时也将触及泛型，而 lambda 表达式更是将函数式编程范式引入了语言。接受和适应此前不熟悉的编程范式对于任何程序员来说，都是一项巨大的挑战。更不用说诸如右值引用带来的观念冲击，以及 constexpr 对于变量生存期和生命周期的重新认识等等。即使是像静态成员变量默认值这样看起来人畜无害者，也能悄悄地引发关于源码文件组织上不小的策略变动。C++从来就以复杂厚重为特点，而现代 C++的学习更是急切地需要一本全面、系统、及时更新的教材，才能满足广大 C++社区成员的普遍需要。靠直接阅读标准文档，对于大多数人来说是不现实的，正如你不可能通过阅读辞典来写出好文章。而目前比较流行的权威教材，如 Bjarne 本人的《C++程序设计语言》和 Stanley Lippman 的 *C++ Primer*，都已经久未更新，或未译成简体中文版。所以，在此时此刻，甚至数年之内，Marc Gregoire 的这本《C++23 高级编程(第 6 版)》都会是一本内容全、材料新、技术权威的 C++理想教材。

我与 Marc 在 CppCon 上见过，并有幸聆听过他关于“设计和实现与 STL 良好配合的算法与数据结构”的演讲。今年的 CppCon 上他的演讲环节更是构成了主旨演讲的重要组成部分，可见他在 C++标准中形成的贡献之巨。而这样一部鸿篇巨著，其翻译也所托得人。我与其中的何荣华同学借助网络平台彼此结识，至今已逾多年。现在何荣华也是使用 C++作为日常工作的开发语言，以开发自动驾驶底层算法，可以说他的背景与 Marc 的早期开发经历相比甚至更具优势，因而也非常有潜力成为未来的 C++标准委员会成员。通过阅读样章，我也感觉无论是技术准确性还是语言通畅性方面，译者团队都属于一流水准。作为《C++覆辙录》的译者，我当然感同身受地知道，译出这样的大部头需要投入怎样巨量的时间和心血。在此，诚挚地向广大 C++社区成员和 C++爱好者们推荐这部质量上乘的优秀作品。

高博，卷积传媒创始人
《C++覆辙录》译者

推 荐 序 四

C++标准委员会从 2011 年开始，每三年定期发布新标准，稳步增添语言特性，不断扩充标准库。光阴飞逝，C++11 标准发布至今已十年余，而 C++23 也于最近正式面世，语言特性越来越丰富。若 C++使用者有志于技术精进，则持续学习是不二途径。《C++23 高级编程(第 6 版)》恰如称手兵刃，能助读者在研习的征途上披荆斩棘。

《C++高级编程》历经 6 版修订，每一版本都更新相当篇幅，始终紧随技术发展趋势，与业界发展保持高度同步。目前涵盖 C++23 标准的书籍为数不多，本书正是其中之一。在前面 I~IV 部分，先以 C++的核心基础为切入点快速起步，后续章节则对 C++11~ C++23 的新特性施以浓墨重彩的描绘，系统、全面地逐一展开讲解。书中文字平实，简明易懂；所举代码范例质量上乘，并且紧密贴合真实世界场景，而非与现实脱节的学院风格，值得认真揣摩仿效。作者还特意编排对比了 C++的传统特性与新特性，譬如内存管理与智能指针、迭代器与范围、函数对象/函数指针与 lambda 等，让读者既能纵观语言全貌又能领会特性精华。更值得称许的是，这些部分并未局限于 C++语言层面，还穿插了编码风格、代码复用和程序设计方法论等议题，引领读者提升编程的思维水平。

本书仅凭上述内容便足以领先，又因第 V 部分而更加脱颖超群。作者借 C++为依托，在最后几个章节介绍了编程最佳实践的各种议题，包括软件工程方法、性能调优、测试除错、设计模式和框架开发等。这些领域举足轻重，且与编程开发密不可分，却鲜有编程书籍着墨，而往往以独立专著论述。第 V 部分承接前文对 C++本身的讲解说明，向读者一鼓作气深入揭示代码背后的设计思想和一系列工程方法。我们学习 C++，目的是为了编写规范代码，构建高质量软件。倘若拘泥于 C++语言自身，仅掌握语法和运用特性，恐怕只能胜任最底层的编程任务。必须提升思想高度，超脱单纯的编程语言，采纳行之有效的设计理念，并运用工程手段。本书正好概括了编程实践经验的精髓，提供了学以致用的最佳指南，为读者开启了更高、更宽广的视角。一旦领悟本书阐述的思想，肯定能受益，明显提升工程能力，并为进一步研读相关专著奠定良好基础。

作者 Gregoire 先生长期从事软件开发，经验老到，本书凝聚了他(和以往版本合作者)的学识和经验。另一方面，四位译者都对 C++语言怀有巨大热忱，他们当中既有在职 C++程序员，又有计算机专业的在读研究生，前者在实际工作中直接运用新标准的 C++特性，深谙各项技术细节，后者能从学习者的视角考虑译文的表达和遣词造句，四人合作促进了对原书的精准理解，从而保证了高品质翻译，相信读者必能从本书获得相当不错的学习体验。

吴天明
资深软件工程师

推 荐 序 五

在学习使用 C++的这些年里，我注意到 C++的语言、它的应用场景和C++程序员中有趣的相似与同构特征。C++这门语言特性丰富、历史包袱/积累厚重，全球 440 万 C++开发者使用它，构造了计算机工业的编译器、数据库、操作系统、求解器等基石，也夜以继日地为工业、科学、车船、生物医药等领域提供计算和控制的支撑。编程语言特性的丰富、应用场景的广泛、语言社区的多元等因素相互作用，推动了 C++的演化，出现了 STL 这样的“天成”之作，造就了支持多种编程范式的“现代”C++，也培养了众多如作者 Marc Gregoire 和译者何荣华这样的优秀 C++程序员。

要将 C++这门语言用好，在参考书籍的选择方面值得动一点心思。当然，C++之父 Bjarne Stroustrup 撰写的 TCPPPL 和 Scott Meyers 撰写的 Effective 系列属于必读。不过对于 C++这样的语言，只选择几本公认的权威经典参考书恐怕还不足以覆盖专业程序员的日常参考需求。本书的英文原名为 *Professional C++, Sixth Edition*，读者如果浏览过本书目录，再随机翻开读上几页，就不难发现，这首先是一本优秀的专业案头参考书，适合放在键盘旁随时翻阅。Pro，意味着专业和职业。本书从一线编程实践的角度，结合代码实例，全面展示了 C++语言特性。而更有价值的是，作者把 C++的语言特性，包括库和相关的工具和方法论放到软件工程的大背景下，结合语言探讨了建模、测试、设计和发布等诸多话题。说是探讨，可能不大贴切，本书的语言风格平实精炼，更像工作中 C++高手和专家针对某个具体问题给出的简短而确定的意见，点到为止，讲求实效。

作为 C++的同龄人，也作为用 C++谋生和创作的码农，我再次感谢何荣华老师和他的合作者们将这本著作翻译出来，让它能帮助更多的同行写出更多专业而优美的 C++代码。也在此期待 C++26 后本书的再次更新。

杨文波
资深嵌入式软件工程师

译 者 序

本书堪称经典之作，内容详实且丰富多元，不仅涵盖 C++基础知识，还深入剖析了诸多 C++高级特性，特别是 C++23 的新特性。目前介绍 C++基础知识的书籍很多，但聚焦于 C++23 新特性的书籍却寥寥无几，而既介绍 C++基础知识又介绍 C++23 新特性的书更是凤毛麟角。本书还重点介绍很多编程哲学，包括 C++的设计方法论，从专业角度分析 C++的编程艺术，并介绍 C++的软件工程和调试技术。可以说本书的出版是 C++开发人员的福音，本书既适合新手学习 C++基础知识，又适合中高级开发者实现技术进阶，是 C++领域一本不可多得的佳作。

近十年来，C++引入了很多新特性，有 C++11 新特性、C++14 新特性、C++17 新特性、C++20 新特性，近期又更新了 C++23 新特性。作为一名 C++程序员，很有必要了解语言最新的变革。读者在学习本书 C++23 新特性的时候，可以多做一层思考，思考为什么标准委员要引入此新特性。

若想跻身资深 C++开发人员之列，必须扎实理解 C++语言的底层原理，了解编程哲学、软件工程方法论、设计、编码、测试、调试和优化等。令人欣喜的是，本书恰好涵盖了这些至关重要的知识，为有志于成为资深 C++开发人员的学习者提供了全面且系统的指引。

本书包括 6 部分。第 I 部分是专业的 C++简介，第 II 部分介绍专业的 C++软件设计，第 III 部分从专业角度分析 C++编码方法，第 IV 部分讲解如何真正掌握 C++的高级特性，第 V 部分重点介绍 C++软件工程技术。最后，在附录部分提供了 C++技术面试、参考文献、 C++标准头文件和 UML 简要介绍。

近年来，软件开发领域掀起一股由人工智能驱动的新浪潮，术语“Vibe Coding”逐渐进入大众视野。为方便读者学习，本书特别附赠文档“Vibe Coding 浪潮下的 C++：审视、比较、适应与未来”，以深入探讨 Vibe Coding 的核心理念，帮助 C++开发者和团队理解如何在 AI 时代合理地、负责任地利用新工具，同时坚守 C++开发的核心原则。

对于这本经典之作，译者在翻译过程中始终秉持严谨态度，力求精准忠实于原文，深度再现原文风格。但鉴于译者水平有限，失误在所难免，如有任何意见和建议，请不吝指正。

感谢清华大学出版社编辑的精心组稿、认真审阅和细心修改，感谢妻子和父母在各个方面的支持和理解。

最后，希望读者通过阅读本书能在 C++领域有更深的造诣，深度领略 C++语言的独特魅力，进而达成“精通”C++的目标。

译者

作 者 简 介

Marc Gregoire 是一位软件项目经理/软件架构师，深耕 C/C++开发，尤精 Microsoft VC++及 MFC 框架，拥有开发 7×24 小时运行于 Windows 和 Linux 平台的 C++程序的经验(如 KNX/EIB 家庭自动化软件)。除了 C/C++，Marc 也擅长 C#。

Marc 是比利时 C++用户组创始人，畅销技术图书 *Professional C++*(第 2~6 版)的作者，*C++ Standard Library Quick Reference*(第 1~2 版)的共同作者，多家出版社多部技术书籍的特约编辑，CppCon C++大会常驻演讲嘉宾，CodeGuru 论坛成员(用户名：Marc G)。自 2007 年以来，他凭借在 Visual C++领域的技术影响力，连续十多年荣获微软 MVP 年度奖项。

Marc 毕业于比利时鲁汶大学，先后获得计算机科学工程硕士学位和 AI 专业的高级硕士学位。职业生涯初期，Marc 加入比利时软件咨询公司 Ordina，担任技术顾问，主导开发 Siemens 和 Nokia Siemens Networks 中面向电信运营商的关键 2G/3G 系统 (基于 Solaris 平台)，项目团队横跨南美、美国、欧洲、中东、非洲及亚洲多地。Marc 现任职于精密光学仪器与工业检测技术领军企业尼康计量(Nikon Metrology)，负责 X 射线、CT 及三维几何检测领域的软件架构设计与项目管理。

技术编辑简介

Bradley Jones 精通多种编程语言和工具，从 C 语言到 Unity，从 Windows 到移动设备平台(包括 Web 开发)，甚至还涉足一些虚拟现实和嵌入式设备开发。除了编程，Bradley 还撰写了关于 C、C++、C#、Windows、Web 等众多技术主题的书籍，以及一些非技术主题的书籍。Bradley 是 Lots of Software 公司的创始人，并因其在行业内的影响力获得广泛认可。Bradley 曾被授予微软 MVP 称号，担任 CODiE 奖评委，是国际技术演讲者，拥有多重身份和荣誉。

Arthur O'Dwyer 是一位专业的 C++培训师、软件工程师、作家，以及 WG21 委员会成员。Arthur 是《精通 C++17 STL》一书的作者，创立了 CppCon 的“回归基础”专题(2019)，实现了 libc++的 <memory_resource>头文件(2022)，并为 C++20 和 C++23 中的简化“隐式移动”语义做出了贡献。他与妻子居住在纽约。

致　　谢

我要感谢 John Wiley & Sons 的编辑和制作团队对本书的支持。特别感谢组稿编辑 Jim Minatel 给了我撰写本书第 6 版的机会；感谢资深管理编辑 Pete Gaughan、管理编辑 Ashirvad Moses Thyagarajan、项目经理 Kathryn Hogan 博士、内容优化专家 Archana Pragash 及文稿编辑 Kim Wimpsett。

特别感谢技术编辑 Bradley Jones 和 Arthur O'Dwyer，他们对本书的技术准确性进行了核实。他们的反馈和大量贡献使本书更加完善，我对此深表感谢。

当然，我的父母和弟弟的支持与耐心对本书的完成也起到了至关重要的作用。我还要衷心感谢我的雇主 Nikon Metrology 在完成本项目期间给予我的支持。

最后，我要感谢你们——读者们，感谢你们多年来支持我的工作，并认可我在专业 C++软件开发方面的这种写作方法，感谢你们对本书多个版本的持续关注与支持。

——Marc Gregoire

前　言

丹麦计算机科学家 Bjarne Stroustrup 于 1982 年发明了 C++；C++继承于 C，同时引入了类。1985 年，发布了第一版的“C++程序设计语言”。第一个标准化版本的 C++在 1998 年发布，称为 C++98。在 2003 年，C++03 发布并包含了一些小的更新。在那之后，C++沉默了一段时间，但吸引力开始慢慢增强，导致该语言在 2011 年进行了重大更新，称为 C++11。从那以后，C++标准委员会以 3 年为周期发布更新的版本，出现了 C++14、C++17、C++20 及现在的 C++23。总之，2023 年发布了 C++23 之后，C++已经将近 40 岁了，并且仍然很强大。在 2023 年的大多数编程语言排名中，C++都排在前 4 位。它被广泛用于各种硬件，从带有嵌入式微处理器的小型设备一直到超级计算机。除了广泛的硬件支持，C++还可以用来完成几乎任何编程工作，包括移动平台上的游戏、对性能要求极高的人工智能(AI)和机器学习(ML)软件、自动驾驶汽车的组件、实时 3D 图形引擎、底层硬件驱动程序、完整的操作系统、网络设备的软件栈、网页浏览器等。C++程序很难与任何其他编程语言相匹配，因此，多年来，C++都是编写性能卓越、功能强大的企业级面向对象程序的事实标准语言。大型科技公司，如微软、Facebook、亚马逊、谷歌等，使用用 C++编写的服务来运行其基础设施。尽管 C++语言已经风靡全球，但这种语言难以完全掌握。专业 C++程序员使用一些简单但高效的技术，这些技术并未出现在传统教材中；即使是经验丰富的 C++程序员，也未必完全了解 C++中某些很有用的特性。

编程书籍往往重点描述语言的语法，而不是语言在真实世界中的应用。典型的 C++教材在每一章中介绍语言中的大部分知识，讲解语法并列举示例。本书不遵循这种模式。本书并不讲解语言的大量细节并给出少量真实世界的场景，而是教你如何在真实世界中使用 C++。本书还会讲解一些鲜为人知的让编程更简单的特性，以及区分编程新手和专业程序员的编程技术。

读者对象

就算使用 C++已经多年，你仍可能不熟悉 C++的一些高级特性，或仍不具有使用这门语言的全面能力。也许你编写过实用的 C++代码，但还想学习更多有关 C++中设计和良好编程风格的内容。也许你还不太了解最新版本 C++23 中引入的所有新特性。也许你是 C++新手，想在入门时就掌握“正确”的编程方式。本书能满足上述需求，将你的 C++技能提升到专业水准。

因为本书专注于将你从对 C++具有基本或中等了解水平蜕变为一名专业 C++程序员，所以本书假设你对该语言具有一定程度的认识。第 1 章涵盖 C++的一些基础知识，可以当成复习材料，但是不能替代实际的语言培训和语言使用手册。如果你刚开始接触 C++，但有十分丰富的 C、Java 或 C#语言经验，那么应该能从第 1 章获得所需的大部分知识。

不管属于哪种情况，都应该具有很好的编程基础。应该知道循环、函数和变量。应该知道如何组织一个程序，而且应该熟悉基本技术，例如递归。应该了解一些常见的数据结构(如队列)及有用的算法(如排序和搜索)。不需要预先了解有关面向对象编程的知识——这是第 5 章讲解的内容。

你还应该熟悉开发代码时使用的编译器。稍后将简要介绍 Microsoft Visual C++和 GCC 这两种编译器。要了解其他编译器，请参阅编译器自带的指南。

本书主要内容

阅读本书是学习 C++语言的一种方法，通过阅读本书既能提升编码质量，又能提升编程效率。本书贯穿对 C++23 新特性的讨论。这些新的 C++特性并不是独立在某几章中，而是穿插于全书，在有必要的情况下，所有例子都已更新为使用这些新特性。

本书不仅讲解 C++语法和语言特性，还强调编程方法论、可重用的设计模式以及良好的编程风格。本书讲解的方法论覆盖整个软件开发过程——从设计和编码，到调试以及团队协作。这种方法可让你掌握 C++语言及其独有特性，还能在大型软件开发中充分利用 C++语言的强大功能。

想象一下有人学习了 C++的所有语法但没有见过一个使用 C++例子的情形。他所了解的知识会让他处于非常危险的境地。如果没有示例的引导，他可能认为所有源代码都要放在程序的 main()函数中，还可能认为所有变量都应该为全局变量——这些都不是良好的编程实践。

专业的 C++程序员除了理解语法外，还要正确理解语言的使用方式。他们知道良好设计的重要性、面向对象编程的理论及使用现有库的最佳方式。他们还开发了大量有用的代码并了解可重用的思想。

通过阅读和理解本书的内容，你也能成为一名专业的 C++程序员。你在 C++方面的知识会得到扩充，将接触到鲜为人知和常被误解的语言特性。你还将领略面向对象设计，掌握卓越的调试技能。最重要的或许是，通过阅读本书，你的头脑中有了大量“可重用”思想，可将这些思想贯彻到日常工作中。

有很多好的理由让你努力成为一名专业的 C++程序员，而非只是泛泛了解 C++。了解语言的真正工作原理有助于提升代码质量。了解不同的编程方法论和过程可让你更好地和团队协作。探索可重用的库和常用的设计模式可提升日常工作效率，并帮助你避免白费力气去做重复的工作。所有这些学习课程都在帮助你成为更优秀的程序员，同时成为更有价值的雇员。

本书结构

本书包括 6 部分。

第 I 部分“专业的 C++简介”是 C++基础速成教程，能确保读者掌握 C++的基础知识。在速成教程后，该部分深入讨论字符串和字符串视图的使用，因为字符串在示例中应用广泛。该部分的最后一章介绍如何编写清晰易读的 C++代码。

第 II 部分“专业的 C++软件设计”介绍 C++设计方法论。你会了解设计的重要性、面向对象方法论和代码重用的重要性。

第 III 部分“C++编码方法”从专业角度概述 C++技术。你将学习在 C++中管理内存的最佳方式，如何创建可重用的类，以及如何利用重要的语言特性，例如继承。你还会学习输入输出技术、错误处理、字符串本地化和正则表达式的使用，学习如何利用模块组织可重用的代码。该部分还会讨论如何实现运算符重载，如何编写模板，如何使用概念限制模板参数，以及如何解锁 lambda 表达式和函数对象的功能。该部分还解释了 C++标准库，包括容器、迭代器、范围和算法。在该部分你还将了解标准中提供的一些附加库，例如用于处理时间、日期、时区、随机数和文件系统的库。

第 IV 部分“掌握 C++的高级特性”讲解如何最大限度地使用 C++。该部分揭示 C++中神秘的部分，并描述如何使用这些更高级的特性。在该部分你将学习如何定制和扩充标准库以满足自己的需求、高级模板编程的细节(包括模板元编程)，以及如何通过多线程编程来充分利用多处理器和多核系统。

第 V 部分“C++软件工程”重点介绍如何编写企业级质量的软件。在这部分你将学习当今编程组织的工程实践，如何编写高效的C++代码，软件测试概念(如单元测试和回归测试)，C++程序的调试技术，如何在自己的代码中融入设计技术、框架和概念性的面向对象设计模式，跨语言和跨平台代码的解决方案等。

第 VI 部分是四个附录。附录 A 列出在 C++技术面试中取得成功的指南，附录 B 是带注解的参考文献列表，附录 C 总结 C++标准中的头文件，附录 D 简要介绍 UML(Unified Modeling Language，统一建模语言)(附录 A~D 可通过扫描封底二维码获取)。

本书没有列出 C++中每个类、方法和函数的参考。Peter Van Weert 和 Marc Gregoire 撰写的 *C++17 Standard Library Quick Reference* 是 C++17 标准库提供的所有重要数据结构、算法和函数的浓缩版。附录 B 列出了更多参考资料。下面是两个很好的在线参考资料。

```
cppreference.com
```

可使用这个在线参考资料，也可下载其离线版本，在没有连接到互联网时使用。

```
cplusplus.com/reference/
```

本书正文中提到“标准库参考资料”时，就是指上述 C++参考资料。

下面是其他的优质在线资源：

```
github.com/isocpp/CppCoreGuidelines
```

《C++核心指南》由 C++语言发明人 Bjarne Stroustrup 牵头撰写。指南的目的是帮助人们有效地使用现代 C++。这些指导方针侧重于较高级别的问题，如接口、资源管理、内存管理和并发。

```
github.com/Microsoft/GSL
```

这是微软的指南支持库(GSL)的一个实现，它包含了 C++核心指南使用的函数和类型。这是一个只有头文件的库。

```
isocpp.org/faq
```

这是一个频繁被提问的 C++问题的庞大集合。

```
stackoverflow.com
```

可以在这里搜索常见编程问题的回答，或者提出你自己的问题。

使用本书的条件

要使用本书，只需要一台带有 C++编译器的计算机。本书只关注 C++中的标准部分，而没有任何编译器厂商相关的扩展。

任何 C++编译器

可使用任意 C++编译器。如果还没有 C++编译器，可下载一个免费的。这有许多选择。例如，对于 Windows，可下载 Microsoft Visual Studio Community Edition，这个版本免费且包含 Visual C++；对于 Linux，可使用 GCC 或 Clang，它们也是免费的。

下面将简要介绍如何使用 Visual C++和 GCC。可参阅相关的编译器文档了解更多信息。

编译器与 C++23 功能支持

本书包含 C++23 标准引入的新功能。在撰写本书时，还没有编译器可以完全支持 C++23 的所有新功能。某些新功能仅由某些编译器支持，而其他编译器不支持，而有些功能尚不受任何编译器支持。编译器厂商正在努力支持所有新功能，我相信不久就会有完全符合 C++23 标准的编译器可用。可以在 en.cppreference.com/w/cpp/compiler_support 上查看哪些编译器支持哪些功能。

编译器与 C++模块支持

在撰写本书时，还没有编译器可以完全支持 C++的模块。不过，所有主流编译器至少部分支持。本书在各个地方都使用了模块。如果你的编译器尚不支持模块，可以将模块代码转换为非模块代码，具体方法在第 11 章中有简要说明。

示例：Microsoft Visual C++ 2022

首先需要创建一个项目。启动 Visual C++ 2022，在欢迎界面上，单击 Create A New Project 按钮。如果没有出现欢迎界面，单击 File | New | Project。在 Create A New Project 对话框中，使用 C++、Windows 和 Console 标签，找到 Console App 项目模板，然后单击 Next 按钮。指定项目的名称、保存位置，单击 Create 按钮。

加载新项目后，就会在 Solution Explorer 中看到项目文件列表。如果这个停靠窗口不可见，可选择 View | Solution Explorer。一个新创建的项目会包括一个名为<projectname>.cpp 的文件，就在 Solution Explorer 中 Source Files 部分的下方，可以在该文件中开始编写 C++代码。如果想要编译源代码文件(扫描封底二维码获取本书源代码压缩文件)，则必须在 Solution Explorer 中选择<projectname>.cpp 文件并将其删除。在 Solution Explorer 中右击项目名，再选择 Add | New Item 或 Add | Existing Item，就可以给项目添加新文件或已有文件。

在撰写本书期间，Visual C++ 2022 尚未自动启用 C++23 功能。要启用 C++23 功能，可在 Solution Explorer 窗口中右击项目，然后单击 Properties。在 Properties 窗口中，选择 Configuration Properties | General，根据使用的 Visual C++版本，将 C++ Language Standard 选项设置为 ISO C++23 Standard 或 Preview | Features from the Latest C++ Working Draft，并单击 OK 按钮。

最后，使用 Build | Build Solution 编译代码。没有编译错误后，就可以使用 Debug | Start Debugging 运行了。

注意：

Microsoft Visual C++完全支持模块，包括 C++23 标准中的命名模块 std。

示例：GCC

用自己喜欢的任意文本编辑器创建源代码，保存到一个目录下。要编译代码，可打开一个终端，运行如下命令，指定要编译的所有.cpp 文件：

```
g++ -std=c++2b -o <executable_name> <source1.cpp> [source2.cpp ...]
```

-std=c++2b 用于告诉 GCC 启用对 C++23 功能的支持。当 GCC 完全兼容 C++23 后，这个选项将改为-std=c++23。

模块支持

在 GCC 中，使用-fmodules-ts 选项可以启用对模块的支持。

在撰写本书时，GCC 尚不支持 C++23 标准中引入的命名模块 std(在第 1 章中介绍)。为了使这类代码能够编译，你需要将 import std 声明替换为单个标准库头文件的 import 声明。完成替换后，对于标准库头文件的 import 声明(例如以下内容)，你需要对它们进行预编译：

```
import <iostream>;
```

这是一个预编译<iostream>的示例：

```
g++ -std=c++2b -fmodules-ts -xc++-system-header iostream
```

例如，第 1 章中的 AirlineTicket 代码使用了模块。为了使用 GCC 编译它，首先将 std::println() 替换为 std::cout，因为在撰写本书时，GCC 尚不支持 <print> 功能。之后，将 import std; 声明替换为适当的 import 声明，在这个例子中是 <string> 和 <iostream>。你可在可下载的源代码归档中的 Examples\Ch00\AirlineTicket 目录中找到已适配的代码。

接下来，编译标准库头文件<iostream>和<string>：

```
g++ -std=c++2b -fmodules-ts -xc++-system-header iostream
g++ -std=c++2b -fmodules-ts -xc++-system-header string
```

编译模块接口文件：

```
g++ -std=c++2b -fmodules-ts -c -x c++ AirlineTicket.cppm
```

最后，编译应用代码：

```
g++ -std=c++2b -fmodules-ts -o AirlineTicket AirlineTicket.cpp
AirlineTicketTest.cpp AirlineTicket.o
```

当其通过编译后，你可以这样运行它：

```
./AirlineTicket
```

注意：

使用 GCC 编译 C++ 代码时，采用 C++ 模块的过程可能在未来发生变化。同时，C++23 标准中的命名模块 std 将得到支持。届时，请查阅 GCC 文档，了解如何编译此类代码的更新流程。

C++23 的打印范围支持

第 2 章描述了你可以轻松地将标准库容器(如 std::vector)的整个内容打印到屏幕上。这是自 C++23 引入的新特性，在撰写本书时，并非所有编译器都已支持此功能。

例如，第 2 章解释了你可以按如下方式输出 std::vector 的内容。如果你还不理解所有语法，没关系，到第 2 章结束时你就会掌握。

```
std::vector values { 11, 22, 33 };
std::print("{:n}", values);
```

这将输出：

```
11, 22, 33
```

如果你的编译器尚不支持使用 std::print() 打印容器内容的 C++23 功能，你可将代码的第二行改

为以下内容：

```
for (const auto& value : values) { std::cout << value << ", "; }
```

这将输出：

```
11, 22, 33
```

同样，如果你现在还不理解语法，别担心，到第 2 章结束时一切都会变得清晰。

配套下载文件

读者在学习本书中的示例时，可以手动输入所有代码，也可使用本书附带的源代码文件。然而，我建议手动输入所有代码，这对于学习和你的记忆都是有益的。本书使用的所有源代码都可以扫描封底二维码下载。

下载代码后，只需要用自己喜欢的解压缩软件进行解压缩即可。

另外，读者可扫描封底二维码，下载本书附录(附录 A～D)和本书习题答案。

目　录

——以下内容可扫封底二维码下载——

第 VI 部分 附录

第I部分
专业的 C++简介

第1章 C++和标准库速成

本章内容

- 简要回顾C++语言和标准库(Standard Library)最重要的部分及语法
- 如何写一个基本的类
- 作用域解析的基本原理
- 什么是统一初始化
- const的用法
- 什么是指针、引用、异常及类型别名
- 类型推导基础

本章旨在对C++最重要的部分进行简要描述，使你在阅读本书后面的内容之前掌握一定的基础知识。本章不会全面讲述C++编程语言或标准库。一些基本知识(如什么是程序和递归)，以及一些深奥的知识(例如，什么是union和volatile关键字)也会被忽略。此外忽略了与C++关系不大的C语言部分，这些内容将在后续章节详细讨论。

本章主要讲述日常编程中会遇到的部分C++知识。如果你刚接触C++，不了解什么是引用变量，那么可通过阅读本章的内容来了解此类变量的用法。本章还将讲述使用标准库中功能的基础知识，例如vector容器、optional值和string对象。本章将简要介绍标准库的这部分内容，这样我们从本书的一开始就可以在例子中使用这些现代特性。

如果你对C++非常熟悉，请快速浏览本章的内容，看看有没有自己需要复习的某些C++语言基本内容。如果你刚开始接触C++，请仔细阅读本章内容并务必理解所有示例。如果需要更多的介绍性信息，请参阅附录B。

1.1 C++速成

C++语言经常被认为是“更好的C语言”或“C语言的超集”，主要被设计为面向对象的C语言，常被称为“包含类的C”。后来，在设计C++时，对C语言中许多使用不便或不够精细的内容进行了处理。由于C++是基于C的，如果你有C语言编程经验，将发现本节介绍的许多语法非常熟悉。当然，这两种语言并不一样。例如，C23标准规范文档的页数略小于800页，而C++23标准规范文

档则超过 2000 页。因此，如果你是一名 C 程序员，而且你来自其他语言，如 Java、C#、Python 等，请注意新的或不熟悉的语法。

1.1.1 小程序“Hello World”

下面的代码可能是你遇到的最简单的 C++程序。如果你正在使用旧版本的 C++，import std;和 std::println()可能不被支持，这种情况下，需要使用稍后讨论的替代方案。

```
// 01_helloworld.cpp
import std;

int main()
{
    std::println("Hello, World!");
    return 0;
}
```

正如预期的那样，这段代码会在屏幕上输出“Hello, World！”这一信息。这是一个非常简单的程序，好像不会赢得任何赞誉，但是这个程序确实展现出与 C++程序格式相关的一些重要概念。

- 注释
- 模块导入
- main()函数
- 打印文本
- 从函数返回

下面简要描述这些概念。

1. 注释

这个程序的第一行是注释，这只是供编程人员阅读的一条消息，编译器会将其忽略。在 C++中，可通过两种方法添加注释。可以使用两条斜杠表明这一行中在它后面的内容都是注释。

```
// 01_helloworld.cpp
```

使用多行注释也可以实现这个行为(也就是说，两者没什么不同)。多行注释以/*开始，以*/结束。下面的代码使用了多行注释。

```
/*  This is a multiline comment.
    The compiler will ignore it.
 */
```

第 3 章将详细讲述注释。

2. 模块导入

C++20 中最重要的新特性之一就是对模块(module)的支持，用来替代之前所谓的头文件机制。如果你想要使用某个模块中的功能，则需要导入这个模块。这是通过一条 import 声明做到的。从 C++23 开始，可以通过导入一个名为 std 的标准命名模块来访问整个 C++标准库。Hello World 小程序的第一行导入了名为<iostream>的模块，它声明了 C++提供的输入输出机制。

```
import std;
```

如果此程序没有导入该模块，它就无法实现打印文本这项唯一的功能。

如果没有 C++23 的标准命名模块支持，则必须显式导入代码所需的所有单个头文件。由于标准库中有 100 多个头文件，因此并不总是很容易知道需要导入哪个特定的头文件才能使用某个功能。作为参考，附录 C“标准库头文件”列出了 C++标准库的所有头文件，包括对其内容的简短描述。例如，可以只导入代码真正需要的头文件，而不是在“Hello World”应用程序中导入 std 模块。在这个例子中，代码只需要导入<print>即可访问文本打印功能。请注意，在导入命名模块 std 时，不需要使用尖括号，但在导入单个头文件时，需要按如下方式使用尖括号：

```
import <print>;
```

由于本书是关于 C++23 的书，会经常使用模块。C++标准库提供的所有功能都在定义好的头文件中。本书中的大多数示例只是导入 std 模块，而不是单个头文件，但总是提到在哪个头文件中提供了某些功能。

模块不限于标准库功能。可以编写自己的模块来提供自定义类型和功能，你将在本书中学习如何做到这一点。

注意：

如果你的编译器还不支持模块，只需要简单地将 import 声明替换为#include 预处理指令，我们将在接下来的章节中讨论。

3. 编译器如何处理源代码

简单来说，生成一个 C++程序共有 3 个步骤。从技术角度看，编译过程还涉及其他一些阶段，但就现在而言，理解下面这个简化后的流程就足够了。

(1) 首先，代码通过预处理器运行，预处理器识别有关代码的元信息并处理预处理器指令，如#include 指令。所有预处理器指令都已被处理的源文件称为翻译单元。

(2) 接下来，所有翻译单元都被独立编译成机器可读的目标文件，其中对函数等的引用尚未定义。

(3) 解析这些引用是在最后阶段由链接器完成的，它将所有目标文件链接在一起，形成最终的可执行文件。

C++23

注意：

从 C++23 开始，标准要求 C++编译器接受以 UTF-8 编码保存的源代码文件。第 21 章“字符串本地化和正则表达式”讨论了不同的编码，包括 UTF-8。我建议将你的工具链配置为使用 UTF-8。这将提高文件在不同平台之间的可移植性，并允许在源文件中使用非英语字符。

要在 Microsoft Visual C++中启用 UTF-8 支持，请将/utf-8 选项添加到“项目属性 | 配置属性 | C/C++ | 命令行”下的“其他选项”设置中。对于 GCC，使用命令行选项-finput-charset=UTF-8。Clang 默认所有文件都是 UTF-8 编码。

4. 预处理指令

如果你的编译器还不支持模块，你需要#include 头文件，而不是导入模块或头文件。也就是说，需要编写如下的预处理指令而不是像 import<print>这样的 import 声明。

```
#include <print>
```

预处理指令以#字符开始，前面示例中的#include <print>就是如此。在此例中，include 指令告诉预处理器：提取<print>头文件中的所有内容并提供给当前文件。<print>头文件提供了打印文本到屏幕

的功能。

第 11 章“模块、头文件和其他主题”将更详细地讨论预处理器指令。但是，如前所述，本书使用模块而不是旧式头文件。

5. main()函数

main()函数是程序的入口。main()函数返回一个 int 值以指示程序的最终执行状态。main()函数要么没有参数，要么具有两个参数，如下所示。

```
int main(int argc, char** argv)
```

argc 给出了传递给程序的实参数目，argv 包含了这些实参。注意 argv[0]可能是程序的名称，也可能是空字符串，所以不应使用它。相反，应当使用特定于平台的功能检索程序名。重要的是要记住，实际参数从索引 1 开始。

C++23

6. 打印文本

在 C++23 之前，需要使用 I/O 流将文本输出到屏幕。将在下一节中简要介绍流，并在第 13 章“C++I/O 揭秘”中详细介绍。然而，C++23 引入了一种新的、更易于使用的机制将文本打印到屏幕上，这在本书中的几乎每个代码片段中都有使用：std::print()和 println()，两者都在<print>中定义。

第 2 章“使用字符串和字符串视图”详细讨论了 std::print()和 println()字符串格式化和打印函数。然而，它们的基本用法很简单，在这里介绍，以便可以在接下来的代码片段中使用它们。在最基本的形式中，println()可用于打印一行以换行符自动结束的文本：

```
std::println("Hello, World!");
```

println()的第一个参数是一个格式字符串，它可以包含替换字段，这些字段可以被作为第二个和后续参数传入的值替换。可通过为每个要包含的字段添加花括号{}来指示替换字段的位置。例如：

```
std::println("There are {} ways I love you.", 219);
```

在这个例子中，数字 219 被插入字符串中，因此输出为：

```
There are 219 ways I love you.
```

可根据需要设置任意数量的替换字段，例如：

```
std::println("{} + {} = {}", 2, 4, 6);
```

在这个例子中，每个字段都是按顺序应用的，因此结果输出为：

```
2 + 4 = 6
```

关于替换字段的格式还有很多内容，但这是第 2 章的内容。

如果使用 print()而不是 println()，打印的文本将不会以换行字符结束。

7. 输入输出流

如果你的编译器还不支持 C++23 的 std::print()和 println()，你需要使用输入输出流来重写这些代码。

第 13 章将深入介绍输入输出流，但基本的输入输出非常简单。可将输出流想象为针对数据的洗衣滑槽，放入其中的任何内容都可以被正确输出。std::cout 就是对应于用户控制台或标准输出的滑槽，此外有其他滑槽，包括用于输出错误信息的 std::cerr。<<运算符将数据放入滑槽，在前面的示例中，

引号中的文本字符串被送到标准输出。输出流可在一行代码中连续输出多个不同类型的数据。下面的代码先输出文本，然后是数字，之后是更多文本。

```
std::cout << "There are " << 219 << " ways I love you." << std::endl;
```

从 C++20 开始，推荐的写法是使用 std::format()，它定义在<format>中，用来格式化字符串。format()函数与 print()和 println()使用相同概念的替换字段，更多细节将在第 2 章讨论。可用它重写之前的代码，十分简单：

```
std::cout << std::format("There are {} ways I love you.", 219) << std::endl;
```

因此，如果你的编译器还不支持 print()和 println()，你可以很容易地使用 cout、format()和 endl 重写这些代码。例如，假设有以下语句：

```
std::println("{} + {} = {}", 2, 4, 6);
```

在此语句中，将 println()替换为 format()，将结果流式传输到 cout，并添加 endl 的输出：

```
std::cout << std::format("{} + {} = {}", 2, 4, 6) << std::endl;
```

std::endl 代表序列的结尾。当输出流遇到 std::endl 时，就会将已送入滑槽的所有内容输出并转移到下一行。表明一行结尾的另一种方法是使用\n 字符，\n 是一个转义字符(escape character)，这是一个换行符。转义字符可以在任何被引用的文本字符串中使用。表 1-1 列出了最常用的转义字符。

表 1-1　最常用的转义字符

转义字符	含义
\n	换行：将光标移到下一行的开头
\r	回车：将光标移到本行的开头
\t	制表符
\\	反斜杠字符
\"	引号

警告：

请记住 endl 会在流中插入新的一行，并且把当前缓冲区中的所有内容刷出滑槽。不建议过度使用 endl，例如在循环中使用，因为这会影响程序的性能。另一方面，在流中插入\n 也会插入新的一行，但不会自动刷新缓冲区。

默认情况下，print()和 println()将文本打印到标准输出控制台 std::cout。可以按如下方式打印到错误控制台 std::cerr：

```
std::println(std::cerr, "Error: {}", 6);
```

流还可用于接收用户的输入，最简单的方法是在输入流中使用>>运算符。std::cin 输入流接收用户的键盘输入。下面是一个例子：

```
import std;
int main()
{
    int value;
    std::cin >> value;
    std::println("You entered {}", value);
```

```
}
```

>>运算符在读取值后遇到空格字符时停止输入。这也意味着不能使用运算符读取包含空格的文本。此外，用户输入可能很棘手，因为你永远不知道用户会输入什么样的数据。第 13 章详细讨论了输入流，包括如何读取带有嵌入空格的文本。

如果你拥有 C 的背景知识但初次接触 C++，你可能想了解过去使用的、可靠的 printf()和 scanf()现在究竟是什么情况。尽管在 C++中仍然可以使用这些函数，但强烈建议你改用 print()、println()、format()和流库，主要原因是 printf()和 scanf()未提供类型安全。

8. 从函数返回

“Hello World”程序的最后一行如下：

```
return 0;
```

由于这是 main()函数，从它返回将会把控制权返回给操作系统。执行此操作时，它会传递值 0，这通常会向操作系统发出信号，表明执行程序时没有错误。对于错误情况，可以返回非零值。

main()中的 return 语句是可选的。如果不编写，编译器将自动隐式地添加 return 0;。

1.1.2 命名空间

命名空间用来处理不同代码段之间的名称冲突问题。例如，你可能编写了一段代码，其中有一个名为 foo()的函数。某天，你决定使用第三方库，其中也有一个 foo()函数。编译器无法判断你的代码要使用哪个版本的 foo()函数。库函数的名称无法改变，而改变自己的函数名称又会感到非常痛苦。

在此类情况下可使用命名空间，从而指定定义名称的环境。为将某段代码加入命名空间，可用 namespace 块将其包含。下面是一个例子：

```
namespace mycode {
    void foo()
    {
        std::println("foo() called in the mycode namespace");
    }
}
```

将你编写的 foo()版本放到命名空间 mycode 后，这个函数就与第三方库提供的 foo()函数区分开来。为调用启用了命名空间的 foo()版本，需要使用::在函数名称之前给出命名空间，::称为作用域解析运算符。

```
mycode::foo();      // Calls the "foo" function in the "mycode" namespace
```

mycode 命名空间中的任何代码都可调用该命名空间内的其他代码，而不需要显式说明该命名空间。隐式的命名空间可使代码清晰并易于阅读。可使用 using 指令避免预先指明命名空间。这个指令通知编译器，后面的代码将使用指定命名空间中的名称。下面的代码中就隐含了命名空间：

```
using namespace mycode;

int main()
{
    foo();  // Implies mycode::foo();
}
```

一个源文件中可包含多条 using 指令；这种方法虽然便捷，但注意不要过度使用。极端情况下，如果你使用了已知的所有命名空间，实际上就是完全取消了命名空间。如果使用了两个同名的命名空间，将再次出现名称冲突问题。另外，应该知道代码在哪个命名空间内运行，这样就不会无意中调用错误版本的函数。

前面已经看到了命名空间的语法——在 Hello World 程序中已经用过命名空间，println 实际上是定义在 std 命名空间中的名称。可使用 using 指令重新编写 Hello World 程序，如下所示。

```
import std;

using namespace std;

int main()
{
   println("Hello, World!");
}
```

注意：

本书的大多数代码片段都假定已经对 std 命名空间使用了 using 指令，因此可以直接使用 C++标准库中的所有内容而不需要在前边添加 std::。

还可以使用 using 指令引用命名空间内的特定项。例如，如果只想使用 std 命名空间中的 print，可以使用如下的 using 声明。

```
using std::print;
```

后面的代码可使用 print 而不需要预先指明这个命名空间，但仍然需要显式说明 std 命名空间中的其他项，例如 println。

```
using std::print;
print("Hello, ");
std::println("World!");
```

警告：

切勿在全局作用域的头文件中使用 using 指令或 using 声明，否则添加你的头文件的每个人都必须使用它。将其放在较小的作用域(例如命名空间或类作用域)中是可以的，甚至可放在文件头部。将 using 指令或 using 声明放在模块接口文件中也是不错的选择，只要你不将它导出。然而，本书总是完全限定了模块接口文件中的所有类型，这样有助于更好地理解接口。稍后将解释模块接口文件和从模块导出实体。

1. 嵌套的命名空间

嵌套的命名空间，即将一个命名空间放在另一个命名空间中。各个命名空间之间由双冒号隔开，例如：

```
namespace MyLibraries::Networking::FTP {
    /* ... */
}
```

这种紧凑的写法是在C++17之后才得到支持的，在C++17之前，必须按如下方式使用嵌套的命名空间。

```
namespace MyLibraries {
    namespace Networking {
        namespace FTP {
            /* ... */
        }
    }
}
```

2. 命名空间别名

可使用命名空间别名，为另一个命名空间指定一个更简短的新名称。例如：

```
namespace MyFTP = MyLibraries::Networking::FTP;
```

1.1.3 字面量

字面量用于在代码中编写数字或字符串。C++支持大量标准字面量。可使用以下字面量指定数字(列出的示例都表示数字 123)。

- 十进制字面量 123
- 八进制字面量 0173(由 0 开头)
- 十六进制字面量 0x7B(由 0x 开头)
- 二进制字面量 0b1111011(由 0b 开头)

警告：
不要在数字字面量前加 0，除非将成为一个八进制字面量。

C++中的其他字面量示例包括：

- 浮点值(如 3.14f)
- 双精度浮点值(如 3.14)
- 十六进制浮点字面量(如 0x3.ABCp-10、0Xb.cp12l)
- 单个字符(如'a')
- 以零结尾的字符数组(如"character array")

字面量可以有一个后缀，比如 3.14f 中的 f，以强制指定某种类型。这种情况下，3.14f 为 float 类型，而 3.14 为 double 类型。

单引号字符可以用作数字字面量中的分隔符。例如：

- `23'456'789`
- `2'34'56'789`
- `0.123'456f`

多个仅由空格分隔的字符串字面量会自动连接成一个字符串。例如：

```
std::println("Hello, "
             "World!");
```

相当于：

```
std::println("Hello, World!");
```

还可以定义自己的字面量类型，这是一项高级功能，将在第 15 章“C++运算符重载”中介绍。

1.1.4　变量

在 C++中，可在任何位置声明变量，并且可在声明一个变量所在行之后的任意位置使用该变量。声明变量时可不指定值，这些未初始化的变量通常会被赋予一个半随机值，这个值取决于当时内存中的内容(这是许多 bug 的来源)。在 C++中，也可在声明变量时为变量指定初始值。下面的代码给出了两种风格的变量声明方式，使用的都是代表整数值的 int 类型。

```
int uninitializedInt;
int initializedInt { 7 };
println("{} is a random value", uninitializedInt);
println("{} was assigned as an initial value", initializedInt);
```

注意：

当代码中使用了未初始化的变量时，多数编译器会给出警告或报错信息。当访问未初始化的变量时，某些 C++环境可能会报告运行时错误。

initializedInt 变量使用统一初始化语法进行初始化。也可以使用下面的赋值语法来初始化：

```
int initializedInt = 7;
```

统一初始化在 2011 年的 C++11 标准中引入。建议使用统一初始化替代旧的赋值语法，这就是本书使用的语法。1.1.25 节会深入讨论它的好处及推荐它的原因。

C++中的变量是强类型的；也就是说，它们总是有一个特定的类型。C++自带一个可以开箱即用的内置类型集合。表 1-2 列出了 C++中最常见的变量类型。

表 1-2　最常见的变量类型

类型	说明	用法
(signed) int signed	正整数或负整数，范围取决于编译器(通常是 4 字节)	int i {-7}; signed int i {-6}; signed i {-5};
(signed) short (int)	短整型整数(通常是 2 字节)	short s {13}; short int s {14}; signed short s {15}; signed short int s {16};
(signed) long (int)	长整型整数(通常是 4 字节)	long l {-7L};
(signed) long long (int)	超长整型整数，范围取决于编译器，但不低于长整型(通常是 8 字节)	long long ll {14LL};
unsigned (int) unsigned short (int) unsigned long (int) unsigned long long (int)	对前面的类型加以限制，使其值≥0	unsigned int i {2U}; unsigned j {5U}; unsigned short s {23U}; unsigned long l {54UL}; unsigned long long ll {140ULL};
float	浮点型数字	float f {7.2f};
double	双精度数字，精度不低于 float	double d {7.2};

(续表)

类型	说明	用法
long double	长双精度数字，精度不低于 double	long double d {16.98L};
char unsigned char signed char	单个字符	char ch {'m'};
char8_t char16_t char32_t	单个 *n* 位 UTF-*n* 编码的 Unicode 字符，*n* 可以是 8、16、32	char8_t c8 {u8'm'}; char16_t c16 {u'm'}; char32_t c32 {U'm'};
wchar_t	单个宽字符，大小取决于编译器	wchar_t w {L'm'};
bool	布尔类型，取值为 true 或 false	bool b {true};

有符号、无符号的整型和 char 类型的范围如表 1-3 所示。

表 1-3 整型和 char 类型的范围

类型	有符号	无符号
char	-128~127	0~255
2 字节整型	-32 768~32 767	0~65 535
4 字节整型	-2 147 483 648~2 147 483 647	0~4 294 967 295
8 字节整型	-9 223 372 036 854 775 808~9 223 372 036 854 775 807	0~18 446 744 073 709 551 615

char 类型与 signed char 和 unsigned char 类型是不同的类型，它只应该用来表示字符。根据你的编译器不同，它既可能是有符号的，也可能是无符号的，所以不应该用它表示有符号或者无符号字符。

浮点类型的范围和精度将在稍后讨论。

与 char 相关的是，<cstddef>提供了 std::byte 类型用来表示单个字节。在 C++17 之前，char 或 unsigned char 用来表示一个字节，但是那些类型使得像在处理字符。std::byte 却能指明意图，即内存中的单个字节。一个 byte 可以用如下的方式初始化：

```
std::byte b { 42 };
```

注意：

C++没有提供基本的字符串类型。但是作为标准库的一部分提供了字符串的标准实现，本章后面的内容和第 2 章将讲述这一问题。

1. 数值极限

C++提供了一种获取数值极限信息的标准方式，例如在当前的平台上一个整数能表示的最大值。在 C 中，你可以使用各种宏定义，例如 INT_MAX。尽管这些方法在 C++中仍然可以使用，但推荐的做法是使用定义在<limits>中的类模板 std::numeric_limits。后续章节将讨论类模板，但那些细节对于理解如何使用 numeric_limits 来说无关紧要。目前，你只需要知道在使用类模板时需要在一对尖括号内指定需要的类型。例如，要获得整型的数值极限，可以编写 std::numeric_limits<int>。请参阅标准库参考(见附录 B)，以了解可以使用 numeric_limits 查询哪些信息。

下面是一些例子：

```
println("int:");
println("Max int value: {}", numeric_limits<int>::max());
```

```
println("Min int value: {}", numeric_limits<int>::min());
println("Lowest int value: {}", numeric_limits<int>::lowest());

println("\ndouble:");
println("Max double value: {}", numeric_limits<double>::max());
println("Min double value: {}", numeric_limits<double>::min());
println("Lowest double value: {}", numeric_limits<double>::lowest());
```

上面的代码段在我的系统上的输出如下：

```
int:
Max int value: 2147483647
Min int value: -2147483648
Lowest int value: -2147483648

double:
Max double value: 1.7976931348623157e+308
Min double value: 2.2250738585072014e-308
Lowest double value: -1.7976931348623157e+308
```

注意 min()和 lowest()之间的区别。对于一个整数，最小值等于最低值。然而对于浮点类型来说，最小值表示该类型能表示的最小正数，最低值表示该类型能表示的最小负数，即-max()。

2. 零初始化

可以用{0}或零初始化器{}将变量初始化为 0。零初始化会将原始的整数类型(例如 char、int 等)初始化为 0，将原始的浮点类型初始化为 0.0，将指针类型初始化为 nullptr，将对象用默认构造函数初始化(稍后讨论)。

下面是 float 和 int 零初始化的例子：

```
float myFloat {};
int myInt {};
```

3. 类型转换

可使用类型转换方式将变量转换为其他类型。例如，可将 float 转换为 int。C++提供了 3 种方法显式地转换变量类型。第一种方法来自 C，并且仍然被广泛使用，但实际上，已经不建议使用这种方法；第二种方法初看上去更自然，但很少使用；第三种方法最复杂，却最整洁，是推荐方法。

```
float myFloat { 3.14f };
int i1 { (int)myFloat };                  // method 1
int i2 { int(myFloat) };                  // method 2
int i3 { static_cast<int>(myFloat) };     // method 3
```

得到的整数是去掉小数部分的浮点数。第 10 章将详细描述这些转换方法之间的区别。在某些环境中，可自动执行类型转换或强制执行类型转换。例如，short 可自动转换为 long，因为 long 代表精度更高的相同数据类型。

```
long someLong { someShort };              // no explicit cast needed
```

当自动转换变量的类型时，应该了解潜在的数据丢失情况。例如，将 float 转换为 int 会丢掉一些信息(数字的小数部分)，并且如果浮点值表示的数字超过了整数可表示的最大值，转换结果有可能是完全错误的。如果将一个 float 赋给一个 int 而不显式执行类型转换，多数编译器会给出警告甚至错误信息。如果确信左边的类型与右边的类型完全兼容，那么隐式转换也完全没有问题。

4. 浮点型数字

处理浮点型数字可能会比处理整型数字复杂。你需要记住几件事情。使用数量级不同的浮点值计算可能会导致错误。此外，计算两个几乎相同的浮点数的差时会导致精度丢失。另外要记住很多十进制数不能精确地表示为浮点数。然而，继续深入讨论使用浮点数的数值问题及如何编写数值稳定的浮点数程序已经超出了本书的范围，这些话题足以用一整本书讨论了。

这里有几个特殊的浮点数。

- +/−infinity：表示正无穷和负无穷，例如非零数除以 0 得到的结果。
- NaN：非数字的缩写，例如 0 除以 0 的结果，这在数学上是未定义的。

可以用 std::isnan()判断一个给定的浮点数是否为非数字，用 std::isinf()判断是否为无穷，这两个函数都定义在<cmath>中。

可以使用 numeric_limits 获取这些特殊的浮点数，例如 numeric_limits<double>::infinity。

C++23

5. 扩展浮点类型

正如前面关于变量的部分所述，C++提供了以下标准浮点类型：float、double 和 long double。

C++23 引入了表 1-4 在某些领域流行的扩展浮点类型。对这些类型的支持是可选的，并非所有编译器都提供这些类型。

表 1-4　扩展浮点类型

类型	说明	字面量后缀
std::float16_t	IEEE 754 标准的 16 位格式	F16 或 f16
std::float32_t	IEEE 754 标准的 32 位格式	F32 或 f32
std::float64_t	IEEE 754 标准的 64 位格式	F64 或 f64
std::float128_t	IEEE 754 标准的 128 位格式	F128 或 f128
std::bfloat16_t	脑浮点，用于某些人工智能领域	BF16 或 bf16

大多数时候，float、double 和 long double 这样的标准类型精度就足够了。其中，double 应该是你的默认类型。使用 float 可能导致精度损失，根据使用情况，这可能是可接受的，也可能是不可接受的。

6. 浮点类型的范围与精度

浮点类型的范围和精度有限。表 1-5 给出了 C++支持的所有标准和扩展浮点类型的详细规范。然而，标准类型(float、double 和 long double)的规范并不是由 C++标准精确指定的。标准规定，只有 long double 应该至少与 double 具有相同的精度，double 应该至少具有与 float 相同的精度。对于这三种类型，该表显示了编译器常用的值。

表 1-5　C++支持的标准和扩展浮点类型

类型	名称	尾数位	有效数字位数	指数位	最小正数	最大值
float	单精度浮点	24	7.22	8	1.18×10^{-38}	3.40×10^{38}
double	双精度浮点	53	15.95	11	2.23×10^{-308}	1.80×10^{308}
long double	扩展精度浮点	64	19.27	15	3.36×10^{-4932}	1.19×10^{4932}
std::float16_t	半精度浮点	11	3.31	5	6.10×10^{-5}	65504
std::float32_t	单精度浮点	24	7.22	8	1.18×10^{-38}	3.40×10^{38}
std::float64_t	双精度浮点	53	15.95	11	2.23×10^{-308}	1.80×10^{308}
std::float128_t	四倍精度浮点	113	34.02	15	3.36×10^{-4932}	1.19×10^{4932}
std::bfloat16_t	脑浮点	8	2.41	8	1.18×10^{-38}	3.40×10^{38}

1.1.5　运算符

如果无法改变变量的值，那么变量还有什么用呢？表 1-6 显示了 C++中最常用的运算符以及使用这些运算符的示例代码。注意，在 C++中，运算符可以是二元的(操作两个表达式)、一元的(仅操作一个表达式)甚至是三元的(操作三个表达式)。在 C++中只有一个条件运算符，1.1.9 节将介绍这个运算符。此外，第 15 章将介绍如何将运算符用到自定义类型上。

表 1-6　运算符

运算符	说明	用法

=	二元运算符，将右边的值赋给左边的变量	int i;
		i = 3;
		int j;
		j = i;

!	一元运算符，改变变量的真/假(非零/零)状态	bool b { !true }; bool b2 { !b };
+	执行加法的二元运算符	int i { 3 + 2 }; int j { i + 5 }; int k { i + j };
- * /	执行减法、乘法以及除法的二元运算符	int i { 5-1 }; int j { 5*2 }; int k { j / i };
%	二元运算符，求除法操作的余数，也称为 mod 运算符。例如 5%2=1	int rem { 5 % 2 };
++	一元运算符，使变量值增加 1。如果运算符在变量之后(后增量)，表达式的结果是没有增加的值；如果运算符在变量之前(前增量)，表达式的结果是增加 1 后的新值	i++; ++i;
--	一元运算符，使变量值减 1	i--; --i;

+=
-=
*=
/=
%=
	i = i+j; 的简写
	i = i-j; 的简写
	i = i * j; 的简写
	i = i / j; 的简写
	i = i % j; 的简写
		i += j;
		i -= j;
		i *= j;
		i /= j;
		i %= j;

& &=	将一个变量的原始位与另一个变量执行按位“与”运算	i = j & k; j &= k;
| |=	将一个变量的原始位与另一个变量执行按位“或”运算	i = j | k; j |= k;

<<
>>
<<=
>>=
	对一个变量的原始位执行移位运算，每一位左移(<<)或右移(>>)指定的位数	i = i << 1;
		i = i >> 4;
		i <<= 1;
		i >>= 4;

^ ^=	执行两个变量之间的按位“异或”运算	i = i ^ j; i ^= j;

形式为 op=的运算符，例如+=，称为复合赋值运算符。

当一个二元运算符应用于两个不同类型的操作数时，编译器会插入一个隐式转换，在应用运算符之前将其中一个转换为另一个。还可以使用显式转换，使用 static_cast()将一种类型转换为另一种类型。

对于隐式转换，编译器有一定的规则来决定将哪种类型转换为其他哪种类型。例如，对于具有小整数类型和大整数类型的二进制运算，较小的类型将被转换为较大的类型。然而，结果可能并不总是如所期望的那样。因此，建议在隐式转换和使用显式转换时要小心，以确保编译器按你的意图运行。

下面的程序显示了最常见的变量类型以及运算符的用法。如果还不确定变量以及运算符的使用方式，请试着判断该程序的输出，然后运行该程序来验证自己的答案是否正确。

```
int someInteger { 256 };
short someShort;
long someLong;
float someFloat;
double someDouble;

someInteger++;
someInteger *= 2;
// Conversion from larger integer type to smaller integer type
// can cause a warning or error, hence static_cast() is required.
someShort = static_cast<short>(someInteger);
someLong = someShort * 10000;
someFloat = someLong + 0.785f;
// To make sure the division is performed with double precision,
// someFloat is explicitly converted to double first.
someDouble = static_cast<double>(someFloat) / 100000;
println("{}", someDouble);
```

关于表达式求值顺序，C++编译器有一套准则。如果某行代码非常复杂，包含多个运算符，其执行顺序可能不会一目了然。因此，最好将复杂表达式分成若干短小的表达式，或使用括号明确地将子表达式分组。例如，除非你记住了 C++运算符的优先级，否则下面的代码将比较难懂。

```
int i { 34 + 8 * 2 + 21 / 7 % 2 };
```

添加括号可清楚地表明首先执行哪个运算。

```
int i { 34 + (8 * 2) + ( (21 / 7) % 2 ) };
```

如果测试这些代码，这两种方法是等价的，其结果都是 i 等于 51。如果你认为 C++按从左到右的顺序对表达式求值，答案将是 1。实际上，C++会首先对/、*以及%求值(从左向右)，然后才执行加减运算，最后是位运算。通过使用圆括号，可明确告诉编译器某个运算应该优先求值。

准确地讲，运算符的计算顺序是由所谓的优先级决定的。优先级高的运算符要先于优先级低的运算符执行。下面列出了表 1-6 中出现的运算符的优先级，越靠上的运算符优先级越高，也将先执行。

- ++ --(后置)
- ! ++ --(前置)
- * / %
- + -
- << >>
- &
- ^
- |

- = += -= *= /= %= &= |= ^= <<= >>=

这仅仅是 C++所有运算符中的一部分。第 15 章将完整介绍所有运算符，以及它们的优先级。

1.1.6 枚举

整数代表某个数字序列中的值。枚举是一种允许你定义自己的序列的类型，这样你就能使用这个序列中的值声明变量。例如，在一个国际象棋程序中，可以用 int 代表所有棋子，用常量代表棋子的类型，代码如下所示。代表棋子类型的整数用 const 标记，表明这个值永远不会改变。

```
const int PieceTypeKing { 0 };
const int PieceTypeQueen { 1 };
const int PieceTypeRook { 2 };
const int PieceTypePawn { 3 };
//etc.
int myPiece { PieceTypeKing };
```

这种表示法存在一定的风险。因为棋子只是一个 int，如果另一个程序增加棋子的值，那么会发生什么？加 1 就可让王变成王后，而这实际上没有意义。更糟的是，有人可能将某个棋子的值设置为 -1，而这个值并没有对应的常量。

强类型的枚举类型通过定义变量的取值范围解决了上述问题。下面的代码声明了一个新类型 PieceType，这个类型具有 4 个可能的值，分别代表 4 种国际象棋棋子：

```
enum class PieceType { King, Queen, Rook, Pawn };
```

这种新的类型可以像下面这样使用：

```
PieceType piece { PieceType::King };
```

事实上，枚举类型只是一个整型值。King、Queen、Rook、Pawn 的实际值分别是 0、1、2、3。还可为枚举成员指定整型值，其语法如下：

```
enum class PieceType
{
   King = 1,
   Queen,
   Rook = 10,
   Pawn
};
```

如果你没有为当前的枚举成员赋值，编译器会将上一个枚举成员的值递增 1，再赋予当前的枚举成员。如果没有给第一个枚举成员赋值，编译器就给它赋予 0。所以，在此例中，King 具有整型值 1，编译器为 Queen 赋予整型值 2，Rook 的值为 10，编译器自动为 Pawn 赋予值 11。

尽管枚举值内部是由整型值表示的，它却不会自动转换为整数。因此，下面的代码是不合法的：

```
int underlyingValue { piece };
```

C++23

从 C++23 开始，可以使用 std::to_underlying()。例如：

```
int underlyingValue { to_underlying(piece) };
```

默认情况下，枚举值的基本类型是整型，但可采用以下方式加以改变：

```
enum class PieceType : unsigned long
{
   King = 1,
```

```
    Queen,
    Rook = 10,
    Pawn
};
```

对于 enum class，枚举值名不会自动超出封闭的作用域，这意味着它们不会与定义在父作用域的其他名称冲突。所以，不同的强类型枚举可以拥有同名的枚举值。例如，以下两个枚举是完全合法的：

```
enum class State { Unknown, Started, Finished };
enum class Error { None, BadInput, DiskFull, Unknown };
```

这个特性的一大好处就是可以给枚举值取较短的名字，例如：Unknown 而不是 UnknownState 或 UnknownError。然而，这同时意味着必须使用枚举值的全名，或者使用 using enum 或 using 声明，下面是一个例子：

```
using enum PieceType;
PieceType piece { King };
```

可以用 using 声明避免使用某个特定枚举值的全名。例如，在下面的代码段中，King 可以不用全名就被使用，但是其他枚举值仍需要使用全名。

```
using PieceType::King;
PieceType piece { King };
piece = PieceType::Queen;
```

警告：

即使 C++允许不使用枚举值的全名，仍然建议谨慎地使用这个特性。至少要使 using 或 using enum 声明的作用域尽量小。如果作用域很大的话，有可能重新引入名称冲突。后续关于 switch 语句的小节会展示使用 using enum 声明时如何合适地限制作用域。

旧式风格枚举

新的代码总是应该使用上一节提到的强类型枚举。然而，在遗留代码库中，你可能会遇到旧式风格的枚举：enum 而不是 enum class。这是一个定义为旧式枚举的 PieceType：

```
enum PieceType { PieceTypeKing, PieceTypeQueen, PieceTypeRook, PieceTypePawn };
```

这种枚举类型的枚举值会被导出到外层的作用域。这意味着可以在父作用域不通过全名而直接使用它们。例如：

```
PieceType myPiece { PieceTypeQueen };
```

当然，这同时意味着它们可能会与父作用域中的名称产生冲突，导致编译错误。例如：

```
bool ok { false };
enum Status { error, ok };
```

这段代码不会成功编译，因为名字 ok 首先被定义为一个布尔类型的变量，之后同一个名称又作为枚举值的名称使用。Visual C++ 2022 会给出如下的错误提示：

```
error C2365: 'ok': redefinition; previous definition was 'data variable'
```

因此，必须确保旧式风格的枚举使用独一无二的枚举值名称，例如 PieceTypeQueen，而不是简单的 Queen。

这些旧式风格的枚举不是强类型的，意味着它们不是类型安全的。它们总是会被解释为整型，因此，你可能会无意中比较来自完全不同枚举类型的枚举值，或者将错误枚举类型的枚举值传递给函数。

警告：
总是使用强类型的 enum class 而不是类型不安全的无作用域的旧式风格 enum 枚举。

1.1.7　结构体

结构体(struct)允许将一个或多个已有类型封装到一个新类型中。数据库记录是结构体的经典示例，如果想要建立一个人事系统来跟踪雇员的信息，那么需要存储名首字母、姓首字母、雇员编号以及每个雇员的薪水。下面的代码给出了 employee.cppm 模块接口文件中的一个结构体，这个结构体包含所有这些信息。这是本书中第一个由你自己写的模块。模块接口文件通常以.cppm 作为后缀(注释：在撰写本书时，还没有模块接口文件的标准化命名。但是，大多数编译器都支持.cppm 扩展名，因此本书使用了该扩展名。查看你使用的编译器的文档以了解要使用的扩展名)。模块接口文件中的第一行是模块声明，并声明该文件正在定义一个名为 employee 的模块。此外，模块需要明确说明其导出的内容，即，在导入该模块的其他位置时可见的内容。从模块导出类型是通过在 struct 前面使用 export 关键字完成的。

```
export module employee;

export struct Employee {
    char firstInitial;
    char lastInitial;
    int employeeNumber;
    int salary;
};
```

声明为 Employee 类型的变量将拥有全部内建的字段。可使用“.”运算符访问结构体的各个字段。下面的示例创建了一条员工记录，然后将其输出。与标准命名模块 std 一样，在导入自定义模块时不使用尖括号。

```
import std;
import employee; // Import our employee module

using namespace std;

int main()
{
    // Create and populate an employee.
    Employee anEmployee;
    anEmployee.firstInitial = 'J';
    anEmployee.lastInitial = 'D';
    anEmployee.employeeNumber = 42;
    anEmployee.salary = 80000;
    // Output the values of an employee.
    println("Employee: {}{}", anEmployee.firstInitial,
        anEmployee.lastInitial);
    println("Number: {}", anEmployee.employeeNumber);
    println("Salary: ${}", anEmployee.salary);
}
```

1.1.8 条件语句

条件语句允许根据某件事情的真假来执行代码。在下面介绍的内容中，你可以了解到C++中有两种主要的条件语句：if/else 语句和 switch 语句。

1. if/else 语句

最常见的条件语句是 if 语句，其中可能伴随着 else 语句。如果 if 语句中给定的条件为 true，就执行对应的代码行或代码块，否则执行 else 语句(如果存在 else 语句)或者执行条件语句之后的代码。下面的代码显示了一种级联的 if 语句，这是一种奇特方式：if 语句伴随着 else 语句，而 else 语句又伴随着另一个 if 语句。

```
if (i > 4) {
    // Do something.
} else if (i > 2) {
    // Do something else.
} else {
    // Do something else.
}
```

if 语句的圆括号中的表达式必须是一个布尔值，或者求值的结果必须是布尔值。零值是 false，非零值则算作 true。例如，if(0)等同于 if(false)。后面将讲到的逻辑条件运算符提供了一种求表达式值的方式，其结果为布尔值 true 或 false。

2. if 语句的初始化器

C++允许在 if 语句中包括一个初始化器，语法如下：

```
if (<initializer>; <conditional_expression>) {
    <if_body>
} else if (<else_if_expression>) {
    <else_if_body>
} else {
    <else_body>
}
```

<initializer>中引入的任何变量只在<conditional_expression>、<if_body>、<else_if_expression>、<else_if_body>和<else_body>中可用。此类变量在 if 语句以外不可用。

此时还不到列举此功能的有用示例的时候，下面只列出它的形式：

```
if (Employee employee { getEmployee() }; employee.salary > 1000) { ... }
```

在这个例子中，初始化器通过调用 getEmployee()函数来获取雇员。本章稍后将讨论函数。该条件检查检索到的雇员的工资是否超过 1000。只有在这种情况下，if 语句的主体才会被执行。本书将给出更具体的例子。

3. switch 语句

switch 是另一种根据表达式值执行操作的语法。在 C++中，switch 语句的表达式必须是整型、能转换为整型的类型、枚举类型或强类型枚举，必须与一个常量进行比较，每个常量值代表一种“情况(case)”，如果表达式与这种情况匹配，随后的代码行将会被执行，直至遇到 break 语句为止。此外可提供 default 情况，如果没有其他情况与表达式值匹配，表达式值将与 default 情况匹配。下面的伪代

码显示了 switch 语句的常见用法：

```
switch (menuItem) {
    case OpenMenuItem:
        // Code to open a file
        break;
    case SaveMenuItem:
        // Code to save a file
        break;
    default:
        // Code to give an error message
        break;
}
```

switch 语句总是可以转换为 if/else 语句。前面的 switch 语句可以转换为：

```
if (menuItem == OpenMenuItem) {
    // Code to open a file
} else if (menuItem == SaveMenuItem) {
    // Code to save a file
} else {
    // Code to give an error message
}
```

如果你想基于表达式的多个值(而非对表达式进行一些测试)执行操作，通常使用 switch 语句。此时，switch 语句可避免级联使用 if-else 语句。如果只需要检查一个值，则应当使用 if 或 if-else 语句。

一旦找到与 switch 条件匹配的 case 表达式，就执行其后的所有语句，直至遇到 break 语句为止。即使遇到另一个 case 表达式，执行也会继续，这称为 fallthrough。在下面的示例中，对 Mode :: Standard 和 Default 都执行了一组语句。如果 Mode 为 Custom，则首先将 value 从 42 改为 84，然后执行与 Default 和 Standard 相同的语句。换句话说，Custom case 会一直执行到最终到达 break 语句或 switch 语句的末尾。此代码段还显示了一个很好的示例，该示例使用适当范围的 enum 声明来避免为不同的 case 标签编写 Mode::Custom、Mode::Standard 和 Mode::Default。

```
enum class Mode { Default, Custom, Standard };

int value { 42 };
Mode mode { /* ... */ };
switch (mode) {
    using enum Mode;

    case Custom:
        value = 84;
    case Standard:
    case Default:
        // Do something with value ...
        break;
}
```

如果你无意间忘掉了 break 语句，fallthrough 将成为 bug 的来源。因此，如果在 switch 语句中检测到 fallthrough，编译器将生成警告信息，除非 case 为空。在上例中，编译器不会发出 Standard 情况跌落进 Default 情况的警告，但是可能会对 Custom 情况的 fallthrough 生成警告信息。为了阻止编译器发出警告，可以使用[[fallthrough]]特性，告诉编译器某个 fallthrough 是有意为之，如下所示。

```
switch (mode) {
   using enum Mode;

   case Custom:
      value = 84;
      [[fallthrough]];
   case Standard:
   case Default:
      // Do something with value ...
      break;
}
```

用大括号括住 case 表达式后面的语句通常是可选的，但有时是必要的，例如在定义变量时。以下是一个示例：

```
switch (mode) {
   using enum Mode;

   case Custom:
      {
         int someVariable { 42 };
         value = someVariable * 2;
         [[fallthrough]];
      }
   case Standard:
   case Default:
      // Do something with value ...
      break;
}
```

对枚举使用 switch 语句时，如果没有处理所有不同的枚举值，大多数编译器都会发出警告；要么显式地为每个枚举值编写情况，要么结合 Default 情况仅为选定的枚举器编写例子。但是，建议不要在枚举的 switch 语句中包含 Default 情况。相反，应该明确列出所有枚举值。原因是，当稍后向枚举中添加更多枚举值时，这使得代码不太容易出错。这种情况下，如果忘记将任何新的枚举值添加到特定的 switch 语句中，编译器将发出警告，而不是使用 default 情况默默地处理新枚举值。

4. switch 语句的初始化器

与 if 语句一样，可以在 switch 语句中使用初始化器。语法如下：

```
switch (<initializer>; <expression>) { <body> }
```

<initializer>中引入的任何变量将只在<expression>和<body>中可用。它们在 switch 语句之外不可用。

1.1.9 条件运算符

C++有一个接收 3 个参数的运算符，称为三元运算符。可将其作为“如果[某事发生了]，那么[执行某个操作]，否则[执行其他操作]”的条件表达式的简写。这个条件运算符由一个“?”和一个“:”组成。下面的代码中，如果变量 i 的值大于 2，将输出 yes，否则输出 no。

```
println("{}", (i > 2) ? "yes" : "no");
```

i > 2 两边的小括号是可选的，因此与下面的代码行是等效的。

```
println("{}", i > 2 ? "yes" : "no");
```

条件运算符的优点是它是一个表达式而不是像 if 或者 switch 那样的语句。因此，条件运算符几乎可在任何环境中使用。在上例中，这个条件运算符在输出语句中执行。记住这个语法的简便方法是将问号前的语句真的当作一个问题。例如，“i 大于 2 吗？如果是真的，结果就是 yes，否则结果就是 no。”

1.1.10　逻辑比较运算符

前面介绍了非正式定义的逻辑比较运算符。>运算符比较两个值的大小，如果左边的值大于右边的值，那么结果为 true。所有逻辑比较运算符都遵循这一模式——其结果都是 true 或 false。

表 1-7 列出了常见的逻辑比较运算符。

表 1-7　逻辑比较运算符

运算符	说明	用法

<
<=
>
>=
	判断左边的值是否小于、小于或等于、大于、大于或等于右边的值	`if (i < 0) {`
		`  print("i is negative");`
		`}`

==	判断左边的值是否等于右边的值，不要将其与赋值运算符=混淆	`if (i = = 3) {`
		`  print("i is 3");`
		`}`

!=	不等于。如果左边的值与右边的值不相等，则语句的结果为 true	`if (i != 3) {`
		`  print("i is not 3");`
		`}`

<=>	三向比较运算符，也称为太空飞船运算符，下一节将会详细解释	`result = i <=> 0;`

!	逻辑非。改变布尔表达式的 true/false 状态，这是一个一元运算符	`if (!someBoolean) {`
		`  print("someBoolean is false");`
		`}`

&&	逻辑与。如果表达式的两边都为 true，其结果为 true	`if (someBoolean && someOtherBoolean) {`
		`  print("both are true");`
		`}`

||	逻辑或。如果表达式两边的任意一边值为 true，其结果就为 true	`if (someBoolean || someOtherBoolean) {`
		`  print("at least one is true");`
		`}`

C++对表达式求值时会采用短路逻辑。这意味着一旦最终结果可确定，就不对表达式的剩余部分求值。例如，当执行如下所示的多个布尔表达式的逻辑或操作时，如果发现其中一个表达式的值为 true，立刻可判定其结果为 true，就不再检测剩余部分。

```
bool result { bool1 || bool2 || (i > 7) || (27 / 13 % i + 1) < 2 };
```

在此例中，如果 bool1 的值是 true，整个表达式的值必然为 true，因此不必对其他部分求值。这

种方法可阻止代码执行多余操作。然而，如果后面的表达式以某种方式影响程序的状态(例如，调用了一个独立函数)，就会带来难以发现的 bug。

下面的代码显示了一条使用了&&的语句，这条语句在第二项之后就会被短路，因为 0 总被当作 false：

```
bool result { bool1 && 0 && (i > 7) && !done };
```

短路做法对性能有好处。在使用逻辑短路时，可将代价更低的测试放在前面，以避免执行代价更高的测试。在指针上下文中，它也可避免指针无效时执行表达式的一部分的情况。本章后面将讨论指针及包含短路的指针。

1.1.11 三向比较运算符

三向比较运算符可用于确定两个值的大小顺序。它也被称为太空飞船操作符，因为其符号<=>类似于太空飞船。使用单个表达式，它可以告诉你一个值是否等于、小于或大于另一个值。因为它必须返回的不仅仅是 true 或 false，所以它不能返回布尔类型。相反，它返回类枚举(enumeration-like)[1]类型，定义在<compare>和 std 命名空间中。如果操作数是整数类型，则结果是所谓的强排序，并且可以是以下之一。

- strong_ordering::less：第一个操作数小于第二个
- strong_ordering::greater：第一个操作数大于第二个
- strong_ordering::equal：第一个操作数等于第二个

以下是它的用法的示例：

```
int i { 11 };
strong_ordering result { i <=> 0 };
if (result == strong_ordering::less) { println("less"); }
if (result == strong_ordering::greater) { println("greater"); }
if (result == strong_ordering::equal) { println("equal"); }
```

某些类型没有全序关系。例如，非数字浮点值永远不会等于、小于或大于任何其他浮点值。因此，这样的比较会导致偏序。

- partial_ordering::less：第一个操作数小于第二个
- partial_ordering::greater：第一个操作数大于第二个
- partial_ordering::equivalent：第一个操作数等价于第二个，也就是!(a<b) && !(b<a)，例如-0.0 等价于+0.0，但它们不相等
- partial_ordering::unordered：如果有一个操作数是非数字或者两个操作数都是非数字

如果你真的需要对浮点值进行强排序，例如，如果你知道它们不可能是非数字，你可以使用 std::strong_order()，它总是产生 std::strong_ordering 结果。

还有一种弱排序，这是一种额外的排序类型，你可以从中选择，为自己的类型实现三向比较。在弱排序的情况下，所有值都是有序的，即没有无序的结果，但排序不强，这意味着可能存在等价的不相等值。一个例子是使用不区分大小写的比较对字符串进行排序，这种情况下，字符串“Hello World”和“hello world”当然不相等，但它们是等价的。以下是弱排序的不同结果：

- weak_ordering::less：第一个操作数小于第二个
- weak_ordering::greater：第一个操作数大于第二个

1 不是真正的枚举类型。这些排序类型不能用在 switch 语句中，也不能使用 using enum 声明。

- weak_ordering::equivalent：第一个操作数等于第二个

这三种不同类型的排序支持某些隐式转换。strong_ordering 可以隐式地转换为 partial_ordering 或 weak_ordering。weak_ordering 可以隐式转换为 partial_ordering。

对于原始类型，与仅使用==、<和>运算符进行单个比较相比，使用三向比较运算符不会带来太多收益。但是，它对于比较昂贵的对象很有用。使用三向比较运算符，可以使用单个运算符对此类对象进行排序，而不必潜在地调用两个单独的比较运算符，从而触发两个昂贵的比较。第 9 章将说明如何为自己的类型增加对三向比较的支持。

最后，<compare>提供命名的比较函数来解释排序结果。这些函数是 std::is_eq()、is_neq()、is_lt()、is_lteq()、is_gt()以及 is_gteq()。如果排序分别表示==、!=、<、<=、>或 >=，则返回 true，否则返回 false。以下是一个例子：

```
int i { 11 };
strong_ordering result { i <=> 0 };
if (is_lt(result)) { println("less"); }
if (is_gt(result)) { println("greater"); }
if (is_eq(result)) { println("equal"); }
```

1.1.12　函数

对于任何大型程序而言，将所有代码都放到 main()中是无法管理的。为使程序便于理解，需要将代码分解为简单明了的函数。

在 C++中，为让其他代码使用某个函数，首先应该声明该函数。如果函数在某个特定的文件内部使用，通常会在源文件中声明并定义这个函数。如果函数是供其他模块或文件使用的，可以从模块接口文件中导出一个函数声明，函数的定义既可以放在同一个模块接口文件中，也可以放在所谓的模块实现文件中。

注意：

函数声明通常称为“函数原型”或“函数头”，以强调这代表函数的访问方式，而不是具体代码。术语“函数签名”指将函数名与形参列表组合在一起，但没有返回类型。

函数的声明如下所示。这个示例的返回值是 void 类型，说明这个函数不会向调用者提供结果。调用者在调用函数时必须提供两个参数—— 一个整数和一个字符。

```
void myFunction(int i, char c);
```

如果没有与函数声明匹配的函数定义，在编译过程的链接阶段会出错，因为使用函数 myFunction()时会调用不存在的代码。以下函数定义输出了两个参数的值：

```
void myFunction(int i, char c)
{
   println("The value of i is {}.", i);
   println("The value of c is {}.", c);
}
```

在程序的其他位置，可调用 myFunction()，并将实参传递给两个形参。函数调用的几个示例如下：

```
int someInt { 6 };
char someChar { 'c' };
myFunction(8, 'a');
myFunction(someInt, 'b');
```

```
myFunction(5, someChar);
```

注意：

与 C 不同，在 C++中没有形参的函数仅需要一个空的参数列表，不需要使用 void 指出此处没有形参。然而，如果没有返回值，仍需要使用 void 指明这一点。

C++函数还可以向调用者返回一个值。下面的函数将两个数字相加并返回结果：

```
int addNumbers(int number1, int number2)
{
    return number1 + number2;
}
```

这个函数可以这样被调用：

```
int sum { addNumbers(5, 3) };
```

1. 函数返回类型的推断

你可以要求编译器自动推断出函数的返回类型。要使用这个功能，需要把 auto 指定为返回类型：

```
auto addNumbers(int number1, int number2)
{
    return number1 + number2;
}
```

编译器根据函数体中用于 return 语句的表达式推导返回类型。可以有多个 return 语句，但它们必须全部解析为完全相同的类型，因为编译器永远不会插入任何隐式转换来推断函数的返回类型。这样的函数甚至可以包括递归调用(对自身的调用)，但是该函数中的第一个 return 语句必须是非递归调用。

2. 当前函数的名称

每个函数都有一个预定义的局部变量__func__，其中包含当前函数的名称。这个变量的一个用途是用于日志记录：

```
int addNumbers(int number1, int number2)
{
    println("Entering function {}", __func__);
    return number1 + number2;
}
```

3. 函数重载

重载功能意味着要提供多个具有相同名称但具有不同参数集的功能。仅指定不同的返回类型是不够的，因为在调用函数时可以忽略返回值；相反，参数的数量和/或类型必须不同。

假设你想提供适用于整型和 double 的 addNumbers()版本。如果不重载，必须想出唯一的名称，例如：

```
int addNumbersInts(int a, int b) { return a + b; }
double addNumbersDoubles(double a, double b) { return a + b; }
```

使用函数重载，则不需要为函数的不同版本提供不同的名称。以下代码片段定义了两个名为 addNumbers()的函数，一个为整型定义，另一个为 double 定义：

```
int addNumbers(int a, int b) { return a + b; }
double addNumbers(double a, double b) { return a + b; }
```

当调用 addNumbers()时，编译器会根据提供的参数自动选择正确的函数重载版本。

```
println("{}", addNumbers(1, 2));          // Calls the integer version
println("{}", addNumbers(1.11, 2.22));    // Calls the double version
```

1.1.13　属性

属性是一种将可选的和/或特定于编译器厂商的信息添加到源代码中的机制。在 C++对属性进行标准化之前，编译器厂商决定了如何指定此类信息，例如__attribute__和__declspec 等。从 C++ 11 开始，通过使用双方括号语法[[attribute]]对属性进行标准化的支持。

在本章的前面，引入了[[fallthrough]]属性，以防止故意在 switch case 语句中使用 fallthrough 时发出编译器警告。C++标准定义了几个在函数上下文中有用的标准属性。

1. [[nodiscard]]

[[nodiscard]]属性可用于有一个返回值的函数，以使编译器在该函数被调用却没有对返回的值进行任何处理时发出警告，以下是一个例子。

```
[[nodiscard]] int func() { return 42; }

int main()
{
   func();
}
```

编译器会发出类似以下内容的警告：

```
warning C4834: discarding return value of function with 'nodiscard' attribute
```

例如，此特性可用于返回错误代码的函数。通过将[[nodiscard]]属性添加到此类函数中，错误代码就无法被忽视。

更笼统地说，[[nodiscard]]属性可用于类、函数和枚举。将属性应用于整个类的一个示例是，当有一个表示错误条件的类时。通过将[[nodiscard]]应用于这样的类，编译器将对每个返回此错误条件的函数调用发出警告，并且调用者对此不做任何操作。

可以字符串的形式为[[nodiscard]]属性提供原因。如果函数的调用者忽略了返回的值，则此原因将显示在编译器生成的警告消息中。以下是一个示例：

```
[[nodiscard("Some explanation")]] int func();
```

2. [[maybe_unused]]

[[maybe_unused]]属性可用于禁止编译器在未使用某些内容时发出警告，如下例所示：

```
int func(int param1, int param2)
{
   return 42;
}
```

如果将编译器警告级别设置得足够高，则此函数定义可能导致两个编译器警告。例如，Microsoft Visual C++给出以下警告：

```
warning C4100: 'param2': unreferenced formal parameter
warning C4100: 'param1': unreferenced formal parameter
```

通过使用[[maybe_unused]]，可以阻止这种警告：

```
int func(int param1, [[maybe_unused]] int param2)
{
    return 42;
}
```

在这种情况下，第二个参数标记有禁止其警告的属性。现在，编译器仅对 param1 发出警告：

```
warning C4100: 'param1': unreferenced formal parameter
```

[[maybe_unused]]属性可用于类和结构体、非静态数据成员、联合、typedef、类型别名、变量、函数、枚举及枚举值。你可能尚不知道其中的某些术语，本书稍后将进行讨论。

3. [[noreturn]]

向函数添加[[noreturn]]属性意味着它永远不会将控制权返回给调用点。通常，函数要么导致某种终止(进程终止或线程终止)，要么引发异常。异常会在本章的后面部分讨论。使用此属性，编译器可以避免发出某些警告或错误，因为它现在可以更多地了解该函数的用途。下面是一个例子：

```
import std;
using namespace std;

[[noreturn]] void forceProgramTermination()
{
    exit(1);  // Defined in <cstdlib>
}

bool isDongleAvailable()
{
    bool isAvailable { false };
    // Check whether a licensing dongle is available...
    return isAvailable;
}

bool isFeatureLicensed(int featureId)
{
    if (!isDongleAvailable()) {
        // No licensing dongle found, abort program execution!
        forceProgramTermination();
    } else {
        // Dongle available, perform license check of the given feature...
        bool isLicensed { featureId == 42 };
        return isLicensed;
    }
}

int main()
{
    bool isLicensed { isFeatureLicensed(42) };
    println("{}", isLicensed);
}
```

此代码段可以正常编译，没有任何警告或错误。但是，如果删除[[noreturn]]属性，编译器将生成以下警告(来自 Visual C++的输出)。

```
warning C4715: 'isFeatureLicensed': not all control paths return a value
```

4. [[deprecated]]

[[deprecated]]可用于将某些内容标记为已弃用，这意味着仍可以使用它，但不鼓励使用。此属性接受一个可选参数，该参数可用于解释弃用的原因，如以下示例所示。

```
[[deprecated("Unsafe function, please use xyz")]] void func();
```

如果使用了已弃用的函数，你将收到编译错误或警告。例如，GCC 会给出如下的警告信息：

```
warning: 'void func()' is deprecated: Unsafe function, please use xyz
```

5. [[likely]]和[[unlikely]]

这些可能性属性[[likely]]和[[unlikely]]可用于帮助编译器优化代码。例如，这些属性可用于根据某个分支被采用的可能性来标记 if 和 switch 语句的分支。但是，很少需要这些属性。如今，编译器和硬件具有强大的分支预测功能，可以自行解决。但某些情况下，例如对性能至关重要的代码，可能需要帮助编译器。语法如下：

```
int value { /* ... */ };
if (value > 11) [[unlikely]] { /* Do something ... */ }
else { /* Do something else ... */ }

switch (value)
{
   [[likely]] case 1:
      // Do something ...
      break;
   case 2:
      // Do something ...
      break;
   [[unlikely]] case 12:
      // Do something ...
      break;
}
```

C++23

6. [[assume]]

[[assume]]属性允许编译器假设某些表达式为真，而不必在运行时对其进行求值。编译器可以使用这些假设来更好地优化代码。作为一个例子，让我们看看以下函数：

```
int divideBy32(int x)
{
   return x / 32;
}
```

该函数接收一个有符号整数，因此编译器必须生成代码以确保除法对正数和负数都有效。如果你确定 x 永远不会是负数，并且由于某种原因不能使 x 的类型为 unsigned，你可以添加一个假设，如下所示：

```
int divideBy32(int x)
```

```
{
    [[assume(x >= 0)]];
    return x / 32;
}
```

有了这个假设，编译器可以省略任何处理负数的代码，并将除法优化为单条指令，即简单的五位右移。

1.1.14 C 风格的数组

警告：

本节简要介绍 C 风格的数组，因为你将在遗留代码中遇到它们。然而，在 C++中，最好避免使用 C 风格的数组，而是使用标准库功能，如 std::array 和 vector，这将在以下两节中讨论。

数组具有一系列值，所有值的类型相同，每个值都可根据它在数组中的位置进行访问。在 C++中声明数组时，必须声明数组的大小。数组的大小不能用变量表示—— 必须用常量或常量表达式(constexpr)表示数组大小。常量表达式将在第 9 章讨论。在下面的代码中，首先声明了具有 3 个整数的数组，之后的 3 行语句将每个元素初始化为 0。

```
int myArray[3];
myArray[0] = 0;
myArray[1] = 0;
myArray[2] = 0;
```

警告：

在 C++中，数组的第一个元素始终在位置 0，而非位置 1！数组的最后一个元素的位置始终是数组大小减 1！

下一节将讨论循环，使用循环可初始化每个元素。但不使用循环或前面的初始化机制，也可以使用如下单行代码完成将零初始化的操作。

```
int myArray[3] = { 0 };
```

甚至可以省略 0，如下所示：

```
int myArray[3] = {};
```

最后，等号也是可选的，所以可以写出如下所示的代码：

```
int myArray[3] {};
```

数组也可以用初始化列表初始化，此时编译器可自动推断出数组的大小。例如：

```
int myArray[] { 1, 2, 3, 4 }; // The compiler creates an array of 4 elements.
```

如果指定了数组的大小，而初始化列表包含的元素数量少于给定大小，则将其余元素设置为 0。例如，以下代码仅将数组中的第一个元素设置为 2，将其他元素设置为 0。

```
int myArray[3] { 2 };
```

要获取基于栈的 C 风格数组的大小，可使用 std::size()函数(需要<array>)。它返回 size_t 类型，这是在<cstddef>中定义的无符号整数类型。这是一个例子：

```
std::size_t arraySize { std::size(myArray) };
```

注意：

在遗留代码中，可能会看到 size_t 的使用没有 std 命名空间限定，没有 using 命名空间 std 指令，也没有 using std:: size_t 声明。当使用 import std 时，这将不再有效，因为这会将所有内容导入 std 命名空间。因此，需要使用 std::size_t 或使用适当的 using 指令或声明。另外，第 11 章中提到，可以导入命名模块 std.compat 而不是 std，但不建议将其用于新代码。

C++23

注意：

C++23 为 std::size_t 类型引入了一个字面量后缀 uz，例如 42uz。

获取基于栈的 C 风格数组的大小的一个更老的技巧是使用 sizeof 运算符。sizeof 运算符返回其参数的大小(以字节为单位)。要获取基于栈的数组中的元素数，可以将数组的大小(以字节为单位)除以第一个元素的大小(以字节为单位)。这是一个例子：

```
std::size_t arraySize { sizeof(myArray) / sizeof(myArray[0]) };
```

前面的代码显示了一个一维数组，可将其当作一行整数，每个数字都具有自己的编号。C++允许使用多维数组，可将二维数组看成棋盘，每个位置都具有 x 坐标值和 y 坐标值。三维数组可以被描绘成立方体，而高维数组则更难可视化。下面的代码显示了用于井字游戏棋盘的二维字符数组，然后在位于中央的方块中填充一个 o。

```
char ticTacToeBoard[3][3];
ticTacToeBoard[1][1] = 'o';
```

图 1-1 显示了这个棋盘的图形表示，并给出了每个方块的位置。

ticTacToeBoard[0][0]	ticTacToeBoard[0][1]	ticTacToeBoard[0][2]
ticTacToeBoard[1][0]	ticTacToeBoard[1][1]	ticTacToeBoard[1][2]
ticTacToeBoard[2][0]	ticTacToeBoard[2][1]	ticTacToeBoard[2][2]

图 1-1　棋盘的图形表示

1.1.15　std::array

上一节讨论的数组来自 C，仍能在 C++中使用。但 C++有一种固定大小的特殊容器 std::array，这种容器在<array>头文件中定义。它基本上是对 C 风格的数组进行了简单包装。

用 std::array 替代 C 风格的数组会带来很多好处。它总是知道自身大小；不会自动转换为指针，从而避免了某些类型的 bug；具有迭代器，可方便地遍历元素。第 17 章将详细讲述迭代器。

下例演示了 array 容器的用法。array 类型是一个接受多个类模板参数的类模板，这些参数允许指定要在容器中存储的元素数量及其类型。可在 array 后的尖括号之间指定类模板参数，如 array<int,3>。第 12 章将详细讨论模板。目前，你只需要记住，必须在尖括号中指定两个参数。第一个参数表示数组中元素的类型，第二个参数表示数组的大小。

```
array<int, 3> arr { 9, 8, 7 };
println("Array size = {}", arr.size());
println("2nd element = {}", arr[1]);
```

C++支持所谓的类模板参数推导(CTAD)，如第 12 章详细讨论的那样。目前请记住，这可以避免为某些类模板指定尖括号之间的模板类型。CTAD 仅在初始化时起作用，因为编译器使用此初始化自动推导模板类型。这适用于 std::array，允许你按以下方式定义以前的数组：

```
array arr { 9, 8, 7 };
```

注意：

C 风格的数组和 std::array 都具有固定的大小，在编译期必须知道这一点。在运行时数组不会增大或缩小。

如果希望数组的大小是动态的，推荐使用下一节介绍的 std::vector。在 vector 中添加新元素时，vector 的大小会自动增加。

1.1.16 std::vector

标准库提供了多个不同的非固定大小容器，可用于存储信息。std::vector 就是此类容器的一个示例，它在<vector>中声明，用一种更灵活和安全的机制取代 C 风格数组的概念。用户不需要担心内存的管理，因为 vector 将自动分配足够的内存来存放其元素。vector 是动态的，意味着可在运行时添加和删除元素。第 18 章将详细讨论容器，但 vector 的基本用法很简单。所以本书一开头就介绍它，以便在示例中使用。下面的示例演示了 vector 的基本功能：

```
// Create a vector of integers.
vector<int> myVector { 11, 22 };

// Add some more integers to the vector using push_back().
myVector.push_back(33);
myVector.push_back(44);

// Access elements.
println("1st element: {}", myVector[0]);
```

myVector 被声明为 vector<int>，尖括号用来指定模板参数，与本章前面的 std::array 一样。vector 是一个泛型容器，几乎可容纳任何类型的对象，但是 vector 中的所有元素必须是同一类型，在尖括号内指定这个类型。模板将在第 12 章和第 26 章详细讨论。

与 std::array 相同，vector 类模板支持 CTAD，允许你按下列方式定义 myVector。

```
vector myVector { 11, 22 };
```

再次说明，需要初始化器才能使 CTAD 正常工作。以下是非法的：

```
vector myVector;
```

为向 vector 中添加元素，可使用 push_back()方法。可使用类似于数组的语法(即 operator[])访问各个元素。

1.1.17 std::pair

std::pair 类模板定义在<utility>中。它将两个可能不同类型的值组合在一起。可通过 first 和 second

公共数据成员访问这些值。这是一个例子：

```
pair<double, int> myPair { 1.23, 5 };
println("{} {}", myPair.first, myPair.second);
```

pair 也支持 CTAD，所以你可按以下方式定义 myPair：

```
pair myPair { 1.23, 5 };
```

注意：

虽然你可以编写一个返回 std::pair 的函数，但建议编写一个包含这两个值的小结构体或类，并从函数中返回。返回 pair 的缺点是，客户代码必须使用 first 和 second 来访问这两个值。通过返回一个正确的结构体或类，可以为这两个值提供更有意义的名称。

1.1.18　std::optional

在<optional>中定义的 std::optional 保留特定类型的值，或者不包含任何值。在第 1 章就介绍它，因为在整本书的某些示例中它是一种有用的类型。

基本上，如果想要允许值是可选的，则可以将 optional 用于函数的参数。如果函数可能返回也可能不返回某些内容，则通常也将 optional 用作函数的返回类型。这消除了从函数中返回“特殊”值的需要，例如返回 nullptr、−1、EOF。它还消除了将函数编写为返回代表成功或失败的布尔值的需求，同时将函数的实际结果存储在作为输出参数传递给函数的实参中(类型为对非 const 的引用的参数，在本章稍后讨论)。

optional 类型是一个类模板，因此必须在尖括号之间指定所需的实际类型，如 optional<int>。此语法类似于指定存储在 vector 中的类型的方式，例如 vector <int>。

这是一个返回 optional 的函数的例子：

```
optional<int> getData(bool giveIt)
{
    if (giveIt) {
        return 42;
    }
    return nullopt;   // or simply return {};
}
```

可以按下列方式调用这个函数：

```
optional<int> data1 { getData(true) };
optional<int> data2 { getData(false) };
```

可以用 has_value()方法判断一个 optional 是否有值，或简单地将 optional 用在 if 语句中。

```
println("data1.has_value = {}", data1.has_value());
if (!data2) {
    println("data2 has no value.");
}
```

如果 optional 有值，可以使用 value()或解引用运算符*访问它。本章稍后将在指针的上下文中详细讨论此运算符。

```
println("data1.value = {}", data1.value());
```

```
println("data1.value = {}", *data1);
```

如果你对一个空的 optional 使用 value()，将抛出 std::bad_optional_access 异常。异常将会在本章稍后介绍。

value_or()可用来返回 optional 的值，如果 optional 为空，则返回指定的值。

```
println("data2.value = {}", data2.value_or(0));
```

不能将引用(将在本章稍后讨论)保存在 optional 中，所以 optional<T&>是无效的。但是，可以将指针保存在 optional 中。

1.1.19 结构化绑定

结构化绑定允许声明多个变量，这些变量使用数组、结构体、pair 或元组中的元素以初始化。

例如，假设有下面的数组：

```
array values { 11, 22, 33 };
```

可声明 3 个变量 x、y 和 z，像下面这样使用数组中的 3 个值进行初始化。注意，必须为结构化绑定使用 auto 关键字。例如，不能用 int 替代 auto。

```
auto [x, y, z] { values };
```

使用结构化绑定声明的变量数量必须与右侧表达式中的值数量匹配。

如果所有非静态成员都是公有的，也可将结构化绑定用于结构体。例如：

```
struct Point { double m_x, m_y, m_z; };
Point point;
point.m_x = 1.0; point.m_y = 2.0; point.m_z = 3.0;
auto [x, y, z] { point };
```

正如最后一个例子，以下代码段将 pair 中的元素分解为单独的变量：

```
pair myPair { "hello", 5 };
auto [theString, theInt] { myPair };  // Decompose using structured bindings.
println("theString: {}", theString);
println("theInt: {}", theInt);
```

通过使用 auto&或 const auto&代替 auto，还可用结构化绑定语法创建一组对非 const 的引用或 const 引用。稍后将讨论对非 const 的引用和 const 引用。

1.1.20 循环

计算机擅长重复执行类似的任务。C++提供了 4 种循环结构：while 循环、do/while 循环、for 循环和基于范围的 for 循环。

1. while 循环

只要条件表达式的求值结果为 true，while 循环就会重复执行一个代码块。例如，下面的代码会输出"This is silly."5 次。

```
int i { 0 };
while (i < 5) {
   println("This is silly.");
```

```
    ++i;
}
```

break 关键字可以在循环中使用，以立即退出循环，并从循环后的代码行开始恢复程序的执行。关键字 continue 可用来返回到循环顶部并对 while 表达式重新求值。然而，在循环中使用 continue 经常被认为是不良的编码风格，因为它们会使程序的执行产生无规则的跳转，应该慎用。

2. do/while 循环

C++还有一个版本的 while 循环，称为 do/while 循环。其运行方式类似于 while 循环，但会首先执行代码，而判断是否继续执行的条件检测被放在结尾处。如果想让代码块至少执行一次，并且根据某一条件确定是否多次执行，就可以使用这个循环版本。下面的代码尽管条件为 false，但仍会输出"This is silly."一次。

```
int i { 100 };
do {
    println("This is silly.");
    ++i;
} while (i < 5);
```

3. for 循环

for 循环提供了另一种循环语法。任何 for 循环都可转换为 while 循环，反之亦然。然而，for 循环的语法一般更简便，因为可看到循环的初始表达式、结束条件以及每次迭代结束后执行的语句。在下面的代码中，i 被初始化为 0；只要 i 小于 5，循环就会继续执行；每次迭代结束时，i 的值会增 1。这段代码的功能与 while 循环示例相同，但这段代码看起来更清晰，因为在一行中显示了初始值、结束条件以及每次迭代结束后执行的语句。

```
for (int i { 0 }; i < 5; ++i) {
    println("This is silly.");
}
```

4. 基于范围的 for 循环

基于范围的 for 循环是第 4 种循环，这种循环允许方便地迭代容器中的元素。这种循环类型可用于 C 风格的数组、初始化列表(稍后讨论)，也可用于任何具有返回迭代器的 begin()和 end()方法的类型(见第 17 章)，例如 std::array、vector 及第 18 章讨论的其他所有标准库容器。

下例首先定义一个包含 4 个整数的数组，此后“基于范围”的 for 循环遍历数组中的每个元素的副本，输出每个值。为在迭代元素时不制作副本，应使用本章后面讨论的引用变量。

```
array arr { 1, 2, 3, 4 };
for (int i : arr) { println("{}", i); }
```

基于范围的 for 循环的初始化器

可以在基于范围的 for 循环中使用初始化器，与 if 和 switch 语句中的用法相似，语法如下：

```
for (<initializer>; <range-declaration> : <range-expression>) { <body> }
```

任何在<initializer>中引入的变量只能用于< range-declaration >、< range-expression>和<body>中，不能被用于基于范围的 for 循环之外。下面是一个例子：

```
for (array arr { 1, 2, 3, 4 }; int i : arr) { println("{}", i); }
```

1.1.21 初始化列表

初始化列表在<initializer_list>头文件中定义；利用初始化列表，可轻松地编写能接收可变数量参数的函数。std::initializer_list 是一个类模板，要求在尖括号之间指定列表中的元素类型，这类似于指定 vector 中存储的元素类型。下例演示如何使用初始化列表：

```
import std;
using namespace std;

int sum(initializer_list<int> values)
{
   int total { 0 };
   for (int value : values) {
      total += value;
   }
   return total;
}
```

通过接收整数的初始化列表作为参数，可以使用带大括号的整数初始化器作为参数调用函数 sum()。函数体使用基于范围的 for 循环来累加总和。可按如下方式使用该函数：

```
int a { sum({ 1, 2, 3 }) };
int b { sum({ 10, 20, 30, 40, 50, 60 }) };
```

初始化列表是类型安全的，列表中所有元素必须为同一类型。对于此处的 sum()函数，初始化列表中的所有元素都必须是整数。尝试使用 double 数值进行调用，将导致编译器生成错误或警告，表示从 double 到 int 的转换是一种窄化，如下所示：

```
int c { sum({ 1, 2, 3.0 }) };
```

1.1.22 C++中的字符串

在 C++中使用字符串有两种方法。

- C 风格字符串：用字符数组表示字符串
- C++风格字符串：将 C 风格的表示封装到一种易于使用的、更安全的 string 类型中

第 2 章将会有详细论述。现在，只需要知道，C++ 中 std::string 类型在<string>头文件中定义，C++ string 的用法与基本类型几乎相同。下面的示例说明了 string 如何像字符数组那样使用：

```
string myString { "Hello, World" };
println("The value of myString is {}", myString);
println("The second letter is {}", myString[1]);
```

1.1.23 作为面向对象语言的 C++

如果你是一位 C 程序员，可能会认为本章讲述的内容到目前为止只是传统 C 语言的补充。顾名思义，C++语言在很多方面只是“更好的 C”。这种观点忽略了一个重点：与 C 不同，C++是一种面向对象的语言。

面向对象程序设计(OOP)是一种完全不同的、更趋自然的编码方式。如果习惯使用过程语言，如 C 或者 Pascal，不要担心。第 5 章讲述将观念转换到面向对象范式需要的所有背景知识。如果你已经了解 OOP 的理论，下面的内容将帮助你加速了解(或者回顾)基本的 C++对象语法。

1. 定义类

类定义了对象的特征。在 C++中，类通常在模块接口文件(.cppm)中定义和被导出，然而类的方法定义既可以在相同的模块接口文件中，也可以在对应的模块实现文件(.cpp)中。第 11 章将会深入讨论模块。

下面的示例定义了一个基本的机票类。这个类可根据飞行的里程数以及顾客是不是“精英超级奖励计划”的成员计算票价。

这个定义首先声明一个类名，在大括号内声明了类的数据成员(属性)及方法(行为)。每个数据成员及方法都具有特定的访问级别：public、protected 或 private。这些标记可按任意顺序出现，也可重复使用。public 成员可在类的外部访问，private 成员不能在类的外部访问，protected 成员可以通过派生类访问，详见第 10 章关于继承的部分。推荐把所有数据成员都声明为 private，在需要时，可通过 public 或 protected 的获取器(getter)来访问它们，通过设置器(setter)来改变它们。这样，就很容易改变数据的表达方式，同时使 public/protected 接口保持不变。

请记住，当写一个模块接口文件时，不要忘记使用 export module 声明来表明你正在写哪个模块，同时不要忘记将那些你希望对模块的使用者可用的类型显式地导出。

```
export module airline_ticket;

import std;

export class AirlineTicket
{
   public:
      AirlineTicket();
      ~AirlineTicket();

      double calculatePriceInDollars();

      std::string getPassengerName();
      void setPassengerName(std::string name);

      int getNumberOfMiles();
      void setNumberOfMiles(int miles);

      bool hasEliteSuperRewardsStatus();
      void setHasEliteSuperRewardsStatus(bool status);
   private:
      std::string m_passengerName;
      int m_numberOfMiles;
      bool m_hasEliteSuperRewardsStatus;
};
```

本书遵循这样一个约定：在类的每个数据成员之前加上小写字母 m 的前缀，后跟一个下画线，如 m_passengerName。

与类同名但没有返回类型的方法是构造函数，当创建类的对象时会自动调用构造函数。~之后紧接着类名的方法是析构函数，当销毁对象时会自动调用。

模块接口文件(.cppm)中定义了类，然而在本例中，方法的实现位于模块实现文件(.cpp)中。源文件以如下的模块声明开头，告诉编译器这是 ariline_ticket 模块的源文件。

```
module airline_ticket;
```

可通过几种方法初始化数据成员。一种方法是使用构造函数初始化器(constructor initializer)，即在构造函数名称之后加上冒号。下面是包含构造函数初始化器的 AirlineTicket 构造函数：

```
AirlineTicket::AirlineTicket()
    : m_passengerName { "Unknown Passenger" }
    , m_numberOfMiles { 0 }
    , m_hasEliteSuperRewardsStatus { false }
{
}
```

第二种方法是将初始化任务放在构造函数体中，如下所示。

```
AirlineTicket::AirlineTicket()
{
    // Initialize data members.
    m_passengerName = "Unknown Passenger";
    m_numberOfMiles = 0;
    m_hasEliteSuperRewardsStatus = false;
}
```

然而，如果构造函数只是初始化数据成员，而不做其他事情，实际上就没必要使用构造函数，因为可在类定义中直接初始化数据成员，也称为类内初始化(in-class initializer)。例如，不编写 AirlineTicket 构造函数，而是修改类定义中数据成员的定义，如下所示。

```
    private:
        std::string m_passengerName { "Unknown Passenger" };
        int m_numberOfMiles { 0 };
        bool m_hasEliteSuperRewardsStatus { false };
```

如果类还需要执行其他一些初始化类型，如打开文件、分配内存等，则需要编写构造函数进行处理。

下面是 AirlineTicket 类的析构函数：

```
AirlineTicket::~AirlineTicket()
{
    // Nothing to do in terms of cleanup
}
```

这个析构函数什么都不做，因此可从类中删除。这里之所以展示它，是为了让你了解析构函数的语法。如果需要执行一些清理，如关闭文件、释放内存等，则需要使用析构函数。第 8 章和第 9 章详细讨论析构函数。

一些 AirlineTicket 类方法的定义如下所示：

```
double AirlineTicket::calculatePriceInDollars()
{
    if (hasEliteSuperRewardsStatus()) {
        // Elite Super Rewards customers fly for free!
        return 0;
    }
    // The cost of the ticket is the number of miles times 0.1.
    // Real airlines probably have a more complicated formula!
    return getNumberOfMiles() * 0.1;
}

string AirlineTicket::getPassengerName() { return m_passengerName; }
```

```
void AirlineTicket::setPassengerName(string name) { m_passengerName = name; }

int AirlineTicket::getNumberOfMiles() { return m_numberOfMiles; }
void AirlineTicket::setNumberOfMiles(int miles) { m_numberOfMiles = miles; }

bool AirlineTicket::hasEliteSuperRewardsStatus()
{
   return m_hasEliteSuperRewardsStatus;
}
void AirlineTicket::setHasEliteSuperRewardsStatus(bool status)
{
   m_hasEliteSuperRewardsStatus = status;
}
```

如本节开头所述，也可以将方法实现直接放在模块接口文件中。语法如下：

```
export class AirlineTicket
{
   public:
      double calculatePriceInDollars()
      {
         if (hasEliteSuperRewardsStatus()) { return 0; }
         return getNumberOfMiles() * 0.1;
      }

      std::string getPassengerName() { return m_passengerName; }
      void setPassengerName(std::string name) { m_passengerName = name; }

      int getNumberOfMiles() { return m_numberOfMiles; }
      void setNumberOfMiles(int miles) { m_numberOfMiles = miles; }

      bool hasEliteSuperRewardsStatus() { return m_hasEliteSuperRewardsStatus; }
      void setHasEliteSuperRewardsStatus(bool status)
      {
         m_hasEliteSuperRewardsStatus = status;
      }
   private:
      std::string m_passengerName { "Unknown Passenger" };
      int m_numberOfMiles { 0 };
      bool m_hasEliteSuperRewardsStatus { false };
};
```

2. 使用类

为了使用 AirlineTicket 类，需要首先导入它的模块。

```
import airline_ticket;
```

下面的示例程序使用了 AirlineTicket 类。这个示例创建的 AirlineTicket 对象基于栈。

```
AirlineTicket myTicket;
myTicket.setPassengerName("Sherman T. Socketwrench");
myTicket.setNumberOfMiles(700);
double cost { myTicket.calculatePriceInDollars() };
println("This ticket will cost ${}", cost);
```

AirlineTicket 示例展示了创建和使用类的一般语法。当然，还有很多东西要学，这就是第 8 章、

第 9 章和第 10 章的主题。

1.1.24 作用域解析

作为 C++程序员，需要熟悉作用域(scope)的概念，作用域定义了项目的可见性。程序中的每个名称(包括变量、函数和类名称)都在某个作用域内。可以使用命名空间、函数定义、用花括号分隔的块和类定义来创建作用域。在 for 循环和基于范围的 for 循环的初始化语句中初始化的变量的作用域为 for 循环之内，在 for 循环之外不可见。同样，在 if 或 switch 语句中初始化的变量的作用域为 if 或 switch 语句，并且在该语句之外不可见。当你尝试访问一个变量、函数或类时，将首先在最近的封闭作用域内查找名称，然后在父作用域内查找，以此类推，直到全局作用域。不在命名空间、函数、用花括号分隔的块或类中的任何名称都被视为在全局作用域内。如果在全局作用域内未找到，编译器将提示未定义的符号错误。

有时，作用域中的名称会覆盖其他作用域中相同的名称。有时，你所需的作用域不是程序中某特定行的默认作用域。如果你不希望名称使用默认作用域解析，则可以使用作用域解析运算符::限定特定作用域的名称。下面的示例演示了这一点。该示例定义了一个 Demo 类，类中有一个 get()方法，还定义了一个全局作用域下的 get()函数，以及一个 NS 命名空间里的 get()函数。

```
class Demo
{
   public:
      int get() { return 5; }
};

int get() { return 10; }

namespace NS
{
   int get() { return 20; }
}
```

全局作用域是未命名的，但是你可以单独使用作用域解析运算符(不带名称前缀)来专门访问它。可以按以下方式调用不同的 get()函数。在此示例中，代码本身位于 main()函数中，该函数始终位于全局范围内。

```
int main()
{
   Demo d;
   println("{}", d.get());       // prints 5
   println("{}", NS::get());     // prints 20
   println("{}", ::get());       // prints 10
   println("{}", get());         // prints 10
}
```

如果将前边名为 NS 的命名空间定义为未命名/匿名(即没有名字)的命名空间，如下所示：

```
namespace
{
   int get() { return 20; }
}
```

则以下代码将导致有歧义的名称解析的编译错误，因为你会有一个定义在全局作用域中的 get()，以及一个定义在未命名的命名空间中的 get()。

```
println("{}", get());
```

如果你在 main 函数之前使用了如下的 using 命令，也会发生同样的错误。

```
using namespace NS;
```

1.1.25　统一初始化

在 C++11 之前，各类型的初始化并非总是统一的。例如，考虑下面的两个定义，其中一个作为结构体，另一个作为类。

```
struct CircleStruct
{
    int x, y;
    double radius;
};

class CircleClass
{
    public:
        CircleClass(int x, int y, double radius)
            : m_x { x }, m_y { y }, m_radius { radius } {}
    private:
        int m_x, m_y;
        double m_radius;
};
```

在 C++11 之前，CircleStruct 类型变量和 CircleClass 类型变量的初始化是不同的。

```
CircleStruct myCircle1 = { 10, 10, 2.5 };
CircleClass myCircle2(10, 10, 2.5);
```

对于结构体版本，可使用{...}语法。然而，对于类版本，需要使用函数符号(...)调用构造函数。自 C++11 以后，允许一律使用{...}语法来初始化类型，如下所示。

```
CircleStruct myCircle3 = { 10, 10, 2.5 };
CircleClass myCircle4 = { 10, 10, 2.5 };
```

定义 myCircle4 时将自动调用 CircleClass 的构造函数。甚至等号也是可选的，因此下面的代码与前面的代码等价：

```
CircleStruct myCircle5 { 10, 10, 2.5 };
CircleClass myCircle6 { 10, 10, 2.5 };
```

在本章前面“结构体”一节中出现的另一个例子中，一个 Employee 结构体用如下方式初始化。

```
Employee anEmployee;
anEmployee.firstInitial = 'J';
anEmployee.lastInitial = 'D';
anEmployee.employeeNumber = 42;
anEmployee.salary = 80'000;
```

使用统一初始化，可以写成这样：

```
Employee anEmployee { 'J', 'D', 42, 80'000 };
```

使用统一初始化并不局限于结构体和类，它还可用于初始化 C++中的任何内容。例如，下面的代

码把所有 4 个变量都初始化为 3。

```
int a = 3;
int b(3);
int c = { 3 };  // Uniform initialization
int d { 3 };    // Uniform initialization
```

统一初始化还可用于对变量进行零初始化，只需要指定一对空的大括号。例如：

```
int e { };      // Uniform initialization, e will be 0
```

这种语法也可以用于结构体。如果按如下方式创建 Employee 结构体的实例，则其数据成员将被默认初始化，对于 char 和 int 等基本类型，这意味着它们将包含内存中剩余的任何随机数据：

```
Employee anEmployee;
```

但是，如果按如下方式创建实例，则所有数据成员都将初始化为零：

```
Employee anEmployee { };
```

使用统一初始化的一个优点是可以阻止窄化(narrowing)。当使用旧式风格的赋值语法初始化变量时，C++隐式地执行窄化。例如：

```
int main()
{
   int x = 3.14;
}
```

在 main()的代码中，C++在对 x 赋值之前，会自动将 3.14 截断为 3。有些编译器可能会针对窄化给出警告信息，而另一些编译器则不会。在任何情况下，窄化转换都不应被忽视，因为它们可能会引起微妙的错误。使用统一初始化，如果编译器完全支持 C++11 标准，x 的赋值会生成编译错误。

```
int x { 3.14 };     // Error because narrowing
```

如果你需要窄化转换，建议使用准则支持库(GSL)中提供的 gsl::narrow_cast()函数。

统一初始化也可以在构造函数初始化器中用于初始化作为类成员的数组。

```
class MyClass
{
   public:
       MyClass()
          : m_array { 0, 1, 2, 3 }
       {
       }
   private:
       int m_array[4];
};
```

统一初始化还可用于标准库容器，如本章前面提到的 std::vector。

注意：

考虑到所有这些好处，建议使用统一初始化，而不是使用赋值语法初始化变量。因此，本书尽可能使用统一初始化。

指派初始化器

指派初始化器(designated initializer)可以使用它们的名称初始化所谓“聚合”的数据成员。聚合类

型是满足以下限制的数组类型的对象(或结构体或类的对象)：仅有 public 数据成员，无用户声明或继承的构造函数，无虚函数，无 virtual、private 或 protected 基类(请参见第 10 章)。指派初始化器以点开头，后跟数据成员的名称。指派初始化的顺序必须与数据成员的声明顺序相同。不允许混合使用指派初始化器和非指派初始化器。未使用指派初始化器初始化的任何数据成员都将使用其默认值进行初始化，这意味着：

- 拥有类内初始化器的数据成员会得到该值。
- 没有类内初始化器的数据成员会被零初始化。

让我们看一下略微修改的 Employee 结构体，这次，salary 数据成员的默认值为 75 000。

```
struct Employee {
    char firstInitial;
    char lastInitial;
    int  employeeNumber;
    int  salary { 75'000 };
};
```

在本章的前面，这种 Employee 结构体是使用如下的统一初始化语法初始化的：

```
Employee anEmployee { 'J', 'D', 42, 80'000 };
```

使用指派初始化器，可以被写成这样：

```
Employee anEmployee {
    .firstInitial = 'J',
    .lastInitial = 'D',
    .employeeNumber = 42,
    .salary = 80'000
};
```

使用指派初始化器的好处是，与使用统一初始化语法相比，它更容易理解指派初始化器正在初始化的内容。

使用指派初始化器，如果对某些成员的默认值感到满意，则可以跳过它们的初始化。例如，在创建员工时，可以跳过初始化 employeeNumber，这种情况下，employeeNumber 初始化为零，因为它没有类内初始化器。

```
Employee anEmployee {
    .firstInitial = 'J',
    .lastInitial = 'D',
    .salary = 80'000
};
```

如果使用统一初始化语法，上述做法是不可行的，必须像下面这样指定 employeeNumber 为 0。

```
Employee anEmployee { 'J', 'D', 0, 80'000 };
```

如果你像下面这样跳过了初始化 salary 数据成员，它就会得到它的默认值，即它的类内初始化值，75 000。

```
Employee anEmployee {
    .firstInitial = 'J',
    .lastInitial = 'D'
};
```

使用指派初始化器的最后一个好处是，当新成员被添加到数据结构时，使用指派初始化器的现有代码将继续起作用。新的数据成员将使用其默认值进行初始化。

1.1.26　指针和动态内存

动态内存允许所创建的程序具有在编译期大小可变的数据，大多数复杂程序都会以某种方式使用动态内存。

1. 栈和自由存储区

C++程序中的内存分为两部分——栈(stack)和自由存储区(free store)。将栈可视化的一种方法就是将其看作一副纸牌，当前顶部的牌代表程序当前的作用域，通常是当前正在执行的函数。当前函数中声明的所有变量将占用顶部栈帧(也就是最上面的那张牌)的内存。如果当前函数(称为 foo())调用了另一个函数 bar()，一张新牌就会被放在牌堆上面，这样 bar()就会拥有自己的栈帧供其运行。任何从 foo()传递给 bar()的参数都会从 foo()栈帧复制到 bar()栈帧。图 1-2 显示了执行假想的 foo()函数时栈的情形，foo()函数中声明了两个整型值。

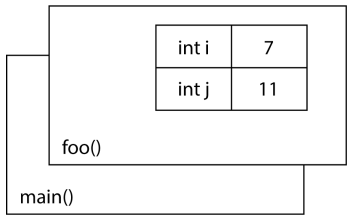

图 1-2　foo()函数中声明了两个整型值

栈帧很好，因为它为每个函数提供了独立的内存空间。如果在 foo()栈帧中声明了一个变量，那么除非专门要求，否则调用 bar()函数不会更改该变量。此外，foo()函数执行完毕时，栈帧就会消失，该函数中声明的所有变量都不会再占用内存。在栈上分配内存的变量不需要由程序员释放内存(删除)，这个过程是自动完成的。

自由存储区是与当前函数或栈帧完全独立的内存区域。如果想在函数调用结束之后仍然保存其中声明的变量，可以将变量放到自由存储区中。自由存储区的结构不如栈复杂，可以将它当作一堆位。程序可在任何时候向其中添加新的位或修改已有的位。必须确保释放(删除)在自由存储区上分配的任何内存，这个过程不会自动完成，除非使用了智能指针。智能指针将在第 7 章详细讨论。

警告：

这里介绍指针是因为你将会遇到它们，尤其是在遗留代码中。但是，在新代码中，仅在不涉及所有权的情况下，才允许使用此类原始/裸指针。否则，应该使用第 7 章介绍的智能指针之一。

2. 使用指针

可以通过显式分配内存的方式将任何东西放到自由存储区中。例如，要将一个整数放在自由存储区中，需要为其分配内存，但是首先需要声明一个指针：

```
int* myIntegerPointer;
```

int 类型后面的*表示，所声明的变量引用/指向某个整数内存。可将指针看作指向动态分配自由存储区中内存的一个箭头，它还没有指向任何内容，因为你还没有把它指派给任何内容，它是一个未初始化的变量。在任何时候都应避免使用未初始化的变量，尤其是未初始化的指针，因为它们会指向内

存中的某个随机位置。使用这种指针很可能使程序崩溃。这就是总是应该同时声明和初始化指针的原因。如果不希望立即分配内存，可以把它们初始化为空指针 nullptr(详见后面的“空指针常量”小节)。

```
int* myIntegerPointer { nullptr };
```

空指针是一个特殊的默认值，有效的指针都不含该值，在布尔表达式中使用时会被转换为 false。例如：

```
if (!myIntegerPointer) { /* myIntegerPointer is a null pointer. */ }
```

使用 new 操作符分配内存：

```
myIntegerPointer = new int;
```

在此情况下，指针指向一个整数值的地址。为访问这个值，需要对指针解引用。可将解引用看作沿着指针箭头寻找自由存储区中实际的值。为给自由存储区中新分配的整数赋值，可采用如下代码：

```
*myIntegerPointer = 8;
```

注意，这并非将 myIntegerPointer 的值设置为 8，在此并没有改变指针，而是改变了指针所指的内存。如果真要重新设置指针的值，它将指向内存地址 8，这可能是一个随机的无用内存单元，最终导致程序崩溃。

使用完动态分配的内存后，需要使用 delete 操作符释放内存。为防止在释放指针指向的内存后再使用指针，建议把指针设置为 nullptr。

```
delete myIntegerPointer;
myIntegerPointer = nullptr;
```

警告：

在解引用之前指针必须有效。对 null 或未初始化的指针解引用会导致未定义的行为。程序可能崩溃，也可能继续运行，却给出奇怪的结果。

指针并非总是指向自由存储区内存，可声明一个指向栈中变量甚至指向其他指针的指针。为让指针指向某个变量，需要使用“取址”运算符&。

```
int i { 8 };
int* myIntegerPointer { &i };  // Points to the variable with the value 8
```

C++使用特殊语法处理指向结构体或类的指针。从技术角度看，如果指针指向某个结构体或类，可以首先用“*”对指针解引用，然后使用普通的“.”语法访问其中的字段，如下面的代码所示。代码还演示了如何动态分配 Employee 实例及如何解除分配。

```
Employee* anEmployee { new Employee { 'J', 'D', 42, 80'000 } };
println("{}", (*anEmployee).salary);
delete anEmployee; anEmployee = nullptr;
```

此语法有一点混乱。->(箭头)运算符允许同时对指针解引用并访问字段。下面的代码与前面的代码等效，但阅读起来更方便。

```
println("{}", anEmployee->salary);
```

还记得本章前面介绍的短路逻辑吗？这种做法可与指针一起使用，以免使用无效指针，如下

所示。

```
bool isValidSalary { anEmployee && anEmployee->salary > 0 };
```

或者稍微详细一点：

```
bool isValidSalary { anEmployee != nullptr && anEmployee->salary > 0 };
```

仅当 anEmployee 有效时，才对其进行解引用以获取 salary。如果它是一个空指针，则逻辑运算短路，不再解引用 anEmployee 指针。

3. 动态分配的数组

自由存储区也可以用于动态分配数组。使用 new[]操作符可给数组分配内存：

```
int arraySize { 8 };
int* myVariableSizedArray { new int[arraySize] };
```

这条语句分配足够的内存，用于存储 arraySize 个整数。图 1-3 显示了执行这条语句后栈和自由存储区的情况。可以看到，指针变量仍在栈中，但动态创建的数组在自由存储区中。

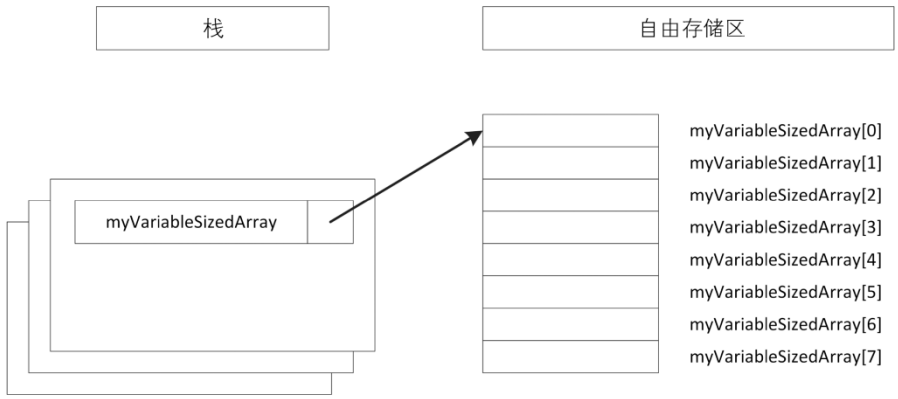

图 1-3　自由存储区

现在已经分配了内存，可将 myVariableSizedArray 当作基于栈的普通数组使用。

```
myVariableSizedArray[3] = 2;
```

使用完这个数组后，应该将其从自由存储区中删除，这样其他变量就可以使用这块内存。在 C++中，可使用 delete[]操作符完成这一任务。

```
delete[] myVariableSizedArray;
myVariableSizedArray = nullptr;
```

delete 后的方括号表明所删除的是一个数组！

注意：

如果你确实需要动态分配内存，请避免使用 malloc()和 free()。相反，使用 new 和 delete，或 new[]和 delete[]。然而，在现代 C++中，目标是完全避免 new、delete、new[]和 delete[]，并使用更现代的构造，例如标准库容器(如 std::vector)和第 7 章讲述的智能指针。

警告：

在 C++中，每次调用 new 时，都必须相应地调用 delete；每次调用 new[]时，都必须相应地调用 delete[]，以避免内存泄漏。如果未调用 delete 或 delete[]，或调用不匹配，会导致内存泄漏。第 7 章将讨论内存泄漏。

4. 空指针常量

在 C++11 之前，常量 NULL 用于表示空指针，它定义在<cstddef>中。无法使用任何 import 声明访问此常量，必须使用#include<cstddef>。NULL 只是被简单地定义为常量 0，这会导致一些问题。分析下面的例子：

```
#include <cstddef>

void func(int i) { /* … */ }

int main()
{
    func(NULL);
}
```

这段代码定义了一个 func()函数，它有一个整型参数。main()函数通过参数 NULL 调用 func()，NULL 被当作一个空指针常量。但是，NULL 不是指针，而等价于整数 0，所以实际调用的是 func(int)。这可能不是预期的行为，因此，有些编译器会给出警告。

可引入真正的空指针常量 nullptr 来解决这个问题。下面的代码使用了真正的空指针，并且导致了编译错误，因为我们没有重载参数为指针的 func()版本。

```
func(nullptr);
```

1.1.27　const 的用法

在 C++中有多种方法使用 const 关键字。所有用法都是相关的，但存在微妙的差别。const 的细微之处造就了绝佳的面试问题。

基本上，关键字 const 是 constant 的缩写，它表示某些内容保持不变。编译器通过将任何试图将其更改的行为标记为错误，来保证此要求。此外，启用优化后，编译器可以利用此知识生成更好的代码。

1. const 修饰类型

如果已经认为关键字 const 与常量有一定关系，就正确揭示了它的一种用法。在 C 语言中，程序员经常使用预处理器的#define 机制声明一个符号名称，其值在程序执行时不会变化，例如版本号。在 C++中，鼓励程序员使用 const 取代#define 定义常量。使用 const 定义常量就像定义变量一样，只是编译器保证代码不会改变这个值。下面是几个例子：

```
const int versionNumberMajor { 2 };
const int versionNumberMinor { 1 };
const std::string productName { "Super Hyper Net Modulator" };
const double PI { 3.14159265358979238462 };
```

可以将任何变量标记为 const，包括全局变量和类中的数据成员。

const 与指针

当变量通过指针包含一层或多层间接时，应用 const 将变得更加棘手。考虑以下代码：

```
int* ip;
ip = new int[10];
ip[4] = 5;
```

假设你决定对 ip 使用 const。暂时不要考虑这样做的用处，考虑它意味着什么。你是要阻止 ip 变量本身被更改，还是要阻止其指向的值被更改？也就是说，你要阻止第二条语句还是第三条？

为了防止指向的值被修改(如第三条所示)，可以用下面这种方式将关键字 const 添加到 ip 的声明中。

```
const int* ip;
ip = new int[10];
ip[4] = 5; // DOES NOT COMPILE!
```

现在，你无法更改 ip 指向的值。一种替代的但在语义上等效的书写方式如下：

```
int const* ip;
ip = new int[10];
ip[4] = 5; // DOES NOT COMPILE!
```

将 const 放在 int 之前还是之后在功能上并没有区别。

如果想将 ip 本身标记为 const，而不是它指向的值，需要这样写：

```
int* const ip { nullptr };
ip = new int[10];    // DOES NOT COMPILE!
ip[4] = 5;           // Error: dereferencing a null pointer
```

现在，ip 本身无法更改，编译器要求你在声明它时对其进行初始化，可以使用如先前代码中的 nullptr 或如下所示的新分配的内存。

```
int* const ip { new int[10] };
ip[4] = 5;
```

也可以像下面这样，将指针本身和指针所指的值都标记为 const。

```
int const* const ip { nullptr };
```

这是另一种等效的写法：

```
const int* const ip { nullptr };
```

尽管此语法可能看起来令人困惑，但实际上存在一个简单的规则：const 关键字作用于与其紧挨着的左侧的内容。再次考虑这一行：

```
int const* const ip { nullptr };
```

从左到右，第一个 const 直接位于单词 int 的右侧。因此，它适用于 ip 指向的 int。它指定你不能更改 ip 指向的值。第二个 const 直接位于“*”的右侧。它适用于指向 int 的指针，该指针是 ip 变量。因此，它指定你不能更改 ip(指针)本身。

该规则令人困惑的原因是一个例外。也就是说，第一个 const 可放在变量之前，如下所示。

```
const int* const ip { nullptr };
```

这种“例外”的语法比其他语法更常遇到。

可以将这个规则扩展到任意级别的间接等级，如以下示例所示：

```
const int * const * const * const ip { nullptr };
```

注意：

*有另一个易于记忆的规则，可以用于读懂复杂的变量声明：从右向左读。例如，int * const ip 从右到左读取，得到“ip 是指向 int 的 const 指针。”另外，int const * ip 读为“ip 是指向 const int 的指针”，而 const int * ip 读为“ip 是指向 int 常量的指针”。*

使用 const 保护参数

在 C++中，可将非 const 变量转换为 const 变量。为什么想这么做呢？这提供了一定程度的保护，防止其他代码修改变量。如果你调用同事编写的一个函数，并且想确保这个函数不会改变传递给它的实参，可以告诉同事让函数采用 const 参数。如果这个函数试图改变参数的值，就不会通过编译。

在下面的代码中，调用 mysteryFunction()时 string*自动转换为 const string*。如果编写 mysteryFunction()的人员试图修改所传递字符串的值，代码将无法编译。有绕过这个限制的方法，但是需要有意识地这么做，C++只是阻止无意识地修改 const 变量。

```
void mysteryFunction(const string* someString)
{
    *someString = "Test";  // Will not compile
}

int main()
{
    string myString { "The string" };
    mysteryFunction(&myString);  // &myString is a string*
}
```

还可以在原始类型参数上使用 const，以防止在函数体中意外更改它们。例如，以下函数具有 const 整数参数。在函数体中，无法修改整数 param。如果尝试对其进行修改，则编译器将生成错误。

```
void func(const int param) { /* Not allowed to change param... */ }
```

2. const 成员函数

const 关键字的第二个用途是将成员函数标记为 const，以防止它们修改类的数据成员。可以修改前面介绍的 AirlineTicket 类，以将所有只读成员函数标记为 const。const 必须同时添加到成员函数声明及其定义中。如果任何 const 成员函数尝试修改 AirlineTicket 数据成员之一，则编译器将提示错误。

```
export class AirlineTicket
{
   public:
      double calculatePriceInDollars() const;

      std::string getPassengerName() const;
      void setPassengerName(std::string name);

      int getNumberOfMiles() const;
      void setNumberOfMiles(int miles);
```

```
        bool hasEliteSuperRewardsStatus() const;
        void setHasEliteSuperRewardsStatus(bool status);
    private:
        std::string m_passengerName { "Unknown Passenger" };
        int m_numberOfMiles { 0 };
        bool m_hasEliteSuperRewardsStatus { false };
};

std::string AirlineTicket::getPassengerName() const
{
    return m_passengerName;
}
// Other member functions omitted...
```

注意：

为了遵循 const-correctness 原则，建议将不改变对象的任何数据成员的成员函数声明为 const。与非 const 成员函数被称为赋值函数(mutator)相对，这些成员函数也称为检查器(inspector)。

1.1.28 引用

专业的 C++代码，包括本书中的许多代码，都广泛使用了引用。C++中的引用(reference)是另一个变量的别名。对引用的所有修改都会更改其引用的变量的值。可以将引用视为隐式指针，它省去了获取变量地址和解引用指针的麻烦。另外，可将引用视为原始变量的另一个名称。可以创建独立的引用变量，在类中使用引用数据成员，接受引用作为函数和方法的参数，并从函数返回引用。

1. 引用变量

引用变量必须在创建时被初始化，例如：

```
int x { 3 };
int& xRef { x };
```

给类型附加一个&，则指示相应的变量是一个引用。它仍然像正常变量一样被使用，但是在幕后，它实际上是指向原始变量的指针。变量 x 和引用变量 xRef 指向同一个值，也就是说，xRef 只是 x 的另一个名称。如果通过其中一个更改值，则也可在另一个中看到更改。例如，以下代码通过 xRef 将 x 设置为 10：

```
xRef = 10;
```

不允许在类定义之外声明一个引用而不对其进行初始化。

```
int& emptyRef; // DOES NOT COMPILE!
```

警告：

引用变量必须总是在创建时被初始化。

修改引用

引用始终指向它初始化时的那个变量，引用一旦创建便无法更改。对于刚开始使用 C++的程序员来说，语法可能会令人困惑。如果在声明引用时将变量赋值给引用，则引用指向该变量。但是，如果之后将变量赋值给引用，则引用所指向的变量会更改为赋值的变量的值。原来的引用不会改为指向新的变量。下面是一个代码示例：

```
int x { 3 }, y { 4 };
int& xRef { x };
xRef = y; // Changes value of x to 4. Doesn't make xRef refer to y.
```

你可能想尝试使用 y 的地址对 xRef 赋值来规避此限制。

```
xRef = &y; // DOES NOT COMPILE!
```

这句代码会编译失败。y 的地址是一个指针，但是 xRef 被声明为一个对 int 的引用，而不是对指针的引用。

一些程序员在绕过引用的预期语义的尝试中走得更远。如果将引用赋值给引用会怎么办？这样会使第一个引用指向第二个引用指向的变量吗？你可能会想尝试以下代码：

```
int x { 3 }, z { 5 };
int& xRef { x };
int& zRef { z };
zRef = xRef; // Assigns values, not references
```

最后一行不会更改 zRef 的指向，而是将 z 的值设置为 3，因为 xRef 引用 x，即 3。

警告：

一旦将引用初始化为引用特定变量，就无法将引用更改为引用另一个变量。只能更改引用所指向的变量的值。

const 引用

应用于引用的 const 通常比应用于指针的 const 容易，这有两个原因。首先，引用默认是 const，因为你不能更改它们的指向。因此，不需要显式标记它们为 const。其次，你无法创建对引用的引用，因此通常只有一个间接引用级别。获得多个间接级别的唯一方法是创建对指针的引用。

因此，当 C++程序员提起 const 引用时，他们的意思是这样的：

```
int z;
const int& zRef { z };
zRef = 4; // DOES NOT COMPILE
```

通过将 const 应用于 int&，可以阻止对 zRef 的赋值，如上所示。类似于指针，const int& zRef 等价于 int const& zRef。但是请注意，将 zRef 标记为 const 对 z 无效。仍然可以通过直接更改 z 的值而不是通过引用来更改 z 的值。

不能创建对未命名值的引用，例如整数字面量，除非该引用是 const 值。在下面的示例中，unnamedRef1 会编译失败，因为它是对非 const 的引用，却指向了一个常量。那意味着你可以更改常数 5 的值，这没有任何意义。unnamedRef2 之所以有效，是因为它是 const 引用，因此不能编写诸如 unnamedRef2 = 7 的代码。

```
int& unnamedRef1 { 5 };             // DOES NOT COMPILE
const int& unnamedRef2 { 5 };       // Works as expected
```

临时对象也是如此。不能为临时对象创建对非 const 的引用，但是 const 引用是可以的。例如，假设具有以下返回 std :: string 对象的函数：

```
string getString() { return "Hello world!"; }
```

可以为 getString()的结果创建一个 const 引用，该引用将使临时 std::string 对象保持生命周期，直到该引用超出作用域。

```
string& string1 { getString() };        // DOES NOT COMPILE
const string& string2 { getString() }; // Works as expected
```

指针的引用和引用的指针

可以创建对任何类型的引用，包括指针类型。这是对指向 int 的指针的引用的示例：

```
int* intP { nullptr };
int*& ptrRef { intP };
ptrRef = new int;
*ptrRef = 5;
delete ptrRef; ptrRef = nullptr;
```

语法有点奇怪：你可能不习惯看到*和&彼此相邻。但是，语义很简单：ptrRef 是对 intP 的引用，intP 是对 int 的指针。修改 ptrRef 会更改 intP。对指针的引用很少见，但有时可能有用。

取一个引用的地址与取该引用所指向的变量的地址得到的结果是相同的。这是一个示例：

```
int x { 3 };
int& xRef { x };
int* xPtr { &xRef }; // Address of a reference is pointer to value.
*xPtr = 100;
```

该代码通过取 x 的引用的地址来将 xPtr 设置为指向 x。将 100 赋值给*xPtr 会将 x 的值更改为 100。由于类型不匹配，xPtr == xRef 的比较是无法编译的，xPtr 是指向 int 的指针，而 xRef 是对 int 的引用。比较 xPtr == &xRef 和 xPtr == &x 都可以正确编译。

最后，请注意，不能声明对引用的引用或对引用的指针。例如，int& &和 int&*都是不允许的。

结构化绑定和引用

在本章的前面已经介绍过结构化绑定。给出了一个如下的例子：

```
pair myPair { "hello", 5 };
auto [theString, theInt] { myPair };  // Decompose using structured bindings
```

既然你已经了解了引用和 const 变量的知识，现在你应该知道它们也可以与结构化绑定一起使用。下面是一个示例：

```
auto& [theString, theInt] { myPair };        // Decompose into references-to-non-const
const auto& [theString, theInt] { myPair }; // Decompose into references-to-const
```

2. 引用数据成员

类的数据成员可以是引用。如前所述，引用不能不指向其他变量而存在，并且不可以更改引用指向的变量。因此，引用数据成员不能在类构造函数的函数体内部进行初始化，必须在构造函数初始化器中进行初始化。在语法方面，构造函数初始化器紧跟在构造函数的声明之后，并以冒号开头。以下是一个展示构造函数初始化器的简单示例。第 9 章有更详细的介绍。

```
class MyClass
{
   public:
      MyClass(int& ref) : m_ref { ref } { /* Body of constructor */ }
   private:
```

```
        int& m_ref;
};
```

警告：

引用必须始终在创建时被初始化。通常，引用是在声明时创建的，但是引用数据成员需要在类的构造函数初始化器中初始化。

3. 引用作为函数参数

C++程序员通常不使用独立的引用变量或引用数据成员，引用的最常见用途是用于函数的参数。默认的参数传递语义是值传递(pass-by-value)：函数接收其参数的副本。修改这些参数后，原始实参将保持不变。栈中变量的指针经常在 C 语言中使用，以允许函数修改其他栈帧中的变量。通过对指针解引用，函数可以修改表示该变量的内存，即使该变量不在当前的栈帧中。这种方法的问题在于，它将指针复杂的语法带入了原本简单的任务。

相对于向函数传递指针，C++提供了一种更好的机制，称为引用传递(pass-by-reference)，参数是引用而不是指针。以下是 addOne()函数的两种实现，第一种对传入的变量没有影响，因为它是值传递的，因此该函数将接收传递给它的值的副本。第二种使用引用，因此更改了原始变量。

```
void addOne(int i)
{
    i++;  // Has no real effect because this is a copy of the original
}

void addOne(int& i)
{
    i++;  // Actually changes the original variable
}
```

调用具有整型引用参数的 addOne()函数的语法与调用具有整型参数的 addOne()函数没有区别。

```
int myInt { 7 };
addOne(myInt);
```

注意：

两个 addOne()函数的实现之间存在微妙区别。使用值传递的版本可以接收字面量而不会出现任何问题，例如 addOne(3)是合法的。然而，如果向引用传递的 addOne()函数传递字面量，会导致编译错误。可使用下一节介绍的 const 引用参数解决该问题。

下面是另一个引用派上用场的例子，是一个简单的交换函数，用于交换两个 int 类型的值。

```
void swap(int& first, int& second)
{
    int temp { first };
    first = second;
    second = temp;
}
```

可以像这样调用它：

```
int x { 5 }, y { 6 };
swap(x, y);
```

当使用实参 x 和 y 调用 swap()时，形参 first 被初始化为对 x 的引用，second 被初始化为对 y 的引用。当 swap()修改 first 和 second 时，实际上更改的是 x 和 y。

当你要将一个指针传递给接收引用的函数时，就会产生一个常见的难题。在这种情况下，可以通过对指针解引用将其“转换”为引用。该操作提供了指针所指向的值，编译器随后使用该值初始化引用参数。例如，可以像这样调用 swap()：

```
int x { 5 }, y { 6 };
int *xp { &x }, *yp { &y };
swap(*xp, *yp);
```

最后，如果函数需要返回一个复制成本高昂的类的对象，函数接收一个对该类的非 const 引用的输出参数，此后进行修改，而非直接返回对象。开发人员认为这是防止从函数返回对象时创建副本从而导致性能损失的推荐方法。但是，即使在那时，编译器通常也足够聪明，可以避免任何冗余的复制。

警告：

从函数返回对象的推荐方法是通过值返回，而不是使用一个输出参数。

const 引用传递

const 引用的参数的主要目的是提高效率。将值传递给函数时，便会生成一个完整副本。传递引用时，实际上只是传递指向原始对象的指针，因此计算机无须生成副本。通过 const 引用传递，可以做到二者兼顾：不生成任何副本，并且无法更改原始变量。当处理对象时，const 引用变得更重要，因为对象可能很大，并且对其进行复制可能会产生有害的副作用。下面的示例演示如何将 std::string 作为 const 引用传递给函数：

```
import std;
using namespace std;
void printString(const string& myString) { println("{}", myString); }

int main()
{
   string someString { "Hello World" };
   printString(someString);
   printString("Hello World");  // Passing literals works.
}
```

值传递和引用传递

当要修改参数并希望那些更改能够作用于传给函数的变量时，需要通过引用传递。但是，不应将引用传递的使用局限于那些情况。引用传递避免将实参复制到函数，从而提供了两个附加好处。

- 效率：复制大型的对象可能花费很长时间，引用传递只是将该对象的一个引用传给了函数
- 支持：不是所有的类都允许值传递

如果你想利用这些好处，又不想修改原始对象，则应将参数标记为 const，从而可以传递 const 引用。

注意：

引用传递的这些好处意味着，应该只对简单的内置类型(如 int 和 double)且无须修改实参的时候使用值传递。如果需要将对象传递给函数，则更应该使用 const 引用传递而不是值传递。这样可以防止不必要的复制。如果函数需要修改对象，则使用非 const 引用来对其进行传递。在引入了移动语义之后，第 9 章对该规则进行了稍微修改，允许在某些情况下对对象使用值传递。

4. 引用作为返回值

函数也可以返回引用。当然，只有在函数终止后返回的引用所指向的变量继续存在的情况下，才可以使用此方法。

警告：

切勿返回作用域为函数内部的局部变量的引用，例如在函数结束时将被销毁的自动分配的栈上变量。

返回引用的主要原因之一是，能够直接把返回值作为左值(赋值语句的左侧)对其赋值。几个重载的运算符通常会返回引用，例如，运算符=、+ =等。第 15 章将详细介绍如何编写自己的此类重载运算符。

从函数返回引用的另一个原因是拷贝返回类型的成本很高。通过返回一个引用或对 const 的引用，可以避免拷贝，但要记住前面的警告。这通常用于通过引用类成员函数中的 const 来返回对象，如本章后面所描述的那样。

5. 在引用与指针之间抉择

C++中的引用可能被认为是多余的：使用引用可以做的所有事情都可以使用指针完成。例如，可以这样编写前面出现的 swap()函数。

```
void swap(int* first, int* second)
{
    int temp { *first };
    *first = *second;
    *second = temp;
}
```

但是，此代码比使用引用的版本更杂乱。引用使程序更简洁，更易于理解。它们也比指针安全，因为没有空引用，并且不需要显式解引用，因此不会遇到与指针相关的任何解引用错误。当然，这些关于引用更安全的争论只有在没有任何指针的情况下才有意义。例如，使用下面的函数，该函数接收对 int 的引用。

```
void refcall(int& t) { ++t; }
```

可以声明一个指针并将其初始化以指向内存中的某个随机位置。然后，可以解引用此指针，并将其作为引用参数传递给 refcall()，如以下代码所示。这段代码可以成功编译，但是并不确定执行后会发生什么。例如，它可能导致程序崩溃。

```
int* ptr { (int*)8 };
refcall(*ptr);
```

大多数时候，可以使用引用而不是指针。与指向对象的指针相同，对对象的引用也支持多态性，将在第 10 章进行详细介绍。但是，在某些情况下，需要使用指针。一种情况是需要更改其指向的位置时。回顾一下，不能更改引用所指向的变量。例如，当动态分配内存时，需要将指向结果的指针存储在指针而不是引用中。需要使用指针的第二种情况是，指针是 optional 的，即当它可以为 nullptr 时。另一个用例是，如果想将多态类型(将在第 10 章讨论)存储在容器中。

很久以前，在遗留代码中，选择参数和返回类型中使用指针还是引用的一种方法是考虑内存的所有权。如果接收变量的代码成为所有者，并因此负责释放与对象关联的内存，则它必须接收指向该对

象的指针。如果接收该变量的代码不必释放内存，那么它应接收一个引用。但是，现在应避免使用原始指针，而使用智能指针(请参阅第 7 章)，这是转让所有权的推荐方法。

注意：
尽量选择引用而不是指针，也就是说，只有在无法使用引用的情况下才选择使用指针。

考虑将一个整数数组分为两个数组的函数：分别存放奇数和偶数。该函数不知道源数组中的偶数或奇数个数，因此它应在检查源数组后为目标数组动态分配内存。它还应该返回两个新数组的大小。共有 4 项要返回：指向两个新数组的指针以及两个新数组的大小。显然，必须使用引用传递。规范的 C 的写法如下所示：

```
void separateOddsAndEvens(const int arr[], size_t size, int** odds,
    size_t* numOdds, int** evens, size_t* numEvens)
{
    // Count the number of odds and evens.
    *numOdds = *numEvens = 0;
    for (size_t i = 0; i < size; ++i) {
        if (arr[i] % 2 == 1) {
            ++(*numOdds);
        } else {
            ++(*numEvens);
        }
    }

    // Allocate two new arrays of the appropriate size.
    *odds = new int[*numOdds];
    *evens = new int[*numEvens];

    // Copy the odds and evens to the new arrays.
    size_t oddsPos = 0, evensPos = 0;
    for (size_t i = 0; i < size; ++i) {
        if (arr[i] % 2 == 1) {
            (*odds)[oddsPos++] = arr[i];
        } else {
            (*evens)[evensPos++] = arr[i];
        }
    }
}
```

该函数的最后 4 个参数是“引用”参数。若要更改它们引用的值，separateOddsAndEvens()必须对它们解引用，这会导致函数体内的语法丑陋。此外，当调用 separateOddsAndEvens()时，必须传递两个指针的地址，以便函数可以更改实际的指针，并传递两个 size_t 的地址，以便函数可以更改实际的 size_t。还要注意，调用方要负责删除由 separateOddsAndEvens()创建的两个数组。

```
int unSplit[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int* oddNums { nullptr };
int* evenNums { nullptr };
size_t numOdds { 0 }, numEvens { 0 };

separateOddsAndEvens(unSplit, std::size(unSplit),
    &oddNums, &numOdds, &evenNums, &numEvens);

// Use the arrays...
```

```
delete[] oddNums; oddNums = nullptr;
delete[] evenNums; evenNums = nullptr;
```

如果此语法令你烦恼(它也确实如此)，则可以使用引用编写相同的函数，以获得真正的引用传递语义。

```
void separateOddsAndEvens(const int arr[], size_t size, int*& odds,
    size_t& numOdds, int*& evens, size_t& numEvens)
{
    numOdds = numEvens = 0;
    for (size_t i { 0 }; i < size; ++i) {
        if (arr[i] % 2 == 1) {
            ++numOdds;
        } else {
            ++numEvens;
        }
    }

    odds = new int[numOdds];
    evens = new int[numEvens];

    size_t oddsPos { 0 }, evensPos { 0 };
    for (size_t i { 0 }; i < size; ++i) {
        if (arr[i] % 2 == 1) {
            odds[oddsPos++] = arr[i];
        } else {
            evens[evensPos++] = arr[i];
        }
    }
}
```

这种情况下，参数 odds 和 evens 是对 int*的引用。separateOddsAndEvens()无须解引用就可以修改函数的实参 int*(引用传递)。相同的逻辑适用于 numOdds 和 numEvens，它们是对 size_t 的引用。使用此版本的函数，不再需要传递指针或 size_t 的地址。引用参数会自动为你处理：

```
separateOddsAndEvens(unSplit, std::size(unSplit),
    oddNums, numOdds, evenNums, numEvens);
```

即使使用引用参数已经比使用指针清晰得多，但建议避免使用动态分配的数组。例如，通过使用标准库容器 vector，可将 separateOddsAndEvens()函数重写，从而更安全、更短、更美观且更具可读性，因为所有内存分配和释放都是自动发生的。

```
void separateOddsAndEvens(const vector<int>& arr,
    vector<int>& odds, vector<int>& evens)
{
    for (int i : arr) {
        if (i % 2 == 1) {
            odds.push_back(i);
        } else {
            evens.push_back(i);
        }
    }
}
```

这个版本可以被这样使用：

```
vector<int> vecUnSplit { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
vector<int> odds, evens;
separateOddsAndEvens(vecUnSplit, odds, evens);
```

请注意，你无须释放 odds 和 evens 容器，vector 类负责此工作。该版本比使用指针或引用的版本更容易使用。

使用 vector 的版本已经比使用指针或引用的版本好得多，但是正如之前所建议的那样，应尽可能避免使用输出参数。如果一个函数需要返回一些东西，它应该直接返回而不是使用输出参数！从 C++17 开始，如果 object 是一个无名的临时对象，编译器不允许对 return object 形式的语句执行任何对象的拷贝或移动。这被称为拷贝/移动操作的强制省略，意味着按值返回对象根本不会造成性能损失。若对象是局部变量而非函数参数，则允许非强制省略拷贝/移动操作，这种优化也称为命名返回值优化(NRVO)。标准不保证这种优化。一些编译器仅对发布版本执行此优化，而对调试版本则不执行。通过强制和非强制省略，编译器可以避免对函数返回的对象进行任何拷贝。这样会导致零拷贝值传递语义。请注意，对于 NRVO，即使不会调用拷贝/移动构造函数，它们仍然需要可访问，否则，根据标准，程序的格式是错误的。第 9 章讨论了拷贝/移动操作和构造函数，但这些细节对于当前的讨论并不重要。

以下版本的 separateOddsAndEvens()返回一个简单的包含两个 vector 的结构体，而不是接收两个输出 vector 作为参数。它也使用了指派初始化器。

```
struct OddsAndEvens { vector<int> odds, evens; };

OddsAndEvens separateOddsAndEvens(const vector<int>& arr)
{
    vector<int> odds, evens;
    for (int i : arr) {
        if (i % 2 == 1) {
            odds.push_back(i);
        } else {
            evens.push_back(i);
        }
    }
    return OddsAndEvens { .odds = odds, .evens = evens };
}
```

进行了这些更改之后，用于调用 separateOddsAndEvens()的代码变得紧凑，且易于阅读和理解。

```
vector<int> vecUnSplit { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto oddsAndEvens { separateOddsAndEvens(vecUnSplit) };
// Do something with oddsAndEvens.odds and oddsAndEvens.evens...
```

注意：
不要使用输出参数，如果一个函数需要返回某些东西，直接按值返回即可。

1.1.29 const_cast()

在 C++中，每个变量都有特定的类型。在某些情况下，可能将一种类型的变量转换为另一种类型的变量。为此，C++提供了 5 种类型的转换：const_cast()、static_cast()、reinterpret_cast()、dynamic_cast()和 std::bit_cast()。本节讨论 const_cast()。static_cast()在本章的前面进行了简要介绍，并将在第 10 章进

行更详细的讨论。其余的其他类型的转换也将在第 10 章进行讨论。

const_cast()是 5 种不同类型转换中最简单的，可以使用它为变量添加或取消 const 属性，这是 5 种类型转换中唯一可以消除 const 属性的转换。当然，从理论上讲，不需要 const 转换。如果变量是 const，则应保持 const。但在实际中，有时会遇到这样的情况：指定一个函数接收 const 参数，然后这个参数将在接收非 const 参数的函数中使用，并且可以确保后者不会修改其非 const 参数。“正确”的解决方案是使 const 在程序中一直保持，但这并不总是可行的，尤其是在使用第三方库的情况下。因此，有时需要舍弃变量的 const 属性，但是只有在确定所调用的函数不会修改该对象时，才应这样做。否则，除了重构程序外，别无选择。这是一个例子：

```
void thirdPartyLibraryFunction(char* str);

void f(const char* str)
{
    thirdPartyLibraryFunction(const_cast<char*>(str));
}
```

此外，标准库提供了一个名为 std::as_const()的辅助方法，该方法定义在<utility>中，该方法接收一个引用参数，返回它的 const 引用版本。基本上，as_const(obj)等价于 const_cast<const T&>(obj)，其中 T 是 obj 的类型。与使用 const_cast()相比，使用 as_const()可以使代码更短，更易读。本书稍后将介绍 as_const()的具体用例，其基本用法如下：

```
string str { "C++" };
const string& constStr { as_const(str) };
```

1.1.30　异常

C++是一种非常灵活的语言，但它也允许一些不安全的行为。例如，编译器允许编写改变随机内存地址或者尝试除以 0 的代码(计算机无法处理无穷大的数值)。异常(exceptions)就是试图增加一个安全等级的语言特性。

异常是一种特殊情况，也就是说，在程序的正常执行流程中，你不期望或不想要的情况。例如，如果编写一个获取 Web 页面的函数，就有几件事情可能出错，包含页面的 Internet 主机可能被关闭，页面可能是空白的，或者连接可能会丢失。处理这种情况的一种方法是，从函数返回特定的值，如 nullptr 或其他错误代码。异常提供了处理此类问题的更好方法。

异常伴随着一些新术语。当某段代码检测到异常时，就会抛出(throw)一个异常，另一段代码会捕获(catch)这个异常并执行恰当的操作。下例给出一个名为 divideNumbers()的函数，如果调用者传递给分母的值为 0，就会抛出一个异常。使用 std::invalid_argument 时需要<stdexcept>。

```
double divideNumbers(double numerator, double denominator)
{
    if (denominator == 0) {
        throw invalid_argument { "Denominator cannot be 0." };
    }
    return numerator / denominator;
}
```

当执行 throw 行时，函数将立刻结束而不会返回值。如果调用者将函数调用放到 try/catch 块中，就可以捕获异常并进行处理，如下面的代码所示。第 14 章将详细介绍异常处理，但是现在，请记住，建议通过 const 引用捕获异常，例如以下示例中的 const invalid_argument&。还要注意，所有标准库异

常类都有一个名为 what()的方法，该方法返回一个字符串，其中包含对该异常的简要说明。

```
try {
   println("{}", divideNumbers(2.5, 0.5));
   println("{}", divideNumbers(2.3, 0));
   println("{}", divideNumbers(4.5, 2.5));
} catch (const invalid_argument& exception) {
   println("Exception caught: {}", exception.what());
}
```

第一次调用 divideNumbers()成功执行，结果会输出给用户。第二次调用会抛出一个异常，不会返回值，唯一的输出是捕获异常时输出的错误信息。第三次调用根本不会执行，因为第二次调用抛出了一个异常，导致程序跳转到 catch 块。前面代码块的输出是：

```
5
Exception caught: Denominator cannot be 0.
```

C++的异常非常灵活，为正确使用异常，需要理解抛出异常时栈变量的行为，必须正确捕获并处理必要的异常。另外，如果需要在异常中包含有关错误的更多信息，则可以编写自己的异常类型。最后，C ++编译器不会强迫你捕获所有可能发生的异常。如果你的代码从不捕获任何异常，但是引发了异常，则该程序将终止。这些异常的棘手方面将在第 14 章详细介绍。

1.1.31 类型别名

类型别名(type alias)为现有的类型声明提供新名称。可以将类型别名视为一种语法，用于为现有类型声明引入同义词而无须创建新类型。以下为 int *类型声明赋予新名称 IntPtr：

```
using IntPtr = int*;
```

可以互换使用新的类型别名及其原本的名称。例如，以下两行有效。

```
int* p1;
IntPtr p2;
```

使用新类型名称创建的变量与使用原始类型声明创建的变量完全兼容。因此，使用这些定义，编写以下内容是完全正确的，因为它们不仅是兼容的类型，它们是完全相同的类型。

```
p1 = p2;
p2 = p1;
```

类型别名最常见的用途是在原类型声明过于笨拙时提供便于管理的名称。模板通常会出现这种情况。标准库本身的一个示例是 std::basic_string<T>，用于表示字符串。这是一个类模板，其中 T 是字符串中每个字符的类型，例如 char。每当要引用模板类型参数时，都必须指定它。为了声明变量，指定函数参数等，必须写 basic_string<char>。

```
void processVector(const vector<basic_string<char>>& vec) { /* omitted */ }

int main()
{
   vector<basic_string<char>> myVector;
   processVector(myVector);
}
```

因为 basic_string<char>使用的频率如此之高，标准库便为其提供了一个更短也更有意义的类型

别名。

```
using string = basic_string<char>;
```

有了这个类型别名，之前的代码段可以被写得更加优雅。

```
void processVector(const vector<string>& vec) { /* omitted */ }

int main()
{
   vector<string> myVector;
   processVector(myVector);
}
```

1.1.32　类型定义

类型别名是在 C++11 中引入的。在 C++11 之前，必须使用 typedef 完成类似的操作，但要复杂得多。这里仍将解释这种旧机制，因为你将在遗留代码中看到它。

像类型别名一样，typedef 为现有的类型声明提供新名称。例如，使用以下类型别名：

```
using IntPtr = int*;
```

可用 typedef 将其改写为如下代码：

```
typedef int* IntPtr;
```

正如你所见，它的可读性降低了。顺序是颠倒的，即使对于专业的 C++开发人员，也会引起很多混乱。除了更加复杂之外，typedef 的行为与类型别名相同。例如，可以按如下方式使用 typedef：

```
IntPtr p;
```

但是，类型别名和 typedef 并不完全等效。与 typedef 相比，类型别名与模板一起使用时功能更强大，但这是第 12 章介绍的主题，因为它需要有关模板的更多信息。

警告：
总是优先选择类型别名而不是 typedef。

1.1.33　类型推断

类型推断允许编译器自动推断出表达式的类型。类型推断有两个关键字：auto 和 decltype。

1. 关键字 auto

关键字 auto 有多种不同的用法：

- 推断函数的返回类型，如前所述。
- 定义结构化绑定，如前所述。
- 推断表达式的类型，如前所述。
- 推断非类型模板参数的类型，见第 12 章。
- 简写函数模板的语法，见第 12 章。
- 在 decltype(auto)中使用，见第 12 章。
- 使用其他语法编写函数，见第 12 章。
- 编写泛型 lambda 表达式，见第 19 章。

auto 可用于告诉编译器，在编译期自动推断变量的类型。下面的代码演示了在这种情况下关键字 auto 最简单的用法：

```
auto x { 123 };     // x is of type int.
```

在这个示例中，输入 auto 和输入 int 的效果没什么区别，但 auto 对较复杂的类型会更有用。假定 getFoo()函数有一个复杂的返回类型。如果希望把调用该函数的结果赋予一个变量，可以输入该复杂类型，也可以简单地使用 auto，让编译器推断出该类型。

```
auto result { getFoo() };
```

这样，可方便地更改函数的返回类型，而不需要更新代码中调用该函数的所有位置。

auto&语法

使用 auto 推断类型时去除了引用和 const 限定符。假设有以下函数：

```
const string message { "Test" };
const string& foo() { return message; }
```

可以调用 foo()，把结果存储在一个变量中，将该变量的类型指定为 auto，如下所示。

```
auto f1 { foo() };
```

因为 auto 去除了引用和 const 限定符，且 f1 是 string 类型，因此会建立一个副本。如果希望 f1 是一个 const 引用，就可以明确将它建立为一个引用，并标记为 const，如下所示。

```
const auto& f2 { foo() };
```

本章前面介绍了工具函数 as_const()，它返回其引用参数的 const 引用版本。将 as_const()与 auto 结合使用时要小心。由于 auto 去除引用和 const 限定符，因此以下 result 变量的类型为 string，而不是 const string&，因此将进行复制：

```
string str { "C++" };
auto result { as_const(str) };
```

警告：

始终要记住，auto 去除了引用和 const 限定符，从而会创建副本！如果不需要副本，可使用 auto&或 const auto&。

auto*语法

auto 关键字也可以用于指针，下面是一个例子：

```
int i { 123 };
auto p { &i };
```

p 的类型是 int*。与上一节中讨论的引用不同，此处不存在意外复制的危险。但是，在使用指针时，建议使用 auto*语法，因为它可以更清楚地指出涉及指针。例如：

```
auto* p { &i };
```

此外，使用 auto*代替 auto 确实可以解决将 auto、const 和指针一起使用时的奇怪行为。假设你编写以下内容：

```
const auto p1 { &i };
```

大多数情况下，不会发生你期待的事情！

通常，当使用 const 时，你想保护指针所指向的东西。你可能会认为 p1 的类型为 const int*，但实际上，该类型为 int* const，因此它是指向非 const 整数的 const 指针。按如下所示将 const 放在 auto 后面无济于事；类型仍然是 int* const。

```
auto const p2 { &i };
```

将 auto*与 const 结合使用时，它的行为就会与期望的一样。这是一个例子：

```
const auto* p3 { &i };
```

现在 p3 的类型为 const int*。如果你真的需要一个 const 的指针而不是 const 的整数，需要将 const 放在后边。

```
auto* const p4 { &i };
```

p4 的类型为 int * const。

最后，使用这个语法可以令指针和整数都是 const。

```
const auto* const p5 { &i };
```

p5 的类型是 const int* const。如果省略了*，将不能得到这个结果。

拷贝列表初始化和直接列表初始化

有两种使用带大括号初始化列表的初始化方式。

- 拷贝列表初始化：T obj = {arg1, arg2, ... };
- 直接列表初始化：T obj {arg1, arg2, ... };

与自动类型推断相结合，拷贝列表初始化和直接列表初始化之间就存在重要区别。例如：

```
// Copy list initialization
auto a = { 11 };          // initializer_list<int>
auto b = { 11, 22 };      // initializer_list<int>

// Direct list initialization
auto c { 11 };            // int
auto d { 11, 22 };        // Error, too many elements.
```

对于拷贝列表初始化，带大括号的初始化程序中的所有元素都必须具有相同的类型。例如，以下内容会编译失败。

```
auto b = { 11, 22.33 };  // Compilation error
```

2. 关键字 decltype

关键字 decltype 把表达式作为实参，计算出该表达式的类型。例如：

```
int x { 123 };
decltype(x) y { 456 };
```

在这个示例中，编译器推断出 y 的类型是 int，因为这是 x 的类型。

auto 与 decltype 的区别在于，decltype 未去除引用和 const 限定符。再来分析返回 const string 引用的 foo()函数。按如下方式使用 decltype 定义 f2，导致 f2 的类型为 const string&，从而不生成副本。

```
decltype(foo()) f2 { foo() };
```

刚开始不会觉得 decltype 有多大价值。但在模板的上下文中，decltype 会变得十分强大，详见第 12 和第 26 章。

1.1.34 标准库

C++具有标准库，其中包含许多有用的类，在代码中可方便地使用这些类。使用标准库中类的好处是不需要重新创建它们，也不需要浪费时间去实现系统已经自动实现的内容。另一好处是标准库中的类已经过成千上万用户的严格测试和验证。标准库中的类也经过了性能优化，因此使用这些类在大多数情况下比使用自己的类效率更高。

标准库中可用的功能非常多。第 16~24 章将详细讲述标准库。当开始使用 C++时，最好立刻了解标准库可以做什么。如果你是一位 C 程序员，这一点尤其重要。作为 C 程序员，使用 C++时可能会以 C 的方式解决问题，然而使用 C++的标准库类可以更方便、安全地解决问题。

这就是本章前面介绍标准库中的一些类的原因，例如 std::string、array、vector、pair 和 optional。本书一直在示例中使用这些代码，以确保你能够习惯使用标准库类。第 16~24 章将介绍更多的类。

1.2 第一个大型的 C++程序

下面的程序建立一个雇员数据库，在前面讨论结构体时曾将其用作示例。在此，将使用本章前面讲述的许多特性来完成一个功能完整的 C++程序。这个实际的示例使用了类、异常、流、vector、命名空间、引用及其他语言特性。

1.2.1 雇员记录系统

管理公司雇员记录的程序应该灵活并具有有效的功能。这个程序包含的功能有：

- 添加和解雇雇员
- 雇员晋升和降级
- 查看所有雇员，包括过去及现在的雇员
- 查看所有当前雇员
- 查看所有以前雇员

程序的代码分为三部分：Employee 类封装了单个雇员的信息，Database 类管理公司的所有雇员，单独的 UserInterface 提供程序的交互界面。

1.2.2 Employee 类

Employee 类维护某个雇员的全部信息，该类的方法提供了查询以及修改信息的途径。Employee 还知道如何在控制台显示自身。此外存在调整雇员薪水和雇佣状态的方法。

1. Employee.cppm

Employee.cppm 模块接口文件定义了 Employee 类，此文件的各部分分别在随后进行描述。文件的前几行如下：

```
export module employee;
import std;
namespace Records {
```

第一行是模块声明，并声明该文件导出一个名为 employee 的模块，然后导入标准库功能。此代码还声明花括号中包含的后续代码位于 Records 命名空间中，为使用特定代码，整个程序都会用到 Records 命名空间。

接下来，在 Records 命名空间内定义以下两个常量。本书使用惯例，不在常量前加任何特殊字母，而是以大写字母开头，以便更好地与变量进行区分。

```
const int DefaultStartingSalary { 30'000 };
export const int DefaultRaiseAndDemeritAmount { 1'000 };
```

第一个常量代表新雇员的默认起薪，这个常量是没有被导出的，因为它不需要被本模块之外的代码访问。employee 模块内的代码可以通过 Records::DefaultStartingSalary 访问。

第二个常量是用于晋升或降级雇员的职位。此常量是被导出的，因此此模块外部的代码可以操作它。例如，将雇员晋升的金额设置为默认金额的两倍。

接下来，声明了 Employee 类及其 public 成员函数：

```
export class Employee
{
   public:
      Employee(const std::string& firstName,
             const std::string& lastName);

      void promote(int raiseAmount = DefaultRaiseAndDemeritAmount);
      void demote(int demeritAmount = DefaultRaiseAndDemeritAmount);
      void hire(); // Hires or rehires the employee
      void fire(); // Dismisses the employee
      void display() const; // Prints employee info to console

      // Getters and setters
      void setFirstName(const std::string& firstName);
      const std::string& getFirstName() const;

      void setLastName(const std::string& lastName);
      const std::string& getLastName() const;

      void setEmployeeNumber(int employeeNumber);
      int getEmployeeNumber() const;

      void setSalary(int newSalary);
      int getSalary() const;

      bool isHired() const;
```

提供了一个接收名字和姓氏的构造函数。promote()和 demote()成员函数都具有整数参数，其默认值等于 DefaultRaiseAndDemeritAmount。这样，其他代码可以省略该参数，并且将自动使用默认值。还提供了雇用和解雇雇员的方法，以及显示有关雇员信息的方法。许多获取器和设置器提供了更改信息或查询雇员当前信息的功能。

将数据成员声明为 private，这样其他部分的代码将无法直接修改它们。

```
   private:
      std::string m_firstName;
      std::string m_lastName;
      int m_employeeNumber { -1 };
      int m_salary { DefaultStartingSalary };
```

```
        bool m_hired { false };
    };
}
```

获取器和设置器提供了修改或查询这些值的唯一 public 途径。数据成员在类定义中(而非构造函数中)进行初始化。默认情况下，新雇员无姓名，雇员编号为-1，起薪为默认值，状态为未受雇。

2. Employee.cpp

模块实现文件的前几行如下：

```
module employee;
import std;
using namespace std;
```

第一行指定了此源文件的模块，接下来是 std 的导入，以及一条 using 指令。

构造函数接收姓和名，只设置相应的数据成员。

```
namespace Records {
    Employee::Employee(const string& firstName, const string& lastName)
        : m_firstName { firstName }, m_lastName { lastName }
    {
    }
```

promote()和 demote()成员函数只是用一些新值调用 setSalary()方法。整型参数的默认值不显示在源文件中；它们只能出现在函数声明中，不能出现在函数定义中。

```
    void Employee::promote(int raiseAmount)
    {
        setSalary(getSalary() + raiseAmount);
    }

    void Employee::demote(int demeritAmount)
    {
        setSalary(getSalary() - demeritAmount);
    }
```

hire()和 fire()成员函数正确设置了 m_hired 数据成员。

```
    void Employee::hire() { m_hired = true; }
    void Employee::fire() { m_hired = false; }
```

display()成员函数使用 println()显示当前雇员的信息。由于这段代码是 Employee 类的一部分，因此可直接访问数据成员(如 m_salary)，而不需要使用 getSalary()获取器。然而，使用获取器和设置器(当存在时)是一种好的风格，甚至在类的内部也是如此。

```
    void Employee::display() const
    {
        println("Employee: {}, {}", getLastName(), getFirstName());
        println("-------------------------");
        println("{}", (isHired() ? "Current Employee" : "Former Employee"));
        println("Employee Number: {}", getEmployeeNumber());
        println("Salary: ${}", getSalary());
        println("");
    }
```

最后，许多获取器和设置器执行获取值以及设置值的任务。

```
    // Getters and setters
    void Employee::setFirstName(const string& firstName) {m_firstName = firstName;}
    const string& Employee::getFirstName() const { return m_firstName; }

    void Employee::setLastName(const string& lastName) { m_lastName = lastName; }
    const string& Employee::getLastName() const { return m_lastName; }

    void Employee::setEmployeeNumber(int employeeNumber) {
       m_employeeNumber = employeeNumber; }
    int Employee::getEmployeeNumber() const { return m_employeeNumber; }

    void Employee::setSalary(int salary) { m_salary = salary; }
    int Employee::getSalary() const { return m_salary; }

    bool Employee::isHired() const { return m_hired; }
}
```

即使这些成员函数看起来微不足道，但是使用这些微不足道的获取器和设置器，仍然优于将数据成员设置为 public。例如，将来你可能想在 setSalary()成员函数中执行边界检查，获取器和设置器也能简化调试，因为可在其中设置断点，在检索或设置值时检查它们。另一个原因是决定修改类中存储数据的方式时，只需要修改这些获取器和设置器，而其他使用该类的代码可以保持不变。

3. EmployeeTest.cpp

当编写一个类时，最好对其进行独立测试。以下代码在 main()函数中针对 Employee 类执行了一些简单操作。当确信 Employee 类可正常运行后，应该删除这个文件，或将这个文件注释掉，这样就不会编译具有多个 main()函数的代码。

```
import std;
import employee;

using namespace std;
using namespace Records;

int main()
{
    println("Testing the Employee class.");
    Employee emp { "Jane", "Doe" };
    emp.setFirstName("John");
    emp.setLastName("Doe");
    emp.setEmployeeNumber(71);
    emp.setSalary(50'000);
    emp.promote();
    emp.promote(50);
    emp.hire();
    emp.display();
}
```

可以更好地测试各个类的另一种方法是使用单元测试，详见第 30 章。单元测试是测试特定功能的小段代码，保留在代码库中。所有单元测试都经常执行，例如，它们可以由构建系统自动执行。这样做的好处是，当你对现有功能进行更改时，如果你破坏了某些东西，单元测试会立即警告你。

1.2.3 Database 类

下面实现 Database 类。它使用标准库中的 std::vector 类来存储 Employee 对象。

1. database.cppm

下面是模块接口文件 database.cppm 中的前几行：

```
export module database;
import std;
import employee;

namespace Records {
   const int FirstEmployeeNumber { 1'000 };
```

由于数据库会自动给新雇员指定一个雇员号，因此定义一个常量作为编号的开始。

接下来，Database 类被定义和导出。

```
   export class Database
   {
      public:
         Employee& addEmployee(const std::string& firstName,
                               const std::string& lastName);
         Employee& getEmployee(int employeeNumber);
         Employee& getEmployee(const std::string& firstName,
                               const std::string& lastName);
```

数据库可根据提供的姓名方便地添加一个新雇员。为方便起见，这个方法返回一个新雇员的引用。外部代码也可通过调用 getEmployee()成员函数来获得雇员的引用。这个成员函数声明了两个重载版本，一个允许按雇员号进行检索，另一个要求提供雇员的姓名。

由于数据库是所有雇员记录的中心存储库，因此它具有显示所有雇员、当前在职雇员及已离职雇员的方法。

```
         void displayAll() const;
         void displayCurrent() const;
         void displayFormer() const;
```

最后，private 数据成员被定义如下。

```
      private:
         std::vector<Employee> m_employees;
         int m_nextEmployeeNumber { FirstEmployeeNumber };
   };
}
```

m_employees 数据成员包含 Employee 对象，数据成员 m_nextEmployeeNumber 跟踪新雇员的雇员号，使用 FirstEmployeeNumber 常量进行初始化。

2. Database.cpp

addEmployee()成员函数的实现如下：

```
module database;
import std;

using namespace std;
```

```
namespace Records {
    Employee& Database::addEmployee(const string& firstName,
                                    const string& lastName)
    {
        Employee theEmployee { firstName, lastName };
        theEmployee.setEmployeeNumber(m_nextEmployeeNumber++);
        theEmployee.hire();
        m_employees.push_back(theEmployee);
        return m_employees.back();
    }
```

addEmployee()成员函数创建一个新的 Employee 对象，在其中填充信息并将其添加到 vector 中。注意当使用这个方法后，数据成员 m_nextEmployeeNumber 的值会递增，因此下一个雇员将获得新编号。vector 的 back()成员函数返回 vector 中最后一个元素的引用，即最新添加的雇员。

getEmployee()成员函数之一的实现如下。第二个版本以类似的方式实现，因此未示出。都使用基于范围的 for 循环遍历 m_employees 中的所有雇员，并检查 Employee 是否与传递给该成员函数的信息匹配。如果找不到匹配项，则会引发异常。注意在基于范围的 for 循环中使用 auto&，因为该循环不想处理 Employee 的副本，而是要处理 m_Employees vector 中 Employee 的引用。

```
    Employee& Database::getEmployee(int employeeNumber)
    {
        for (auto& employee : m_employees) {
            if (employee.getEmployeeNumber() == employeeNumber) {
                return employee;
            }
        }
        throw logic_error { "No employee found." };
    }
```

所有显示方法都采用相似的算法。这些方法遍历所有雇员，如果符合显示标准，就通知雇员将自身显示到控制台中。

```
    void Database::displayAll() const
    {
        for (const auto& employee : m_employees) { employee.display(); }
    }

    void Database::displayCurrent() const
    {
        for (const auto& employee : m_employees) {
            if (employee.isHired()) { employee.display(); }
        }
    }

    void Database::displayFormer() const
    {
        for (const auto& employee : m_employees) {
            if (!employee.isHired()) { employee.display(); }
        }
    }
}
```

3. DatabaseTest.cpp

用于数据库基本功能的简单测试如下所示：

```
import std;
import database;

using namespace std;
using namespace Records;

int main()
{
    Database myDB;
    Employee& emp1 { myDB.addEmployee("Greg", "Wallis") };
    emp1.fire();

    Employee& emp2 { myDB.addEmployee("Marc", "White") };
    emp2.setSalary(100'000);

    Employee& emp3 { myDB.addEmployee("John", "Doe") };
    emp3.setSalary(10'000);
    emp3.promote();

    println("All employees:\n==============");
    myDB.displayAll();

    println("\nCurrent employees:\n==================");
    myDB.displayCurrent();

    println("\nFormer employees:\n=================");
    myDB.displayFormer();
}
```

1.2.4 用户界面

程序的最后一部分是基于菜单的用户界面，可让用户方便地使用雇员数据库。

下面的 main()函数包含一个循环，用于显示菜单、执行被选中的操作，然后重复整个过程。对于大多数操作，都定义了独立的函数。对于显示雇员之类的简单操作，则将实际代码放在对应的情况(case)中。

```
import std;
import database;
import employee;

using namespace std;
using namespace Records;

int displayMenu();
void doHire(Database& db);
void doFire(Database& db);
void doPromote(Database& db);

int main()
{
    Database employeeDB;
```

```
    bool done { false };
    while (!done) {
        int selection { displayMenu() };
        switch (selection) {
        case 0:
            done = true;
            break;
        case 1:
            doHire(employeeDB);
            break;
        case 2:
            doFire(employeeDB);
            break;
        case 3:
            doPromote(employeeDB);
            break;
        case 4:
            employeeDB.displayAll();
            break;
        case 5:
            employeeDB.displayCurrent();
            break;
        case 6:
            employeeDB.displayFormer();
            break;
        default:
            println(cerr, "Unknown command.");
            break;
        }
    }
}
```

displayMenu()函数输出菜单并获取用户输入。在此假定用户能够“正确输入”，当需要一个数字时就会输入一个数字，这一点很重要。在阅读了第 13 章有关 I/O 的内容后，你就会知道如何防止输入错误信息。

```
int displayMenu()
{
    int selection;
    println("");
    println("Employee Database");
    println("-----------------");
    println("1) Hire a new employee");
    println("2) Fire an employee");
    println("3) Promote an employee");
    println("4) List all employees");
    println("5) List all current employees");
    println("6) List all former employees");
    println("0) Quit");
    println("");
    print("---> ");
    cin >> selection;
    return selection;
}
```

doHire()函数获取用户输入的新雇员姓名，并通知数据库添加这个雇员。

```
void doHire(Database& db)
{
   string firstName;
   string lastName;

   print("First name? ");
   cin >> firstName;

   print("Last name? ");
   cin >> lastName;

   auto& employee { db.addEmployee(firstName, lastName) };
   println("Hired employee {} {} with employee number {}.",
      firstName, lastName, employee.getEmployeeNumber());
}
```

doFire()和 doPromote()都会要求数据库根据雇员号找到雇员的记录，然后使用 Employee 对象的 public 成员函数进行修改。

```
void doFire(Database& db)
{
   int employeeNumber;
   print("Employee number? ");
   cin >> employeeNumber;

   try {
      auto& emp { db.getEmployee(employeeNumber) };
      emp.fire();
      println("Employee {} terminated.", employeeNumber);
   } catch (const std::logic_error& exception) {
      println(cerr, "Unable to terminate employee: {}", exception.what());
   }
}

void doPromote(Database& db)
{
   int employeeNumber;
   print("Employee number? ");
   cin >> employeeNumber;

   int raiseAmount;
   print("How much of a raise? ");
   cin >> raiseAmount;

   try {
      auto& emp { db.getEmployee(employeeNumber) };
      emp.promote(raiseAmount);
   } catch (const std::logic_error& exception) {
      println(cerr, "Unable to promote employee: {}", exception.what());
   }
}
```

1.2.5 评估程序

前面的程序涵盖了许多主题，从最简单的到较复杂的都有。可采用多种方法扩展这个程序。例如，

用户界面(UI)没有显示 Database 或 Employee 类的全部功能。可修改 UI，以包含这些特性。也可以尝试为这两个类实现一些额外的功能，这是对本章所学内容的绝佳练习。

如果不理解程序的某些部分，可以参考前面的内容以回顾这些主题。如果仍不甚明了，最好的学习方法是编写代码并查看结果。例如，如果不确定如何使用条件运算符，可编写一个简单的 main()函数进行测试。

1.3　本章小结

读完本章关于 C++和标准库的速成内容之后，你已经为成为专业 C++程序员做好了准备。在开始深入学习本书后面的 C++语言知识时，可查阅本章以回顾需要复习的内容。为了回顾那些被遗忘的概念，可能只需要查看本章的一些示例代码。

你编写的每个程序都必须以这样或那样的方式使用字符串。为此，第 2 章将深入讲解如何在 C++中处理字符串。

1.4　练习

通过做以下习题，可以巩固本章涉及的知识点。所有练习的答案都可扫描封底二维码下载。但是，如果你被某些问题难住，请先重新阅读本章的对应内容，以尝试自己找到答案，再从网站上查看解决方案。

练习 1-1　修改本章开头的 Employee 结构体，将其放在一个名为 HR 的命名空间中。你必须对 main()中的代码进行哪些修改才能使用此新实现？此外，修改代码以使用指派初始化器。

练习 1-2　以练习 1-1 的结果为基础，并向 Employee 添加一个枚举数据成员 title，以指定某个雇员是经理、高级工程师还是工程师。你将使用哪种枚举类型，为什么？无论需要添加什么，都将其添加到 HR 命名空间中。在 main()函数中测试新的 Employee 数据成员。使用 switch 语句为 title 打印出易于理解的字符串。

练习 1-3　使用 std::array 存储练习 1-2 中具有不同数据的 3 个 Employee 实例。然后使用基于范围的 for 循环打印出 array 中的雇员。

练习 1-4　进行与练习 1-3 相同的操作，但使用 std::vector 而不是 array，并使用 push_back()将元素插入 vector 中。

练习 1-5　采用练习 1-4 的解决方案，将表示名字和姓氏首字母的数据成员替换为字符串，以表示完整的名字和姓氏。

练习 1-6　既然你已经了解了 const 和引用及其用途，请修改本章前面的 AirlineTicket 类，以尽可能使用引用，并正确使用 const。

练习 1-7　修改练习 1-6 中的 AirlineTicket 类，使其包含一个可选的飞行常客号码。表示此可选数据成员的最佳方法是什么？添加一个设置器和获取器来设置和获取飞行常客号码。修改 main()函数来测试你的实现。

第2章 使用字符串和字符串视图

本章内容

- C 风格字符串和 C++字符串的区别
- C++ std::string 类的细节
- 使用 std::string_view 的原因
- 原始字符串字面量
- 如何格式化文本
- 如何将元素范围格式化为字符串

你编写的每个应用程序都会使用某种类型的字符串。使用老式 C 语言时，没有太多选择，只能使用普通的以 null 结尾的字符数组表示字符串。遗憾的是，这种表示方式会导致很多问题，例如会导致安全漏洞的缓冲区溢出。C++标准库包含了一个安全易用的 std::string 类，这个类没有这些缺点。

字符串十分重要，所以作为本书的前面部分，本章将详细讨论字符串。

2.1 动态字符串

在将字符串当成一等对象支持的语言中，字符串有很多吸引人的特性，例如可扩展至任意大小，或能提取或替换子字符串。在其他语言(如 C 语言)中，字符串几乎就如后加入的功能，C 语言中并没有真正好用的 string 数据类型，只有固定的字节数组。C 语言的“字符串库”只不过是一组非常原始的函数，甚至没有边界检查的功能。C++提供了 string 类型作为一等数据类型。

2.1.1 C 风格字符串

在 C 语言中，字符串表示为字符的数组。字符串中的最后一个字符是 null 字符(\0)，这样，操作字符串的代码就知道字符串在哪里结束。官方将这个 null 字符定义为 NUL，这个拼写中只有一个 L，而不是两个 L。NUL 和 NULL 指针是两回事。尽管 C++提供了更好的字符串抽象，但理解 C 语言中使用的字符串技术非常重要，因为在 C++程序设计中仍可能使用这些技术。最常见的一种情况是 C++程序调用某个第三方库或与操作系统交互时调用基于 C 语言的接口。

目前，程序员使用 C 字符串时最常犯的错误是忘记为\0 字符分配空间。例如，字符串"hello"看上

去有 5 个字符长，但在内存中需要 6 个字符的空间才能保存这个字符串的值，如图 2-1 所示。

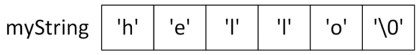

图 2-1　6 个字符的空间

C++包含一些来自 C 语言的字符串操作函数，它们在<cstring>中定义。通常，这些函数不直接操作内存分配。例如，strcpy()函数有两个字符串参数。这个函数将第二个字符串复制到第一个字符串，而不考虑第二个字符串能否恰当地填入第一个字符串。下面的代码试图在 strcpy()函数之上构建一个包装器，这个包装器能够分配正确数量的内存并返回结果，而不是接收一个已经分配好的字符串。这个最初的尝试会被证明是错误的，它通过 strlen()函数获得字符串的长度。调用者负责释放由 copyString()分配的内存：

```
char* copyString(const char* str)
{
   char* result { new char[strlen(str)] };  // BUG! Off by one!
   strcpy(result, str);
   return result;
}
```

copyString()函数的代码这样写是不正确的。strlen()函数返回字符串的长度，而不是保存这个字符串所需的内存量。对于字符串"hello"，strlen()返回的是 5，而不是 6。为字符串分配内存的正确方式是在实际字符所需的空间加 1。一开始看到到处都需要加 1 可能会感到有点奇怪，但这是其工作方式，所以在使用 C 风格的字符串时要记住这一点。正确的实现代码如下：

```
char* copyString(const char* str)
{
   char* result { new char[strlen(str) + 1] };
   strcpy(result, str);
   return result;
}
```

要记住 strlen()只返回字符串中实际字符数目的一种方式是：考虑如果为一个由几个其他字符串构成的字符串分配空间，应该怎么做。例如，如果函数接收 3 个字符串参数，并返回一个由这 3 个字符串串联而成的字符串，那么这个返回的字符串应该有多大？为精确分配足够空间，空间的大小应该是 3 个字符串的长度相加，然后加上 1 留给尾部的\0 字符。如果 strlen()的字符串长度包含\0，那么分配的内存就会过大。下面的代码通过 strcpy()和 strcat()函数执行这个操作。strcat()中的 cat 表示串联(concatenate)：

```
char* appendStrings(const char* str1, const char* str2, const char* str3)
{
   char* result { new char[strlen(str1) + strlen(str2) + strlen(str3) + 1] };
   strcpy(result, str1);
   strcat(result, str2);
   strcat(result, str3);
   return result;
}
```

C 和 C++中的 sizeof()操作符可用于获得给定数据类型或变量的大小。例如，sizeof(char)返回 1，因为字符的大小是 1 字节。但在 C 风格的字符串中，sizeof()和 strlen()是不同的。绝对不要通过 sizeof()获得字符串的大小。它根据 C 风格的字符串的存储方式来返回不同大小。如果 C 风格的字符串存储为 char[]，则 sizeof()返回字符串使用的实际内存，包括\0 字符。例如：

```
char text1[] { "abcdef" };
size_t s1 { sizeof(text1) }; // is 7
size_t s2 { strlen(text1) }; // is 6
```

但是，如果 C 风格的字符串存储为 char*，sizeof()就返回指针的大小！例如：

```
const char* text2 { "abcdef" };
size_t s3 { sizeof(text2) }; // is platform-dependent
size_t s4 { strlen(text2) }; // is 6
```

在 32 位模式下编译期，s3 的值为 4；而在 64 位模式下编译期，s3 的值为 8，因为这返回的是指针 const char*的大小。

可在<cstring>头文件中找到操作字符串的 C 函数的完整列表。

警告：

在 Microsoft Visual Studio 中使用 C 风格的字符串函数时，编译器可能会给出安全相关的警告甚至错误，说明这些函数已经被弃用了。使用其他 C 标准库函数可以避免这些警告，例如 strcpy_s()和 strcat_s()，这些函数是“安全 C 库”(ISO/IEC TR 24731)标准的一部分。然而，最好的解决方案是切换到 C++的 string 类，本章后面的 2.1.3 节“C++ std::string 类”会讨论这个类，但在那之前我们会先讨论字符串字面量。

2.1.2 字符串字面量

注意，C++程序中编写的字符串要用引号包围。例如，下面的代码输出字符串 hello，这段代码包含这个字符串本身，而不是一个包含这个字符串的变量。

```
println("hello");
```

在上面的代码中，"hello"是一个字符串字面量(string literal)，因为这个字符串以值的形式写出，而不是一个变量。字符串字面量实际上存储在内存的只读部分。通过这种方式，编译器可重用相同字符串字面量的引用，从而优化内存的使用。也就是说，即使一个程序使用了 500 次"hello"字符串字面量，编译器也只在内存中创建一个 hello 实例。这种技术称为字面量池(literal pooling)。

字符串字面量可赋值给变量，但因为字符串字面量位于内存的只读部分，且使用了字面量池，所以这样做会产生风险。C++标准正式指出：字符串字面量的类型为“n 个 const char 的数组”，然而为了向后兼容较老的不支持 const 的代码，大部分编译器不会强制你将字符串字面量赋值给 const char*类型的变量。这些编译器允许将字符串字面量赋值给不带有 const 的 char*，而且整个程序可正常运行，除非试图修改字符串。一般情况下，试图修改字符串字面量的行为是没有定义的。例如，它可能导致程序崩溃；可能使程序继续执行，看起来却有莫名其妙的副作用；可能不加通告地忽略修改行为；可能修改行为是有效的，这完全取决于编译器。例如，下面的代码展示了未定义的行为：

```
char* ptr { "hello" };          // Assign the string literal to a variable.
ptr[1] = 'a';                   // Undefined behavior!
```

一种更安全的编码方法是在引用字符串常量时，使用指向 const 字符的指针。下面的代码包含同样的 bug，但由于这段代码将字符串字面量赋值给 const char*，因此编译器会捕获到任何写入只读内存的企图。

```
const char* ptr { "hello" }; // Assign the string literal to a variable.
ptr[1] = 'a';                // Error! Attempts to write to read-only memory
```

还可将字符串字面量用作字符数组(char[])的初始值。这种情况下，编译器会创建一个足以放下这个字符串的数组，然后将字符串复制到这个数组。因此，编译器不会将字面量放在只读的内存中，也不会进行字面量池的操作。

```
char arr[] { "hello" };  // Compiler takes care of creating appropriate sized
                         // character array arr.
arr[1] = 'a';            // The contents can be modified.
```

原始字符串字面量

原始字符串字面量(raw string literal)是可横跨多行代码的字符串字面量，不需要转义嵌入的双引号，像\t 和\n 这种转义序列不按照转义序列的方式处理，而是按照普通文本的方式处理。转义字符在第 1 章讨论过了。如果像下面这样编写普通的字符串字面量，那么会收到一个编译器错误，因为字符串包含了未转义的双引号。

```
println("Hello "World"!");    // Error!
```

对于普通字符串，必须转义双引号，如下所示。

```
println("Hello \"World\"!");
```

对于原始字符串字面量，就不需要转义双引号了。原始字符串字面量以 R"(开头，以)"结尾。

```
println(R"(Hello "World"!)");
```

如果需要一个包含多行的字符串，而不使用原始字符串字面量，就需要在字符串中新行的开始位置嵌入\n 转义序列。例如：

```
println("Line 1\nLine 2");
```

这将输出：

```
Line 1
Line 2
```

而使用原始字符串字面量，不使用\n 转义序列来开始一个新行，只需要在源代码中按下 Enter 键以开始一个真正的新行。这与前面使用嵌入的\n 的代码片段的效果相同。

```
println(R"(Line 1
Line 2)");
```

在原始字符串字面量中忽略了转义序列。例如，在下面的原始字符串字面量中，\t 转义序列没有替换为实际的制表符字符，而是按照字面形式保存(即反斜杠后跟字母 t)。

```
println(R"(Is the following a tab character? \t)");
```

这将输出：

```
Is the following a tab character? \t
```

因为原始字符串字面量以)"结尾，所以使用这种语法时，不能在字符串中嵌入)"。例如，下面的字符串是不合法的，因为这个字符串的中间包含)"。

```
println(R"(Embedded )" characters)");    // Error!
```

如果需要嵌入)"，则需要使用扩展的原始字符串字面量语法，如下所示。

```
R"d-char-sequence(r-char-sequence)d-char-sequence"
```

r-char-sequence 是实际的原始字符串。d-char-sequence 是可选的分隔符序列，原始字符串首尾的分隔符序列应该一致。分隔符序列最多能有 16 个字符。应选择未出现在原始字符串字面量中的序列作为分隔符序列。

上例可改用唯一的分隔符序列。

```
println(R"-(Embedded )" characters)-");
```

在操作数据库查询字符串、正则表达式和文件路径时，原始字符串字面量可以令程序的编写更加方便。第 21 章将讨论正则表达式。

2.1.3　C++ std::string 类

C++提供了一个得到极大改善的字符串概念，并作为标准库的一部分提供了这个字符串的实现。在 C++中，std::string 是一个类(实际上是 basic_string 模板类的一个实例)，这个类支持<cstring>中提供的许多功能，还能自动管理内存分配。string 类在 std 命名空间的<string>头文件中定义，在本书中已经多次使用了 string 类。下面深入学习这个类。

1. C 风格的字符串的问题

为理解 C++ string 类的必要性，需要考虑 C 风格字符串的优势和劣势。

优势：

- 很简单，底层使用了基本的字符类型和数组结构。
- 轻量级，如果使用得当，只会占用所需的内存。
- 可按操作原始内存的方式轻松操作和复制字符串。
- 如果你是一名 C 语言程序员——为什么还要学习新事物？

劣势：

- 为了模拟一等数据类型字符串，需要付出很多努力。
- 很容易产生难以找到的内存 bug，且难以解决。
- 没有利用 C++的面向对象特性。
- 要求程序员了解底层的表示方式。

上面的列表是精心准备的，从而可让人思考也许有更好的方式。如后面所述，C++的 string 类解决了 C 风格字符串的所有问题，并且证明了 C 风格字符串相对于一等数据类型的那些优势实际上是无关紧要的。

2. 使用 std::string 类

尽管 string 是一个类，但是几乎总可将 string 当成内建类型使用。事实上，把 string 想象为简单类型更容易发挥 string 的作用。通过运算符重载的神奇作用，C++的 string 使用起来比 C 字符串容易得多。接下来的两节将通过演示运算符重载如何使连接和比较字符串变得容易来开始讨论。之后讨论了 C++字符串如何处理内存，它们与 C 风格字符串的兼容性，以及可以对字符串执行的一些内置操作。

字符串连接

字符串的+运算符被重新定义为表示“字符串连接”。以下代码产生 1234：

```
string a { "12" };
string b { "34" };
string c { a + b };    // c is "1234"
```

+=运算符也被重载了，通过这个运算符可以轻松地追加一个字符串。

```
a += b;     // a is "1234"
```

字符串比较

C 风格字符串的另一个问题是不能通过==运算符进行比较。假设有以下两个字符串。

```
char* a { "12" };
char b[] { "12" };
```

按照下述方式编写的比较操作始终返回 false，因为它比较的是指针的值，而不是字符串的内容。

```
if (a == b) { /* ... */ }
```

注意C 数组和指针是相关的。可将C 数组(如示例中的b 数组)看成指向数组中第一个元素的指针。第 7 章将深入论述数组-指针的对偶性。

要比较 C 字符串，需要编写这样的代码：

```
if (strcmp(a, b) == 0) { /* ... */ }
```

此外，C 字符串也无法通过<、<=、>=或>进行比较，因此需要通过 strcmp()根据字符串的字典顺序返回−1、0 和 1 的值判断。这样会产生非常笨拙且可读性低的代码，还很容易出错。

在 C++的 string 类中，操作符(==、!=和<等)都被重载了，这些运算符可以操作真正的字符串字符。例如：

```
string a { "Hello" };
string b { "World" };
println("'{}' < '{}' = {}", a, b, a < b);  // 'Hello' < 'World' = true
println("'{}' > '{}' = {}", a, b, a > b);  // 'Hello' > 'World' = false
```

C++ string 类另外提供了一个 compare()方法，它的行为类似于 strcmp()并且具有类似的返回类型。下面是一个例子：

```
string a { "12" };
string b { "34" };

auto result { a.compare(b) };
if (result < 0) { println("less"); }
if (result > 0) { println("greater"); }
if (result == 0) { println("equal"); }
```

与 strcmp() 一样，这使用起来很麻烦，你需要记住返回值的确切含义。此外，由于返回值只是一个整数，很容易忘记这个整数的含义，写出以下错误的代码来比较相等性。

```
if (a.compare(b)) { println("equal"); }
```

compare()为相等返回 0，为不相等返回任何其他值。所以，这行代码与它的意图相反，也就是说，它对不相等的字符串输出“equal”！如果只想检查两个字符串是否相等，不要使用 compare()，只需要使用==。

从 C++20 开始，第 1 章介绍的三向比较运算符改进了这一点。string 类完全支持这个运算符，下面是一个例子：

```
auto result { a <=> b };
if (is_gt(result)) { println("greater"); }
if (is_lt(result)) { println("less"); }
if (is_eq(result)) { println("equal"); }
```

内存处理

如下面的代码所示，当 string 操作需要扩展 string 时，string 类能够自动处理内存需求，因此不会再出现内存溢出的情况了。此代码片段还演示了可以使用方括号运算符[]访问单个字符，就像使用 C 风格的字符串一样。

```
string myString { "hello" };
myString += ", there";
string myOtherString { myString };
if (myString == myOtherString) {
   myOtherString[0] = 'H';
}
println("{}", myString);
println("{}", myOtherString);
```

这段代码的输出如下所示。

```
hello, there
Hello, there
```

在这个例子中有几点需要注意。一是要注意即使字符串被分配和调整大小，也不会出现内存泄漏的情况。所有这些 string 对象都创建为栈中的变量。尽管 string 类肯定需要完成大量分配内存和调整大小的工作，但是 string 类的析构函数会在 string 对象离开作用域时清理内存。析构函数工作的细节会在第 8 章详细讨论。

另外需要注意的是，运算符以预期的方式工作。例如，=运算符复制字符串，这是最有可能预期的操作。如果习惯使用基于数组的字符串，那么这种方式有可能带来全新体验，也可能令你迷惑。不用担心，一旦学会信任 string 类能做出正确的行为，那么代码编写会简单得多。

与 C 风格字符串的兼容

为达到兼容的目的，还可应用 string 类的 c_str()方法获得一个表示 C 风格字符串的 const char 指针。不过，一旦 string 执行任何内存重分配或 string 对象被销毁了，返回的这个 const 指针就失效了。应该在使用结果之前调用这个方法，以便它准确反映 string 当前的内容。永远不要从函数中返回在基于栈的 string 上调用 c_str()的结果。

还有一个 data()方法，在 C++14 及更早的版本中，始终与 c_str()一样返回 const char*。从 C++17 开始，在非 const 字符串上调用时，data()返回 char*。

string 上的操作

string 类支持一系列其他的操作，以下列出了一部分。获取标准库参考(见附录 B)，了解可以对 string 对象执行的所有支持操作的完整列表。

- substr(pos, len)：返回从给定位置开始的给定长度的子字符串。
- find(str)：如果找到了给定的子串，返回它的位置；如果没有找到，返回 string::npos。
- replace(pos, len, str)：将字符串的一部分(给定开始位置和长度)替换为另一个字符串。
- starts_with(str)/ends_with(str)：如果一个字符串以给定的子串开始或者结尾，则返回 true。
- contains(str)/contains(ch)：如果一个字符串包含另一个字符串或某个字符，则返回 true。

C++23

这是一个小代码片段，展示了其中的一些操作。

```
string strHello { "Hello!!" };
string strWorld { "The World..." };
auto position { strHello.find("!!") };
if (position != string::npos) {
```

```
        // Found the "!!" substring, now replace it.
        strHello.replace(position, 2, strWorld.substr(3, 6));
    }
    println("{}", strHello);
    // Test contains().
    string toFind { "World" };
    println("{}", strWorld.contains(toFind));
    println("{}", strWorld.contains('.'));
    println("{}", strWorld.contains("Hello"));
```

输出如下：

```
Hello World
true
true
false
```

C++23

在 C++23 之前，可以通过将 nullptr 传递给其构造函数来构造字符串对象。这将导致运行时出现未定义的行为。从 C++23 开始，尝试从 nullptr 构造字符串会导致编译错误。

3. std::string 字面量

源代码中的字符串字面量通常解释为 const char*或 const char[]。使用用户定义的标准字面量后缀 s 可以把字符串字面量解释为 std::string。例如：

```
auto string1 { "Hello World" };     // string1 is a const char*.
auto& string2 { "Hello World" };    // string2 is a const char[12].
auto string3 { "Hello World"s };    // string3 is an std::string.
```

标准字面量后缀 s 在 std::literals::string_literals 命名空间中定义。但是，string_literals 和 literals 都是内联命名空间。因此，使用以下选项使这些字符串字面量可用于你的代码。

```
using namespace std;
using namespace std::literals;
using namespace std::string_literals;
using namespace std::literals::string_literals;
```

基本上，在内联命名空间中声明的所有内容都会自动在父命名空间中可用。要自己定义内联命名空间，可以使用 inline 关键字。例如，string_literals 内联命名空间定义如下：

```
namespace std {
    inline namespace literals {
        inline namespace string_literals {
            // ...
        }
    }
}
```

4. std::vector 和字符串的 CTAD

第 1 章解释了 std::vector 支持类模板参数推导 (CTAD)，允许编译器根据初始化列表自动推导 vector 的类型，对字符串 vector 使用 CTAD 时必须小心。以 vector 的以下声明为例：

```
vector names { "John", "Sam", "Joe" };
```

推导出的类型将是 vector<const char*>，而非 vector<string>！这是一个很容易犯的错误，可能导

致代码出现一些奇怪的行为，甚至崩溃。这取决于之后对 vector 的处理方式。

如果你需要一个 vector<string>，可以使用上一节提到的 std::string 字面量。注意下例中每个字符串字面量后面的 s：

```
vector names { "John"s, "Sam"s, "Joe"s };
```

2.1.4 数值转换

C++标准模板库同时提供了高级数值转换函数和低级数值转换函数，下一节将详细解释。

1. 高级数值转换函数

std 命名空间包含很多辅助函数，以便完成数值和字符串之间的转换，它们定义在<string>中。它们可使数值与字符串之间的相互转换更加容易。

数值转换为字符串

下面的函数可用于将数值转换为字符串，*T* 可以是(unsigned) int、(unsigned) long、(unsigned) long long、float、double 及 long double。所有这些函数都负责内存分配，它们会创建一个新的 string 对象并返回。

```
string to_string(T val);
```

这些函数的使用非常简单直观。例如，下面的代码将 long double 值转换为字符串。

```
long double d { 3.14L };
string s { to_string(d) };  // s contains 3.140000
```

字符串转换为数值

通过下面这组同样在 std 命名空间中定义的函数，可以将字符串转换为数值。在这些函数原型中，str 表示要转换的字符串，pos 是一个指针，这个指针接收第一个未转换的字符的索引，base 表示转换过程中使用的进制。pos 指针可以是空指针，如果是空指针，则被忽略。如果不能执行任何转换，这些函数会抛出 invalid_argument 异常。如果转换的值超出返回类型的范围，则抛出 out_of_range 异常。

```
int stoi(const string& str, size_t *pos = nullptr, int base = 10);
long stol(const string& str, size_t *pos = nullptr, int base = 10);
unsigned long stoul(const string& str, size_t *pos = nullptr, int base = 10);
long long stoll(const string& str, size_t *pos = nullptr, int base = 10);
unsigned long long stoull(const string& str, size_t *pos = nullptr, int base = 10);
float stof(const string& str, size_t *pos = nullptr);
double stod(const string& str, size_t *pos = nullptr);
long double stold(const string& str, size_t *pos = nullptr);
```

下面是一个示例：

```
const string toParse { "  123USD" };
size_t index { 0 };
int value { stoi(toParse, &index) };
println("Parsed value: {}", value);
println("First non-parsed character: '{}'", toParse[index]);
```

输出如下所示：

```
Parsed value: 123
First non-parsed character: 'U'
```

stoi()、stol()、stoul()、stoll()和 stoull()接收整数值并且有一个名为 base 的参数，表明了给定的数值应该用什么进制来表示。base 的默认值为 10，采用数字为 0~9 的十进制，base 为 16 表示采用十六进制。如果 base 被设为 0，函数会按照以下规则自动计算给定数字的进制。

- 如果数字以 0x 或者 0X 开头，则被解析为十六进制数字。
- 如果数字以 0 开头，则被解析为八进制数字。
- 其他情况下，被解析为十进制数字。

2. 低级数值转换

C++也提供了许多低级数值转换函数，这些都在<charconv>头文件中定义。这些函数不执行内存分配，也不直接使用 std::string，而使用由调用者分配的缓存区。此外，它们还针对性能进行了优化，并且与语言环境无关(有关本地化的详细信息，请参见第 21 章)。最终结果是，这些函数可以比其他高级数值转换函数快几个数量级。这些函数也是为快速往返而设计的，这意味着将数值序列化为字符串表示，然后将结果字符串反序列化为数值，结果与原始值完全相同。

如果希望实现高性能、完美往返、独立于语言环境的转换，则应当使用这些函数。例如，在数值数据与人类可读格式(如 JSON、XML 等)之间进行序列化/反序列化。

数值转换为字符串

要将整数转换为字符，可使用下面一组函数。

```
to_chars_result to_chars(char* first, char* last, IntegerT value, int base = 10);
```

这里，IntegerT 可以是任何有符号或无符号的整数类型或 char 类型。结果是 to_chars_result 类型，类型定义如下所示。

```
struct to_chars_result {
   char* ptr;
   errc ec;
};
```

如果转换成功，ptr 成员将等于所写入字符尾后一位置的指针。如果转换失败(即 ec==errc::value_too_large)，则它等于 last。如果 ec 等于默认构造的 errc，则转换成功。

下面是一个使用示例：

```
const size_t BufferSize { 50 };
string out(BufferSize, ' '); // A string of BufferSize space characters.
auto result { to_chars(out.data(), out.data() + out.size(), 12345) };
if (result.ec == errc{}) { println("{}", out); /* Conversion successful. */ }
```

使用第 1 章介绍的结构化绑定，可以将其写成：

```
string out(BufferSize, ' '); // A string of BufferSize space characters.
auto [ptr, error] { to_chars(out.data(), out.data() + out.size(), 12345) };
if (error == errc{}) { println("{}", out); /* Conversion successful. */ }
```

类似地，下面的一组转换函数可用于浮点类型。

```
to_chars_result to_chars(char* first, char* last, FloatT value);
```

```
to_chars_result to_chars(char* first, char* last, FloatT value,
                         chars_format format);
to_chars_result to_chars(char* first, char* last, FloatT value,
                         chars_format format, int precision);
```

这里，*FloatT* 可以是任意浮点类型，如 float、double 或 long double。可使用 chars_format 标志的组合指定格式：

```
enum class chars_format {
    scientific,                     // Style: (-)d.ddde±dd
    fixed,                          // Style: (-)ddd.ddd
    hex,                            // Style: (-)h.hhhp±d (Note: no 0x!)
    general = fixed | scientific    // See next paragraph.
};
```

默认格式是 chars_format::general，这将导致 to_chars()将浮点值转换为(−)ddd.ddd 形式的十进制表示形式，或(−)d.ddde±dd 形式的十进制指数表示形式，得到最短的表示形式，小数点前至少有一位数字(如果存在)。如果指定了格式，但未指定精度，将为给定格式自动确定最简短的表示形式，最大精度为 6 个数字。例如：

```
double value { 0.314 };
string out(BufferSize, ' '); // A string of BufferSize space characters.
auto [ptr, error] { to_chars(out.data(), out.data() + out.size(), value) };
if (error == errc{}) { println("{}", out); /* Conversion successful. */ }
```

字符串转换为数值

对于相反的转换，即将字符串转换为数值，可使用下面的一组函数[1]。

```
from_chars_result from_chars(const char* first, const char* last, IntegerT& value,
                             int base = 10);
from_chars_result from_chars(const char* first, const char* last, FloatT& value,
                             chars_format format = chars_format::general);
```

from_chars_result 的类型定义如下：

```
struct from_chars_result {
    const char* ptr;
    errc ec;
};
```

结果类型的 ptr 成员是指向第一个未转换字符的指针，如果所有字符都成功转换，则它等于 last。如果所有字符都未转换，则 ptr 等于 first，错误代码的值将为 errc::invalid_argument。如果解析后的值过大，无法由给定类型表示，则错误代码的值将是 errc::result_out_of_range。注意，from_chars()不会忽略任何前导空白。

to_chars()和 form_chars()的完美往返特性可以表示如下：

```
double value1 { 0.314 };
string out(BufferSize, ' '); // A string of BufferSize space characters.
auto [ptr1, error1] { to_chars(out.data(), out.data() + out.size(), value1) };
if (error1 == errc{}) { println("{}", out); /* Conversion successful. */ }

double value2;
```

1 从 C++23 开始，to_chars()和 from_chars()的整数重载被标记为 constexpr。这意味着它们可以在其他 constexpr 函数和类中在编译时进行求值。有关 constexpr 的讨论，请参阅第 9 章。

```
auto [ptr2, error2] { from_chars(out.data(), out.data() + out.size(), value2) };
if (error2 == errc{}) {
    if (value1 == value2) {
        println("Perfect roundtrip");
    } else {
        println("No perfect roundtrip?!?");
    }
}
```

2.1.5　std::string_view 类

在 C++17 之前，为接收只读字符串的函数选择形参类型一直是一件进退两难的事情。它应当是 const char*吗？那样的话，如果客户使用 std::string，则必须调用其上的 c_str()或 data()来获取 const char*。更糟的是，函数将失去 std::string 良好的面向对象的方面及其良好的辅助成员函数。或许，形参应改用 const std::string&？这种情况下，始终需要 std::string。例如，如果传递一个字符串字面量，编译器将默认创建一个临时字符串对象(其中包含字符串字面量的副本)，并将该对象传递给函数，因此会增加一点开销。有时，人们会编写同一函数的多个重载版本，一个接收 const char*，另一个接收 const string&，但显然，这并不是一个优雅的解决方案。

在 C++17 中，通过引入 std::string_view 类解决了所有这些问题，std::string_view 类是 std::basic_string_view 类模板的实例化，在<string_view>中定义。string_view 基本上就是 const string& 的简单替代品，但不会产生开销。它从不复制字符串，string_view 提供字符串的只读视图，并支持类似于字符串的接口，包括 C++23 中引入的 contains()成员函数。一个例外是缺少 c_str()，但 data()是可用的。另外，string_view 添加了 remove_prefix(size_t)和 remove_suffix(size_t)成员函数，前者将起始指针前移给定的偏移量来收缩字符串，后者则将结尾指针倒退给定的偏移量来收缩字符串。就像字符串一样，从 C++23 开始，从 nullptr 构造 string_view 会导致编译错误。

如果知道如何使用 std::string，那么使用 string_view 将变得十分简单，如下面的代码片段所示。extractExtension()函数提取给定文件名的扩展名(包括点号)并返回。注意，通常按值传递 string_views，因为它们的复制成本极低。它们只包含指向字符串的指针以及字符串的长度。rfind()成员函数从后面开始在字符串中搜索另一个给定的字符串或字符。对 string_view 调用的 substr()成员函数返回一个 string_view，该函数被传递给字符串构造函数将其转换为字符串，然后从函数返回。

```
string extractExtension(string_view filename)
{
    // Return a copy of the extension.
    return string { filename.substr(filename.rfind('.')) };
}
```

该函数可用于所有类型的字符串：

```
string filename { R"(c:\temp\my file.ext)" };
println("C++ string: {}", extractExtension(filename));

const char* cString { R"(c:\temp\my file.ext)" };
println("C string: {}", extractExtension(cString));

println("Literal: {}", extractExtension(R"(c:\temp\my file.ext)"));
```

对 extractExtension()的所有这些调用中，没有进行一次实参的复制。extractExtension()函数的 fileName 参数只是指针和长度，该函数的返回类型也是如此。这都十分高效。

还有一个 string_view 构造函数，它接收任意原始缓冲区和长度。这可用于从字符串缓冲区(并非以 NUL 终止)构建 string_view。如果确实有一个以 NUL(\0)终止的字符串缓冲区，但你已经知道字符串的长度，构造函数不必再次统计字符数目，例如：

```
const char* raw { /* ... */ };
size_t length { /* ... */ };
println("Raw: {}", extractExtension({ raw, length }));
```

最后一行代码也可以写成这样：

```
println("Raw: {}", extractExtension(string_view { raw, length }));
```

最后，还可以用一个通用范围构造 string_view，该范围是基于迭代器的范围，自 C++23 以来，它是现代范围。第 17 章讨论了迭代器、通用范围和现代范围。

注意：

每当函数需要将只读字符串作为一个参数时，可用 std::string_view 替代 const std::string&或 const char*。

无法从 string_view 隐式构建一个 string。这是被禁止的，以防止在 string_view 中意外复制字符串，因为从 string_view 构造字符串总是涉及复制数据。要将 string_view 转换为字符串，请使用显式字符串构造函数。这正是 extractExtension()中的 return 语句所做的：

```
return string { filename.substr(filename.rfind('.')) };
```

由于同样的原因，无法连接一个 string 和一个 string_view。下面的代码将无法编译：

```
string str { "Hello" };
string_view sv { " world" };
auto result { str + sv };  // Error, does not compile!
```

可以使用 string 的构造函数将 string_view 转换为 string：

```
auto result1 { str + string { sv } };
```

或者你可以使用 append()：

```
string result2 { str };
result2.append(sv.data(), sv.size());
```

警告：

返回字符串的函数应返回 const std::string&或 string，但不应返回 string_view。返回 string_view 会带来使返回的 string_view 无效的风险，例如当它指向的字符串需要重新分配时。

警告：

将 const string&或 string_view 存储为类的数据成员需要确保它们指向的字符串在对象的生命周期内保持有效状态。强烈建议存储 std::string。

1. std::string_view 和临时字符串

string_view 不应该用于保存一个临时字符串的视图，考虑以下示例：

```
string s { "Hello" };
string_view sv { s + " World!" };
println("{}", sv);
```

此代码段具有未定义的行为，即运行此代码时发生的情况取决于编译器和编译器设置。它可能会崩溃，它可能会打印“ello World！”(没有字母 H)，等等。为什么这是未定义的行为？字符串视图 sv 的初始化表达式将生成一个临时字符串，其中包含“Hello World！”。然后，string_view 存储指向此临时字符串的指针。在第二行代码的末尾，这个临时字符串被销毁，留下一个悬空指针的 string_view。

警告：
永远不要使用 string_view 保存临时字符串的视图。

2. std::string_view 字面量

可使用标准的用户定义的字面量 sv，将字符串字面量解释为 std::string_view。例如：

```
auto sv { "My string_view"sv };
```

标准的用户定义的字面量 sv 需要以下几条 using 命令之一：

```
using namespace std::literals::string_view_literals;
using namespace std::string_view_literals;
using namespace std::literals;
using namespace std;
```

2.1.6　非标准字符串

许多 C++程序员都不使用 C++风格的字符串，这有几个原因。一些程序员只是不知道有 string 类型，因为它并不总是 C++规范的一部分。其他程序员发现，C++ string 没有提供他们需要的行为，或他们不喜欢 std::string 对字符编码不感知这一事实，所以开发了自己的字符串类型。第 21 章将回到字符编码的主题。

也许最常见的原因是，开发框架和操作系统有自己的表达字符串的方式，例如 Microsoft MFC 中的 CString 类。它常用于向后兼容或解决遗留的问题。在 C++中启动新项目时，提前确定团队如何表示字符串是非常重要的。务必注意以下几点：

- 不应当选择 C 风格的字符串表示。
- 可对自己所使用框架中可用的字符串功能进行标准化，如 MFC、QT 内置的字符串功能。
- 如果为字符串使用 std::string，应当使用 std::string_view 将只读字符串作为参数传递给函数，否则，看一下你的框架是否支持类似于 string_view 的类。

2.2　字符串格式化与打印

在 C++20 之前， 字符串的格式化一般是通过 printf()之类的 C 风格函数或 std::cout 之类的 C++的 I/O 流完成的。

- C 风格函数：
  - 不推荐，因为它们不是类型安全的，并且无法扩展支持自定义类型。
  - 因为字符串和参数是分开的，所以可读性高，且容易翻译成不同语言。
    例如：

```
printf("x has value %d and y has value %d.\n", x, y);
```

- C++ I/O 流：
  - 推荐(C++20 之前)，因为类型安全且可扩展。
  - 因为字符串和参数交织在一起，所以可读性差，且难以翻译成不同语言。
    例如：

```
cout << "x has value " << x << " and y has value " << y << '.' << endl;
```

C++20 引入了 std::format()，用来格式化字符串，它定义在<format>中。它基本上结合了 C 风格函数和 C++的 I/O 流的所有优点，是一种类型安全且可扩展的机制。例如：

```
cout << format("x has value {} and y has value {}.", x, y) << endl;
```

C++23 通过引入 std::print()和 println()使它变得更加容易。例如：

```
println("x has value {} and y has value {}.", x, y);
```

此外，std::print()和 println()更好地支持将 UTF-8 Unicode 文本写入兼容 Unicode 的控制台。Unicode 在第 21 章中进行了讨论，但这里有一个简单例子：

```
println("こんにちは世界");
```

这将正确打印字符串“こんにちは世界”，这是“Hello World”的日语版本。如果尝试使用 C++ I/O 流按如下方式打印此字符串，则根据你的控制台[1]设置，输出可能会出现乱码，例如“πüôπéôπül⁄2πüíπü»Ση ûτòî”：

```
cout << "こんにちは世界" << endl;
```

由于 Unicode 支持，你甚至可以打印表情符号。如果你的输出控制台正确支持 Unicode，下面将打印一个笑脸。使用 cout 可能导致输出混乱。

```
println("😀");
```

std::print()和 println()现在是向控制台写入文本的推荐方法，因此，本书中的所有例子都使用了它们。它们是类型安全的，可扩展以支持用户类型，易于阅读，支持 Unicode 输出，支持不同语言的本地化等等。除了所有这些好处，与直接使用 C++ I/O 流相比，print()和 println()的性能也要好得多，尽管在底层，print()和 println()仍然在使用这样的流。

2.2.1 格式字符串

std::format()、print()和 println()使用一个格式字符串，一个指定给定参数在输出字符串中必须如何格式化的字符串。它的基本形式在上一章中介绍过，并在整个示例中使用过。现在是时候看看这些格式字符串到底有多强大了。

格式字符串通常是 format()、print()和 println()的第一个参数。格式字符串可以包含一组花括号{}，它们表示替换字段。可以根据需要设置任意数量的替换字段。format()、print()和 println()的后续参数是用于填充这些替换字段的值。如果你需要在输出中包含{和}字符，那么你需要将它们转义为{{或}}。

到目前为止，替换字段一直是空白的花括号{}，但这只是开始。在这些花括号内可以是格式为[index][:specifier]的字符串：

1 要编译包含 Unicode 字符的源代码，可能需要传递一个编译器开关。对于 Visual C++，必须传递/utf-8 编译器开关。对于 GCC，请使用命令行选项-finput-charset=UTF-8。Clang 默认假设所有文件都是 UTF-8 编码。请查阅编译器文档。

- 可选的 index 是一个参数索引，将在下一节中讨论。
- 可选的 specifier 是一个格式说明符，用于规定值在输出中必须如何格式化，将在“格式说明符”一节中详细解释。

必须向 format()、print()和 println()传递格式字符串。例如，不能直接按如下方式打印值：

```
int x { 42 };
println(x);
```

相反，可以使用以下内容：

```
println("{}", x);
```

也不能仅通过以下内容来打印单个换行符：

```
println();
```

相反，请使用以下内容：

```
println("");
```

2.2.2　参数索引

可从所有替换字段中省略索引，也可以为所有替换字段指定传递给 format()、print()和 println()的值之一的从零开始的索引，作为应用于替换字段的第二个和后续参数。如果你想多次输出某个值，你可以多次使用该索引。如果省略索引，则作为第二个和后续参数传递的值将按给定顺序用于所有替换字段。

以下对 println()的调用省略了替换字段中的显式索引：

```
int n { 42 };
println("Read {} bytes from {}", n, "file1.txt");
```

可以按如下方式指定手动索引：

```
println("Read {0} bytes from {1}", n, "file1.txt");
```

不允许混合使用手动索引和自动索引。以下内容使用了无效的格式字符串：

```
println("Read {0} bytes from {}", n, "file1.txt");
```

输出字符串中格式化值的顺序可以更改，而无需更改参数的实际顺序。如果你想在软件中翻译字符串，这是一个有用的功能。某些语言在句子中有不同的顺序。例如，前面的格式字符串可按如下方式翻译成中文。在中文中，句子中替换字段的顺序是颠倒的，但由于在格式字符串中使用了参数索引，println()的参数顺序保持不变。

```
println("从{1}中读取{0}个字节。", n, "file1.txt");
```

2.2.3　打印到不同的目的地

到目前为止，每次调用 print()和 println()都有一个格式字符串作为第一个参数，后面跟着一些额外的参数。例如：

```
println("x has value {} and y has value {}.", x, y);
```

这将字符串打印到标准输出流，与 std::cout 相同的流。

正如第 1 章所解释的，还有 std::cerr，它流式传输到标准错误控制台。可以使用 print()和 println()打印到错误控制台，如下所示：

```
println(cerr, "x has value {} and y has value {}.", x, y);
```

C++23

2.2.4 格式字符串的编译期验证

从 C++23 开始，format()[1]、print()和 println()的格式字符串必须是编译期常量，以便编译器可以在编译期检查格式字符串中是否有任何语法错误。这意味着以下内容无法编译：

```
string s { "Hello World!" };
println(s); // Error! Does not compile.
```

产生的错误取决于编译器，不幸的是，在撰写本书时，它相当模糊，并不总是能立即帮助确定错误的确切原因。例如，以下是 Microsoft Visual C++2022 编译器的错误：

```
error C7595: 'std::basic_format_string<char>::basic_format_string': call to
immediate function is not a constant expression
```

正确的使用如下：

```
string s { "Hello World!" };
println("{}", s);
```

当然也允许使用 constexpr 格式的字符串，因为它们是编译期常量。第 9 章详细讨论了 constexpr 关键字。

```
constexpr auto formatString { "Value: {}" };
println(formatString, 11); // Value: 11
```

1. 非编译期常量格式字符串

当需要为不同语言本地化/翻译格式字符串时，格式字符串必须是编译期常量的事实可能会有点麻烦。在这种情况下，可以使用 std::vprint_unicode()或 std::vprint_nonunicode()。不过，它们有点难用。你不能像使用 print()那样只传递参数，你需要使用 std::make_format_args()实现。以下是一个示例：

```
use std::make_format_args() to do so. Here's an example:
enum class Language { English, Dutch };

string_view GetLocalizedFormat(Language language)
{
switch (language) {
case Language::English: return "Numbers: {0} and {1}.";
case Language::Dutch: return "Getallen: {0} en {1}.";
}
}

int main()
{
Language language { Language::English };
vprint_unicode(GetLocalizedFormat(language), make_format_args(1, 2));
println("");
language = Language::Dutch;
```

1 这是 std::format()的一个重大变更。在 C++23 之前，format()的格式字符串并没有被强制要求为编译时常量。

```
vprint_unicode(GetLocalizedFormat(language), make_format_args(1, 2));
}
```

输出为：

```
Numbers: 1 and 2.
Getallen: 1 en 2.
```

以下使用 print()的调用无法编译，因为它需要一个编译期常量格式字符串：

```
print(GetLocalizedFormat(language), 1, 2);
```

2. 非编译期常量格式字符串的错误处理

当在运行时而不是在编译期验证格式字符串时，对于任何格式字符串错误都会抛出 std::format_error 异常。如前所述，std: format()、print()和 println()等函数永远不会抛出此类异常，因为格式字符串都是在编译期验证的。但是，std::vformat()和 vprint_unicode()等函数(见上一节)不要求格式字符串是常量，因此不在编译期验证，而是在运行时验证。这些函数可能抛出 format_error 异常。以下是一个示例：

```
try {
   vprint_unicode("An integer: {5}", make_format_args(42));
} catch (const format_error& caught_exception) {
   println("{}", caught_exception.what()); // "Argument not found."
}
```

现在，让我们研究一下格式说明符到底有多强大。

2.2.5　格式说明符

如前所述，格式字符串可以包含由花括号分隔的替换字段。在这些花括号内可以是格式为[index][:specifier]的字符串。本节讨论替换字段的格式说明符部分。index 在前面讨论过。

格式说明符用于控制值在输出中的格式，前缀为冒号。格式说明符的一般形式如下所示(注释：严格来说，在精度和类型之间可以有一个可选的 L。这与地区特定格式有关，本处不做进一步讨论)：

```
[[fill]align][sign][#][0][width][.precision][L][type]
```

方括号里的所有说明符都是可选的。下一小节将详细讨论各个说明符。

1. width

width 指定待格式化的值所占字段的最小宽度。width 也可以是另一组花括号，称为动态宽度。如果在花括号中指定了索引，例如{3}，则动态宽度的 width 取自给定的索引对应的 format()的实参。如果未指定索引，例如{}，则 width 取自实参列表中的下一个参数。

示例如下：

```
int i { 42 };
println("|{:5}|", i);          //|   42|
println("|{:{}}|", i, 7);      //|     42|
println("|{1:{0}}|", 7, i);    //|     42|
```

2. [fill]align

[fill]align 也是可选的，说明使用哪个字符作为填充字符，然后是值在其字段中的对齐方式：

- <表示左对齐(非整数和非浮点数的默认对齐方式)。
- >表示右对齐(整数和浮点数的默认对齐方式)。
- ^表示居中对齐。

fill 字符会被插入输出中，以确保输出中的字段达到说明符的[width]指定的最小宽度。如果未指定[width]，则[fill]align 无效。

示例如下：

```
int i { 42 };
println("|{:7}|", i);    //|     42|
println("|{:<7}|", i);   //|42     |
println("|{:_>7}|", i);  //|_____42|
println("|{:_^7}|", i);  //|__42___|
```

以下是一个有趣的技巧，可以将一个字符输出特定次数。可以在格式说明符中明确指定所需的字符数，而不是自己键入包含正确字符数的字符串文字：

```
println("|{:=>16}|", ""); //|================|
```

3. sign

sign 可以是下列三项之一：

- -表示只显示负数的符号(默认方式)。
- +表示显示正数和负数的符号。
- space 表示对于负数使用负号，对于正数使用空格。

示例如下：

```
int i { 42 };
println("|{:<5}|", i);   //|42   |
println("|{:<+5}|", i);  //|+42  |
println("|{:< 5}|", i);  //| 42  |
println("|{:< 5}|", -i); //|-42  |
```

4.

#启用所谓的备用格式(alternate formatting)规则。如果为整型启用，并且指定了十六进制、二进制或八进制数字格式，则备用格式会在格式化数字前面插入 0x、0X、0b、0B 或 0。如果为浮点类型启用，则备用格式将始终输出十进制分隔符，即使后面没有数字。

以下两节给出了备用格式的示例。

5. type

type 指定了给定值要被格式化的类型，以下是几个选项。

- 整型：b(二进制)、B(二进制，当指定#时，使用 0B 而不是 0b)、d(十进制)、o(八进制)、x(小写字母 a、b、c、d、e、f 的十六进制)、X(大写字母 A、B、C、D、E、F 的十六进制，当指定#时，使用 0X 而不是 0x)。如果 type 未指定，整型默认使用 d。
- 浮点型：支持以下浮点格式。科学、固定、通用和十六进制格式的结果与本章前面讨论的 std::chars_format::scientific、fixed、general 和 hex 相同。
  - e,E：以小写 e 或大写 E 表示指数的科学表示法，按照给定精度或 6(如果未指定精度)格式化。
  - f,F：固定表示法，按照给定精度或 6(如果未指定精度)格式化。
  - g,G：以小写 e 或大写 E 表示指数的通用表示法，按照给定精度或 6(如果未指定精度)格式化。

  - a,A：带有小写字母(a)或大写字母(A)的十六进制表示法。
  - 如果 type 未指定，浮点型默认使用 g。
- 布尔型：s(以文本形式输出 true 或 false)，b、B、c、d、o、x、X(以整型输出 1 或 0)。如果 type 未指定，布尔型默认使用 s。
- 字符型：c(输出字符副本)、?(转义字符被复制到输出，参见“格式化转义字符和字符串”一节)，b、B、d、o、x、X(整数表示)。如果 type 未指定，字符型默认使用 c。
- 字符串：s(输出字符串副本)、?(转义字符被复制到输出，参见“格式化转义字符和字符串”一节)。如果 type 未指定，字符串默认使用 s。
- 指针：p(0x 为前缀的十六进制表示法)。如果 type 未指定，指针默认使用 p。只能格式化 void* 类型的指针。其他指针类型必须首先转换为 void* 类型，例如使用 static_cast<void*>(myPointer)。

整型的示例如下：

```
int i { 42 };
println("|{:10d}|", i);     //|        42|
println("|{:10b}|", i);     //|    101010|
println("|{:#10b}|", i);    //|  0b101010|
println("|{:10X}|", i);     //|        2A|
println("|{:#10X}|", i);    //|      0X2A|
```

字符串的示例如下：

```
string s { "ProCpp" };
println("|{:_^10}|", s);    //|__ProCpp__|
```

浮点型的示例将在下一小节给出。

6. precision

precision 只能用于浮点和字符串类型。它的格式为一个点后跟浮点类型要输出的小数位数，或字符串要输出的字符数。浮点类型的位数包括所有数字，包括小数分隔符之前的数字，除非使用固定浮点表示法(f 或 F)，在这种情况下，精度是小数点之后的位数。

就像 width 一样，这也可以是另一组花括号，在这种情况下，它被称为动态精度。precision 取自 format()的实参列表中的下一个实参或具有给定索引的实参。

浮点型的示例如下：

```
double d { 3.1415 / 2.3 };
println("|{:12g}|", d);                              //|     1.36587|
println("|{:12.2}|", d);                             //|         1.4|
println("|{:12e}|", d);                              //|1.365870e+00|

int width { 12 };
int precision { 3 };
println("|{2:{0}.{1}f}|", width, precision, d);  //|       1.366|
println("|{2:{0}.{1}}|", width, precision, d);   //|        1.37|
```

7. 0

0 表示，对于数值，将 0 插入格式化结果中，以达到 [width]指定的最小宽度(请参阅前面的内容)。这些 0 插在数值的前面，但在符号及任何 0x、0X、0b 或 0B 前缀之后。如果指定了对齐，则将忽略本选项。

示例如下：

```
int i { 42 };
println("|{:06d}|", i);   //|000042|
println("|{:+06d}|", i);  //|+00042|
println("|{:06X}|", i);   //|00002A|
println("|{:#06X}|", i);  //|0X002A|
```

8. L

可选的 L 说明符启用特定于区域设置的格式。此选项仅对算术类型有效，如整数、浮点类型和布尔值。当与整数一起使用时，L 选项指定必须使用特定于区域设置的数字组分隔符。对于浮点类型，这意味着使用特定于区域设置的数字组和小数分隔符。对于以文本形式输出的布尔类型，这意味着使用特定于语言环境的 true 和 false 表示。

使用 L 说明符时，必须将 std::locale 实例作为第一个参数传递给 std::format()。这只适用于 format()，不适用于 print()和 println()。以下是一个使用 nl 语言环境格式化浮点数的示例：

```
float f { 1.2f };
cout << format(std::locale{ "nl" }, "|{:Lg}|\n", f);  //|1,2|
```

第 21 章讨论了本地化。

2.2.6　格式化转义字符和字符串

C++23

C++23 允许使用?类型说明符格式化转义字符串和字符。这种用例并不经常出现，但它对日志记录和调试目的很有帮助。输出类似于在代码中编写字符串和字符文字的方式：它们以双引号或单引号开头和结尾，并且使用转义字符序列。表 2-1 显示了使用转义格式时某些字符的输出。

表 2-1　使用转义格式时某些字符的输出

字符	转义输出
制表符	\t
换行符	\n
回车	\r
反斜杠	\\
双引号	\"
单引号	\'

双引号的转义仅在输出是双引号字符串时发生，而单引号的转义只在输出是单引号字符时发生。不可打印字符的转义输出是\u{十六进制码位}。

以下是一些示例：

```
println("|{:?}|", "Hello\tWorld!\n");   //|Hello\tWorld!\n|
println("|{:?}|", "\"");                 //|"\""|
println("|{:?}|", '\'');                 //|'\''|
println("|{:?}|", '"');                  //|'"'|
```

2.2.7　格式化范围

C++23

第 1 章介绍了用于存储多个数据元素的 std::vector、array 和 pair 容器。第 18 章“标准库容器”

介绍了标准库提供的许多其他容器。从 C++23 开始，可以直接格式化这样的元素范围。对于 vector 和 array 等范围，默认情况下，输出被方括号包围，单个元素用逗号分隔。如果范围的元素是字符串，则默认情况下会转义它们的输出。

可以使用嵌套格式说明符控制范围的格式。一般形式如下：

```
[[fill]align][width][n][range-type][:range-underlying-spec]
```

方括号之间的所有内容都是可选的。与其他格式说明符一样，fill 指定填充字符，align 指定输出的对齐方式，width 指定输出字段的宽度。如果指定了 n，则输出将不包含范围的开括号和闭括号。范围类型可以是表 2-2 中的类型之一。

表 2-2　范围类型

范围类型	描述
m	仅适用于具有两个元素的 pair 和 tuple。默认情况下，这些内容用括号括起来，用逗号分隔。如果指定了 m，则它们不会被任何类型的括号包围，这两个元素用“:”分隔
s	像字符串一样格式化范围(不能与 n 或 range-underlying-spec 组合)
?s	像转义字符串一样格式化范围(不能与 n 或 range-underlying-spec 组合)

range-underlying-spec 是范围中各个元素的可选格式说明符。范围说明符可以嵌套多层。如果元素依然是范围(如 vector 的 vector)，那么 range-underlying-spec 是另一个范围格式说明符，以此类推。

让我们来看一些例子。首先格式化一个数字的 vector：

```
vector values { 11, 22, 33 };
println("{}", values);          // [11, 22, 33]
println("{:n}", values);        // 11, 22, 33
```

如果你想替换开始和结束的方括号，可将 n 说明符与你自己的开始和结束字符包围的格式说明符组合在一起。例如，以下内容用花括号将输出括起来。要在输出中出现的花括号需要转义为{{和}}。

```
println("{{{:n}}}", values);    // {11, 22, 33}
```

下面提供了整个范围的格式说明符。对于这两种情况，范围都在一个 16 个字符宽的字段的中心输出，其中*作为填充字符。对于第二种情况，n 指定应省略开括号和闭括号：

```
println("{:*^16}", values);     // **[11, 22, 33]**
println("{:*^16n}", values);    // ***11, 22, 33***
```

以下内容没有为整个范围提供显式说明符，但它确实指定了如何格式化单个元素。这种情况下，单个元素在一个六个字符宽的字段的中心输出，其中*作为填充字符：

```
println("{::*^6}", values);     // [**11**, **22**, **33**]
```

这可以再次与 n 说明符结合使用：

```
println("{:n:*^6}", values);    // **11**, **22**, **33**
```

下面是一些格式化 string 的 vector 的例子：

```
vector strings { "Hello"s, "World!\t2023"s };
println("{}", strings);         // ["Hello", "World!\t2023"]
println("{:}", strings);        // ["Hello", "World!\t2023"]
println("{::}", strings);       // [Hello, World!      2023]
println("{:n:}", strings);      // Hello, World!      2023
```

如果你有一个字符 vector，你可以将它们格式化为单个字符，或者你可以使用 s 或?s 将整个 vector 视为字符串类型：

```
vector chars { 'W', 'o', 'r', 'l', 'd', '\t', '!' };
println("{}", chars);          // ['W', 'o', 'r', 'l', 'd', '\t', '!']
println("{::#x}", chars);      // [0x57, 0x6f, 0x72, 0x6c, 0x64, 0x9, 0x21]
println("{:s}", chars);        // World      !
println("{:?s}", chars);       // "World\t!"
```

以下是输出 pair 的一些示例。默认情况下，一对元素用括号而不是方括号括起来，两个元素用逗号分隔。使用 n 说明符可以删除开括号和闭括号。m 说明符删除了括号，并用“:”分隔元素。

```
pair p { 11, 22 };
println("{}", p);              // (11, 22)
println("{:n}", p);            // 11, 22
println("{:m}", p);            // 11: 22
```

最后，这是一些输出 vector 的 vector 的例子：

```
vector<vector<int>> vv { {11, 22}, {33, 44, 55} };
println("{}", vv);             // [[11, 22], [33, 44, 55]]
println("{:n}", vv);           // [11, 22], [33, 44, 55]
println("{:n:n}", vv);         // 11, 22, 33, 44, 55
println("{:n:n:*^4}", vv);     // *11*, *22*, *33*, *44*, *55*
```

2.2.8 支持自定义类型

可以扩展格式库以添加对自定义类型的支持。这涉及编写 std::formatter 类模板的特化版本，该模板包含两个方法模板：parse()和 format()。在读到此处时，你还不能理解这个例子中的所有语法，因为用到了以下技术：

- constexpr 函数，将在第 9 章讨论。
- 模板特化，成员函数模板以及缩写模板函数语法，将在第 12 章讨论。
- 异常，将在第 14 章讨论。
- 迭代器，将在第 17 章讨论。

尽管如此，为完整起见，并让你了解可能的情况，让我们看看在本书中进一步学习之后，将如何实现自定义格式化程序，届时你可以回到这个例子。

假设有一个用来存储键值对的类：

```
class KeyValue
{
   public:
      KeyValue(string_view key, int value) : m_key { key }, m_value { value } {}

      const string& getKey() const { return m_key; }
      int getValue() const { return m_value; }

   private:
      string m_key;
      int m_value { 0 };
};
```

可以通过编写以下类模板特化来实现 KeyValue 对象的自定义 formatter。此自定义格式化器还支持：

- 自定义格式说明符：{:k}只输出键，{:v}只输出值，{:b}和{}同时输出键和值。
- 嵌套格式说明符：这些说明符指定键和/或值的可选格式。语法如下：{:b:KeyFormat:ValueFormat}。

```
template <>
class std::formatter<KeyValue>
{
   public:
      constexpr auto parse(auto& context)
      {
         string keyFormat, valueFormat;
         size_t numberOfParsedColons { 0 };
         auto iter { begin(context) };
         for (; iter != end(context); ++iter) {
            if (*iter == '}') { break; }

            if (numberOfParsedColons == 0) { // Parsing output type
               switch (*iter) {
                  case 'k': case 'K':      // {:k format specifier
                     m_outputType = OutputType::KeyOnly;      break;
                  case 'v': case 'V':      // {:v format specifier
                     m_outputType = OutputType::ValueOnly;    break;
                  case 'b': case 'B':      // {:b format specifier
                     m_outputType = OutputType::KeyAndValue;  break;
                  case ':':
                     ++numberOfParsedColons;  break;
                  default:
                     throw format_error { "Invalid KeyValue format." };
               }
            } else if (numberOfParsedColons == 1) { // Parsing key format
               if (*iter == ':') { ++numberOfParsedColons; }
               else { keyFormat += *iter; }
            } else if (numberOfParsedColons == 2) { // Parsing value format
               valueFormat += *iter;
            }
         }
         // Validate key format specifier.
         if (!keyFormat.empty()) {
            format_parse_context keyFormatterContext { keyFormat };
            m_keyFormatter.parse(keyFormatterContext);
         }
         // Validate value format specifier.
         if (!valueFormat.empty()) {
            format_parse_context valueFormatterContext { valueFormat };
            m_valueFormatter.parse(valueFormatterContext);
         }
         if (iter != end(context) && *iter != '}') {
            throw format_error { "Invalid KeyValue format." };
         }
         return iter;
      }

      auto format(const KeyValue& kv, auto& ctx) const
```

```
        {
            switch (m_outputType) {
                using enum OutputType;
                case KeyOnly:
                    ctx.advance_to(m_keyFormatter.format(kv.getKey(), ctx));
                    break;
                case ValueOnly:
                    ctx.advance_to(m_valueFormatter.format(kv.getValue(), ctx));
                    break;
                default:
                    ctx.advance_to(m_keyFormatter.format(kv.getKey(), ctx));
                    ctx.advance_to(format_to(ctx.out(), " - "));
                    ctx.advance_to(m_valueFormatter.format(kv.getValue(), ctx));
                    break;
            }
            return ctx.out();
        }
    private:
        enum class OutputType { KeyOnly, ValueOnly, KeyAndValue };
        OutputType m_outputType { OutputType::KeyAndValue };
        formatter<string> m_keyFormatter;
        formatter<int> m_valueFormatter;
};
```

parse()成员函数负责解析范围[begin(context), end(context)]内的格式说明符。它将解析格式说明符的结果存储在 formatter 类的数据成员中，并且应该返回一个迭代器，该迭代器指向解析格式说明符字符串结束后的下一个字符。其中两个数据成员是 formatter<string>类型的 m_keyFormatter 和 formatter<int>类型的 m_valueFormatter，分别处理格式说明符的 KeyFormat 和 ValueFormat 部分的解析。

format()成员函数根据 parse()解析的格式规范格式化第一个实参，将结果写入 tx.out()，并返回一个指向输出末尾的迭代器。std::format_to()函数类似于 std::format()，但接收一个输出迭代器(指出输出的写入位置)。

可以这样测试 KeyValue formatter：

```
const size_t len { 34 }; // Label field length
KeyValue kv { "Key 1", 255 };
println("{:>{}} {}",   "Default:", len, kv);
println("{:>{}} {:k}", "Key only:", len, kv);
println("{:>{}} {:v}", "Value only:", len, kv);
println("{:>{}} {:b}", "Key and value with default format:", len, kv);
println("{:>{}} {:k:*^11}",    "Key only with special format:", len, kv);
println("{:>{}} {:v::#06X}",   "Value only with special format:", len, kv);
println("{:>{}} {::*^11:#06X}", "Key and value with special format:", len, kv);
try {
    auto formatted { vformat("{:cd}", make_format_args(kv)) };
    println("{}", formatted);
} catch (const format_error& caught_exception) {
    println("{}", caught_exception.what());
}
```

输出如下：

```
                    Default: Key 1 - 255
                  Key only: Key 1
                Value only: 255
Key and value with default format: Key 1 - 255
    Key only with special format: ***Key 1***
  Value only with special format: 0X00FF
Key and value with special format: ***Key 1*** - 0X00FF
Invalid KeyValue format.
```

作为练习，你可以在键和值之间添加对自定义分隔符的支持。有了自定义格式化器，可能性是无穷的，而且一切都是类型安全的！

2.3　本章小结

本章讨论了 C++的 string 和 string_view 类，并讨论了为什么应该用这两个类替换旧式的 C 风格字符数组。还讲解了一些辅助函数，这些函数可以简化数值和字符串之间的双向转换过程，还介绍了原始字符串字面量的概念。

本章最后讨论了字符串格式库，本书中的所有示例都使用了该库。它是一种强大的机制，可以通过细粒度的控制对字符串进行格式化。

第 3 章将讨论良好编码风格的指导原则，包括代码文档、分解、命名约定和代码格式等提示。

2.4　练习

通过完成以下习题，可以巩固本章涉及的知识点。所有练习的答案都可扫描封底二维码下载。但是，如果你被某些问题难住，请先重新阅读本章的对应内容，以尝试自己找到答案，再从网站上查看解决方案。

练习 2-1　编写一个程序，要求用户输入两个字符串，然后使用三向比较运算符将其按字母表顺序打印出来。为了获取一个字符串，可以使用 std::cin 流，曾在第 1 章对其简要介绍。第 13 章将详细解释输入和输出。但是现在，只需要知道如何从控制台中读取字符串。要终止该行，只需要按 Enter 键。

```
std::string s;
getline(cin, s1);
```

练习 2-2　编写一个程序，要求用户提供源字符串(=haystack)、要在源字符串中查找的字符串(=needle)以及替换字符串。编写一个包含 3 个参数的函数：haystack、needle 和 replacement string，该函数返回一个 haystack 的副本，其中所有的 needle 都被替换为 replacement string。要求使用 std::string，不使用 string_view。你将使用哪种类型的参数，为什么？在 main()中调用此函数并打印所有字符串以进行验证。

练习 2-3　修改练习 2-2 中的程序，并尽可能多地使用 std::string_view。

练习 2-4　编写一个程序，要求用户输入未知数量的浮点数，并将所有数字存储在 vector 中。键入每个数字之后都应按 Enter 键。当用户输入数字 0 时，停止输入更多数字。要从控制台读取浮点数，请使用 cin，方法与第 1 章中输入整数值的方法相同。使用一个两列的表来格式化所有数字，其中每

列以不同的方式格式化数字。表中的每一行对应一个输入的数字。

练习 2-5 编写一个程序，要求用户输入未知数量的单词。当用户输入*时停止输入。将所有单个单词存储在一个 vector 中。可以使用以下方式输入单个单词：

```
std::string word;
cin >> word;
```

输入完成后，计算最长单词的长度。最后，按列输出所有单词，一行五个。列的宽度基于最长的单词。输出以列为中心的单词，并用|字符分隔列。

第3章 编码风格

本章内容

- 编写代码文档的重要性以及可以使用的注释风格
- 分解(decomposition)的含义及用法
- 什么是命名约定
- 什么是格式规则

如果你每天花费数小时使用键盘编写代码，你应该为你的工作感到骄傲。编写可以完成任务的代码只是程序员全部工作的一部分而已，任何人都可以学习编写基本的代码，编写具有风格的代码才算真正掌握了编码。

本章讲述如何编写风格优秀的代码，还将展示几种C++风格。简单地改变代码的风格可以令其看起来变化极大。例如，Windows 程序员编写的C++代码通常具有自己的风格，使用了 Windows 的约定。macOS 程序员编写的 C++代码与之相比几乎是完全不同的语言。如果打开的 C++源代码一点都不像你了解的 C++，接触几种不同的风格有助于避免你陷入迷惘。

3.1 良好外观的重要性

编写风格上“良好”的代码很费时间。你或许在几小时内就可以匆匆写出解析 XML 文件的程序。而编写功能分离、注释充分、结构清晰的相同程序可能要花费更长时间。这么做值得吗？

3.1.1 事先考虑

如果一名新程序员在一年之后不得不使用你的代码，你对代码有多少信心？本书作者的一个朋友面对日益混乱的网络应用程序代码，让他的团队联想到：一年后加入的一名实习生。如果没有文档，而函数有好几页长，这名可怜的实习生如何才能赶上代码的进度？编写代码时，可以假定某个新人甚至是你自己在将来不得不维护这些代码。你还记得代码如何运行吗？如果你不能提供帮助会怎么样？由于良好的代码便于阅读和理解，因此不存在这些问题。

3.1.2 良好风格的元素

很难列举“风格良好”的代码具有的特征。随着时间的推移，你会发现自己喜欢的风格，并从他人编写的代码中借鉴有用的技巧。或许更重要的是，你遇到的糟糕代码可以教会你应该避免什么样的风格。当然，良好的代码有一些共同的原则，本章将就此进行讨论。

- 文档
- 分解
- 命名
- 语言的使用
- 格式

3.2 为代码编写文档

在编程环境下，文档通常指源文件中的注释。当编写相关代码时，注释用来说明你当时的想法。注释给出的信息并不能通过阅读代码轻易地获取。

3.2.1 使用注释的原因

很明显，使用注释是一个好主意，但为什么代码需要注释？有时程序员意识到注释的重要性，但没有完全理解为什么注释如此重要。使用注释有几个原因，本章将一一解释它们。

1. 说明用途的注释

使用注释的原因之一是说明客户如何与代码交互。通常而言，开发人员应当能够根据函数名、返回值的类型以及参数的类型和名称来推断函数的功能。但是，代码本身不能解释一切。函数的前置条件、后置条件(注释：前置条件指客户代码调用函数前必须满足的条件，后置条件指函数执行完毕后必须满足的条件)及函数可能抛出的异常，都是可以在注释中解释的内容。在笔者看来，只有当注释能提供有用的信息时才添加注释，例如前置和后置条件以及异常，否则，忽略注释也是可以接受的。经验丰富的程序员能可靠地确定这一点，但经验不足的开发人员则未必能做出正确的决策。因此，一些公司制定规则，要求模块或头文件中每个公有访问的函数或方法都应该带有注释，明确列出每个方法的目的、参数、返回值、前置和后置条件及可能抛出的异常。

通过注释，可用自然语言陈述在代码中无法陈述的内容。例如，在 C++中无法说明：数据库对象的 saveRecord()成员函数只能在 openDatabase()之后调用，否则将抛出异常。但可以使用注释提示这一限制，如下所示：

```
// Throws:
//   DatabaseNotOpenedException if openDatabase() has not been called yet.
int saveRecord(Record& record);
```

saveRecord()成员函数接收一个对 Record 对象的非 const 引用，用户可能想知道为什么不是 const 引用，所以这是需要在注释中解释的内容：

```
// Parameters:
//   record: If the given record doesn't yet have a database ID, then saveRecord()
//   modifies the record object to store the ID assigned by the database.
// Throws:
//   DatabaseNotOpenedException if openDatabase() has not been called yet.
```

```
int saveRecord(Record& record);
```

C++语言强制要求指定方法的返回类型，但是无法说明返回值实际代表了什么。例如，saveRecord()的声明可能指出这个方法返回 int 类型(这是一种不良的设计决策，见下一节的讨论)，但是阅读这个声明的客户不知道 int 的含义。注释可解释其含义：

```
// Saves the given record to the database.
//
// Parameters:
//   record: If the given record doesn't yet have a database ID, then saveRecord()
//   modifies the record object to store the ID assigned by the database.
// Returns: int
//   An integer representing the ID of the saved record.
// Throws:
//   DatabaseNotOpenedException if openDatabase() has not been called yet.
int saveRecord(Record& record);
```

前面的注释以正式的方式记录了有关 saveRecord()方法的所有内容，包括描述该方法功能的句子。一些公司需要这样正式和详细的文档，但是，我不建议一直使用这种注释方式。例如，第一行是相当无用的，因为函数名的意义是不言自明的。参数的描述和关于异常的注释一样重要，所以这些肯定应该保留。

说明这个版本的 saveRecord()的返回类型到底代表什么是必要的，因为它返回一个通用的 int。但是，更好的设计是返回 RecordID 而不是普通的 int，这样就不需要为返回类型添加任何注释。RecordID 可以是一个具有单个 int 数据成员的简单类，但它传达了更多信息，并且允许在将来根据需要添加更多数据成员。因此，以下是对 saveRecord()的建议：

```
// Parameters:
//   record: If the given record doesn't yet have a database ID, then saveRecord()
//   modifies the record object to store the ID assigned by the database.
// Throws:
//   DatabaseNotOpenedException if openDatabase() has not been called yet.
RecordID saveRecord(Record& record);
```

注意：

如果你的公司并未强制要求为函数编写正式的注释，那么在编写注释时，要遵循常识。那些可通过函数名、返回值类型及形参的类型和名称明显看出的信息，就不必添加到注释中。

有时函数的参数和返回值是泛型，可用来传递任何类型的信息。在此情况下应该清楚地用文档说明所传递的确切类型。例如，Windows 的消息处理程序接收两个参数 LPARAM 和 WPARAM，返回 LRESULT。这些参数和返回值可以传递任何内容，但是不能改变它们的类型。使用类型转换，可以用它们传递简单的整数，或者传递指向某个对象的指针。文档应该是这样的：

```
// Parameters:
//   WPARAM wParam: (WPARAM)(int): An integer representing...
//   LPARAM lParam: (LPARAM)(string*): A string pointer representing...
// Returns: (LRESULT)(Record*)
//   nullptr in case of an error, otherwise a pointer to a Record object
//   representing...
LRESULT handleMessage(WPARAM wParam, LPARAM lParam);
```

公开文档应该描述代码的行为，而不是实现。行为包括输入、输出、错误条件和处理、预期用途

和性能保证。例如，如果一个公开文档描述生成单个随机数的调用，则应指定它不带参数，返回先前指定范围内的整数，并应列出出现问题时可能抛出的所有异常。此公开文档不应解释用于实际生成数字的线性同余算法的详细信息。在针对代码用户的注释中提供过多的实现细节可能是编写注释时最常见的一个错误。

2. 用来说明复杂代码的注释

在实际源代码中，好的注释同样重要。在一个处理用户输入并将结果输出到控制台的简单程序中，阅读并理解所有代码可能很容易。然而，在专业领域，经常需要编写算法复杂或过于深奥而无法通过简单阅读来理解的代码。

考虑下面的代码。这段代码写得很好，但是可能无法一眼就看出其作用。如果以前见过这个算法，你可能会认出它来，但是新人可能无法理解代码的运行方式。

```
void sort(int data[], std::size_t size)
{
   for (int i { 1 }; i < size; ++i) {
      int element { data[i] };
      int j { i };
      while (j > 0 && data[j - 1] > element) {
         data[j] = data[j - 1];
         j--;
      }
      data[j] = element;
   }
}
```

较好的做法是使用注释描述函数的参数、所使用的算法和循环的不变量。不变量(invariant)是执行一段代码的过程中必须为真的条件，例如循环的迭代条件。下面是改良后的函数，顶部的注释解释了参数的含义，函数开头的注释概述了这个算法，行内的注释解释了可能令人感到疑惑的特定行。

```
// Implements the "insertion sort" algorithm.
// data is an array containing the elements to be sorted.
// size contains the number of elements in the data array.
void sort(int data[], std::size_t size)
{
   // The insertion sort algorithm separates the array into two parts--the
   // sorted part and the unsorted part. Each element, starting at position
   // 1, is examined. Everything earlier in the array is in the sorted part,
   // so the algorithm shifts each element over until the correct position
   // is found to insert the current element. When the algorithm finishes
   // with the last element, the entire array is sorted.

   // Start at position 1 and examine each element.
   for (int i { 1 }; i < size; ++i) {
      // Loop invariant:
      //    All elements in the range 0 to i-1 (inclusive) are sorted.

      int element { data[i] };
      // j marks the position in the sorted part where element will be inserted.
      int j { i };
      // As long as the value in the slot before the current slot in the sorted
      // array is higher than element, shift values to the right to make room
      // for inserting element (hence the name, "insertion sort") in the correct
      // position.
```

```
            while (j > 0 && data[j - 1] > element) {
                // invariant: elements in the range j+1 to i are > element.
                data[j] = data[j - 1];
                // invariant: elements in the range j to i are > element.
                j--;
            }
            // At this point the current position in the sorted array
            // is *not* greater than the element, so this is its new position.
            data[j] = element;
        }
    }
```

新代码的长度有所增加，但通过注释，不熟悉排序算法的读者就能理解这个算法。

3. 传递元信息的注释

警告：

本节中提到的所有元信息都来自过去的实践。如今，这种元信息是非常不值得鼓励的，因为如第28章所述，必须使用版本控制系统。版本控制方案提供了带注释的修改历史，包括修改日期、修改人、对每个修改的注释(如果使用得当)，包括对修改请求和bug报告的引用。应当使用描述性注释，分别签入(check-in)、提交每个修改请求或bug修复。使用这样的系统，你不必直接在源代码文件中手动跟踪元信息。

在旧的遗留代码库中，可能遇到用于提供比代码本身更高级别信息的注释。这些元信息提供了有关代码创建的详细信息(但没有描述具体行为的细节)，例如每个函数的原始作者、编写代码的日期、函数所提供的特定功能、对一行代码做出响应的错误号、重新访问代码中可能存在的问题的提醒、更改日志等。以下是一个示例：

```
// Date      | Change
//------------+--------------------------------------------------
// 2001-04-13 | REQ #005: <marcg> Do not normalize values.
// 2001-04-17 | REQ #006: <marcg> use nullptr instead of NULL.

// Author:  marcg
// Date:    110412
// Feature: PRD version 3, Feature 5.10
RecordID saveRecord(Record& record)
{
    if (!m_databaseOpen) { throw DatabaseNotOpenedException { }; }
    RecordID id { getDB()->saveRecord(record) };
    if (id != -1) {     // Added to address bug #142 - jsmith 110428
        record.setId(id);
    }
    // TODO: What if setId() throws an exception? - akshayr 110501
    return id;
}
```

然而，值得重申的是，这种遗留元信息在新代码中没有位置。

4. 版权注释

另一种元信息类型是版权声明。有些公司要求在每个源文件的开头添加此类版权信息。尽管自1886年《伯尔尼公约》以来，实际上并不需要明确写版权声明来拥有你的作品的版权。

3.2.2 注释的风格

每个组织注释代码的方法都不同。在某些环境中，为让代码文档具有统一标准，需要使用特定的风格。在其他环境中，注释的数量和风格由程序员决定。下面给出了注释代码的几种方法。

1. 每行都加入注释

避免缺少文档的方法之一是在每行都包含一条注释。每行都加入注释，可以保证已编写的所有内容都有特定的理由。但在实际中，如果代码非常多，过多的注释会非常笨拙、凌乱和乏味。例如，下面的注释没有意义：

```
int result;                    // Declare an integer to hold the result.
result = doodad.getResult();   // Get the doodad's result.
if (result % 2 ==  0) {        // If the result modulo 2 is 0 ...
   logError();                 // then log an error,
} else {                       // otherwise ...
   logSuccess();               // log success.
}                              // End if/else.
return result;                 // Return the result.
```

代码中的注释好像把每行代码当成容易阅读的故事来讲述。如果读者掌握基本的 C++技能，这完全没有用。这些注释没有给代码引入任何附加信息。例如下面这行：

```
    if (result % 2 == 0) {       // If the result modulo 2 is 0 ...
```

这行代码中的注释只是将代码翻译成自然语言，并没有说明为什么程序员用 2 对结果求模。较好的注释应该是：

```
    if (result % 2 == 0) {       // If the result is even ...
```

修改后的注释给出了代码的附加信息，尽管对于大多数程序员而言这非常明显。用 2 对结果求模是因为代码需要检测结果是不是偶数。

更好的是，如果某个表达式做了一些可能不是对每个人来说都很明显的事情，建议将其转换为具有精心选择的名称的函数。这使代码自文档化，不需要在使用函数的地方编写注释，并生成一段可重用的代码。例如，可以定义一个函数 isEven()，如下：

```
bool isEven(int value) { return value % 2 == 0; }
```

然后就可以这样使用它，不需要任何注释：

```
if (isEven(result)) {
```

尽管注释太多，会使代码冗长、多余，但当代码很难理解时，这样做还是有必要的。下面的代码也是每行都有注释，但是这些注释确实有用。

```
// Calculate the doodad. The start, end, and offset values come from the
// table on page 96 of the "Doodad API v1.6."
result = doodad.calculate(Start, End, Offset);
// To determine success or failure, we need to bitwise AND the result with
// the processor-specific mask (see "Doodad API v1.6," page 201).
result &= getProcessorMask();
// Set the user field value based on the "Marigold Formula."
```

```
// (see "Doodad API v1.6", page 136)
setUserField((result + MarigoldOffset) / MarigoldConstant + MarigoldConstant);
```

这段代码的环境不明，但注释说明了每行代码的作用。如果没有注释，就很难解释与&相关的计算和神秘的“Marigold Formula”。

注意：

通常没必要给每行代码都添加注释，但当代码非常复杂，需要这样做时，不要只是将代码翻译成自然语言，而要解释代码实际上在做什么。

2. 前置注释

团队可能决定所有的源文件都以标准的注释开头。可以在该位置记录程序和特定文件的重要信息。在每个文件顶部加入的说明信息有：

- 版权信息
- 文件或类的简要说明
- 未完成的特性*
- 已知的bug(通常由你的bug和功能跟踪系统处理，见第30章)。

下面列出了一些永远不应该包含在此类注释中的信息示例，因为这些信息是由版本控制系统自动处理的(见第28章)。

- 最近的修改日期
- 原始作者
- 前面所讲的修改日志
- 文件实现的特性ID

下面是一个前置注释的示例：

```
// Implements the basic functionality of a watermelon. All units are expressed
// in terms of seeds per cubic centimeter. Watermelon theory is based on the
// white paper "Algorithms for Watermelon Processing."
//
// The following code is (c) copyright 2023, FruitSoft, Inc. ALL RIGHTS RESERVED
```

3. 固定格式的注释

以标准格式编写可被外部文档生成器解析的注释是一种日益流行的编程方法。在Java语言中，程序员可用标准格式编写注释，允许JavaDoc工具自动为项目创建超链接文档。对于C++而言，免费工具Doxygen (doxygen.org)可解析注释，自动生成HTML文档、类图、UNIX man页面和其他有用的文档。Doxygen甚至可辨别并解析C++程序中JavaDoc格式的注释。下面的代码给出Doxygen可以识别的JavaDoc格式的注释。

```
/**
 * Implements the basic functionality of a watermelon
 * TODO: Implement updated algorithms!
 */
export class Watermelon
{
   public:
       /**
        * @param initialSeeds The starting number of seeds, must be > 5.
        * @throws invalid_argument if initialSeeds <= 5.
```

```
     */
    Watermelon(std::size_t initialSeeds);

    /**
     * Computes the seed ratio, using the Marigold algorithm.
     * @param slow Whether or not to use long (slow) calculations.
     * @return The marigold ratio.
     */
    double calculateSeedRatio(bool slow);
};
```

Doxygen 可识别 C++语法和特定的注释指令，例如@param 和@return，并生成定制的输出。图 3-1 给出了 Doxygen 生成的 HTML 类参考的示例。

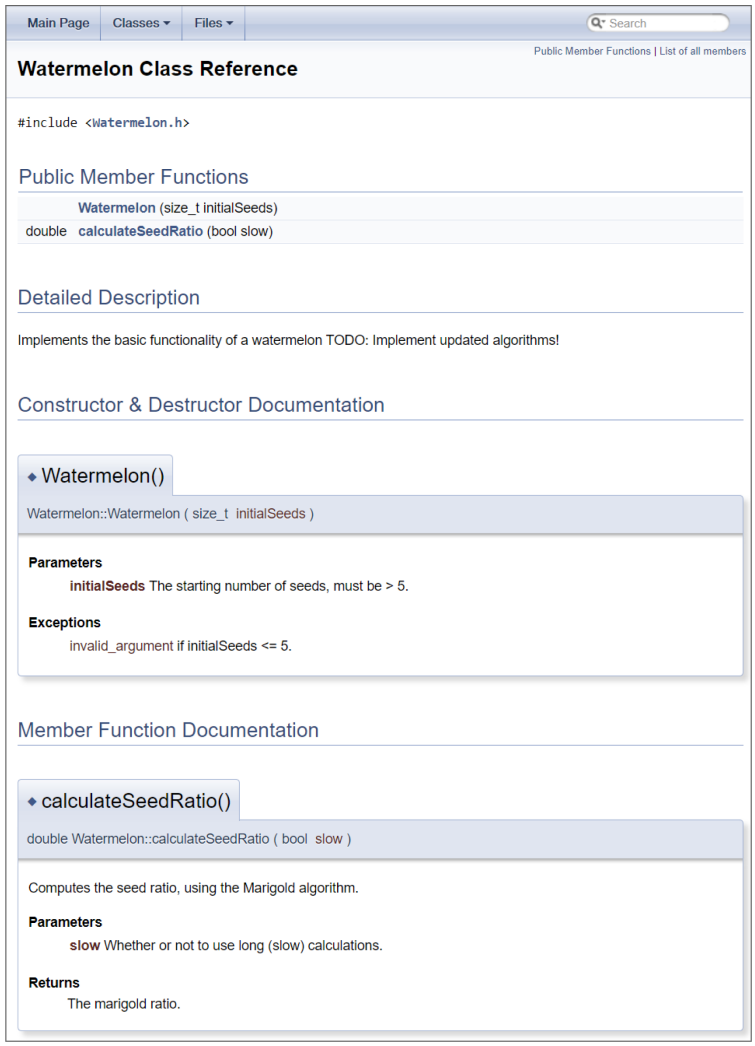

图 3-1　Doxygen 生成的 HTML 类参考的示例

注意，你仍然应当避免编写无用的注释，在使用工具自动生成文档时同样如此。分析一下前面代码中的 Watermelon 构造函数。它的注释忽略了说明信息，只描述参数和异常。在下例中，添加说明信息是多余的。

```
/**
 * The Watermelon constructor.
 * @param initialSeeds The starting number of seeds, must be > 5.
 * @throws invalid_argument if initialSeeds <= 5.
 */
Watermelon(std::size_t initialSeeds);
```

自动生成如图 3-1 所示的文档在开发时很有用，因为这些文档允许开发人员浏览类和类之间关系的高层描述。团队可以方便地定制类似 Doxygen 的工具，以处理所采用的注释格式。理想情况下，团队应该专门布置一台计算机来编写日常文档。

4. 特殊注释

通常应根据需要编写注释，下面是在代码内使用注释的一些指导方针：

- 在添加注释前，首先考虑能否通过修订代码来避免使用注释。例如，重命名变量、函数和类，重新排列代码步骤的顺序，引入完好命名的中间变量等。
- 假想某人正在阅读你的代码。如果有一些不太明显的微妙之处，就应当加上注释。
- 不要在代码中加入自己姓名的缩写。源代码控制解决方案会自动跟踪这类信息。
- 如果处理不太明显的 API，应在解释 API 的地方包含对 API 文档的引用。
- 更新代码时记得更新注释。如果代码的文档中充斥着错误信息，会让人非常困惑。
- 如果使用注释将某个函数分为多节，考虑这个函数能否分解为多个更小的函数。
- 尽量避免使用冒犯性或令人反感的语言，因为你不知道将来谁会查看代码。
- 开一些内部的玩笑通常没有问题，但应该让经理检查是否合适。

5. 自文档化代码

编写良好的代码并非总是需要充裕的注释，优秀的代码本身就容易阅读。如果给每行代码都加入注释，考虑是否可重写这些代码，以更好地匹配注释中所讲的内容。例如给函数、参数、变量、类等使用描述性名称。合理使用 const，也就是说，如果不准备修改变量，就将其标记为 const。重新排列函数中步骤的顺序，使人更容易理解其作用。引入命名良好的中间变量，使算法更易懂。记住 C++ 是一门语言，其主要目的是告诉计算机做什么，但语言的语义也可以向读者解释其含义。

编写自文档化(self-documenting)代码的另一种方法是将代码分解为小段。后面将详细介绍分解。

注意：
优秀的代码本身就容易阅读，注释只需要提供有用的附加信息。

3.3 分解

分解(decomposition)指将代码分为小段。如果打开一个源代码文件，发现一个有 300 行的函数，其中有大量嵌套的代码块，在编程的世界里没有什么比这更令人恐惧了。理想状况下，每个函数或方法都应该只完成一个任务。任何非常复杂的子任务都应该分解为独立的函数或方法。例如，如果有人问你某个方法做什么，你回答“首先做 A，然后做 B，最后，如果满足条件 C，那么做 D，否则做 E”，就应该将该方法分割为辅助方法 A、B、C、D、E。

这种分解并不精密。某些程序员认为，函数的长度不该超过一页，这或许是一个很好的经验法则，但有时，某个只有 1/4 页的代码段也需要分解。另一个经验法则是，如果只查看代码的格式，而不阅读其实际内容，代码在任何区域都不应该显得太拥挤。例如，图 3-2 和图 3-3 被故意弄模糊了，看不清内容。但显然，图 3-3 中代码的分解优于图 3-2。

图 3-2　代码的分解(1)

图 3-3　代码的分解(2)

3.3.1　通过重构分解

喝口咖啡，进入编程状态，开始飞快地编写代码，代码的行为确实符合预期，但远远谈不上优雅。每个程序员都常常这么做。在某个项目中，有时会在短时间内编写大量代码，此时效率最高。在修改代码的过程中，也会得到大量代码。当有新的要求或者修订 bug 时，会对现有的代码进行少量改动。计算机术语 cruft 就是指逐渐累积少量的代码，最终把曾经优雅的代码变成一堆补丁和特例。

重构(refactoring)指重新构建代码的结构。Martin Fowler 的《重构：改进现有代码的设计》第二版是关于重构最有影响力的书籍之一(见附录 B 中的参考书目)。以下列表包含重构代码的一些示例技术：

- 增强抽象的技术
  - **封装数据成员**：将数据成员设置为私有，使用获取器方法和设置器方法访问它们。
  - **使类型通用**：创建更通用的类型，以更好地共享代码。
- 分割代码以使其更合理的技术
  - **提取成员函数**：将较大成员函数的一部分转换成便于理解的新成员函数。
  - **提取类**：将现有类的部分代码转移到新类中。
- 改善代码名称和位置的技巧
  - **移动成员函数或数据成员**：移到更合适的类或源文件中。
  - **重命名成员函数或数据成员**：改为更能体现其作用的名称。
  - **上移(pull up)**：在 OOP 中，移到基类中。
  - **下移(push down)**：在 OOP 中，移到派生类中。

无论代码一开始就是一堆难以理解的稠密代码还是逐渐变成这样的，都需要重构，以定期清理堆

积的代码。通过重构，再次访问已有的代码，重写代码，使代码更容易阅读和维护。重构是重新考虑代码分解的一次机会，如果代码的目的已经改变，或代码在一开始就没有分解，当重构代码时，应扫描一下代码，判断是否需要将其分解为更小的部分。

重构代码时，必须依靠测试框架来捕获可能引入的缺陷。第 30 章讨论的单元测试十分适于帮助在重构期间捕获错误。

3.3.2　通过设计分解

如果使用模块化分解，并通过考虑哪些部分可以推迟到以后来处理每个模块和函数，程序通常不会像在编码时完整实现每个功能的代码那样密集，结构也更合理。

当然，仍然应该在编写代码之前设计程序。

3.3.3　本书中的分解

本书有很多分解示例。许多情况下，方法都没有给出实现代码，因为实现代码与示例无关，并且占用太多篇幅。

3.4　命名

C++编译器有几个命名规则：

- 名称可以包含大小写字母、数字和下画线。
- 字母不限于英语字母表，也可以是任何语言的字母，如日语、阿拉伯语等。
- 名称不能以数字开头(例如 9to5)。
- 包含两个下画线的名称(例如 my__name)是标准库的保留名称，不应当使用。
- 以下画线开头跟着大写字母的名称(例如_Name)是标准库的保留名称，不应当使用。
- 全局命名空间中以下画线开头的名称(例如_name)是保留的，不应当使用。

除了这些规则之外，名称的存在只是为了帮助你和同事处理程序的各个元素。考虑此目的，程序员使用不明确或不适当的名称的频率之高令人惊讶。

3.4.1　选择恰当的名称

变量、方法、函数、参数、类或命名空间的名称应能精确描述其目的。名称还可表达额外的信息，例如类型或者特定用法。当然，真正的考验是其他程序员是否理解你试图通过某个名称传达的意思。

命名并没有固定的规则，但组织可能制定命名规则。然而，有些名称通常是不恰当的。表 3-1 显示了一些适当的名称和不当的名称。

表 3-1　适当的和不当的名称

适当的名称	不当的名称
sourceName、destinationName 区别两个对象	thing1、thing2 太笼统
m_nameCounter 表明了数据成员身份	m_NC 太简单，太模糊
calculateMarigoldOffset() 简单，明确	doAction() 太宽泛，不准确

(续表)

适当的名称	不当的名称
mTypeString 赏心悦目	typeSTR256 只有计算机才会喜欢的名称
g_settings 表示全局身份	m_IHateLarry 不恰当的内部玩笑
errorMessage 描述性名称	string 非描述性名称
sourceFile、destinationFile 无缩写	srcFile、dstFile 缩写

3.4.2 命名约定

选择名称通常不需要太多的思考和创造力。许多情况下，可使用标准的命名技术。下面给出可使用标准名称的数据类型。

1. 计数器

以前编程时，代码可能把变量“i”用作计数器。程序员习惯把 i 和 j 分别用作计数器和内部循环计数器。然而要小心嵌套循环。当想表示“第 j 个”元素时，经常会错误地使用“第 i 个”元素。使用二维数据时，与使用 i 和 j 相比，将 row 和 column 用作索引会更容易。有些程序员更喜欢使用 outerLoopIndex 和 innerLoopIndex 等计数器，一些程序员甚至不赞成将 i 和 j 用作循环计数器。

2. 前缀

许多程序员在变量名称的开头用一个字母提供与变量的类型或用法有关的信息。然而，许多程序员并不赞成使用前缀，因为这会使相关代码在将来难以维护。例如，如果某个成员变量从静态变为非静态，这意味着所有用到这个名称的地方都要修改。如果你没有修改它们的名称，名称会继续表达语义，实际上这个语义是错误的。

当然，通常别无选择，只能遵循公司的约定。表 3-2 显示了一些可用的前缀。

表 3-2 可用的前缀

前缀	示例名称	前缀的字面意思	用法
m m_	mData m_data	“成员”	类的数据成员
s ms ms_	sLookupTable msLookupTable ms_lookupTable	“静态”	静态变量或数据成员
k	kMaximumLength	“konstant”，德语表示的常量	常量值。有些程序员使用全大写字母的名称(不使用前缀)来表示常量
b is	bCompleted isCompleted	“布尔值”	表示布尔值

3. 匈牙利表示法

匈牙利表示法是关于变量和数据成员的命名约定，在 Microsoft Windows 程序员中很流行。其基本思想是使用更详细的前缀而不是一个字母(例如 m)表示附加信息。下面这行代码显示了匈牙利表示法的用法：

```
char* pszName; // psz means "pointer to string, zero-terminated"
```

术语“匈牙利表示法”源于其发明者 Charles Simonyi 是匈牙利人。也有人认为这准确地反映了一个事实：使用匈牙利表示法的程序好像是用外语编写的。为此，一些程序员不喜欢匈牙利表示法。本书使用前缀，而不使用匈牙利表示法。合理命名的变量不需要前缀以外的附加上下文信息，例如，用 m_name 命名数据成员就足够了。

注意：
好的名称会传递与用途有关的信息，而不会使代码难以阅读。

4. 获取器和设置器

如果类包含了数据成员，例如 m_status，习惯上会通过获取器 getStatus()和设置器 setStatus()访问这个成员。要访问布尔数据成员，通常将 is(而非 get)用作前缀，例如 isRunning()。C++语言并未指定如何命名这些方法，但组织可能会采用这种命名形式或类似的形式。

5. 大写

在代码中大写名称有多种不同的方法。与编码风格的大多数元素类似，最重要的是团队采用一个标准的方法，且所有成员都采用这种方法。如果某些程序员用全小写字母命名类，并用下画线表示空格(priority_queue)，而另一些程序员将每个单词的首字母大写(PriorityQueue)，代码将乱成一团。变量和数据成员几乎总以小写字母开头，并用下画线(my_queue)或大写字母(myQueue)分隔单词。在 C++中，函数和方法通常将首字母大写。但是，正如你所见，本书采用小写风格的函数和方法，把它们与类名区别开来。大写字母可用于为类和数据成员名指明单词的边界。

6. 把常量放到命名空间中

假定编写一个带图形用户界面的程序。这个程序有几个菜单，包括 File、Edit 和 Help。用常量代表每个菜单的 ID。Help 是代表 Help 菜单 ID 的一个好名字。

名称 Help 一直运行良好，直到有一天在主窗口上添加了一个 Help 按钮。还需要一个常量来代表 Help 按钮的 ID，但是 Help 已经被使用了。

在此情况下，建议将常量放到不同的命名空间中，命名空间参见第 1 章。可以创建两个命名空间：Menu 和 Button。每个命名空间中都有一个 Help 常量，其用法为 Menu::Help 和 Button::Help，更推荐的方法是使用枚举类型，参见第 1 章。

3.5 使用具有风格的语言特性

C++语言允许执行各种非常难以读懂的操作。看看下面的古怪代码：

```
i++ + ++i;
```

这行代码很难懂，但更重要的是，C++标准没有定义它的行为。问题在于 i++使用了 i 的值，还递增了 i 的值。C++标准没有说明什么时候递增其值，这个副作用(递增)只有在“;”之后才能看到，

但是编译器执行到这一行时，可以在任意点执行递增。无法知道哪个 i 值会用于++i 部分，在不同的编译器和平台上执行这行代码，会得到不同的值。

以下面的表达式为例：

```
a[i] = ++i;
```

在 C++17 中，它的行为是确定的，在对赋值运算的左侧求值之前会保证完成对右侧的所有操作的求值。所以，在本例中，i 首先递增，然后在 a[i]中用作索引。即使如此，为清晰起见，仍然建议避免使用此类表达式。

在使用 C++语言提供的强大功能时，一定要考虑如何以良好的(而不是丑陋的)风格使用语言特性。

3.5.1 使用常量

不良代码经常乱用“魔法数字”。在一些函数中，代码可能使用 2.718 28、24 或 3600 等。为什么呢？这些值有什么含义？具有数学背景的人会发现，第一个数字代表 e 的近似值，但多数人不知道这一点。C++语言提供了常量，可以把一个符号名称赋予某个不变的值，例如 2.718 28、24、3600 等，下面是几个示例：

```
const double ApproximationForE { 2.7182818284590452536 };
const int HoursPerDay { 24 };
const int SecondsPerHour { 3'600 };
```

注意：

标准库包含一组预定义的数学常量，所有这些常量都定义在 std::numbers 命名空间的 <numbers> 中。例如，它定义了 std::numbers::e、pi、sqrt2、phi 等。

3.5.2 使用引用代替指针

以前，C++程序员通常开始学的是 C。在 C 中，指针是按引用传递的唯一机制，多年来一直运行良好。在某些情况下仍然需要指针，但在许多情况下可以用引用代替指针。如果开始学习的是 C，可能认为引用实际上没有给 C++语言增加新的功能，只是引入了一种新的语法，其功能已经由指针提供。

用引用替换指针有许多好处。首先，引用比指针安全，因为引用不会直接处理内存地址，也不会是 nullptr。其次，引用在风格上比指针好，因为引用使用与栈上变量相同的语法，也就是说，它们不要求你使用*明确地记下它们的地址，也不要求你用*明确地解除对它们的引用。引用易于使用，因此将引用加入编码风格库中没有任何问题。遗憾的是，某些程序员认为，如果在函数调用中看到&，被调用的函数将改变对象；如果没有看到&，对象一定是按值传递。而使用引用，就无法判断函数是否将改变对象，除非看到函数原型。这种思路是错误的。用指针传递未必意味着对象将改变，因为参数可能是 const T*。传递指针或引用是否会修改对象，都取决于函数原型是否使用了 const T*、T*、const T&或 T&。因此，只有查看函数原型，才能判断函数是否改变对象。

使用引用的另一个好处是它明确了内存的所有权。如果你编写了一个方法，另一个程序员传递给它一个对象的引用，很明显可以读取并修改这个对象，但是无法轻易地释放对象的内存。如果传递的是一个指针，就不那么明显。需要删除对象来清理内存吗？还是调用者需要这样做？尽管在现代 C++中，含义很明确：任何原始指针都是非拥有的，处理所有权和所有权转移是使用智能指针完成的，如第 7 章所述。

3.5.3 使用自定义异常

C++可以很方便地忽略异常。这一语言的语法没有强制处理异常，理论上，可以很方便地用传统的机制，例如返回特殊值(如-1 或 nullptr)，或者设置错误标志，来编写容错程序。当返回特殊值来处理错误时，可以使用第 1 章介绍的[[nodiscard]]属性强迫函数的调用者处理返回值。

异常提供了更丰富的错误处理机制，自定义异常允许根据需要进行使用。例如，Web 浏览器的自定义异常类型可以包括指定包含错误的网页、发生错误时的网络状态及其他上下文信息的字段。

第 14 章将详细讲述 C++中的异常。

注意：
语言特性是用来帮助程序员的，应该理解并使用有助于形成良好编程风格的特性。

3.6 格式

对于编码格式的争论使许多编程团队四分五裂，友谊荡然无存。在大学时，笔者的一个朋友与同行就 if 语句中的空格使用进行了辩论，辩论十分激烈，人们不得不停下手中的工作以确保一切正常。

如果组织有编码格式标准，你就是幸运的。你或许不喜欢这个标准，但至少不需要讨论这个问题。

如果没有现成的编码格式标准，建议你的组织制定这样的标准。标准化的编码指导原则确保团队中的所有编程人员都遵循相同的命名约定、格式规则等；这样，代码将更趋统一，更容易理解。

有一些可用的自动化工具可以在将代码提交到源代码控制系统之前根据某些规则格式化代码。一些 IDE 内置了此类工具，例如，可以在保存文件时自动格式化代码。

如果团队中的每个人都以自己的方式编写代码，要尽量容忍。你将知道，一些做法只是品位问题，但是另一些做法会使得团队协作变得困难。

3.6.1 关于大括号对齐的争论

或许被议论最多的是在哪里使用界定代码块的大括号。大括号的使用有多种格式，在本书中，除了类、函数和方法名之外，大括号与起始语句放在同一行。下面的代码显示了这种格式(本书整本书都是如此)：

```
void someFunction()
{
    if (condition()) {
        println("condition was true");
    } else {
        println("condition was false");
    }
}
```

这种格式节省了垂直空间，同时仍然通过缩进显示代码块。有些程序员认为，节省垂直空间与现实世界的编码无关。下面显示了一段冗长的代码：

```
void someFunction()
{
    if (condition())
    {
        println("condition was true");
    }
```

```
    else
    {
        println("condition was false");
    }
}
```

有些程序员更自由地使用水平空间，编写的代码如下：

```
void someFunction()
{
    if (condition())
        {
            println("condition was true");
        }
    else
        {
            println("condition was false");
        }
}
```

另一个争论点在于，是否在一条语句周围放置大括号，例如：

```
void someFunction()
{
    if (condition())
        println("condition was true");
    else
        println("condition was false");
}
```

当然，我们不会推荐任何特定的格式。就笔者个人而言，我总是使用大括号，即使是单个语句，因为它可以避免某些写得不好的 C 风格的宏带来的危害(参见第 11 章)，并且在将来添加语句时更安全。

注意：

当选择表示代码块的风格时，最重要的事情是应该能够让读者一眼就看出某个代码块对应的条件。

3.6.2 关于空格和圆括号的争论

单行代码的格式也能够引起争论。再次说明，我同样不支持任何特定的方法，但给出可能遇到的几种格式。

本书在任何关键字之后都会使用空格，在运算符前后都会使用空格，在参数列表或函数调用中的每个逗号之后都会使用空格，并用圆括号表明操作顺序，如下所示：

```
if (i == 2) {
    j = i + (k / m);
}
```

另一种格式将 if 当作函数，在关键字和左括号之间没有空格，如下所示。另外，if 语句内用于明确操作顺序的圆括号也被省略了，因为它们没有语义相关性。

```
if( i == 2 ) {
    j = i + k / m;
}
```

区别十分微妙，请读者自行判断哪种方法更好，然而在此必须指出，if不是函数。

3.6.3 空格、制表符、换行符

空格和制表符的使用并不只是风格上的偏好。如果团队没有使用空格和制表符的约定，当程序员一起工作时会出大问题。最明显的问题是：Alice使用4个空格的制表符缩进代码，而Bob使用5个空格的制表符。当他们使用同一文件时，将无法正确显示代码。如果Bob用制表符重新整理代码格式，同时Alice编辑同样的代码，情况更糟糕，许多源代码控制系统不能合并Alice所做的修改。

大多数(但不是全部)编辑器可设置空格和制表符。某些环境甚至在读取代码时会调整代码的格式，或者即使编写代码时用的是制表符，保存时也总是使用空格。如果环境比较灵活，使用他人的代码会更容易。记住，制表符和空格是不同的，因为制表符的长度不定，而空格始终是空格。

最后，并非所有平台都以相同的方式表示换行。例如，Windows使用\r\n作为换行符，而基于Linux的平台通常使用\n。如果你在公司中使用多个平台，那么需要就使用哪种换行样式达成一致。同样，IDE很可能被配置为使用需要的换行符样式，或者可以使用自动化工具来自动修复换行符，例如，在将代码提交到源代码控制系统时。

3.7 风格的挑战

许多程序员在项目开始时都保证他们将做好每件事。只要变量或参数永远不变，就将其标记为const。所有变量都具有清楚的、简明的、容易阅读的名称。每个开发人员都将左大括号放在后续行，采用标准文本编辑器，并遵循关于制表符和空格的约定。

维持这种层次的格式一直非常困难，原因有很多。当涉及const时，有些程序员不知道如何用它。总会遇到不支持const的旧代码或库函数。例如，假设你正在编写一个接收const参数的函数，并且需要调用一个接收非const参数的遗留函数。如果你无法修改遗留代码使其兼容const，可能是因为它是第三方库，并且你绝对确定遗留函数不会修改其非const参数，经验丰富的程序员会使用const_cast(见第1章)暂时取消变量的const属性，但缺少经验的程序员会取消来自调用函数的const属性，导致程序从不使用const。

有时，标准化的格式会与程序员的个人口味和偏好发生冲突。或许团队文化无法强制使用严格的风格准则。此类情况下，必须判断哪些元素需要标准化(例如变量名称和制表符)，哪些元素可以由个人决定其风格(或许空格和注释格式可以这样)。甚至可以获取或编写脚本，自动纠正格式“bug”，或将格式问题与代码错误一起标记。一些开发环境，例如Microsoft Visual C++，支持根据指定的规则自动格式化代码，这样就很容易编写出始终遵循指定规则的代码。

3.8 本章小结

C++语言提供了许多格式工具，但没有正式说明如何使用这些工具。从根本上讲，风格约定取决于其应用范围和对代码可读性的贡献。如果你是编程团队的一员，在讨论使用什么语言和工具时就应该意识到这个问题。

应该承认，风格是编程的一个重要方面。把代码交给其他人之前，应该检查代码的风格。了解好的编码风格，并采用自己和组织认为有用的约定。

在结束本章时，请记住这句话：

编码时总是应该认为最终维护你代码的人会是一个知道你住在哪里的暴力精神病患者。尽可能写

可读性高的代码。

本章是本书第 I 部分的最后一章，第 II 部分将在较高层次上讨论软件设计。

3.9 练习

通过完成以下习题，可以巩固本章涉及的知识点。所有练习的答案都可扫描封底二维码下载。但是，如果你被某些问题难住，请先重新阅读本章的对应内容，以尝试自己找到答案，再从网站上查看解决方案。

代码注释和编码风格是主观的。以下练习没有一个完美的答案。网站上的解答为练习提供了许多可能的正确答案之一。

练习 3-1 第 1 章讨论了一个雇员记录系统的示例。该系统有一个数据库，该数据库的方法之一是 displayCurrent()。这是该成员函数的实现，并带有一些注释：

```
void Database::displayCurrent() const        // The displayCurrent() member function
{
   for (const auto& employee : m_employees) {  // For each employee...
      if (employee.isHired()) {              // If the employee is hired
         employee.display();                 // Then display that employee
      }
   }
}
```

你发现注释中的错误了吗？为什么？你能写出更好的注释吗？

练习 3-2 第 1 章的雇员记录系统包含一个 Database 类。以下是该类的片段，只有 3 个成员函数。向此代码片段添加适当的 JavaDoc 风格注释。请参阅第 1 章以了解这些成员函数的具体作用。

```
class Database
{
   public:
      Employee& addEmployee(const std::string& firstName,
         const std::string& lastName);
      Employee& getEmployee(int employeeNumber);
      Employee& getEmployee(const std::string& firstName,
         const std::string& lastName);
   // Remainder omitted...
};
```

练习 3-3 下面的类有许多命名问题，你能找出它们并提出更好的名字吗？

```
class xrayController
{
   public:
      // Gets the active X-ray current in µA.
      double getCurrent() const;

      // Sets the current of the X-rays to the given current in µA.
      void setIt(double Val);

      // Sets the current to 0 µA.
      void 0Current();

      // Gets the X-ray source type.
      const std::string& getSourceType() const;
```

```
        // Sets the X-ray source type.
        void setSourceType(std::string_view _Type);

    private:
        double d; // The X-ray current in µA.
        std::string m_src__type; // The type of the X-ray source.
};
```

练习 3-4 给定以下代码片段，格式化代码片段 3 次：首先将大括号单独写为一行，然后缩进每个大括号，最后删除单语句代码块的大括号。本练习让你了解不同的格式样式以及它们对代码可读性的影响。

```
Employee& Database::getEmployee(int employeeNumber)
{
    for (auto& employee : m_employees) {
        if (employee.getEmployeeNumber() == employeeNumber) {
            return employee;
        }
    }
    throw logic_error { "No employee found." };
}
```